DESIGN AND IMPLEMENTATION OF A MOBILE WIRELESS SENSOR NETWORK TESTBED

by

David Michael Johnson

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

May 2010

Copyright © David Michael Johnson 2010

All Rights Reserved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

David Michael Johnson

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Jay Lepreau (signed by: Martin Berzins)
	John B. Carter
	Sneha Kumar Kasera

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of David Michael Johnson in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date	Jay Lepreau (signed by: Martin Berzins) Chair, Supervisory Committee

Approved for the Major Department

Martin Berzins
Chair/Dean

Approved for the Graduate Council

Charles A. Wight
Dean of The Graduate School

ABSTRACT

Network simulation continues to be the dominant method of experimental evaluation in wireless networking. However, much research has established the failure of simulator models to adequately express wireless signal propagation. These shortcomings can lead to incomplete evaluation of wireless protocols and applications. When wireless research includes mobility, real evaluation becomes still more difficult due to the difficulty of creating and controlling mobile nodes in a real environment.

The primary goal of *Mobile Emulab*, the testbed presented in this thesis, is to encourage real mobile wireless research in the wireless community and provide a sound, usable testbed platform for experimentation. Mobile Emulab is both software designed to control and monitor a mobile wireless testbed, and a testbed providing access to mobile wireless resources. The testbed consists of several robots, each with a small computer and small wireless devices called motes, manueverable in an area surrounded by fixed motes. Through a variety of interfaces, remote researchers can control these robots interactively over the Web. Mobile Emulab provides an overhead tracking system that localizes the robots to within 1 cm, providing repeatable positioning and valuable knowledge to researchers studying how signal propagation affects their experiments. Additional software tools that were developed to ease the evaluation process for wireless sensor network researchers can be used by Mobile Emulab experimenters. Finally, the testbed extends Emulab, which provides researchers with well-known experimental interfaces and automation capabilities.

This thesis presents Mobile Emulab's design and implementation, and establishes its usability and utility through several experiments.

For my parents

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I'd like to thank my advisor, Jay Lepreau, for giving me an interesting research project and a chance to build a real collection of systems software, and valuable advice, friendship, and support. By including me in his research group, Jay altered the direction of my career and life. He provided wonderful opportunities to learn about and experience systems software design and development, and I am very thankful to have been part of his life. I also thank Sneha Kumar Kasera and John Carter, members of my committee, for their advice, time, and patience!

I owe Karen Feinauer, who wears the Graduate Coordinator hat in the School of Computing, a massive debt. She has patiently steered me through requirements, answered questions, and prodded and encouraged me to completion of my degree. In another department, or in another university, I might not have graduated.

I am grateful for the significant contributions, ideas, and support from many members of the Flux Research Group—especially Dan Flickinger, Tim Stack, Russ Fish, Leigh Stoller, Mike Hibler, Robert Ricci, Eric Eide, Kirk Webb, and Mark Minor. More generally, the opportunity to work with the members of the Flux Group has helped me increase my knowledge of the computer systems field and given me the ability to partipate in its research community. Most importantly, my experiences in the Flux Group have taught me how to build better systems software!

Finally, I thank my parents for the direction they gave to my life, the encouragement and expectation they provided for my path through higher education, and the care and love with which they have blessed me. This thesis is dedicated to them.

This material is based upon work supported by the National Science Foundation under Grant Nos. 0520311, 0321350, and 0335296. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

CHAPTER 1

INTRODUCTION

Researchers experimenting with mobility in wireless sensor networks face a range of evaluation methods to test their work. The choice of simulation, emulation, or testing on live networks (or a combination thereof) heavily influences evaluation time, cost, and quality. Each type of evaluation method is useful at different times during research and development. For instance, a simulator can be an excellent debugging aid for early development and helpful for testing application scalability beyond what is possible on limited research hardware. Emulated networks provide experimenters with access to real hardware, generally under controlled conditions. Emulated environments provide a measure of repeatability while still allowing researchers to test on real hardware. Finally, evaluation on live networks is an important prerequisite to declaring that an application or protocol really works "in the real world," but it is hard or impossible to control the conditions in a live network.

1.1 Motivation

In the mobile and wireless research community, many research evaluations are performed *only* in simulation. An increasing number of papers also provide experimental data from live or emulated network tests, but the majority evaluate most thoroughly in simulation. There are many reasons that simulation dominates. First and foremost, it tends to be the easiest way to evaluate a protocol or application. Simulations may be as complex or as simple as the researcher deems necessary. Many simulators come prepackaged with different models of wireless signal propagation, basic environment effects, and mobility models. Furthermore, judging by the preponderance of papers that evaluate only by simulation, results from simulation alone is still an accepted means of establishing system quality in the wireless research community.

Simulation tends to be a simpler alternative to real-world experimentation, but it can lack fidelity that real devices and environments provide, especially when considering

wireless and mobile systems. Sometimes this loss of real-world modeling accuracy can significantly affect the evaluation of the protocol or application. Judging from personal interaction with researchers from around the world, the mobile wireless research community generally agrees that simulation is insufficient for many kinds of network experimentation. Simulation simply cannot model effectively many of the physical intricacies of wireless signal propagation [2, 18, 28, 32, 33]. Wireless communication is often inhibited by obstacles or nonuniformities in an environment, which produce multipath and fading effects, or even simple interference, that can be difficult to model in simulation. Slight defects in manufacturing processes or different antennae placement may influence transmission characteristics. For instance, these real world effects can significantly impact the behavior of wireless routing and MAC protocols [33]. Simulation can mask these effects from experimenters, leading to incomplete or even flawed evaluation. When researchers add mobility to their wireless applications and protocols, the situation can only become worse. If simulation cannot accurately model signal propagation in a static environment, how can anyone reasonably expect it to model propagation under motion? Motion simply increases the difficulty of modeling all signal propagation effects in simulation.

There is a clear, strong need for inclusion of evaluation based on emulated or real hardware in real wireless signal propagation environments. Unfortunately, experimentation in these environments presents many difficulties not found in simulation. Researchers should prefer to deploy a dedicated experimental network (a testbed) rather than experiment on production networks since this can avoid external interference that may disrupt experimentation. However, designing and implementing a testbed is difficult and reduces the time available for research and evaluation. If software systems are not in place to maintain and quickly reconfigure basic properties of the testbed, the testbed quickly degenerates into a single use system, lacking flexibility and means to easily extract data from the network.

Wireless testbeds present additional difficulties for researchers. External wireless sources may disrupt experimentation in ways that are hard to observe and understand; removing external sources of wireless interference is not always possible. Repeatability is much more difficult to obtain in wireless testbeds, due to the complex nature of the physical media. There may also be limited channel availability and external wireless sources that interfere with ongoing experiments.

The introduction of sensor network devices to a wireless testbed further increases

the burden on the researcher. Because sensor networks are relatively immature, especially when compared to IP networks, they lack the variety of standard toolchains and application interface software that has existed for many years for IP networks. Many tools, such as `netcat` [13], `nmap` [10], and `tcpdump` [11], exist for IP networks and are invaluable to researchers trying to understand or capture behavior of network protocols. The software that is currently available for use in sensor networks is still maturing, and when combined with the difficulties inherent in dealing with small, resource-constrained embedded devices, protocol and application development and debugging can easily become a nightmare.

Experimentation with real mobile wireless devices presents a final set of issues. Precise, repeatable control and placement of mobile devices (in both time and position) is difficult to achieve. If the experimenter is unconcerned with repeatability, it may be easy to place devices on mobile objects such as people or automobiles. However, if positioning itself, or if relative positioning with respect to time, is important, these methods are insufficient. On the other end of the spectrum are systems that emulate mobility by routing packets through the real device nearest the emulated sending location; however, this does not provide exposure to the effects of real motion and will lack location precision. To achieve repeatable motion, the researcher must develop software and hardware infrastructure (including location and guidance services) to track and control the mobile nodes.

Despite the difficulties inherent in real, mobile wireless experimentation, such experimentation is a valuable part of demonstrating claimed and proper functionality of new network protocols and applications. *In this thesis, we demonstrate that real, mobile wireless sensor network experimentation can be made both practical and useful by creation of an emulation testbed, Mobile Emulab, that provides real wireless devices and mobility.*

1.2 Goals

Several important goals influenced the design of Mobile Emulab. First, it must provide simple, expressive, and precise motion to experimenters. Precise motion enables fine-grained experimental analysis and is a prerequisite for repeatability. The combination of real world motion and wireless devices provides researchers with a useful platform for experimentation. Second, Mobile Emulab's design should keep both hardware and software costs low. This will enable other research groups to more easily create their own

mobile testbeds, although the software must also be easily adapted to different hardware. By fostering easy testbed creation, we can encourage many groups to place testbeds in radically different radio environments. The testbed must be remotely accessible from the Internet and should provide interfaces for controlling and observing motion and collecting experimental data. Finally, it should specifically ease sensor network application testing. Interactive sensor network control interfaces provide experimenters with much greater control, debugging, and exploration capabilities, which already exist for older network environments, such as TCP/IP.

1.3 Contributions

The focus of the research described in this work is the design and implementation of Mobile Emulab, a mobile wireless network testbed for use by remote researchers. Mobile Emulab extends the Emulab [30] network testbed, allowing it to leverage Emulab's powerful capabilities and well-known interfaces. We added wireless devices attached to small computers, in turn mounted atop mobile robotic nodes. To allow researchers to control and dynamically position these nodes from remote sites, we developed control and tracking software for the robots. The control software includes simple path planning and obstacle avoidance algorithms; this support allows us to present simple motion models to researchers and abstracts details of low-level motion control. We track the robots via a computer vision-based localization system. This allows precise positioning and motion, and provides researchers with detailed location information for use in evaluation. Finally, we implemented several applications that enable experimenters to easily explore the wireless characteristics of our environment, and manage and interact with sensor network devices.

In this thesis, I present my contributions to the Emulab mobile sensor network testbed. I contributed substantially to the overall system design, localization subsystem, and wrote application software and libraries to improve testbed utility for experimenters. I made only minor contributions to robot control and monitoring software, although I discuss key aspects of their implementation for clarity.

1.4 Structure

Chapter 2 provides background information about Emulab and discusses several testbeds that relate to this work. Chapter 3 presents the design of the Mobile Emulab software

system and implementation of key services and network protocols. Chapter 4 details the design and development of a computer vision-based localization system. Chapter 5 discusses additional software designed to provide experimenters with more information, control, and logging facilities. Chapter 6 presents several case studies demonstrating experiment interaction with testbed facilities and evaluation of various sensor network applications, which demonstrates that the testbed is a viable and valuable tool for researchers. Finally, Chapter 7 discusses the testbed's impact on the research community and suggests future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

First, this section provides a brief introduction to the Emulab network testbed software, on which Mobile Emulab is based, and highlights its important and useful features. We then discuss a variety of related testbeds.

2.1 Emulab

Emulab is itself a well-known and widely-used network testbed, but more generally is a software framework for controlling testbeds. Emulab enables remote researchers to develop and evaluate network protocols and applications on real hardware. The software automatically instantiates custom network topologies on physical resources present in the testbed, with none of the pain associated with configuration of a single-use testbed in a lab. Emulab software enables full space sharing so that many researchers can run concurrent, but separate, experiments. The capability of the software framework to handle a variety of hardware platforms with slight modification, and the experiment paradigm, coupled to a web interface, provide a strong base platform for supporting the mobile extensions.

Emulab provides an intuitive methodology and interface for researchers. They create *experiments* interactively or via the well-known network simulation language, *ns-2*. When an experiment is "swapped in," Emulab software reserves the requested resources and sets up the network topology along with any custom network parameters. Once the experiment is fully configured, Emulab's event system begins processing any scheduled events, such as running user programs, tweaking link (or other network) parameters, reporting results, etc. Emulab's web interface provides the researcher with current status of nodes and other general information about the experiment. When finished, the researcher may "swap out" the experiment, perhaps saving state for later experimentation via disk imaging.

2.2 Mobile and Fixed Wireless Testbeds

Significant research and development effort has been invested in wireless network testbeds, but few have attempted to add mobility to static nodes. One that has is MiNT [6], a testbed designed to be deployed in small environments. For its wireless platform, MiNT uses desktop PCs containing 802.11 wireless cards with external antennae and attenuation. The antennae extend to mobile robots running on a table. However, the mobile nodes are each confined to a sector in which they can move, lest the wires connecting to the PCs entangle. Finally, MiNT does not provide a localization system, other than presumably the onboard robot odometry. In comparison, our testbed provides experimenters with the ability to move robots anywhere in the experimental area, and to receive ground truth location information.

The ORBIT testbed [26] provides emulated mobility in a multihop network of several hundred 802.11 wireless nodes. Experimenters can run code on PCs that binds to network interfaces on the wireless nodes. Thus, the testbed can emulate mobility by changing the program-interface bindings and provide a rough approximation of motion. In our testbed, we provide real node mobility and thus have no need for proxying packet streams through various emulated devices. We also provide wireless sensor network devices instead of 802.11 nodes.

TWINE [34] is a software framework that provides hybrid simulation, emulation, and live network support for wireless networks. However, TWINE differs from other approaches in that their emulation does not route packets to real interfaces. Instead, the MAC and PHY layers are simulated at a very detailed level so as to provide better modeling realism while retaining repeatability. Both emulation and simulation facilities provide several different mobility simulation models including random, group, and trace-based models.

Finally, there are several fixed wireless sensor network testbeds. MoteLab [29] is a software framework providing access to a building-scale sensor network testbed. Experimenters may request time-slotted reservations over the Internet. MoteLab provides automatic mote programming at experiment swapin and logging of packets sent out of the serial port to a MySQL database. The mote logger application in our testbed (described later in Section 5.3) is inspired by Motelab's mechanism in that they both use the information stored in TinyOS-generated Java Active Message wrapper classes to extract information from packets and store them in a MySQL database. However, our

method also presents this information in multiple, human-readable ways in the database, instead of as a data byte array.

EmStar [14] is a software framework that provides hybrid emulation on real motes, potentially combined with simulation. In emulated mode, code runs on PCs, but physical radios provide real communication effects. EmStar also provides a hybrid mode of operation in which some motes may run the code natively, while others operate in emulation mode. Although the software is not strictly designed for testbed management, it can manage hybrid networks of simulated and emulated motes. The Emulabtestbed, on which our software is based, provides numerous tools for controlling and managing testbed resources.

The Re-Mote Testbed [9] provides novel logging into a MySQL database from motes attached to a control network. The TWIST [16] sensor network testbed provides mote reprogramming and data logging over a control network, and also includes remote power control selection between battery and wall power.

CHAPTER 3

SYSTEM OVERVIEW

Creating a testbed for real, mobile wireless experimentation as discussed in Chapter 1 requires solutions to several subproblems. First, in order to control and maneuver the mobile nodes precisely, the testbed must provide a powerful localization service capable of providing fast, high-precision position information for each robot. Precise localization data may often aid experimenters in development of applications or protocols that require an input node location, or in evaluation of location services themselves.

Second, to provide usable and accessible motion to users, the testbed must provide algorithms in place to aid in control of the mobile nodes. These algorithms must, at minimum, perform obstacle avoidance and rudimentary following of user-specified paths so that experimenters can work without monitoring all details of motion. In mobility simulation, the most often used models are derivatives of simple waypoint models. However, support for full path-based mobility models is also valuable since providing conformity to specific paths may increase experiment repeatability.

Finally, there are several aspects of hardware and environmental control that must be considered when building a system for mobile wireless experimentation. In Mobile Emulab, mobile nodes are surrounded with fixed wireless nodes, since hybrid systems are often of interest in wireless research. Adding fixed nodes also increases testbed scalability and requires fewer mobile nodes. Since this testbed was deployed in a small, public space, it was necessary to choose fixed node placement and mobile node physical radio and antenna setup with care. To create interesting multihop topologies in such a small environment, experimenters will need to use low-power transmissions. At low power, antenna position has greater effect on signal reception.

To ease the transition to real-world mobile experimentation, we designed and built our system as an extension to the Emulab network testbed. By extending Emulab, we provide researchers with a well-known network experimentation interface. This effort

involved building a backend that translates motion requests from users and acts as an information broker.

We discuss the system architecture and several of the important design aspects of these subproblems below.

3.1 Software Design

Emulab provides researchers quick access to automatically-instantiated custom network topologies. Researchers access Emulab primarily through a web interface, through which they can configure and run experiments. The core of Emulab is a large database that maintains state for current experiments, both those currently running on physical resources, and others that have been created in the past but are not running. A large number of configuration programs, scripts, and management daemons perform experiment swapin and swapout and aid the researcher in experiment control, primarily using information stored in this database.

To leverage Emulab's extensive testbed management services, we link Mobile Emulab subcomponents to the database and the web interface. This largely happens in the *embroker* daemon, which functions as an information broker, through which user motion requests and status data flow. Other subcomponents of the mobile testbed are *robotd*, which executes requested robot motion commands through instances of the *pilot* program running on each robot, and *visiond*, which tracks all robots using information gathered by per-camera instances of the *vmc-client* program through computer vision localization techniques. Several new Java applets and other enhancements to the web interface expose new mobile and sensor network functionality to Emulab experimenters. The software architecture of our system appears in Figure 3.1.

3.1.1 Component Initialization and Dataflow

When an experiment swaps in a mobile Emulab experiment, the swapin process spawns off instances of *embroker*, *robotd*, and *visiond*. *embroker* reads in a simple configuration file, generated from the Emulab database, which specifies robots the experimenter has requested, static obstacles in the area of motion, and the bounds in which the robot can move, and listens on a Unix socket for connections. *visiond* and *robotd* both immediately connect to this socket and receive configuration information. Once these three daemons have successfully initialized, *visiond* identifies the robots and begins tracking and sending

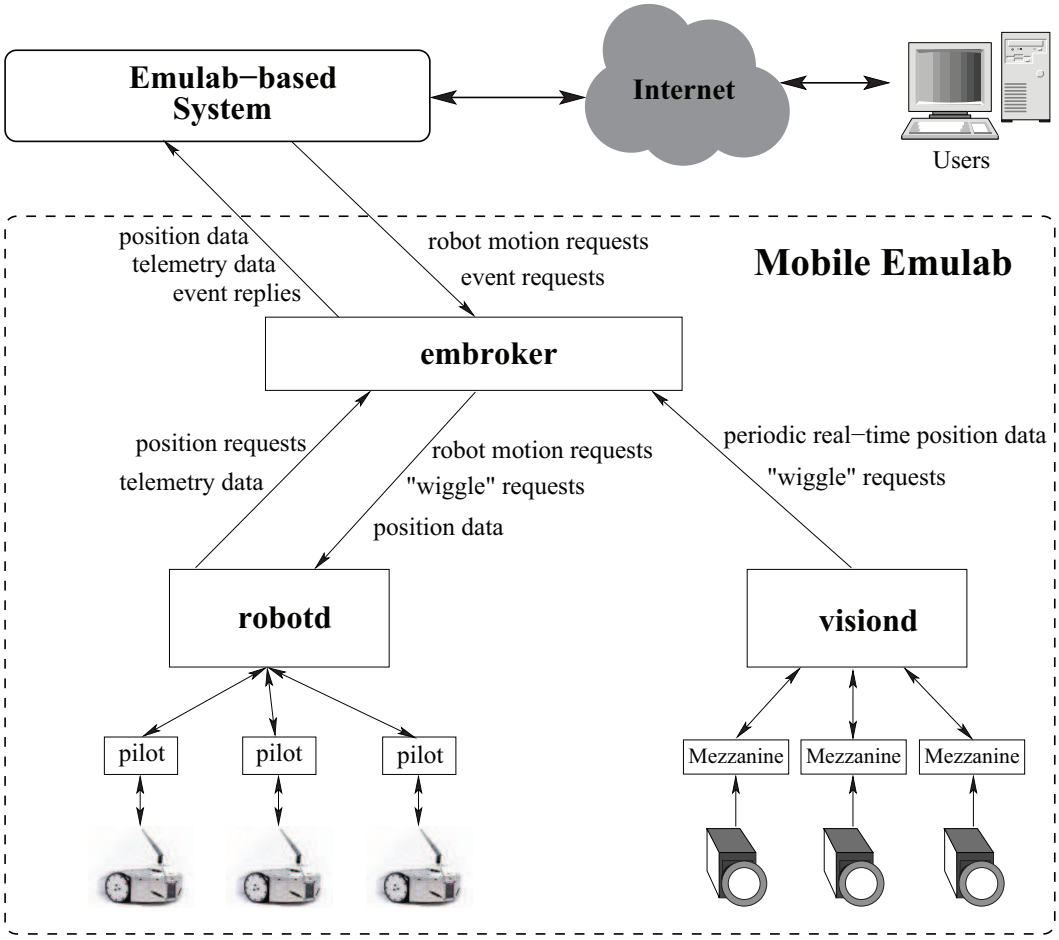


Figure 3.1. Mobile Emulab software architecture.

location information to *embroker*. Once the system has reached this point, it is available to accept user motion requests.

As mentioned above, *embroker* is responsible for passing data between Emulab (and by extension, the users) and the other components that comprise the mobile testbed extensions. *embroker* is connected to Emulab through the event system. The Emulab event system is a publish-subscribe service that supports event generation and reception by testbedobjects (i.e., nodes, shaped links). *embroker* communicates with Emulab primarily through this system, and transforms events into commands for *robotd* and *visiond*. Communication between *embroker*, *robotd*, and *visiond* (and also *pilot* and *vmc-client*) takes place over *mtp*, the Mobile Testbed Protocol.

When an experimenter sends a motion request to Emulab, either through the web interface or from a script, it is passed to *embroker*, which performs bounds-checking on

the destination to ensure the robots do not leave the area in which they can be localized. The request is then passed to *robotd*, which requests the latest location data for the robot to be moved, and then plans a path to the specified destination. *robotd* breaks this path up in the manner required by the motion model being employed, and passes incremental, relative motion requests to *pilot*. Throughout this process, *visiond* tracks each robot and reports positions when requested or streams position data, depending on the motion model in use.

3.1.2 Communication: *mtp*

Communication between the components comprising the mobile extension to Emulab takes place over *mtp*. *mtp* is a message-based protocol that defines a set of messages that testbed components require. For instance, all components must understand robot location updates. Each mobile subcomponent responds appropriately to commands that it supports. Originally, *mtp* used a custom packed data format that was interpreted in a platform-independent manner; it was later wrapped with XDR routines by a colleague.

3.1.3 Robot Control: *robotd*

The two primary components that implement motion control and guidance are *robotd*, which runs on a central control computer, and *pilot*, an instance of which runs on each robot. *robotd* handles motion requests at a high level, while *pilot* implements low-level motion directly on the robot. If the waypoint motion model is used, *robotd* plans a path to the destination consisting of linear segments that efficiently avoids known, static obstacles. *robotd* sends *pilot* relative motion commands, which *pilot* executes through a robot platform-specific API that translates requests to commands understood by the microcontrollers governing the drive wheels. Due to limitations in this API, *pilot* calls back to *robotd* after the robot has traveled a single segment to ensure its heading is correct. Heading may become incorrect due to slippage between the drive wheels and the floor, or unreliable wheel odometry-based methods for calculating distance covered. When *pilot* estimates that it has reached the destination position, it refines the position to correct for potential drift error as instructed by *robotd*, which requests location data from *visiond* to obtain the most recent "ground truth" robot position.

By using the continuous path motion model, the cumbersome nature of the segment-based approach is eliminated. In this model, *robotd* will generate a velocity profile for the path computed from waypoints, or specified directly by the experimenter. This "profile"

consists of wheel speeds that ensure the robot stays on the path. By sending low-level wheel speed commands to *pilot* instead of high-level straight-line moves and pivots, execution times for lengthy moves are reduced. *embroker* streams robot location data directly to *robotd* so that any deviations from the path will be discovered and corrected immediately by modifying wheel speed commands based on the original velocity profile. Flickinger, the implementer of this subsystem, provides significantly more detail about continuous motion in Mobile Emulab in [8] and [22].

3.1.4 Robot Localization: *visiond*

Mobile Emulab identifies and tracks the robots through a computer vision-based tracking system called *visiond*, using ceiling-mounted video cameras aimed directly down at the plane of robot motion. As described in greater detail in Chapter 4, we improved Mezzanine [21], an open source object tracking software package, to transform the overhead camera video into x, y coordinates and orientation for detected objects. The system consists of six cameras covering an approximately 60 m^2 area. An instance of Mezzanine processes video from each camera, translating pixel locations of objects to x, y coordinates in meters.

An instance of *vmc-client* connects to the shared IPC segment used by a Mezzanine instance and extracts object position and orientation for each object, and converts the local coordinates to globally-understood coordinates. Each *vmc-client* forwards individual robot location data to *visiond* for global processing.

Individual robot locations from *vmc-client* instances are aggregated by *visiond* into a single, canonical set of tracks that can be used by the other components. These tracks are reported at 30 frames per second to *embroker* since queries from *robotd* require low-latency replies and high precision. *embroker* in turn reports snapshots of the data (one frame per second) to the Emulab database, for use by the user interfaces. This reduction in data rate is an engineering tradeoff intended to reduce the communication bandwidth with, and resulting database load on, the Emulab core.

3.1.5 Motion Models

Mobile Emulab's initial design supported only a simple waypoint motion model. In this model, experimenters can specify lists of destination points for each robot, but the path to these positions is nondeterministic from the experimenter's point of view; even the straight-line path between waypoints is not guaranteed, since the area of motion is open

to people, work carts, etc. These dynamic obstacles may force a path that is different than expected. Finally, the waypoint model is implemented using a primitive API that supports straight-line motion and pivots. Consequently, the robot must stop and reorient itself to move to the next waypoint, which leads to slower motion execution times.

To improve the execution time of a user-supplied path, and to provide much more expressive motion, we designed a continuous motion model that generates a velocity profile containing wheel speeds that are then sent to and set on each moving robot. This model can take as input a B-spline with enforced motion constraints, or more simply a set of waypoints. Flickinger describes the theory and implementation behind this model in [8]. This model allows the robots to continuously move along a specified path to their destination, significantly enhancing motion request execution times and providing advanced experimenters with much greater potential for motion control.

3.1.6 User Interfaces

Mobile Emulab inherits many useful features of Emulab's web interface. However, many more are necessary to allow experimenters to make full use of both mobile and wireless sensor network aspects of the testbed. For instance, a Java-based motion control applet allows experimenters to interactively position robots and monitor motion in real-time. Another Java applet displays connectivity information for all static motes at different power levels, aiding experimenters in choosing which motes to include in experiments to create topologies with different connectivity properties, without inspecting the space themselves. Finally, we developed an application called `SNAP-M` that is specifically designed to support mobile, wireless sensor device experimentation. Although this application is useful in a standalone manner apart from Emulab, some of its subcomponent features rely heavily on Emulab. These interfaces and others are described in Chapter 5.

3.2 Environment

Currently, the mobile testbed is deployed in an L-shaped area of approximately 60 m^2 and 2.5-2.7m high. This area is "live"; people and work carts may move through the area at any time. This adds a definite aspect of realism to wireless experiments. However, it potentially reduces repeatability. Uncontrolled movement by objects in the area of robot motion can force robots to halt, both by physically impeding their progress, or by preventing *visiond* from maintaining a track by obscuring line of sight between a robot and one or more videocameras.

This deployment environment also contains potentially damaging wireless interference across the 900 MHz spectrum in which the mote radios communicate. While the physical properties of the environment may help experimenters enhance quality of application evaluation, external wireless sources may interfere enough with testbed node communication to ruin analysis. Section 6.1 provides a characterization of our environment.

3.3 Hardware

To increase the types of experimentation we can support, we deployed 25 Crossbow Mica2 motes [3] in fixed locations surrounding the area of motion. All motes are attached to serial programming boards, and each serial line is attached to a console machine. As is the case for most other Emulab node types, each mote's serial port can be exported over a network, or accessed on an Emulab PC functioning as a proxy for direct, programmatic access to several mote serial devices. Testbed software fosters easy mote reprogramming.

The fixed motes are deployed in a partial (subject to the "L"-shaped constraint) grid, spaced approximately 2 m apart. Several motes with attached Crossbow MTS310 sensor boards [4] are distributed evenly around the area of motion and are placed at the height of the robots. Since the robot-mounted motes also have attached sensor boards, the sensors are all in the same horizontal plane. We hope that some applications may be tested by using the robots as emulated, sensor-detectable devices. Other motes, without sensors, cover the ceiling. The overall deployment is such that it is possible to create a variety of different multihop networks. However, since the area is small, experimenters will need to reduce radio power to enable multiple hops. We have found this to work reasonably well in practice, although the maximum number of hops is not large.

Mobile Emulab uses Acroname Garcia robots [1] (pictured in Figure 3.2) as courier devices. Each Garcia has an Intel Stargate with a 400 Mhz XScale processor [5], and attached to one of the Stargate's serial ports is a Mica2 mote. Thus, experimenters can remotely login to the robot and run programs that connect directly to the mote's serial port, or they can access the serial devices at an Emulab PC or over the network. Sensor boards are attached to all robot-mounted motes. The robots are controlled remotely via an 802.11 card plugged into the Stargate.

We have also added antenna extenders that place the mote's antenna approximately 1 m above the ground. We hope that these devices will allow experimenters to emulate human-carried wireless devices. However, in our environment, this lowers the number of

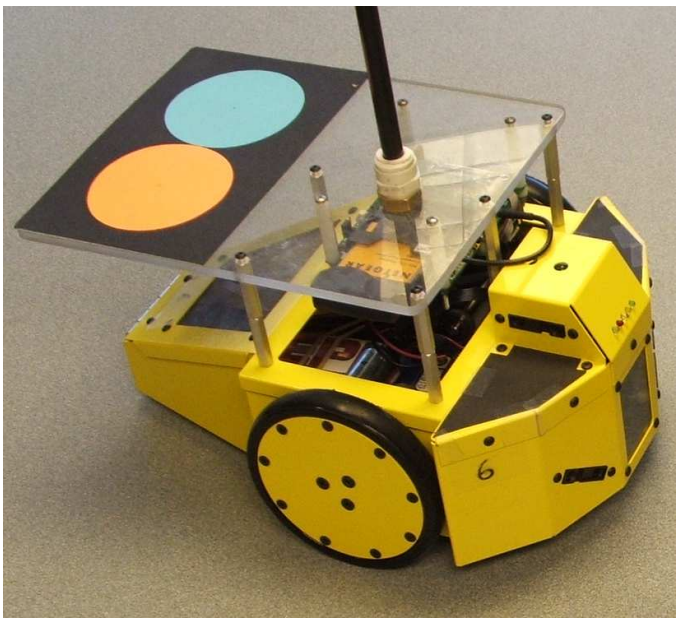


Figure 3.2. Garcia robot with two-circle, two-color fiducial and antenna extender.

static motes that the mobile can communicate. When the antenna is lower to the ground (i.e., closer to the mote itself), a ground-capture effect increases reception capability from ceiling-mounted motes. When the antenna is raised, this effect is lessened and the robot has more difficulty communicating with the ceiling motes. At the same time, many mobile sensor devices have much more power than static motes, so increasing power for mobile radios may be permissible for many applications.

CHAPTER 4

LOCALIZATION

To enable accurate and repeatable experiments, Mobile Emulab must guarantee that all mobile devices and associated antennae are at the specified positions and orientations within a small tolerance. To do this, we need to accurately determine the location (position and orientation), or *localize*, each robot. The system is designed to achieve subcentimeter localization accuracy to ensure that wireless experimenters can clearly understand environmental effects on radio signals.

Although clearly important for positioning guarantees, precise localization is also important for efficient robot motion. An incorrect location estimate at the beginning of a motion can subtly alter the robot's initial trajectory and thereby increase time required to correct during execution, and when homing in on the final destination. When dealing with advanced kinematic controllers that require stable and on-time data, this precision becomes even more important. Any amount of localization data jitter will adversely affect controller performance.

Another important design goal is that the localization system should be flexible over a variety of quality and cost specifications. Because we wanted to create a mobile testbed that other universities would duplicate to create their own mobile wireless testbeds, the localization software should deliver acceptable levels of quality depending largely on available hardware. In this implementation, we attempted to minimize cost while still meeting the goal of precise localization. This was necessary to ensure scalability for future expansions of the testbed motion area. A scalable system is also necessary so robots may be localized across a sufficiently large area to enable interesting multihop wireless experiments. While this implementation of the localization system lowers costs, other adopters of this testbed could use either higher or lower price hardware to obtain desired location precision.

Finally, the localization system must not interfere with user experiments. Although the initial system was primarily designed with specific robots and sensor network gear

attached, we eventually added external antenna risers and contemplated additional sensor attachments. The system we have implemented enables expansion and adjustment to the mobile platform and devices that it carries.

As is typically the case in robotic systems, the robots' on-board odometry could not localize the robots with sufficient accuracy for our purposes. Consequently, we developed a computer vision-based localization system to track devices throughout our experimental area. Vision algorithms process image data from video cameras mounted above the plane of robot motion. These algorithms recognize markers with specific patterns of colors and shapes, called *fiducials*, on each robot, and then extract position and orientation data.

This chapter discusses many of the decisions made while designing the mobile testbed's localization system, describes aspects of its implementation, and presents an analysis of its effectiveness.

4.1 Possible Localization Methods

Because we must have precise, low-cost localization for motion control and experimenter data analysis, we considered several different methods of localization. First, precision requirements almost immediately rule out any form of GPS, since ordinary GPS provides accuracy only to within 1-3 m [15]. Differential GPS [7] raises this accuracy to the centimeter level, but may be difficult to use indoors due to multipath effects. Finally, GPS does not provide target orientation. One could use GPS to find this by placing multiple devices onboard a single tracked object, but the tracked object would need to be fairly large to prevent orientation estimates from being lost in the error of the GPS system itself.

Many schemes have been devised for localization in wireless sensor networks using motes and sensorboards, such as Cricket [25] and the Active BAT system [17]. Primarily, these methods use acoustic or ultrasonic ranging in TDOA (Time Distance Of Arrival) techniques to find intranode ranges. Nodes send wireless signals at the same time as they generated sonic or ultrasonic waves, and receiver nodes can determine range based on how far apart these signals were received. However, since we must keep each mote's sensor and radio free for experimentation, we cannot use these techniques without causing potential experiment disruption. The system could avoid radio usage conflicts by adding a second mote and sensorboard to each robot, but would still need to require that the experimenter avoid the resulting radio and sensor interference.

There are many different robot self-localization schemes. Meltzer et al. present a SLAM (Simultaneous Location and Mapping) algorithm [24] in which a video camera mounted on a mobile robot records environment features, and the robot can localize itself when it sees these features again. Other methods such as [23] use an omnidirectional video camera to recognize specific landmarks, and estimate their positions given current line of sight to these known landmarks. Computer vision-based robot self-localization, while desirable for some applications, is inappropriate for our system because it often requires large amounts of onboard image processing to localize the robot. We would be forced to use low-resolution camera models to reduce CPU processing. Such cameras cannot be used onboard because we cannot extract high-precision data from them. Additional hardware and processing on the robots also increases the required power and hence decreases the amount of time an experiment can run uninterrupted.

After studying these and other systems, we chose to use a computer vision-based localization technique that does not interfere with robot and mote experiments. The resulting system utilizes open-source computer vision software, makes use of a number of simplifying design assumptions that improve quality, and permits the use of higher or lower quality hardware to adapt to other testbed implementers' needs.

4.2 Design Issues

To obtain high-precision data while limiting hardware costs, we made a number of design choices that tend to simplify the system and reduce cost while still attaining desired scalability and localization precision. These design points are discussed further below.

4.2.1 Recognition Software

One of the design goals for this localization system was low development time. Consequently, we investigated a variety of software packages ranging from computer vision libraries to object tracking products. Few libraries that we found provided enough high-level functionality to quickly build a highly-accurate localization system. Furthermore, many commercially-available object recognition and video processing tools and libraries cost several thousands of dollars to license. Due to these prohibitive costs and time constraints, we attempted to find a relatively complete open-source tool that would provide most of the functionality we needed. We eventually used an open-source tool, called Mezzanine, that provides basic object recognition from live video streams and complies with our other design choices.

4.2.2 Fiducial Patterns

Since Mobile Emulab's localization system does not use self-localization or wireless identification techniques, it must identify each robot uniquely by using a family of patterns or by other means. Complex fiducial patterns may be more difficult to detect with low-quality videocameras, which may have lower resolution and produce noisier image data. Furthermore, more complex software is needed to detect anything more basic than a simple system of lines (i.e., a bar code). Also, there is no guarantee that there is enough physical space atop the robot hardware to mount a pattern family that can support enough patterns to uniquely identify all robots in the testbed. Since we originally planned to support 50 to 100 robots in a large empty room, we were forced to choose a much more simple pattern. Finally, pattern selection also impacts future flexibility for hardware and sensor modifications to the robots. By using a simple pattern, many future modifications remain possible.

Mezzanine's default pattern is compatible with these constraints and we use the simple two-circle fiducial it supports. The circles are colored by two different colors widely separated in color space. By using a two-color circle pair, Mezzanine can determine both position and orientation of the object bearing the fiducial. By placing the fiducial on a raised platform, behind the robot's pivot axis, we can add numerous sensors near the front of the robot in the future. This placement, combined with the simple nature of the fiducial, simplified the addition of antenna extenders to the robot. The antenna extenders only slightly worsen the precision of the localization data because they obscure primarily a small portion of one of the circles at any one time. Refer to Figure 3.2 for an illustration of a testbed robot with antenna extender and fiducial.

Mezzanine does not support multiple fiducial color pairs by default, and there is little reason to extend it to create this capability. Due to light variance, the number of unique fiducial pairs is not large enough to support a large number of robots as might be desired in the future. Consequently, each robot bears the same fiducial and is uniquely identified through motion algorithms (see Section 4.3.3).

4.2.3 Vision Hardware

We use video cameras and lenses that combine to produce high-precision localization, yet are not prohibitively expensive. However, digital cameras with resolutions higher than 640×480 pixels all exceeded our cost constraints. We evaluated standard analog security cameras, and found that the analog resolution produced is too low to extract sharp fiducial

outlines. Standard security cameras also lack manual controls for light and color settings, which are needed in our environment to combat the effects of lighting variability. After extensive evaluation, we chose the Hitachi KP-D20A analog CCD camera [20], which provides sufficient analog resolution and good, manual control of light and color settings. The camera cost was $460.

To cover the testbed with as few cameras as possible, we used wide-angle lenses. Such lenses produce barrel distortion, which can be partially accounted for in software, but which decreases our system's precision. Since low-distortion wide-angle lenses can cost many thousands of dollars, we used inexpensive lenses and corrected for distortion in software, using better camera geometry models and interpolative error correction. We are using Computar 2.8–6.0 mm varifocal lenses set at focal lengths of 2.8 mm, each costing approximately $60.

4.2.4 Deployment

Mobile Emulab's localization software utilizes video cameras mounted above the plane of robot movement, looking down, instead of installing one on each robot. This solution scales better than robot self-localization for dense deployments when there will be at least a one-to-one robot-to-camera ratio, since each overhead camera can track many robots. This method also removes processing requirements from the robots.

Futhermore, since the video cameras are pointed straight down, perpendicular to the plane of robot movement, the geometry of the system is greatly simplified. We describe the resulting improvements to precision in Section 4.4.

4.3 Localization Software

4.3.1 Mezzanine

We use Mezzanine [21], an open-source computer vision software that recognizes colored fiducials on objects and extracts position and orientation data for each recognized fiducial. Each fiducial consists of two 2.7 in circles that are widely separated in color space, placed next to each other on top of a robot. Mezzanine's key functionality includes a video image processing phase, a "dewarping" phase, which attempts to eliminate barrel distortion in wide-angle images, and an object identification phase.

During the image processing phase, Mezzanine reads an image from the frame grabber and classifies each matching pixel into user-specified color classes. Each color class must be specified by the user so that all of the pixels in one of the circles in a fiducial

can be classified into that class. This can be problematic because observed color can be distorted by environmental lighting conditions. To operate in an environment with nonuniform and/or variable lighting conditions, the user must specify a wider range of colors to match a single circle on a fiducial. This obviously limits the total number of colors that can be recognized, and consequently, we cannot uniquely identify robots through different fiducials. We obtain unique identification by commanding and detecting movement patterns for each robot (the "wiggle" algorithm), and thereafter maintain an association between a robot's identification and its current location as observed by the camera network. Mezzanine then combines adjacent pixels, all of which are in the same color class, into color blobs. Finally, each blob's centroid is computed in image coordinates for later processing (i.e., object identification).

4.3.2 *vmc-client*

As specified in Chapter 3, an instance of *vmc-client* runs for each videocamera. *vmc-client* connects to the shared memory segment in which Mezzanine writes extracted location data, and registers itself to be notified whenever Mezzanine has processed a new video frame. When notified, *vmc-client* reads the shared memory and passes the location data for any detected objects to any connected *visiond* daemons.

Each *vmc-client* obtains location data for objects in Mezzanine's local x, y coordinates, in which $0, 0$ is approximately at the center of the camera's focal axis. *vmc-client* converts these locations to global coordinates that cover all instances of *vmc-client*s in the testbed. Furthermore, since the robot fiducials are raised off the ground by approximately 20 cm and are offset from the robot's pivot axis, *vmc-client* modifies the global location data to account for these factors. *vmc-client* accepts parameters for these values, but since robots are not uniquely identified, using different types of robots would necessitate multiple *vmc-client* and Mezzanine instances.

4.3.3 *visiond*

An instance of *visiond* is spawned for each mobile experiment. *visiond* connects to all *vmc-client*s provided by *embroker*'s configuration messages, and immediately begins the identification process for all robots reserved in this experiment (see "Unique Identification" below). As soon as a robot is matched to an object location forwarded by one or more *vmc-client*s, *visiond* constructs a track for it and updates the track on subsequent location data from a *vmc-client*.

4.3.3.1 Unique Identification

Because we use the same fiducial on each robot, *visiond* must perform robot identification; that is, it must map objects identified by Mezzanine to robots controlled by the testbed. When *visiond* is initialized by *embroker*, or loses track of a robot, it must re-identify any unidentified robots. Since the testbed design must account for the possibility that robots will be moving in an area in which they may be briefly obscured from the videocameras, *visiond* must also re-identify a robot if such obscuration continues for too long a time.

When a per-experiment *visiond* is launched by *embroker*, *embroker* configures it with the node identifiers that the user has reserved in the experiment. *visiond* immediately begins a serialized identification process. For each robot, *visiond* sends a "wiggle request" message to *embroker*, where it is forwarded to *robotd*, and then to the appropriate *pilot* for motion execution. A "wiggle request" typically pivots the robot by 180°. This motion is easily detectable, and very unlikely to create tracking confusion, since at no time can a user request a motion resulting in a stationary pivot of more than 180°. This occurs because *robotd* always minimizes the arc through which it must turn to execute a pivot. *visiond* saves the current state of all tracked objects and waits for *robotd* to signal that the wiggle has finished. When signaled, *visiond* compares the current set of tracks with the saved set. Whichever track has remained nearly stationary, and has an orientation of 180° difference, is matched to the robot that was commanded to wiggle.

There are a number of cases in which, strictly speaking, Emulab does not need to re-identify a robot. For instance, at the end of each experiment, each robot returns to a parked location that is stored in the database. Consequently, Emulab could assume that each robot is at its parked location at the beginning of each experiment. However, if a robot is obscured at experiment swapin, is at a location slightly different than that stored in the database, or has been mistakenly switched to another robot's parked location by a testbed operator, the wiggle algorithm will prove invaluable to avoid mistaking robots. Such mistakes could potentially result in user motion request execution errors. Finally, if a robot's fiducial is ever obscured during experiment runtime, *visiond* will need to reacquire the robot again through the wiggle process. Therefore, it is in the best interests of the system as a whole to always wiggle to discover true robot-to-object mappings.

4.3.3.2 Track Maintenance

Once *visiond* constructs tracks for all recognized robots, object location updates from all *vmc-client* instances are matched against the current set of known tracks. New object locations match a track if the new location is within a small distance of the latest location in the track, and if the heading is likewise similar. If a track remains unmatched for three subsequent updates, it is cleared and *visiond* no longer associates the robot with the track. Generally, this occurs due to extended obscuration of line of sight from one or more cameras to a tracked robot. Once *visiond* loses track of a robot, it immediately attempts to re-identify the lost robot. If the first re-identification fails, subsequent attempts are spaced at approximately 20 seconds.

Since we require a multicamera localization system, *visiond* also performs track aggregration across multiple cameras. Since the fiducial atop the robots is larger and more complex than a simple, single LED dot, wherever a robot can cross a videocamera boundary, there must be an overlap zone in the cameras' coverage. This is unfortunate and leads to a reduction of system scalability, but it is necessary to maintain stability of data and reduce jitter. For instance, when a robot moves into another camera, *visiond* only begins using the position reported by that camera once the robot has left the original camera. This reduces jitter because even the adjacent cameras and their *vmc-client*s may have slightly different parameterizations and offsets for calculating global coordinates of objects. Were a robot to move back and forth on the camera boundary, the jitter could increase significantly if *visiond* constantly selected the other camera's reported location.

4.3.3.3 Scalability

The *visiond* process for each experiment connects directly to each video camera's *vmc-client* process. To properly eliminate duplicate objects seen by multiple videocameras, *visiond* waits until it has a full frame's worth of object locations from each *vmc-client*. Consequently, *visiond* will very likely not scale well beyond an estimated 30 to 50 cameras. System phase lag will begin to increase, harming advanced robotic motion controllers. Although bandwidth used should not become a problem on high-speed networks, it scales poorly. Due to the small size of the initial implementation of the testbed and time constraints, a permanent localization system scalability increase was beyond the scope of this thesis. However, we have considered how to modify *visiond* and *vmc-client* to increase system scalability, and provide suggestions below.

As stated, the scalability problems arise due to the need to remove duplicate object

locations from adjacent cameras. One simple way to solve this problem is to use a processing hierarchy in which an aggregator instance is allocated to each $m \times n$ grid of *vmc-client* processes. The aggregator would remove duplicates discovered in this grid, and pass on the resulting set of object positions to another aggregator instance. By constructing an aggregation hierarchy in this manner, we can remove duplicates, but also reduce the amount of network bandwidth required since each *visiond* instance will now only need to connect to the aggregator root. Furthermore, by running the aggregator instances on as few machines as possible, we can limit the extra latency caused by network communication with aggregators higher up in the hierarchy. Unfortunately, every additional processing and communication step adds lag to the system. It is possible that when scaling to hundreds or thousands of cameras, phase lag would begin causing more severe problems to robotic controllers.

4.3.3.4 Jitter Reduction

Due to light variance (fluorescent lights, combined with outdoor light), ceiling vibration, etc., there is an amount of jitter in the data reported by *visiond*. We discussed designing a Kalman filter for our system, but a Kalman filter would require significant tuning (perhaps in each environment in which the testbed would be used) and implementation time. Consequently, we implement two different types of smoothing functions.

We first implemented a simple moving window smoothing function. This alleviated difficulties in the initial system, where *embroker* would be deceived by a large enough difference in the reported orientation ($\pm$ several degrees), decide that the robot had moved from its currently assigned position, and attempt to generate motion commands to drive it back. In certain, isolated areas of particularly variable lighting conditions, this resulted in system-generated motion loops, thus acting as a denial of service to the user. When using the moving window estimator with a window size of five, we found that these loops did not appear. Furthermore, since the initial motion control implementation in *robotd* did not require constant vision data at 30 Hz and stopped every 1.5 m, the resulting phase lag introduced into the data did not harm motion.

Unfortunately, the moving window average proved incompatible with the later re-implementation of *robotd* with continuous, path-based motion. *robotd* required data at as fast a rate as possible (only 30 Hz with the testbed video cameras, without any interpolation or predictive filtering). The phase lag introduced by the moving window

average, coupled with relatively (relative to the newer *robotd* controller's needs) noisy data, made it much more difficult for the controller to operate successfully. Thus, we implemented an EWMA filter that we hoped would reduce jitter at least as well as the SMA filter, but with a smaller impact on localization data phase lag. All aspects of smoothing are configurable by the user; however, if the user does not provide a specific α parameter for the EWMA filter, we calculate it using $\alpha = 2/(N+1)$, where N is the window size.

4.4 Dewarping Improvements

The original Mezzanine detected blobs quickly and effectively, but the supplied dewarping transform did not provide nearly enough precision to position robots as exactly as we required. The dewarping transform is computed by a calibration phase in which grid points in the plane of motion are provided to the application. The supplied dewarping algorithm is a global function *approximation*. An approximating function does not need to match the provided data points exactly, whereas an interpolating function must. Although Mezzanine's approximation method worked well for us with slightly wide-angle angles, it began to exhibit strange data discontinuities in reported position estimates as angle of view increased. For instance, moving a fiducial 1-2 cm resulted in position estimate jumps of 10-20 cm.

We enhanced Mezzanine with a different dewarping transformation that takes advantage of the fact that our overhead cameras point directly downwards. My colleague, Russ Fish, noticed that the barrel distortion pattern could be very closely modeled by a cosine and developed a mathematical basis for our model. Thus, we can transform image position estimates to real-world coordinates by dividing the image coordinate vector by the cosine of the angle between the vertical camera optical axis and a line from the optical center of the camera to the fiducial. An additional multiplier inside the cosine, the "warp factor," corresponds to the amount of distortion in the image.

Results indicate that these improvements have removed the strange discontinuities and jumps observed under Mezzanine's original approximation transform. Furthermore, the vision system with these changes reduces error to 1-2 cm. With additional interpolative error correction modifications from Russ Fish, error is reduced to subcentimeter levels.

4.5 Validation

In this section, we validate our localization system by comparing location estimates generated by it for over two hundred grid points, spaced at half-meter intervals around the area of motion. We also examine location estimate *jitter*, or how much estimates vary at the 30 Hz camera rate.

4.5.1 Location Estimate Precision

To obtain as much precision as possible, before modifying Mezzanine's dewarping algorithm, we measured a half-meter grid over the mobile area. Consequently, we could calibrate the algorithm and measure its effectiveness with high precision. Using simple measuring tools and surveying techniques, we set up a grid accurate to 2 mm.

Table 4.1 shows the results of applying these algorithms to a fiducial located by a pin at each of the 211 measured grid points and comparing the result to the surveyed world coordinates of these points. (Points in the overlap between cameras are gathered twice.) The *original* column contains statistics from the original approximate dewarping function, gathered from only one camera. Data for the *cosine dewarping*, and *cosine dewarping + error interpolation* columns were gathered from all six cameras.

Figures 4.1 and 4.2 graphically compare location errors at grid points before and after applying the error interpolation algorithm. Figure 4.1 shows measurements of the cosine dewarped grid points and remaining error vectors across all cameras. The circles are the grid points, and the error vectors *magnified by a factor of 50* are shown as "tails." Since the half-meter grid points are 50 cm apart, a tail one grid-point distance long represents a 1 cm error vector. Points with two tails are in the overlap zones covered by two cameras. Figure 4.2 shows the location errors after applying the error correction and interpolation algorithm.

Table 4.1. Location error measurements.

	Algorithm		
Metric	original	cosine dewarp	+ error interp
Max error	11.36 cm	2.40 cm	1.02 cm
RMS error	4.65 cm	1.03 cm	0.34 cm
Mean error	5.17 cm	0.93 cm	0.28 cm
Std dev	2.27 cm	0.44 cm	0.32 cm

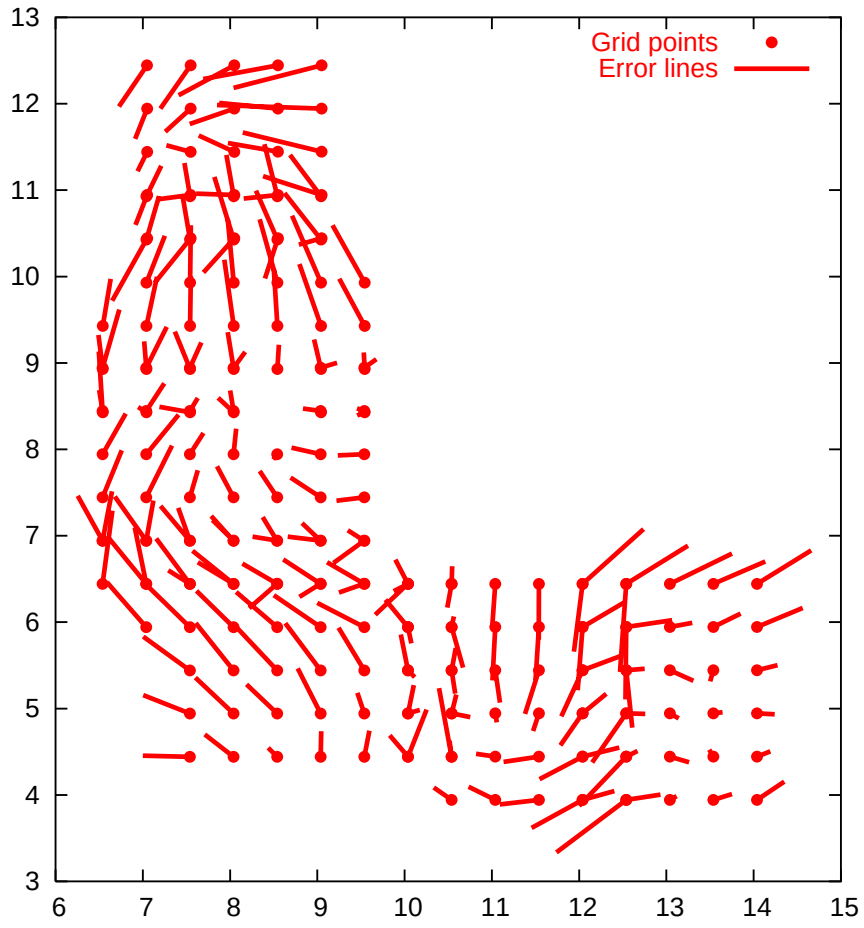


Figure 4.1. Location errors with cosine dewarping.

4.5.2 Jitter Analysis

Although the basic linear waypoint motion controller is unaffected by location estimate jitter, the advanced kinematic controller discussed briefly in Section 3.1.3 is extremely sensitive to noise in location estimates. Even slight amounts of jitter as shown in our data could result in controller instability, leading to stale wheel speed commands sent to the robot.

We collected location estimates generated by *visiond* for a single robot by capturing, timestamping, and logging the messages at *embroker*. In this section, we analyze deltas between subsequent location estimates for x, y, and θ components of location. In addition, we analyze message interarrival times (the time in between reception of two subsequent messages). The advanced controller requires both smooth, timely data.

Although we are primarily interested in jitter behavior while a tracked robot is moving,

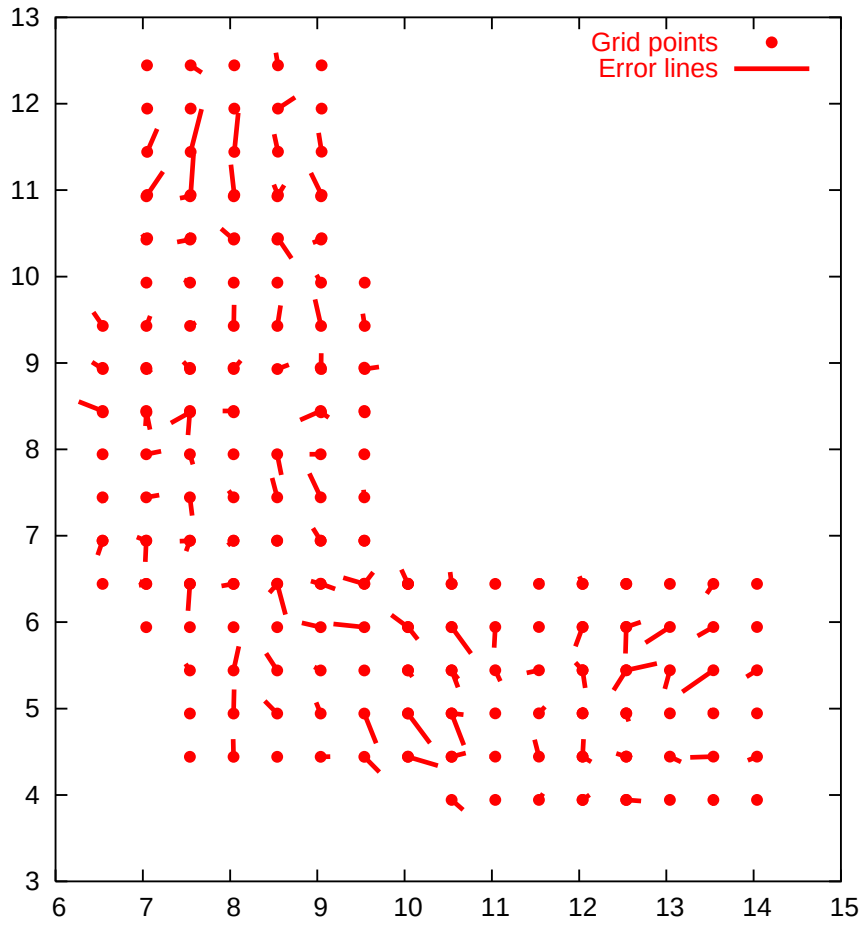


Figure 4.2. Location errors with cosine dewarping and error interpolation.

we first present jitter results from monitoring a single, unmoving robot. This provides a baseline estimate of system noise (at least at the time we monitored—recall that the localization system is sensitive to different and varying lighting conditions, and these are nearly uncontrollable in our deployment), which is helpful in analyzing filter performance.

Figures 4.3, 4.4, and 4.5 show jitter data (the figures show the x, y, and θ components, respectively) collected from logged location estimates for a nonmoving robot in one location (location "P3" in subsequent data tables) in the motion area.

Tables 4.2, 4.3, and 4.4 provide statistical data for location estimates for a nonmoving robot in three different locations. For each location, the data are parameterized by filter window size and type ("—" if no filter was applied). Filters were only applied to real data streams; we did not filter the logged unfiltered data stream after collecting it to specifically analyze filter performance alone. It is important to measure end-to-end system jitter to

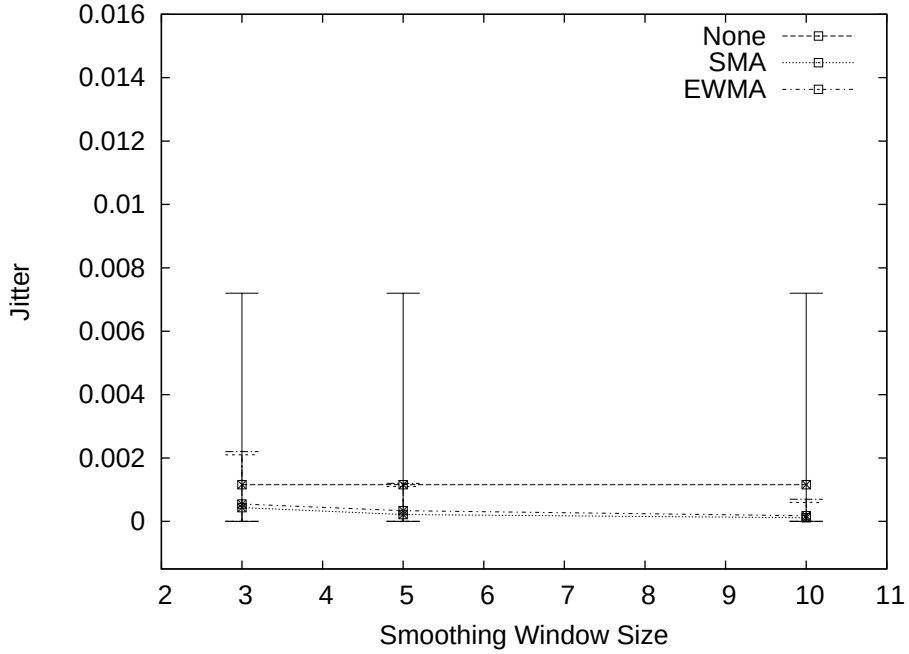


Figure 4.3. Average jitter (error bars show minimum and maximum) in x component.

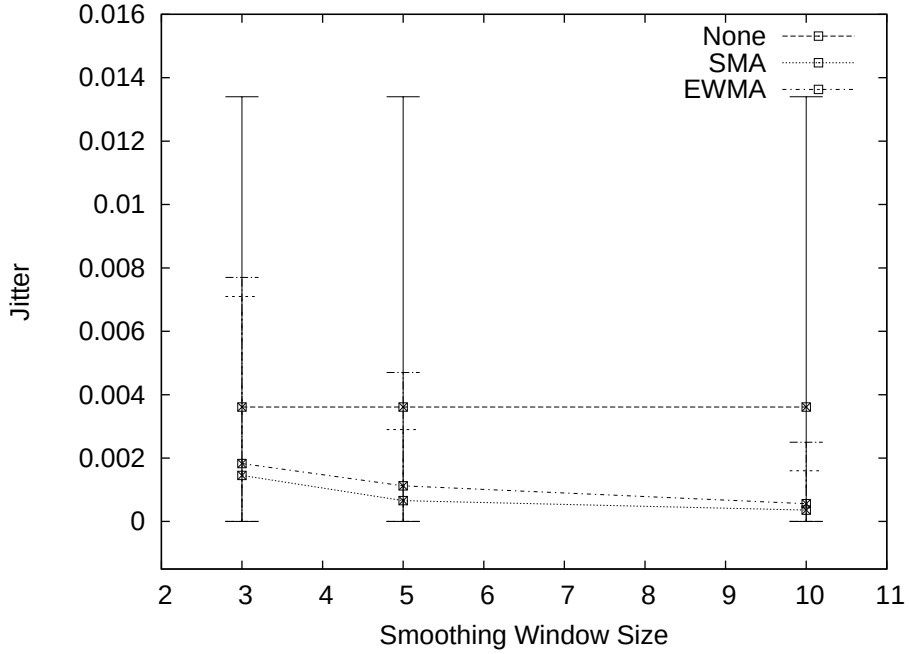


Figure 4.4. Average jitter (error bars show minimum and maximum) in y component.

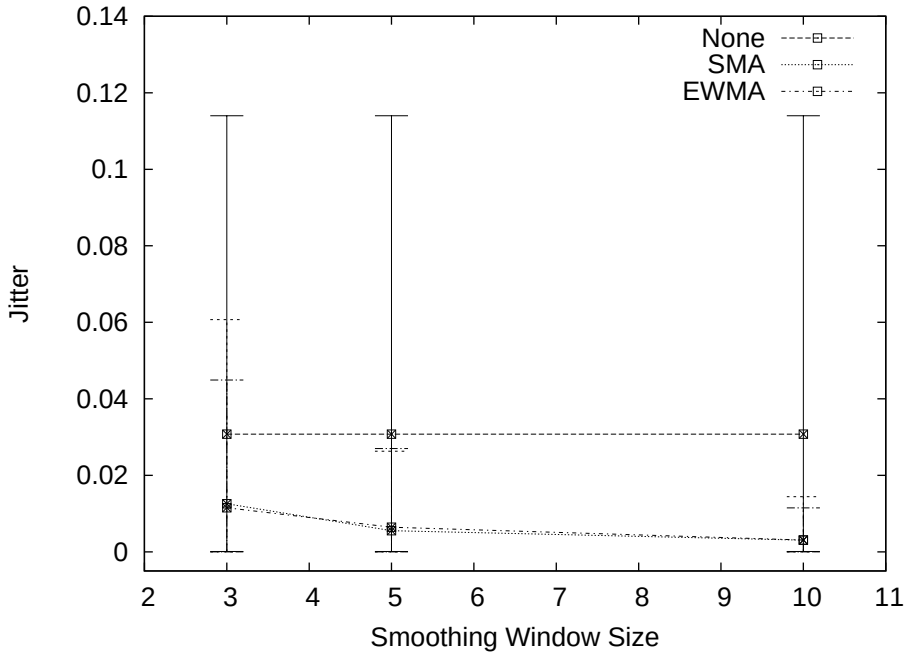


Figure 4.5. Average jitter (error bars show minimum and maximum) in θ component.

analyze data *timeliness* as well as location component jitter—since the location estimate messages must cross at least one network link, there is a chance for the network or OS to add message delivery latency, in addition to any jitter introduced by Mobile Emulab software.

Obviously, it would be ideal if the location estimates exhibited no jitter, but this was not possible in the Mobile Emulab deployment due to variable lighting conditions and low-cost equipment, among other factors. Aside from the θ component, the figures for location P3 are representative of smoothing trends for positions P1 and P2. Somewhat surprisingly, the SMA filter performs very slightly better than the EWMA filter. However, we still suspect that the SMA filter, while providing slightly less noisy data, is introducing more phase lag. The choice of which filter to use may then be based on real-world testing and requirements of the particular kinematic controller driving the robot wheels—perhaps the controller requires very smooth data so that it does not make radical and sudden course corrections, or perhaps it requires very precise data and can cope with slightly more noise.

The smoothing results for the θ component indicate that the EWMA filter performs slightly better at low window sizes than the SMA filter. This is likely occuring because

Table 4.2. Location estimate jitter, x coordinate.

Position	Window	Filter	Mean	Stddev	Variance	Min	Max
P1	—	—	0.001382	0.001751	0.000003	0.000000	0.013400
	3	sma	0.000568	0.000493	0.000000	0.000000	0.003300
		ewma	0.001034	0.000802	0.000001	0.000000	0.006400
	5	sma	0.000339	0.000304	0.000000	0.000000	0.002300
		ewma	0.000587	0.000453	0.000000	0.000000	0.003000
	10	sma	0.000203	0.000179	0.000000	0.000000	0.001200
		ewma	0.000378	0.000300	0.000000	0.000000	0.001900
P2	—	—	0.001048	0.000897	0.000001	0.000000	0.007000
	3	sma	0.000498	0.000382	0.000000	0.000000	0.002000
		ewma	0.000633	0.000507	0.000000	0.000000	0.003800
	5	sma	0.000219	0.000182	0.000000	0.000000	0.001200
		ewma	0.000308	0.000262	0.000000	0.000000	0.002500
	10	sma	0.000102	0.000096	0.000000	0.000000	0.000700
		ewma	0.000210	0.000170	0.000000	0.000000	0.001100
P3	—	—	0.001158	0.000928	0.000001	0.000000	0.007200
	3	sma	0.000436	0.000335	0.000000	0.000000	0.002100
		ewma	0.000549	0.000400	0.000000	0.000000	0.002200
	5	sma	0.000216	0.000171	0.000000	0.000000	0.001100
		ewma	0.000338	0.000260	0.000000	0.000000	0.001200
	10	sma	0.000114	0.000100	0.000000	0.000000	0.000600
		ewma	0.000178	0.000143	0.000000	0.000000	0.000700

Table 4.3. Location estimate jitter, y coordinate.

Position	Window	Filter	Mean	Stddev	Variance	Min	Max
P1	—	—	0.004007	0.004497	0.000020	0.000000	0.033800
	3	sma	0.001548	0.001312	0.000002	0.000000	0.010600
		ewma	0.003042	0.002334	0.000005	0.000000	0.022800
	5	sma	0.000972	0.000799	0.000001	0.000000	0.006700
		ewma	0.001644	0.001310	0.000002	0.000000	0.009600
	10	sma	0.000557	0.000483	0.000000	0.000000	0.002700
		ewma	0.001043	0.000869	0.000001	0.000000	0.005800
P2	—	—	0.003002	0.002422	0.000006	0.000000	0.016000
	3	sma	0.001427	0.001071	0.000001	0.000000	0.005900
		ewma	0.001944	0.001555	0.000002	0.000000	0.011300
	5	sma	0.000673	0.000571	0.000000	0.000000	0.003400
		ewma	0.000944	0.000793	0.000001	0.000000	0.006500
	10	sma	0.000328	0.000264	0.000000	0.000000	0.002500
		ewma	0.000631	0.000489	0.000000	0.000000	0.003100
P3	—	—	0.003610	0.002682	0.000007	0.000000	0.013400
	3	sma	0.001453	0.001140	0.000001	0.000000	0.007100
		ewma	0.001828	0.001323	0.000002	0.000000	0.007700
	5	sma	0.000656	0.000521	0.000000	0.000000	0.002900
		ewma	0.001124	0.000802	0.000001	0.000000	0.004700
	10	sma	0.000359	0.000285	0.000000	0.000000	0.001600
		ewma	0.000560	0.000428	0.000000	0.000000	0.002500

Table 4.4. Location estimate jitter, θ coordinate.

Position	Window	Filter	Mean	Stddev	Variance	Min	Max
P1	—	—	0.033818	0.039660	0.001573	0.000000	0.278100
	3	sma	0.012940	0.010670	0.000114	0.000000	0.085000
		ewma	0.017922	0.014358	0.000206	0.000000	0.101300
	5	sma	0.008314	0.006775	0.000046	0.000000	0.054500
		ewma	0.009132	0.007144	0.000051	0.000100	0.043100
	10	sma	0.004745	0.004075	0.000017	0.000000	0.024400
		ewma	0.006000	0.004656	0.000022	0.000000	0.028500
P2	—	—	0.048632	0.042012	0.001765	0.000000	0.313200
	3	sma	0.023737	0.017557	0.000308	0.000000	0.091000
		ewma	0.023599	0.018493	0.000342	0.000100	0.116500
	5	sma	0.010949	0.009655	0.000093	0.000000	0.055000
		ewma	0.010076	0.008860	0.000079	0.000000	0.066300
	10	sma	0.005188	0.004273	0.000018	0.000000	0.043100
		ewma	0.006823	0.005423	0.000029	0.000000	0.027600
P3	—	—	0.030757	0.023086	0.000533	0.000100	0.114000
	3	sma	0.012566	0.009948	0.000099	0.000000	0.060700
		ewma	0.011506	0.008169	0.000067	0.000000	0.044900
	5	sma	0.005536	0.004317	0.000019	0.000000	0.026300
		ewma	0.006425	0.004861	0.000024	0.000000	0.027000
	10	sma	0.003044	0.002439	0.000006	0.000000	0.014400
		ewma	0.003085	0.002283	0.000005	0.000000	0.011500

the θ component is so much noisier than the x and y components—so the exponential weighting of the EWMA filter of the early window data does not appear to "add" more noise than the SMA filter. Consequently, if the kinematic controller is sensitive to θ jitter, the EWMA filter will be more appropriate to use in *visiond*.

Table 4.5 shows jitter statistics for message interarrival times for a nonmoving robot. Naturally, interarrival times are not affected by filter, since filter computation runtime should be a very tiny portion of wall clock interarrival time. However, the minimum, maximum, and standard deviation statistics are of particular interest. First, the standard deviations suggest that the messages typically arrive approximately 0.33 seconds after the preceding message. However, the minimum and maximum interarrival times can vary from almost 0 seconds to around 0.66 seconds. The maxima could occur if the robot is not tracked in a single frame, but is found again in the next frame (the tracking code allows this to occur). Clearly the code could be extended with an option to always send the latest position for a robot even if it was not found in one frame, as long as it had been found in a recent frame.

There are two potential solutions to reducing message interarrival times. First, the tracking code in *visiond* could produce updates according to a nearly real-time schedule,

Table 4.5. Jitter in location estimate message interarrival times.

Position	Window	Filter	Mean	Stddev	Variance	Min	Max
P1	—	—	0.033404	0.001188	0.000001	0.026700	0.065800
	3	sma	0.033378	0.000563	0.000000	0.030200	0.046500
		ewma	0.033382	0.001080	0.000001	0.019900	0.060900
	5	sma	0.033377	0.000507	0.000000	0.030900	0.044800
		ewma	0.033404	0.001792	0.000003	0.028500	0.068100
	10	sma	0.033379	0.000520	0.000000	0.028400	0.042100
		ewma	0.033404	0.001448	0.000002	0.024400	0.075700
P2	—	—	0.033367	0.000431	0.000000	0.027600	0.039400
	3	sma	0.033367	0.000329	0.000000	0.026800	0.040000
		ewma	0.033368	0.000813	0.000001	0.025000	0.046200
	5	sma	0.033401	0.000890	0.000001	0.026000	0.053000
		ewma	0.033366	0.000351	0.000000	0.032500	0.034500
	10	sma	0.033429	0.001922	0.000004	0.031800	0.090900
		ewma	0.033367	0.000378	0.000000	0.025000	0.039100
P3	—	—	0.033375	0.001025	0.000001	0.028500	0.045000
	3	sma	0.033404	0.001122	0.000001	0.031400	0.066700
		ewma	0.033381	0.000464	0.000000	0.032300	0.046300
	5	sma	0.033376	0.000949	0.000001	0.018800	0.051600
		ewma	0.033367	0.001594	0.000003	0.000400	0.066000
	10	sma	0.033367	0.001495	0.000002	0.002300	0.064000
		ewma	0.033367	0.001132	0.000001	0.026700	0.039700

always sending an update for each tracked robot at every time quantum. This may well require a real-time OS underneath the Mezzanine and *vmc-client* instances. Alternatively, *robotd* or *pilot* could be enhanced with a predictive filter, such as an EKF, that predicts based on the real estimates from *visiond* (and perhaps wheel odometry as well, or any other location data that can be gathered). This filter could certainly operate at speeds much greater than 30 Hz—perhaps matching the execution speed of the algorithm producing the drive wheel speeds for the robot. The best solution is clearly the real-time solution, if possible given available hardware and the number of camera frames to process per quanta, since it nearly eliminates the issue of *visiond*-caused phase lag in the data (except for the case when a robot is temporarily lost for one or more frames).

Table 4.6 provides statistical data for location estimates during a 1 m linear move. For each location, the data are parameterized by filter window size and type ("—" if no filter was applied).

Table 4.6. Jitter for linear motion location estimates and message interarrival times.

Data Type	Window	Filter	Mean	Stddev	Variance	Min	Max
x	—	—	0.004431	0.003850	0.000015	0.000000	0.023400
	3	sma	0.001660	0.001288	0.000002	0.000000	0.006700
		ewma	0.002269	0.001888	0.000004	0.000000	0.013600
	5	sma	0.000923	0.000734	0.000001	0.000000	0.005000
		ewma	0.001231	0.000896	0.000001	0.000000	0.004200
	10	sma	0.001066	0.000619	0.000000	0.000000	0.002900
		ewma	0.001326	0.000756	0.000001	0.000000	0.003600
y	—	—	0.005880	0.002511	0.000006	0.000400	0.015200
	3	sma	0.005751	0.001953	0.000004	0.000900	0.009800
		ewma	0.005868	0.002083	0.000004	0.000700	0.010500
	5	sma	0.005894	0.001907	0.000004	0.002200	0.009400
		ewma	0.005819	0.001968	0.000004	0.001500	0.009500
	10	sma	0.005796	0.001860	0.000003	0.002200	0.009100
		ewma	0.006016	0.001782	0.000003	0.002700	0.009200
θ	—	—	0.035807	0.034841	0.001214	0.000500	0.234800
	3	sma	0.011703	0.010182	0.000104	0.000000	0.064800
		ewma	0.013115	0.011207	0.000126	0.000000	0.061300
	5	sma	0.007475	0.007330	0.000054	0.000000	0.056800
		ewma	0.006677	0.005165	0.000027	0.000100	0.027200
	10	sma	0.003800	0.003179	0.000010	0.000100	0.018400
		ewma	0.003672	0.002829	0.000008	0.000000	0.015800
time	—	—	0.033349	0.001006	0.000001	0.030300	0.036800
	3	sma	0.033367	0.000093	0.000000	0.033200	0.033600
		ewma	0.033373	0.000362	0.000000	0.032300	0.034400
	5	sma	0.033365	0.000438	0.000000	0.032400	0.034600
		ewma	0.033367	0.000366	0.000000	0.032300	0.034400
	10	sma	0.033367	0.000109	0.000000	0.032900	0.033700
		ewma	0.033365	0.000346	0.000000	0.032200	0.034400

CHAPTER 5

USABILITY TOOLS

Because of its dynamic nature, a mobile wireless testbed must present researchers with more control options and data interfaces. Users must be able to control and script motion easily. The addition of sensor network motes to experiments necessitates new kinds of monitoring interfaces to motes, since few such tools already exist for mote applications. Whereas users of Emulab's classic wired network testbed can employ a whole host of tools, such as netcat [13], iperf [27], nmap [10], tcpdump [11], and the like, to control and monitor their experiments, as well as many custom Emulab tools, motes simply do not support the same experiment paradigms. Emulab's wired network testbed also makes guarantees to users with respect to link bandwidth, loss rate, and many other characteristics, but a live wireless testbed can make no such guarantees and must also cope with the problem of external interference. Additional tools can significantly ease the pain of real-world experimentation with sensor network devices.

5.1 Wireless Characteristics

5.1.1 Connectivity

We provide wireless connectivity information for different radio power levels so that experimenters can quickly choose which motes in our testbed to use for experimentation, and also for later use in results analysis. For instance, one could imagine evaluating the effectiveness of a link estimation routing protocol by comparing generated routes with testbed maps of link quality measurements taken at an earlier time. We developed a Java applet that displays wireless connectivity between wireless nodes in a geographic manner.

We run a simple TinyOS program on selected groups of motes (generally, all the fixed motes in the testbed) to collect this data. For each power level, and for each mote, the mote will broadcast 100 packets at a rate of 8 pps. All other motes listen and record the number of packets heard and their RSSI values. This information is analyzed and stored for later use, and recent data can be sent to the web interface for display in an interactive

Java applet. The applet uses a loosely-defined data file, allowing easy display of any link characteristic required in the future. The data format can also associate a set of statistics with each link characteristic value so that experimenters can better ascertain how links change over time.

In the applet itself, using our default data set, experimenters can select different power levels, view statistics by selecting source or destination mote sets, and filter out links below a certain threshold or by a best neighbor limit.

Figures 5.1 and 5.2 show this application with packet reception percentage and RSSI values, respectively. (These figures show only the three best-connected neighbors for each node, at power level 0x8.) Figure 5.1 shows that at power level 0x8, most motes in the testbed can communicate with at least three other motes with relatively low packet loss rates. However, many of the "links" shown do not have a bidirectional component. This is due to the fact that many of these links are assymetric, and when combined with the k-best neighbor filter, both directions may not be part of both nodes' k-best neighbor sets. Figure 5.2 provides insight to the degree with which the best three neighbors (in terms of RSSI) correlate with the best three packet reception neighbors, for each node. Some correlation is evident; for instance, mote114 and mote115, at the bottom of the topology shown, exhibit almost 100% packet delivery rates, and rather high RSSI values. However, comparative inspection of the figures reveals that many packet reception links do not have corresponding RSSI links. Much additional information may be quickly obtained by inspecting the testbed's sensor network topology using this application.

5.1.2 Active Frequency Detection

Although robot resources cannot currently be space-shared, the static motes may be used by multiple experimenters at once. Consequently, we need to do our best to prevent frequency overlap between experiments. We developed a TinyOS-based frequency-hopping radio sniffer for motes. This program listens for packets on channels for a few seconds and sends information to the serial port including the frequency, received packet count, and number of valid packet CRCs. This program receives not only transmissions generated from our system, but from external sources as well, making it a valuable tool with which to periodically scan for many types of interference.

Unfortunately, since we expect experimenters to reduce radio power to create interesting multihop topologies, this application must run from several different observation points to ensure that all transmissions are monitored. Although this application is not

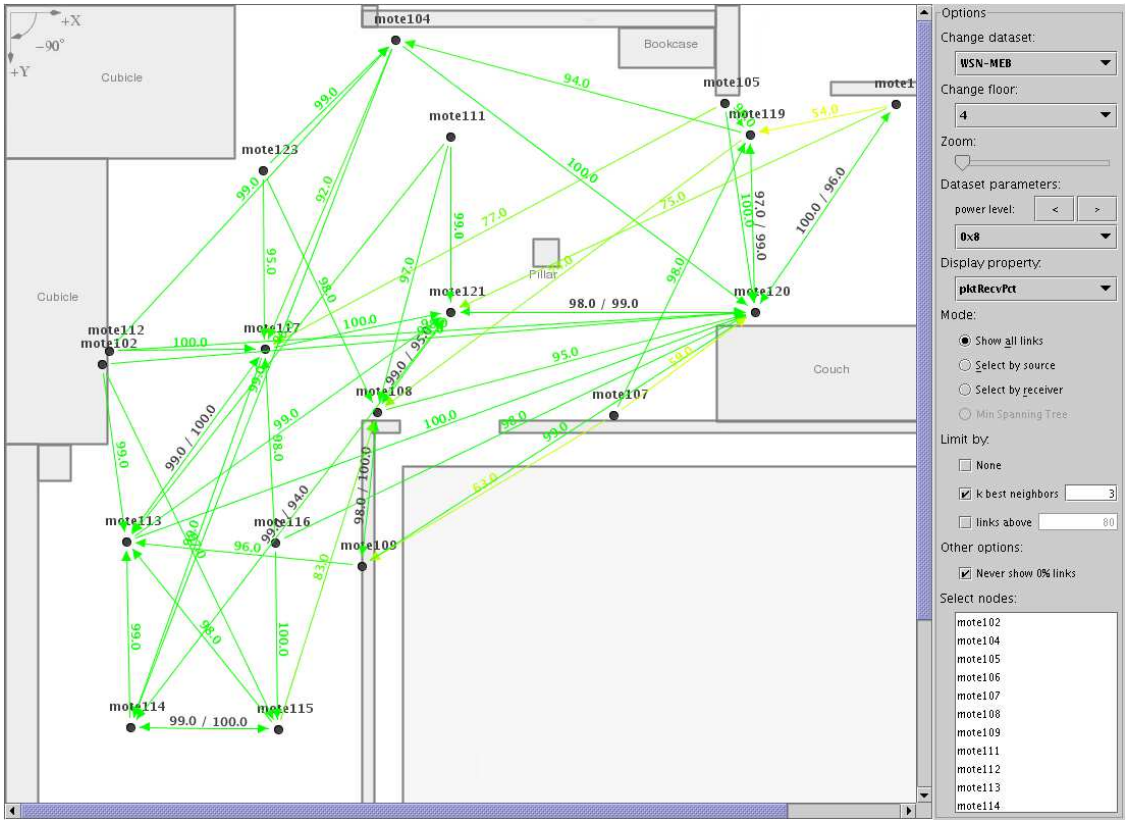


Figure 5.1. Screenshot of wireless connectivity applet showing received packet statistics.

currently integrated into Emulab, users can install the application binaries on their motes before beginning experimentation, as well as during runtime, to find an appropriate frequency for their experiments. By installing it on their motes, they can determine which frequencies their motes can overhear, and adapt their applications accordingly.

5.2 Sensor Network Application Management

A debilitating problem in sensor network research is the lack of readily-available tools for interfacing with mote devices. From a quick survey of the TinyOS contributors development tree, one can conclude that researchers generally construct their own interface programs for sensor network applications.

Developers may often wish to manually interact with a running application by sending command messages to alter program state or to dump a set of debugging data. Beyond manual interaction, real-time data display about the state of the network at large may be useful. One may imagine wanting access to the current routing and sensing states on

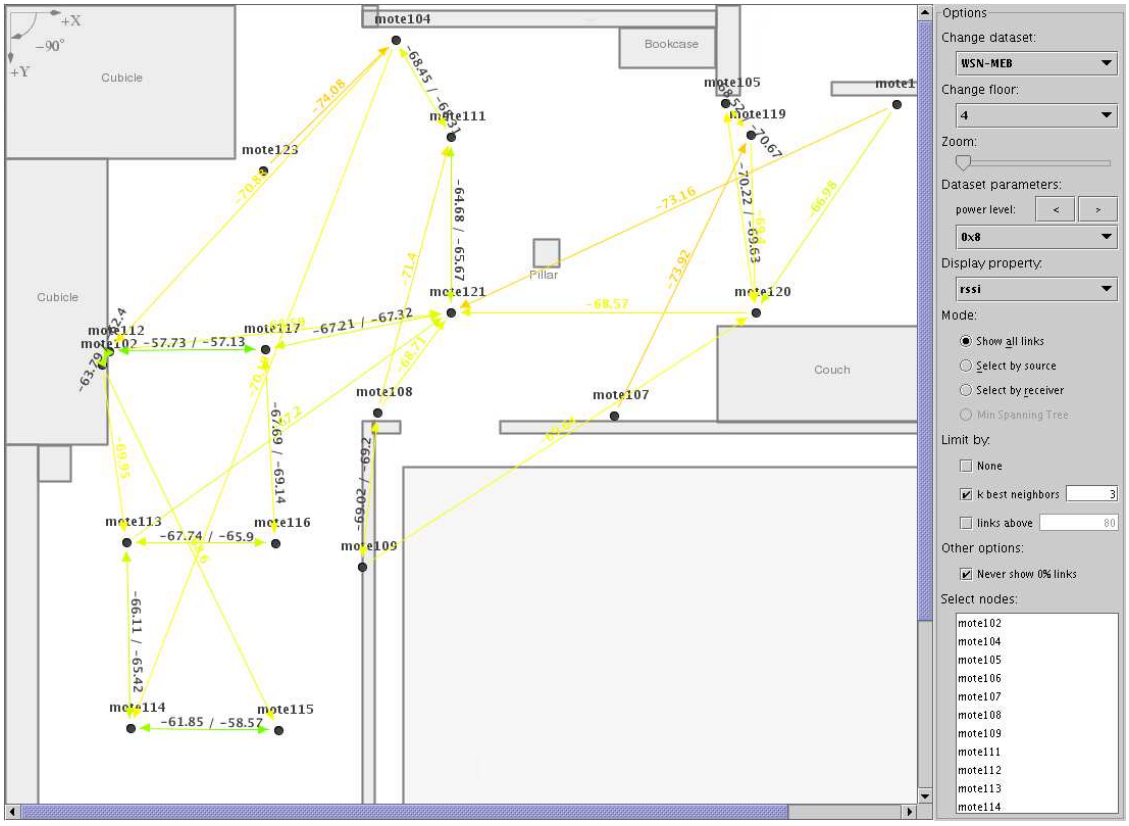


Figure 5.2. Screenshot of wireless connectivity applet showing RSSI statistics.

each node, for instance. When nodes can move, current information about their location and intended destination will also be useful. The following sections describe a set of useful applications and tools that enable sensor network developers (and Mobile Emulab users) to more easily communicate with their applications, explore data, and automate command tasks.

5.2.1 Mote Management and Interaction

`EmulabMoteManager` is an application that provides users with a basic set of functionality for communicating and otherwise interacting with sets of motes. This functionality is provided by the main application, as well as a set of stock modules that use the APIs provided by `EmulabMoteManager`. Figure 5.3 shows a screenshot of `EmulabMoteManager` and several plugins, including one application-specific plugin for the `MobiQ` application, which we discuss in Section 6.2. `EmulabMoteManager` also allows users to load new modules conforming with its API, and create multiple instances of them. Modules can

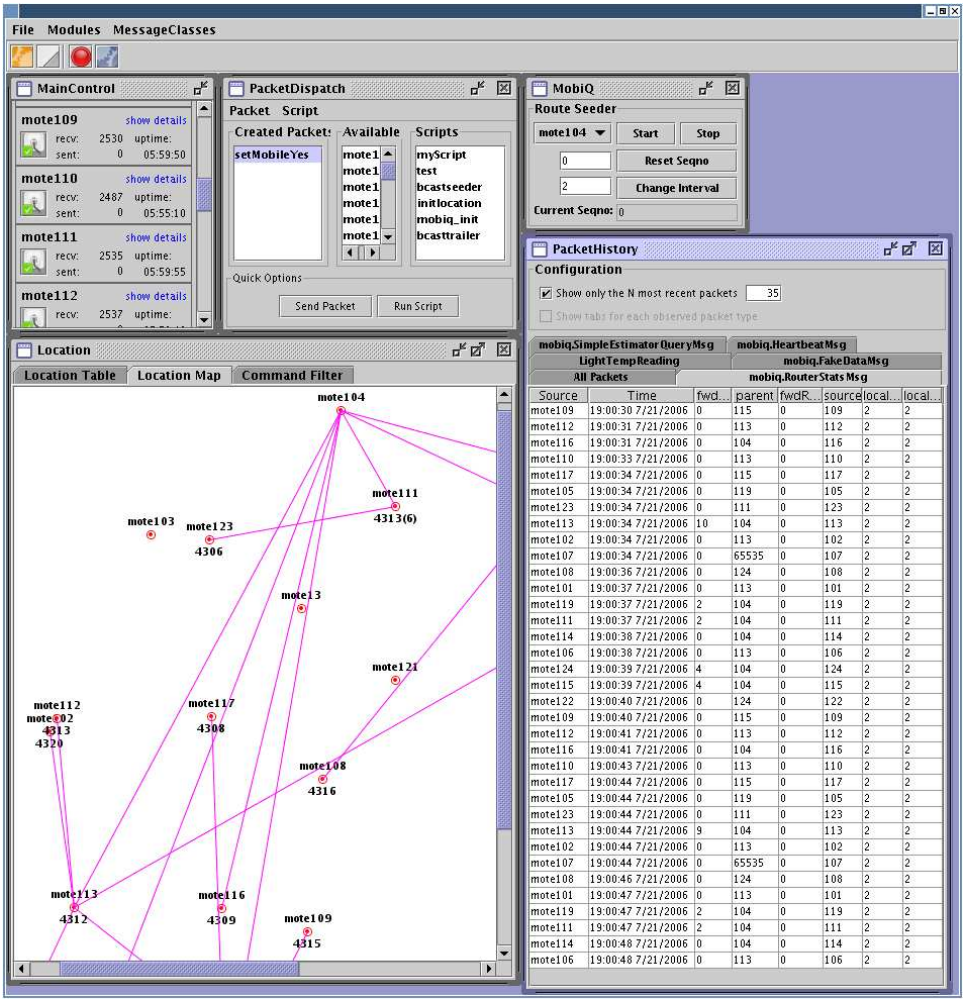


Figure 5.3. Screenshot of the EmulabMoteManager application.

save state to a session file, which is read when the module is next loaded. Finally, `EmulabMoteManager` provides two different APIs for additional module loads via the *ManagerModule* and *AppModule* interfaces; we discuss these interfaces in Section 5.2.2.

5.2.2 Key Interfaces

The *ManagerModule* interface should be implemented by modules that are capable of managing connections to motes and anything deemed related to that function. It requires implementation of several functions, such as commands to retrieve available motes and to connect or disconnect from motes. Several subscription methods enable interface users to easily listen to a subset of motes and be notified when they send messages. Subscriptions are parameterized by message *type*, source mote, or a combination. In order to support

users who may require more than just the message bytes, one can use similar subscription methods to receive message metadata (timestamp, source mote name, and a boolean representing the result of the CRC check on the received bytes) in addition to the message itself.

The *AppModule* interface should be implemented by modules that provide generally-useful functionality that can interoperate with any valid *ManagerModule* and its motes, or more simply, extra application-specific functionality not already supplied by existing modules. Through this interface, users may export sets of GUI menu and icon items that are added to the top of the GUI to streamline ease of use. More importantly, *AppModule*s must implement an `initModule` method that notifies them of the parent module container, any MoteManager interfaces already created, and any previous session state saved by this module. The parent module container interface, *MoteAppContainer*, contains a simple method to save session state at any time. However, state is only written to the filesystem if the user has enabled the auto-save option, or specifically requests that the session be saved. Finally, any loaded module may be multiply instantiated in the GUI.

5.2.3 Message Handling

`EmulabMoteManager` makes use of lower-level abstractions provided by the TinyOS distribution. When writing TinyOS applications, users are encouraged to communicate using "Active Messages", which use a standard format and are supported by modules in the TinyOS kernel. Generally, each message is defined by a packed C structure, and associated with an "Active Message ID". The TinyOS distribution supplies a tool named `mig` that converts these Active Message C structures into Java classes. Each message wrapper class has accessor methods for each variable in the original structure, as well as private methods for packing the data to the mote platform specified to `mig`.

`EmulabMoteManager` and `MoteLogger` both derive much of their usefulness from these Java wrapper classes. Using Java's reflection mechanism, a library included in both applications parses messages classes and extracts the original variable names and types. With this information, the library can print full details contained in a message for which the class is available, and also construct messages from user-supplied field values, which it parses as appropriate for the original message type.

Unfortunately, `mig` does not maintain the original structure of complex data types. For instance, members of a member that is itself a structure appear as members with under-

scores in the accessor method name in the resulting Java class. Since `EmulabMoteManager` makes use of the way in which field accessor methods are named in the Java message class wrappers, it cannot reconstruct the exact original message structure. This has severe implications for proper display of data when logging or displaying messages to users. To work around this loss of information, an `EmulabMoteManager` user must supply a *spec* file in addition to the Java message wrapper class. `mig` produces a specification of the structure that contains the missing nesting information, although most users typically ignore it. With the nesting information, the message handling libraries can display arbitrary messages with complex types.

5.2.4 Stock Plugins

This section describes several of the stock plugins available for use with this application and discusses their general usefulness apart from Emulab.

5.2.4.1 EmulabMoteControl Plugin

This plugin implements the *ManagerModule* interface for motes in the Emulab testbed and also provides a GUI interface for managing them. Figure 5.3 shows an instance of the EmulabMoteControl plugin near the upper left corner. Users can add motes to this manager either by directly connecting to the network-exported serial device, or by connecting to an instance of the TinyOS `SerialForwarder` utility, which in turn attaches to the Emulab network-exported serial device. Other TinyOS Java libraries allow users to write messages (in TinyOS's Java message class wrapper abstraction) to `SerialForwarder` instances. By using the second interface, we provide experimenters with the same abstraction that they may already use in their existing data collection or monitoring programs. This can ease the pain of transforming all or part of their applications into plugins for the `EmulabMoteManager`. The plugin creates a simple GUI interface for adding motes, connecting to and disconnecting from motes, and displaying detailed statistics on packet transfer.

In addition to connection management facilities, the plugin integrates the arbitrary TinyOS Java message-parsing libraries described in the previous section. Users may load Java classes representing TinyOS messages and they will be recognized by the mote connections. Packet creation and display methods are also then made available to any *AppModule* plugin instances that are utilizing this manager, as soon as a new message class is loaded.

By loading multiple instances of the EmulabMoteControl plugin, users will be able to communicate with multiple mote subsets at once, if their plugins can support multiple *ManagerModule* interfaces, or selection of the appropriate interface to use.

5.2.4.2 PacketHistory Plugin

The PacketHistory plugin implements the *AppModule* interface, and creates a GUI for displaying packets in a series of tables. The plugin subscribes to a *ManagerModule* for all message types and mote sources. In the first table, it displays received packet metadata including source mote name, message type (numeric ID if matching Java message class not loaded; otherwise message name) reception timestamp, a boolean representing success of the CRC calculation, and message length. When messages received have a matching, loaded Java message class, it creates a new table for all subsequent messages of that type. Users can double-click on packets in any of the tables to see the packet metadata and interpreted message fields. Figure 5.3 shows an instance of the PacketHistory plugin with several different message types (the message table shown is displaying messages from the `MobiQ` application discussed in Section 6.2) in the lower right corner.

5.2.4.3 PacketDispatch Plugin

The PacketDispatch plugin enables users to send messages to subsets of motes, and also to script actions involving sending. Using a GUI interface, users can create packets for Java message classes that they have loaded by filling in desired field values in a form. Once a packet is created, the user can associate it with one or more motes known to this plugin through the manager, and a simple click will dispatch it to the selected motes. These target associations with motes are dynamic and can be easily modified.

Since the goal of this software is to aid sensor network application designers by reducing the amount of application-dependent interface programs they must write, this plugin also allows users to script packet dispatching. Scripts can take the place of hand-tooled packet sending code, and can be easily reused in other applications without recompiling a new interface.

Using a GUI interface, users can create scripts and add action items. The most important type of action item is a packet send, which users can add to scripts in different ways. First, users can append already created packets and their target mote associations to a preexisting script. Alternatively, they may manually add a new packet and new mote

targets by editing the script. In addition to packet sending, another function that pauses script execution is supported.

The ScriptEditor allows users to add, remove, and change positions of action items in the sequence. Individual script action items may also be edited in a similar GUI manner. The data format used to store scripts is very intuitive, allowing scripts to be hand-edited easily. This plugin is completely independent from the Emulab sensor network testbed, and could just as easily be used inside the `EmulabMoteManager` to interface to another testbed.

Figure 5.3 shows an instance of the PacketDispatch plugin with a created packet and several scripts in the upper middle portion of the application.

5.2.4.4 Location Plugin

The location plugin provides methods for users to monitor mobile mote motion. By connecting to a *LocationSource*, the Location plugin monitors the current location, intended destination, and state of motion execution.

Currently, the GUI implementation of this plugin allows users to connect to a network-based *LocationSource*. Once connected to the source, mote locations (x, y coordinates in centimeters, and orientation in degrees) read from the *LocationSource* are placed into a table and updated when subsequent notifications arrive. In addition, a graphical representation of location status is also displayed. A map showing the current location and straight-line path to the intended destination is updated when location updates arrive at the source. Users may then monitor mote-controlled motion, or control it themselves with the `PacketDispatch` plugin.

Some aspects of this plugin are closely tied to the Emulab testbed, although the plugin could be used by another mobile testbed with only slight modification of either the other testbed, or the plugin itself. For instance, the GUI currently accepts only a network data source. The message classes by which the location plugin communicates with the motes themselves are also closely related to Emulab's needs, and may need to be adapted to suit other users or testbeds.

Figure 5.3 shows an instance of the Location plugin in the lower left portion of the application. The motes displayed are located in the corner of the "L"-shaped testbed deployment, and the lines indicate a multihop routing topology.

5.3 Mote Data Logging

In addition to using `EmulabMoteManager` to monitor and manage their mote networks in real-time, experimenters may also want to record all packets sent from the motes on their serial lines. Mobile Emulab provides this functionality by logging packets to a MySQL database. Figure 5.4 shows an example of logged packet data within a MySQL database via the database command-line interface; the packets are of type `ABPacket`, a simple message with two integer fields.

As previously described in Section 5.2.3, I have written a set of Java libraries to process arbitrary TinyOS Active Messages that have been transformed into Java message wrappers with `mig`. These libraries extract structure and type information describing the original C structures that each message originated as on the mote from the Java message wrappers. These libraries provide the basis for mote message logging in Emulab.

First, any user wishing to use mote logging must upload Java message classfiles through extensions to the Emulab web interface. If they desire more structure in the database so that the schema more greatly resembles any complex types existing in the messages, they may upload the associated spec data produced by `mig` as well. Once the user has uploaded these files, they may create an experiment with any necessary motes, robots, PCs, and any other necessary resources; in addition, they must reserve a single PC running a specific OS that provides a recent MySQL database version for use by the mote logger. When the user swaps out their experiment, or when the user requests a restart of the mote logger, the database is restarted, allowing the software to synchronize the database files with the permanent experiment storage on centralized Emulab servers. Users may restart the mote logger during experiment runtime by using Emulab's XML-RPC command interface. When restarting the mote logger, the user may specify a new name for the dataset, recorded in a master metadata table. Since all mote logging associated with each experiment is kept in a single database, users may associate notes with this name to easily recall which tables in the database correspond to each run of the logger.

The mote logger application creates tables as necessary in the database, and records name and associated notes for each logging run in the metadata table. It records a timestamp, source mote name, message type, CRC check value, length, and raw data in a single table for each message received. When the logger matches a Java message wrapper with the message type and length of an inbound message, it creates a number

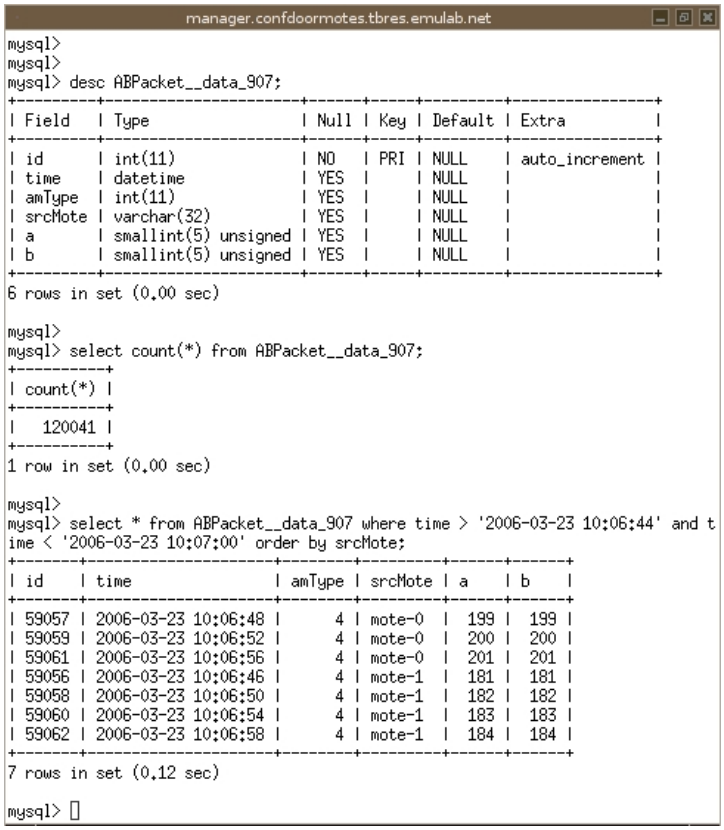


Figure 5.4. Screenshot of logged data in a MySQL database.

of tables in the database if they do not already exist, and inserts data from the message into each of these tables. Several different tables are created so that the user can see the message data for each recognized message type in a format resembling its original C structure definition. For instance, for messages with a field that is either a simple or multidimensional array, an auxiliary table containing rows of array indices and their matching value, in which each row is tagged with the source packet ID, is created. Data from each packet are inserted each time a recognized message is logged. Then, users can select over this table and view the data in an array format, such as *[0,1,1,2,3,...,N]*. This enables simple viewing, and later, offline parsing of data by analysis applications. Users see the data just as they would have if they themselves called the functions in the Java message wrapper class to extract the data. (Unfortunately, although the logger supports multidimensional array views in the database schema, it does not yet support viewing of C `structs`—structure members are flattened into individual fields.) For more complex data structures such as multidimensional arrays, more complex auxiliary data tables are

created. However, the user essentially only needs to select from one of two tables: either the table containing the metadata with the raw message byte array data, or another table containing the metadata and the data extracted from the messages in human-readable form.

5.4 User-available Location Information

To properly correlate application or protocol events with location, experimenters must be able to access the ground truth mote location generated by the localization system. We have implemented an application, `placeholder`, that provides this information via a network source, as well as to a mote application through custom TinyOS components.

`placeholder` functions as a proxy between the Emulab event system and either mote or PC-based applications. When run by an experimenter on an Emulab PC, it subscribes to the MODIFY, SETDEST, and COMPLETE events for all nodes in the experiment. The MODIFY event is generated by *embroker* whenever a node tracked by our localization system changes position (however, recall that these events are generated at approximately 1 Hz instead of the 30 Hz speed of the localization system). The SETDEST event is generated whenever a robot is commanded to move through a variety of interfaces. The COMPLETE event represents the completion of the SETDEST event.

Once subscribed to the event stream, `placeholder` listens for client connections to a network socket, and relays information contained in the events listed above to all connected clients using a simple text format. `placeholder` accepts motion requests from clients, and converts these requests into appropriate motion commands in the Emulab event system, where they are executed. This provides experimenters with another scriptable location source that can easily be accessed by a number of different languages.

Because robots and the motes they carry are separate node entities in Emulab, this program also accepts a list of aliases for nodes. This is important since all localization estimates and motion commands are reported for and excutable only by robots, but not for motes. Thus, if the experimenter wishes to write an application that sees the robot-mounted motes as mobile, they can simply alias the names of each robot node to the associated robot mote name, or any other renaming scheme.

Many mote experimenters who design mobile applications may actually wish to drive the robots from their onboard motes, or in some other way, abstract the mobility platform into a mobile interface that the mote applications may themselves access. We have written

TinyOS components that implement such an interface. These components communicate with an instance of `placeholder` through the Location Plugin to `SNAP-M`. Consequently, mote applications can drive their robot couriers themselves and be notified of current location, status, and completion events. The interface implemented by these components supports three-dimensional motion, but may require modification for nontraditional motion platforms in other testbeds, such as underwater sensor robots.

CHAPTER 6

CASE STUDIES

An important part of validation of a new testbed is to show that it supports the needs of experimenters. This chapter demonstrates the usefulness of the testbed through three very different case studies and analyzes some of its strengths and weaknesses.

6.1 Environment Analysis

In order to characterize the wireless properties of the mobile arena, we have run experiments in which several robots move across the floor at regularly-spaced intervals, and while stopped at each interval, record packets broadcast by a single fixed mote. This experiment demonstrates how users may programmatically script motion, but also illustrates the irregular nature of the signal propagation environment and also provides insight into the nature of repeatability in our testbed.

6.1.1 Experiment Basis and Methodology

The experiment is entirely automated from within a single *ns* file, and makes heavy use of the Emulab event system. In particular, the scriptable nature of motion events is provided by event system components such as *program agents* and event *groups* and *sequences*. The following portion from an *ns* file originally written by a colleague, shown in Figure 6.1, illustrates the manner in which users can programmatically script motion.

In the first few lines, this code reserves a robot named **$walker**, constructs a program agent named **$logger** to run a logging utility, and creates an event sequence named **$rowwalk**. The code is parameterized by maximum height and width of an area, and the step size for each move (**$XINCR** and **$YINCR**). The variable **$ltor** controls the direction of motion of the robot down each row. Rows are oriented parallel to the X axis. The main body of the loop chooses an x, y position and checks that it is inside the bounds of the motion arena and that it does not lie in any of the known obstacles in the Emulab database. For each valid position, a node SETDEST event is appended to the sequence

```
1:  set walker [$ns node]
2:  set logger [$walker program-agent \
3:           -command ``/proj/tbres/johnsond/pkt_listen'']
4:  set rowwalk [$ns event-sequence {}]
5:  set ltor 1
6:  for {set y 0} {$y <= $HEIGHT} {set y [expr $y + $YINCR]} {
7:      set row($rowcount) [$ns event-sequence {}]
8:      for {set x 0} {$x <= $WIDTH} {set x [expr $x + $XINCR]} {
9:          if {$ltor} {
10:             set newx [expr $XSTART + $x]
11:             set newy [expr [$walker set Y_] + $y]
12:         } else {
13:             set newx [expr $XSTART + $WIDTH - $x]
14:             set newy [expr [$walker set Y_] + $y]
15:         }
16:         if {[$topo checkdest $walker $newx $newy]} {
17:             $row($rowcount) append \
18:                 ``$walker setdest $newx $newy 0.2''
19:             $row($rowcount) append \
20:                 ``$logger run -tag $newx-$newy''
21:         }
22:     }
23:     $rowwalk append ``$row($rowcount) run''
24:     incr rowcount
25:     set ltor [expr !$ltor]
26: }
```

Figure 6.1. ns-2 code that moves a robot through a grid and logs output.

for this row. An event that runs the **$logger** program-agent and records its output in a file labeled with the appropriate x, y position is appended following the motion event. Finally, each row event sequence is itself appended to the main rowwalk event sequence. By running this event sequence, either automatically from later sections of the *ns* file, or by hand using the Emulab `tevc` event injector, users can start a run of this experiment. Since events in a sequence are not fired until their predecessor has completed, logging only occurs when the robot is at a stop.

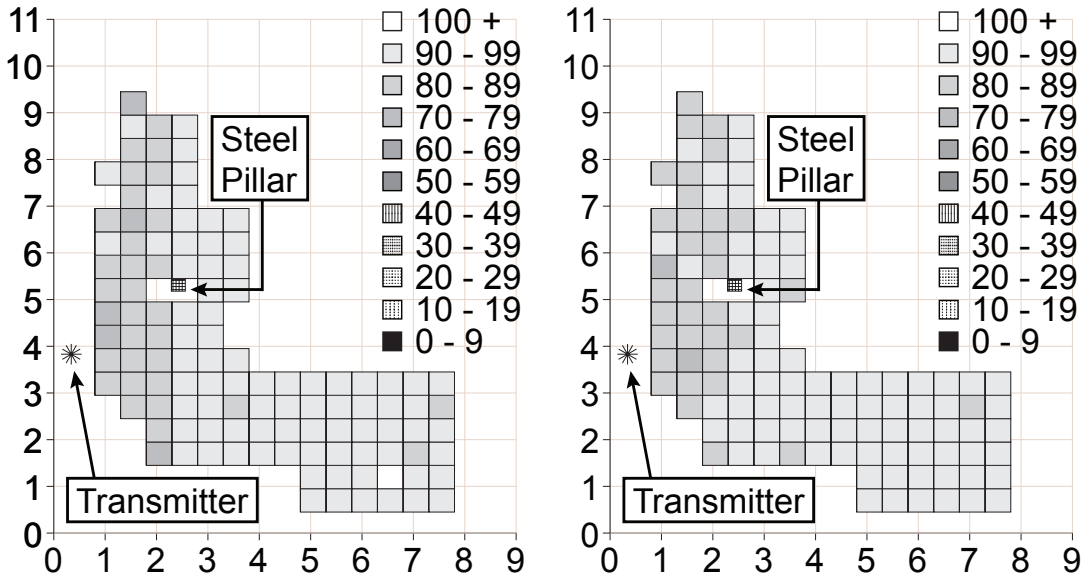
6.1.2 Results

We ran a modified version of this experiment with three robots, using repeated runs with two different fixed transmitter motes and several transmit radio power levels. The experiments were conducted from approximately 0630 until 1710, which provides conditions similar to those that experimenters would face. Both fixed transmitter motes have excellent line of sight to most portions of the L-shaped experimental area, except where obscured by the pillar. The three robots divide the L-shaped area so that the first is assigned all area left of the pillar; the second, all area to the right of the pillar and slightly into the lower portion of the L-shaped area; the third, all remaining area. They move

independently of each other, so one robot may be moving while another is measuring. The fixed transmitter mote antennae are positioned approximately 18 cm above ground, while the robot antenna are extended to approximately 1 m of height. Each run requires 18–21 minutes, largely depending on time required to re-identify the robots if they are ever obscured from localization system view when people move through the area. No attempt was made to stop people from walking through the area during these experiments.

First, Figures 6.2(a) and 6.2(b) show maps of packet reception across the mobile area from the first two runs at transmit power level 0xff, with the transmitter situated left of the area. Immediately obvious is that the first robot is generally too near the transmitter, and that the onmidirectional antennae on the transmitter is lobed so that it cannot transmit at such large angles as well as to the other robots, which have a much smaller angle to the transmitter. The two figures appear to be much the same.

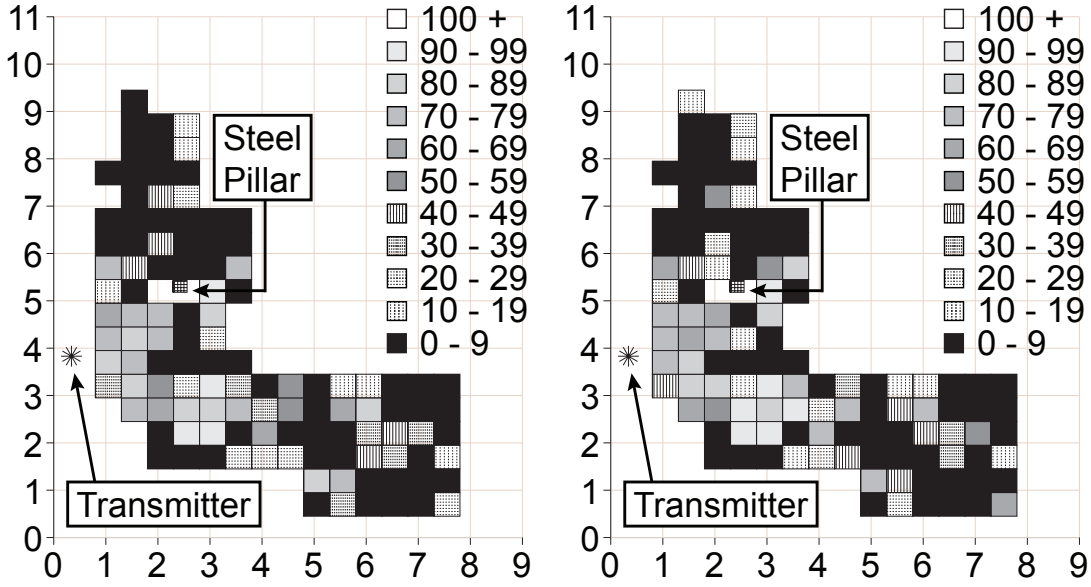
Figures 6.3(a) and 6.3(b) show maps of packet reception at transmit power level 0x03. Once again, many areas appear to have similar packet reception. However, areas of poorer reception are not distributed according to range from the transmitter, as would be expected in an open, flat area. There are remnants of this distribution, as many areas farther away from the transmitter exhibit increasingly poor reception. However, even these areas are interspersed with areas of medium-quality reception.



(a) First run.

(b) Second run.

Figure 6.2. Packet reception at power level 0xff; two runs.



(a) First run. (b) Second run.

Figure 6.3. Packet reception at power level 0x03; two runs.

Figure 6.4 shows a plot of average RSSI values for the first run at power level 0x03. When compared with Figures 6.3(a), this figure demonstrates a partial correlation between packet reception and RSSI. Locations with 0-9 packets received often exhibit the highest average RSSI values. However, there are also areas (especially near the crux of the L) in which RSSI values are quite low, but reception is quite high. These disparities are possibly caused by multipath effects.

To demonstrate similarity over only a few runs, we chose to use a metric of maximum range (i.e., the highest number of received packets less the lowest, at each location). This reflects the worst possible packet reception disparity at each location. Figure 6.5(a) shows a map of these ranges calculated over three runs. Generally, most locations show a maximum packet reception range of less than 6-8 packets. There are a few extreme outliers; however, one can be discounted due to the presence of the pillar, others by nearness to the transmitter, and even by the presence of people walking through at random times. Since each run requires approximately 20 minutes and there are 3 runs, people will generally walk through the area several times, and only one such interference at any location is required to increase the range significantly. Figure 6.5(b) shows packet reception range at power level 0x03. The ranges in this figure tend to fluctuate more than for the case of power level 0xff. This may occur if in a complex signal environment

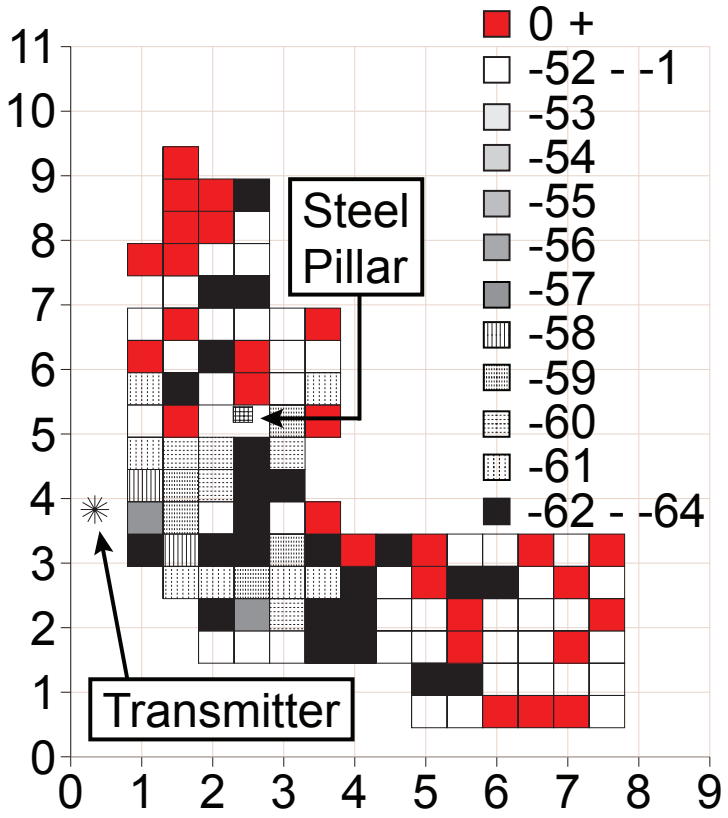
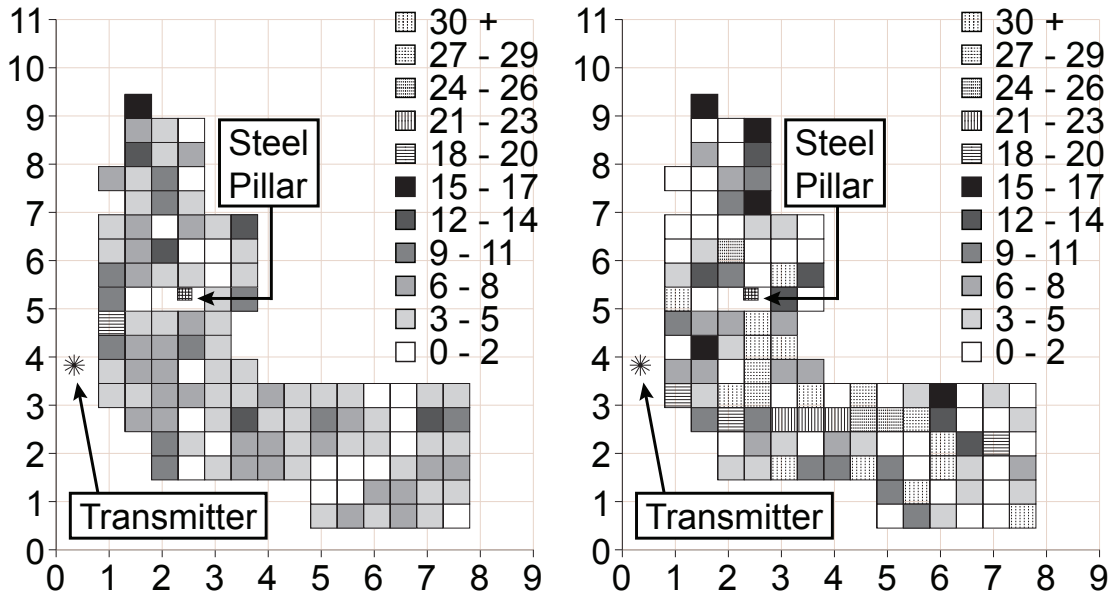


Figure 6.4. Average RSSI at power level 0x03.

exhibiting multipath effects, low-power transmissions are more likely to be garbled than at higher power.

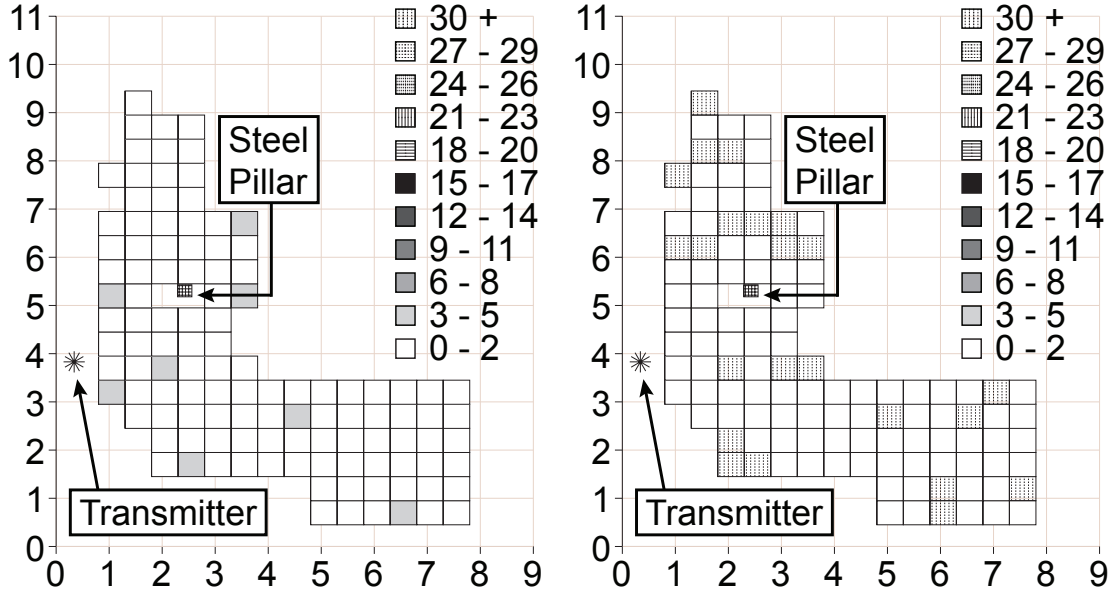
An interesting and unexpected result is that RSSI value ranges are very consistent across multiple runs across an area. This further demonstrates the lack of association of packet reception with RSSI, and perhaps indicates the strong presence of disruptive multipath effects in our environment. We would expect RSSI to be more uniform with multipath effects; however, they may cause more bit errors in packet transmission. Figures 6.6(a) and 6.6(b) show observed average RSSI ranges across power levels 0xff and 0x03, respectively.

Although we did not show plots for all runs, we also experimented with a second transmitter mote placed at approximately X=1.0,Y=3.5. Tables 6.1 and 6.2 show statistics for packet reception and average RSSI maximum ranges for each set of runs at various power levels and with multiple fixed transmitters.



(a) Power level 0xff. (b) Power level 0x03.

Figure 6.5. Packet reception ranges at two power levels.



(a) Power level 0xff. (b) Power level 0x03.

Figure 6.6. Average RSSI ranges at two power levels.

Table 6.1. Packet reception range statistics.

Transmitter	Power Level	min	max	mean	stddev	variance
mote104	0xff	1	19	5.461	3.358	11.275
mote104	0x0a	0	95	14.304	16.863	284.351
mote104	0x08	0	66	9.487	10.412	108.406
mote104	0x03	0	71	11.252	13.714	188.084
mote102	0xff	0	73	8.609	8.854	78.395
mote102	0x08	0	62	11.374	12.673	160.599
mote102	0x03	0	64	4.391	10.131	102.638

Table 6.2. Average RSSI range statistics.

Transmitter	Power Level	min	max	mean	stddev	variance
mote104	0xff	0	5	1.000	0.978	0.957
mote104	0x0a	0	62	2.574	9.730	94.679
mote104	0x08	0	63	2.765	11.222	125.936
mote104	0x03	0	65	12.878	24.896	619.829
mote102	0xff	0	3	0.704	0.710	0.504
mote102	0x08	0	65	10.200	22.542	508.143
mote102	0x03	0	64	3.991	14.995	224.843

6.2 Mobility-enhanced Sensor Network Quality

Certain problems arise in static deployments of wireless sensor networks that can potentially be improved by introducing mobile sensor nodes into the network. This section describes `MobiQ`, an application we designed that uses mobile devices as a means of improving the quality of sensor networks. `MobiQ` employs *mobile data sinks* to collect sensed network data, instead of the traditional approach in which a single, static base station is used. In the `MobiQ` design, mobile sinks may not only collect data, but could also *increase the overall quality* of the sensor network by strategically alleviating common sensor network concerns of connectivity, congestion, and sensed data quality. However, the implementation does not yet fully realize the design, and focuses primarily on improving sensor network lifetime by amortizing sensor data tranmission costs across all the nodes in the network.

We present the design of the application and evaluate it in Mobile Emulab. We also discuss how the development of the application drove codevelopment of tools for sensor testbeds, and highlight shortcomings of Mobile Emulab and present potential solutions.

6.2.1 The Rationale for Mobility

The "quality" of a sensor network can be established by several metrics, and is likely to be determined by the deployment goal for any specific network. For instance, the deployment may prioritize one or more of data quality, data timeliness, deployment coverage, and network lifetime (alternatively, power usage) as the overall goal; it could then adjust goals based on sensed data, very temporarily or longer-term. These different goals are very much related to each other. For instance, maximizing lifetime will almost certainly require minimal sensing and collection rates. If nodes in the network die (especially bottleneck nodes in a tree-based routing protocol, where a "funnel" effect results in nodes closest to the base station expending more power more quickly than nodes further from it), the overall coverage and sensing capability of the network will be degraded.

The ideas in `MobiQ` spring from the bottleneck effect—the primary goal is to amortize the energy cost of forwarding data messages to a base station across static nodes in the network (in other words, moving the bottlenecks around periodically over the lifetime of the network). However, since `MobiQ` postulates one or more mobile sinks as the solution, it then becomes natural to use them to enhance the quality of the network in other ways as well. For instance, a mobile sensor could move to hotspots if the static nodes implied that one existed, and investigate further. Alternatively, they could move to poorly covered areas at certain times, or supplement failed nodes. If all mobile nodes function as *sinks*, static sensors may report to those sinks, which without loss of generality may report to a single base station. However, by making base station location dynamic, we may be able to extend lifetime and increase quality.

These goals cannot be fully satisfied at the same time. For instance, a mobile node cannot be acting as a relay, alleviating congestion, and then also try to physically move and discover why a sensor failure happened. Consequently, we need an *impact estimator* to determine what motions are effective for each mobile sink. The goal of such an estimator is to maximize quality of the network. Our goal is to build a framework in which mobility can aid in solving the quality problem based on a prioritization of the goals discussed above.

6.2.2 Design and Implementation

`MobiQ` is implemented in `NesC` [12], a C-like language that supports building applications in an event-based programming paradigm, by wiring together well-defined

component interfaces. It is built to work within the TinyOS 1.x sensor network operating system [19] environment, which supports several well-known sensor network devices and provides a number of useful libraries and interfaces.

`MobiQ` itself is composed of several primary components: a routing protocol (based on our own mobile-aware extensions to the MintRoute protocol [31]), interfaces and modules for obtaining location updates and commanding motion to occur, and a congestion-based estimator that determines where to move mobile base stations. Other code generates emulated sensor data messages into the network.

Developing a hybrid routing protocol that better suits a sensor deployment composed of both static and mobile devices is a worthwhile goal, but its complexity places it beyond the scope of this thesis. (For instance, one might imagine a dynamic tree-based model, since the goal is still to route sensed data to base stations—but since they are mobile, the static nodes should know this and adjust their routes with intelligence with respect to base station motion, taking advantage of temporarily close mobile nodes, but always ready to fall back to a good, known static route. Such an approach must avoid the cost of reassembling the tree every time a mobile base station changes location.)

Instead, we extended the MintRoute reliable multihop routing protocol, with extensions to support multiple mobile base stations. MintRoute builds an inverted tree-like forwarding structure terminating at the base station, based on link quality estimates. With the `MobiQ` extensions, mobile nodes broadcast while moving, and fixed nodes increase route calculation and the rate at which they notify neighbors of new routes. Thus, the tree can more quickly adjust during and after times of movement. However, these modifications are simplistic and unlikely to function nearly as well as a new design that supports mobility from the beginning.

We implemented a simple estimator that periodically moves a mobile base station to areas in the network that appear congested based on known routing statistics. The estimator supports multiple base stations, so there will likely be multiple inverted routing trees, one rooted at each base station. Fixed motes are free to route to any base station—we assume that base stations are powerful, and could all communicate with a gateway device to upload or aggregate data. Each mobile base station is restricted to a distinct area in which only it can move. This will likely prove difficult to enforce in a real deployment, but it could be approximated by dividing the deployment area into spheres of equal radius, and loosely restricting each base station's movement within its own sphere.

Since this estimator attempts to equalize forwarding cost across all fixed nodes in the network over its full lifetime, it must track the amount of messages that pass through each mote. We maintain these totals at the base stations, and thus, to support multiple mobile base stations, the estimator in the base station must handle fixed mote handoff from one base station to another by migrating statistics. When a child moves to another sink, the new sink will query all others to see who has the current total. If none reply, it is a new child. If one replies, that total is added to whatever the new parent has heard.

Unlike the sampling application described in Section 6.1, base station motion in `MobiQ` is conditional on the behavior of the application itself. Thus, it is natural to integrate motion requests and status notifications *into* the application running on the mote itself, even if the actual motion control is implemented away from the mote (as in Mobile Emulab, where it is implemented in *robotd* and *pilot*). We wrote a TinyOS component that communicates motion requests from robot-mounted motes for their robots to Emulab via the `placeholder` service described in Section 5.4. This component also supports location change updates, which are propagated from *embroker* through the Emulab event system to `placeholder` and `SNAP-M`, to which the motes are connected.

Finally, we developed a simple component that emulates a location service. Applications that use this component can get and set locations for mote IDs for which the component has been given locations. This can easily happen from `SNAP-M` over mote serial lines in Mobile Emulab.

6.2.3 Assumptions

In `MobiQ`, we make the following assumptions about the nature of mobile and fixed devices and their knowledge of their environment. Here, we comment briefly on their feasibility in a real deployment.

Location knowledge. All nodes know their location. This assumption is reasonable due to the large number of localization services for sensor networks. All mobile sinks can obtain the location of any node in the network. In a real deployment, this would probably be obtained once after the localization algorithm was run and the routing protocol had been given time to initialize.

Location stability. We assume for evaluation purposes that after initial deployment, only mobile node locations change. This seems likely to be true in many real-world sensor deployments.

Powerful mobile sinks. Mobile sinks have essentially infinitely more power than static motes. Even if the mobile platform carrying the mote cannot carry "infinitely" more energy capacity, one might imagine deploying charging substations. At any rate, if power is more limited, it is more important for the motion estimator to quickly direct the robots to survey the deployment area, and pre-select a few target locations to move to during network lifetime.

Connectivity between mobile sinks. All mobile sinks can communicate directly with each other. Powerful mobile sinks could certainly have more powerful radios (or use the same radios with which they upload data out of the network location). In a large deployment with many mobile base stations, however, this will not suffice—a better solution could be to modify the estimator to query other base stations during child handoff using a distinct multihop routing topology used by sinks only.

6.2.4 Evaluation

We compared the number of routing messages required for each nonbase station mote in the network in two cases: a single, static base station, and a case with two mobile base stations; this is shown in Figure 6.7. In each case, the static motes sent sensor data messages containing fake data every 5 seconds. Both runs lasted for approximately 45 minutes, although the mobile run lasted for 90 seconds longer. Each mote dumped periodic routing statistics out of its serial port, which we collected and processed offline to analyze.

It is immediately obvious that `MobiQ` with mobile base stations actually increased per-mote message overhead. However, this is to be expected—our test runs examined how well the simple estimator could target areas of congestion, and was allowed to move the base station as much as once every minute. Such a motion rate is perhaps an order of magnitude higher than necessary during a real deployment. Moreover, since our extended protocol must operate during base station motion, and adjust the routing topology to account for changing locations, it requires more routing messages to be exchanged. Thus, we are more interested to see if in the mobile case, this estimator distributed the message load across more motes than in the single base station case. Unfortunately, it proved less effective than hoped for—Figure 6.7 shows hints of wider distribution, especially for motes 116, 117, and 122, but a clear trend cannot be discerned. We would expect our simple radius-based method of moving to a location "close" to congested motes would

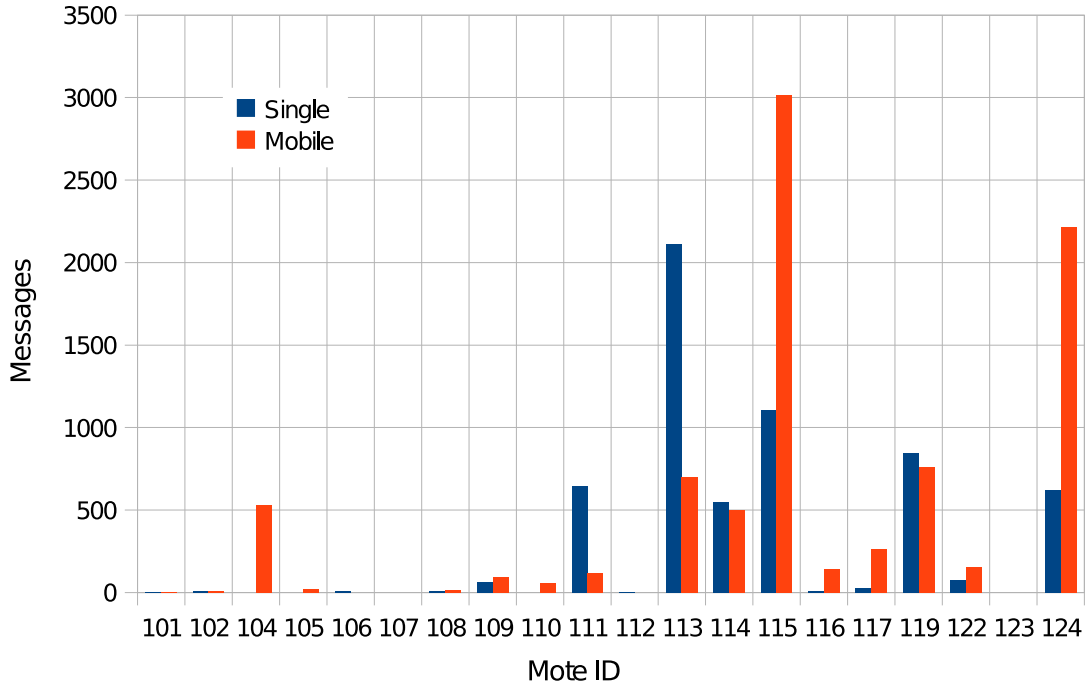


Figure 6.7. Routing messages sent by each mote.

function much better in an outdoor environment, but may not work nearly as well in the small indoor Mobile Emulab deployment.

6.2.5 Lessons for Mobile Testbeds

Many parts of `SNAP-M` were specifically codeveloped with `MobiQ`, or to support it in some way. It is natural, then, that we found them extremely useful as a tool to understand `MobiQ`'s behavior—we could visualize motion, routing topology changes, see live routing protocol statistics dumped from each mote's serial interface, and change parameters of some components via messages sent to mote serial interfaces. `MobiQ` required significant preruntime configuration after Emulab had flashed the motes with the application binary; for instance, the mobile motes had to be informed of the locations of all fixed motes, configured with their motion bounds parameters, and other things. Since `MobiQ` relies on TinyOS Active Messages to perform these configuration tasks, as well as subsequent motion requests and status notifications, `SNAP-M`'s ability to create messages based on user input for message fields and send them to a subset of motes under its control, as well as to script this behavior, was extremely useful. Initializing a mobile application was

then simply a matter of running a script from `SNAP-M`.

By interfacing Mobile Emulab's location and motion control services to `MobiQ` through `placeholder` and the TinyOS location and motion components discussed above, it was possible to build an entire mobile application inside the motes. Some mobile sensor applications might not be built this way, and could involve other, more powerful computers, but it is nevertheless valuable to expose *visiond*'s ground truth location data to the sensor network. A useful feature we did not implement in the location modules (nor in `placeholder`) is the ability to alter the ground truth location estimates based on error models; in real deployments, localization services will almost certainly produce more error than *visiond*.

Finally, we believe that Mobile Emulab needs a larger environment with more opportunities for uniformity (i.e., sections of the environment should be more open in an omnidirectional sense). We were able to successfully build multihop topologies and test applications that required them, but it would have been useful to analyze how `MobiQ`'s simple estimator behaved in a setting more similar to an open, outdoor field, in addition to the Mobile Emulab deployment.

6.3 Heterogeneous Sensor Network Experimentation

An important part of demonstrating the utility of the mobile testbed is to show the ease with which mobile and fixed motes can be integrated into our testbed. I have constructed a large, heterogeneous emulation experiment that utilizes the original Emulab core as well as the mobile and sensor network extensions. In this section, I discuss this emulation in more detail and provide a subjective analysis that evaluates the ease with which mobility and sensor network devices may be integrated with other types of network devices.

6.3.1 Scenario

To demonstrate how Emulab enables experimentation with many diverse node types and software, we created an experiment to emulate the following scenario. Companies that require large amounts of compute power or maintain a large Internet presence typically manage their own data centers, often at several remote sites. Temperature monitoring and automatic response to heightened temperatures, such as partial or complete cluster shutdown, is a necessity.

We have constructed an emulation of this scenario in Emulab. The included NS file creates several cluster server networks, emulating remote data or service centers, and several diverse client networks that access the service clusters. In the default configuration, there are two service clusters. The first cluster contains 3 PCs, where each PC is actually an Emulab virtual node, running in a FreeBSD jail on a single physical PC. The second consists of several physical PCs running FreeBSD or Linux. Each cluster node runs an `iperf` [27] server that listens for incoming connections and acts as a data sink.

There are several client networks whose members all attempt to access the service cluster nodes. Each node runs an iperf source that sends tcp data as quickly as possible to a server chosen randomly in any of the clusters. First, there is a single cable network, emulating cable modem clients. All clients in the network are in a single LAN with 100Mb bandwidth and 90ms latency. Connected to the first cable modem node is a small SDR network. A single SDR node acts as an access point for two more SDR nodes, which are connected to a single wired PC. In this way, they emulate "wireless backhauls" or even neighborhood wireless access points, providing access to another wired PC network. The cable LAN nodes run a mix of FreeBSD, Linux, and Windows XP.

Second, there is a DSL network composed of multiplexed links (each client in the network contacts a router over an unshared link; thus, only upstream bandwidth is shared). Each link is rated at 6Mb bandwidth and 25ms latency, and the upstream connection from the router is 100Mb and 0ms. Connected to the first DSL node is an 802.11g wireless access point. Several other wireless clients are connected to this access point, emulating a neighborhood wireless LAN sharing a single DSL connection.

Finally, to emulate temperature monitoring in the several service clusters (the "data centers"), we employ static sensor motes and mobile sensor motes mounted on robots. Several static motes and a single robot are associated with each service cluster. Each forward the data they sense to the cluster manager. If the cluster manager detects temperatures exceeding a preset threshold from both a static mote and a robot associated with a single cluster, it shuts down the cluster (in the emulation, it shuts down the iperf sinks on each server in the cluster). All client nodes previously connected to this cluster are disconnected, and then connect to new cluster servers in another cluster.

Once the temperatures go back down (again, with robot confirmation if desired), the iperf servers that were killed are restarted, and the clients are restarted to emulate a load-balanced cluster.

Figure 6.8 shows a visualization of this heterogeneous network, produced by Emulab. Motes, robots, and PlanetLab nodes remain free-floating in the visualization since they do not have wired links to the experimental, emulated networks.

6.3.2 Analysis

This experiment demonstrates the powerful opportunity that Mobile Emulab provides to experimenters. Robots, motes, and wired and wireless PCs can be combined in the same experiment under the common Emulab interface, even though methods of user interaction with specific devices may differ. Although the *ns* file that generates this experiment is complex, it is only complex because it is a programmed experiment designed to be parameterized across a large number of variables. Normally, integrating sensor network motes and robots with other Emulab device types is easy because motes and robots are simply different node types, with different software requirements. As small, embedded sensor devices come into more widespread use, it may benefit researchers to already have a testbed that provides a framework for heterogeneous network research and experimentation.

Since sensor devices are still a developing technology, it may not be useful to add Mobile Emulab support to preconfigure a group of motes into a network topology in the same way Emulab automatically configures wired PCs into IPv4 networks. Applications may be a single custom binary provided by the user containing custom protocols and network software that Emulab cannot configure automatically, or may utilize new addressing schemes.

Finally, as we ran this experiment, we experimented with a real heat source to trigger alarms based on the amount of heat detected. We had primarily focused on the network aspects of Mobile Emulab, since we realized the difficulty in providing real data sources that the sensors could detect that were also controllable by experimenters. It is difficult to solve this problem with real emulation in a testbed without building devices that are essentially opposites of common sensor boards (i.e., can generate heat, light, sound, and other events), and colocating them with sensors. However, one solution is to develop TinyOS components that function as parameterizeable data sources, atop which a user could model data events to drive the application. Clearly, this is a capability that Mobile Emulab needs.

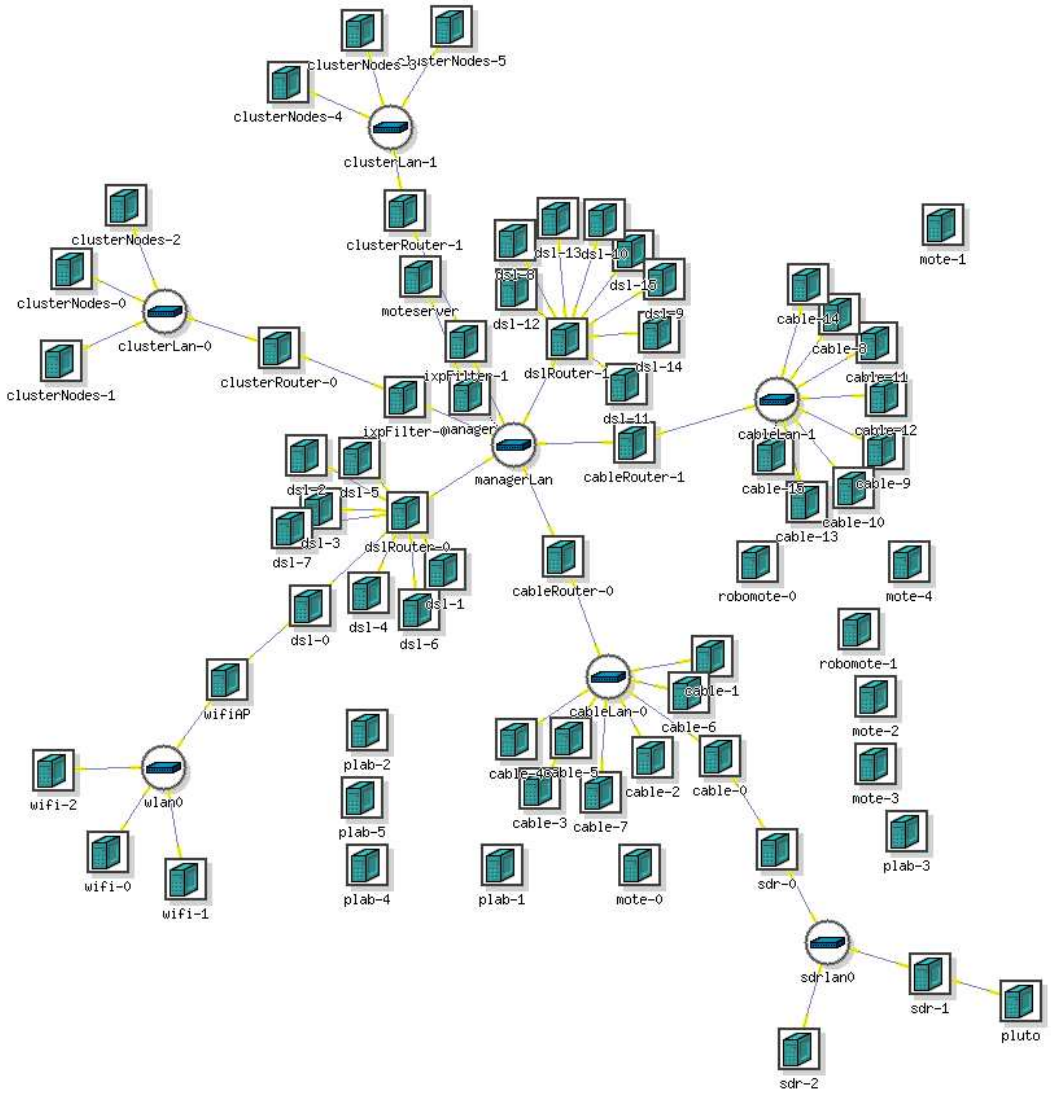


Figure 6.8. Visualization of the heterogeneous topology generated by Emulab.

CHAPTER 7

CONCLUSION

In this document, I have described the design and implementation of Mobile Emulab, a mobile sensor network testbed. Mobile Emulab employs precise localization techniques and dynamic motion control to create an environment for real mobile sensor network experimentation. The testbed provides accessible and precise motion to experimenters, yet is cost-effective to implement and duplicate. Our experiments demonstrate the precision of the location system, yet reveal areas that will require improvement to support larger testbeds and more advanced motion controllers. The case studies we presented establish the power of Mobile Emulab in the context of three very different experiments, and we believe our experiences could be valuable to other implementors of new mobile testbeds. By providing a large body of tools supporting experimentation in Mobile Emulab, we have greatly eased the processes of mote control, application development and debugging, and data storage and inspection.

In this chapter, I discuss the status of our testbed within the research community and describe a large body of potential future work.

7.1 Users

Despite aggressive advertisement in the sensor network community, Mobile Emulab has no real external users. I personally presented and demonstrated this system at five conferences, and talked to many potential users, some of whom seemed quite keen to begin using the system. Furthermore, our testbed extends Emulab and thereby presents a familiar user interface to the network research community. However, no external users have used Mobile Emulab to conduct experiments, although a few research groups have applied for accounts. There are a number of possible reasons why this has occurred, and I discuss several of them below.

First, the initial deployment was constrained to a small area and had only a few mobile nodes. While some mobile sensor network research may only require several mobile nodes,

many fixed nodes are generally assumed to exist. More importantly, there is not a large body of mobile sensor network research. If our testbed supplied 802.11a/b/g wireless devices, or software-defined radios, we would immediately have a much larger pool of researchers from which to draw. Due to technical limitations, this is impossible with our current robotic platform; moreover, a mobile 802.11 testbed would require a *significantly* larger area of motion.

As with any testbed, Mobile Emulab presents users with a learning curve. To an overworked researcher (the status quo), learning how to use a new system for experimentation may seem impractical. Mobile Emulab is different than *ns* mobility simulation in many ways, for instance. It is simply beyond the scope of this work to implement an interface by which users can easily run existing simulations on our real hardware.

Wireless simulation appears to continue to remain a respectable alternative to real experimentation. Judging from publishing trends at major conferences, and my own knowledge, researchers acknowledge that simulation modeling inaccuracies represent a real problem. However, it is also possible that there is too much "simulation inertia" in the mobile and wireless research community that prevents a speedy inclusion of real evaluation. There are doubtlessly many people who still believe simulation is the appropriate and preferred method for evaluation, and that modeling capability can be increased to match real circumstances. Finally, even if researchers recognize abstractly that simulation is bad, if the community at large does not act upon the knowledge, simulation-only publications will continue to persist for some time.

7.2 Analysis of Component Modularity

One of the minor design goals in creation of this testbed is easy adoption and adaptation to new hardware by prospective testbed owners, and replacement of key components by other software. Due to unexpected, hardware-specific requirements on some of the modules, satisfying these goals was not always possible. However, some components may be replaced by others providing similar functionality, and others may be used with different hardware types. In this section, we analyze code modularity and discuss component requirements and overall system adaptability to both hardware and software changes.

7.2.1 *mtp*

This application protocol is used for all communication between the components in our testbed. If any component is modified so that it has a communication need (i.e., more or new data, different configuration) different than currently supported, the protocol will also require modification. A more abstract protocol would be required to further reduce the number of modifications.

7.2.2 *robotd* and *pilot*

robotd implements several different motion controllers, some of which are strongly linked to our chosen robot platform. The basic, waypoint motion model controller is simple and makes use of basic robotic motion (straight line motion and pivots) that COTS robots may already support, as does our. The more advanced controllers depend on the kinematic properties of the robot. Theoretically, they could be used with other robotics platforms without modification, although the gains in the control state equations would likely require modification. The software container does not provide an easy means to add and trigger new controllers, although it would not be difficult to add. However, the advanced controllers are completely dependent on a robot that can execute a two-wheeled pivot, and would not work with a robot that cannot.

pilot is highly dependent on the C++ libraries provided by our robot manufacturer. Our current set of controller functionality requires support by these libraries to perform basic manuevers, such as straight line motion parameterized by length and speed and pivot parameterized by arc angle. The high-level controllers require an interface to set the drive wheel speeds as well. We could easily reduce this dependency by adding a high-level layer exposing this functionality that could call into submodules that implement the high-level functionality with platfom-specific methods.

7.2.3 *visiond* and *vmc-client*

Our localization system was specifically designed to permit a range of hardware so that other implementors could obtain a level of precision best fitting their needs. The only specific hardware requirement is the use of color video cameras, since Mezzanine depends on color to recognize fiducials. Of course, the use of video cameras similar to those we used will likely result in similar performance.

visiond implements high-level algorithms for object tracking and identification. Its purposes are to aggregate object locations, remove duplicates, and smooth data as nec-

essary. Although the architecture does not specifically support smoothing modules, it is quite easy to add them. For instance, the addition of a basic Kalman filter would be quite easy. An advanced Kalman filter that used additional inputs, such as current wheelspeeds of each robot, would require only additional *mtp* modifications. Different aggregation mechanisms would be slightly more difficult to integrate.

7.2.4 *embroker*

The least modular component in our mobile control subsystem is *embroker*, since its purpose is to connect all other key mobile subcomponents to Emulab facilities. Thus, if implementing this testbed on an Emulabsystem, no modifications should be required. However, if not, *embroker* would require a completely new implementation. For instance, one could envision a simple implementation that communicates with the robot subcomponents in the same manner as the current implementation, but provides a much simpler motion command and monitoring interface.

7.3 Future Work

I now turn to a discussion of several areas in which the mobile testbed should be extended and improved. There are many different areas in which increased functionality can benefit, including localization, mobile control, and mote application management.

7.3.1 Localization

There remain two primary outstanding concerns in our current localization system. First, the problems previously discussed concerning jitter in data caused primarily by inexpensive hardware, combined with the low rate of data possible to extract from such hardware, can seriously inhibit the performance of advanced kinematic motion controllers. By using more expensive hardware that produces higher data rates, such as cameras rated at 60 or 120 fps, we could possibly alleviate these concerns. However, since variable lighting is present in our current environment, and perhaps any subsequent environment we could foresee utilitizing, we would still need to somehow smooth the data.

One option to perform smoothing operations that reduce jitter while keeping phase lag low is the Extended Kalman Filter. While such a customized filter is beyond the scope of this work, the predictive elements provided by the filter would greatly enhance possibilities for smoothing. However, from discussions with members of the Mechanical Engineering

Department, jitter being produced by light variability or low-quality hardware may be difficult to resolve well with an Extended Kalman Filter.

The other primary concern that remains with our current localization system is setup cost. Significant time must be spent in manual calibration in order to achieve the levels of precision described in this document. To make it easier for outside entities to create their own testbeds, we would probably want to supply a simplified calibration procedure.

7.3.2 Mobile Control

To ease the difficulty of the remote motion control experience for remote researchers, my colleague Leigh Stoller implemented an interactive, real-time Java motion control applet. Experimenters may drag and drop robots to new destinations on a map of the area, and watch as they move toward their final positions. The applet displays interaction with obstacles. Embedded webcam streams allow experimenters to visualize motion and inspect the environment. The applet also provides rudimentary boundary and obstacle checks to ensure that an experimenter does not maneuver the robot out of our localization system's coverage or into an obstacle.

Originally, this applet only supported a waypoint motion model. We had planned to extend the applet to provide continuous path generation from waypoints and user-constructed continuous motion paths. In this mode, the experimenter would choose a set of waypoints. The applet then constructs a continuous curve by trimming the ends of each line between waypoints, and inserting a circular arc between trimmed ends where a waypoint originally was. The length of the trim will be determined by the desired rate of curvature in the arg segments, but will have a minimum trim length to ensure that the robot can attain the necessary curvature while remaining on the path.

To allow experimenters to construct their own continuous motion curves, I prototyped a path editing mechanism similar to Bezier path editors in common vector drawing software, such as Adobe Illustrator or the open-source Inkscape. While the interface is similar, the underlying representation is slightly different. Our paths are interpolated 6th-order Bezier curves, with four control points in addition to the knots at the beginning and end of each segment. Adjacent segments share knots.

By interpolating at the knots and enforcing that the control point nearest to the knot be collinear with the appropriate middle control handle, we can enforce the C2 continuity constraint. A curve that is C2 continuous is continuously second-order differentiable, and thus provides us with the assurance that there will be *no discontinuities* in the acceleration

profile that the robot must maintain to remain on the path. In other words, the robot cannot sustain instantaneous changes in acceleration; the C2 continuity constraint allows the system to avoid them entirely.

When integration of this protoype into the current interactive motion control interface is complete, potential testbed users will gain faster motion execution times as well as guaranteed robot trajectories.

7.3.3 Mote Application Management

There are nearly endless opportunities to improve interfaces for controlling and managing sensor network applications. For instance, one could write testbed interface modules not only for Emulab, but also for other testbeds like Motelab. Users could run their applications from a common application, perhaps even running instances of an application on multiple testbeds at the same time.

Many of the modules I wrote for the sensor network application manager could be enhanced. A fully-featured message scripting language could provide valuable features such as waiting for messages or responses to sent messages from testbed motes. This could be generalized even further into stored subscripts called in response to an incoming message. In addition, one could allow modules to publish standardized interfaces that would then allow other modules to easily interoperate with them. For instance, the Location module I described earlier could be extended to provide an Overlay interface, so that users could display their own custom statistics on the Location map, associated with links and nodes as necessary. Features like these would greatly reduce the amount of time needed to develop interface applications.

Finally, better integration with existing tools is important. For instance, one could write a *ManagerModule* to support easy integration with Emstar [14], which could potentially increase scalability in evaluation for testbeds with fewer numbers of real motes. Users could assign real motes to more important evaluation points in the network, and use simulated Emstar motes to increase raw numbers of motes. Such a hybrid scheme could be of great value to sensor network experimenters.

REFERENCES

[1] Acroname, Inc. Garcia custom robot, Apr. 2009. http://www.acroname.com/garcia/garcia.html.

[2] Aguayo, D., Bicket, J., Biswas, S., Judd, G., and Morris, R. Link-level measurements from an 802.11b mesh network. In *Proceedings of SIGCOMM '04* (Portland, OR, Aug. 2004).

[3] Crossbow Corp. MPR400/410/420 MICA2 mote — Crossbow Technology, Apr. 2010. http://www.xbow.com/Products/productsdetails.aspx?sid=72.

[4] Crossbow Corp. MTS sensor boards – Crossbow Technology, Apr. 2010. http://www.xbow.com/Products/productsdetails.aspx?sid=177.

[5] Crossbow Corp. Stargate datasheet, Apr. 2010. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/Stargate_Datasheet.pdf.

[6] De, P., Raniwala, A., Sharma, S., and Chiueh, T. MiNT: A miniaturized network testbed for mobile wireless research. In *Proceedings of IEEE INFOCOM* (Mar. 2005).

[7] DGPS general information – USCG navigation center, Apr. 2010. http://www.navcenter.org/dgps.

[8] Flickinger, D. M. Motion planning and coordination of mobile robot behavior for medium scale distributed wireless network experiments. Master's thesis, University of Utah, Dec. 2007. 169 pages.

[9] Fonseca, J. remote-testbed – Re-Mote testbed framework, Jan. 2010. http://code.google.com/p/remote-testbed.

[10] Fyodor. Nmap — free security scanner for network exploration & security audits, Apr. 2010. http://insecure.org/nmap.

[11] Garcia, L. M. TCPDUMP/LIBPCAP public repository, Apr. 2010. http://www.tcpdump.org.

[12] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. The nesC language: A holistic approach to networked embedded systems. In *Proc. of the ACM SIGPLAN '03 Conf. on Programming Language Design and Implementation (PLDI)* (San Diego, CA, 2003), pp. 1–11.

[13] Giacobbi, G. The GNU Netcat — Official Homepage, Jan. 2004. http://netcat.sourceforge.net.

[14] Girod, L., Stathopoulos, T., Ramanathan, N., Elson, J., Estrin, D., Osterweil, E., and Schoellhammer, T. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *SenSys '04: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems* (Baltimore, MD, 2004), pp. 201–213.

[15] GPS general information – USCG navigation center, Apr. 2010. `http://www.navcenter.org/gps`.

[16] Handziski, V., Kopke, A., Willig, A., and Wolisz, A. Twist: A scalable and reconfigurable wireless sensor network testbed for indoor deployments. Tech. Rep. TKN-05-008, Telecommunication Networks Group, Technical University Berlin, Nov. 2005.

[17] Harter, A., Hopper, A., Steggles, P., Ward, A., and Webster, P. The anatomy of a context-aware application. In *Proceedings of 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking* (Seattle, WA, Aug. 1999), pp. 59–68.

[18] Heidemann, J., Bulusu, N., Elson, J., Intanagonwiwat, C., chan Lan, K., Xu, Y., Ye, W., Estrin, D., and Govindan, R. Effects of detail in wireless network simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation* (Phoenix, AZ, Jan. 2001), pp. 3–11.

[19] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. System architecture directions for networked sensors. In *Proc. of the Ninth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Cambridge, MA, Nov. 2000), pp. 93–104.

[20] Hitachi Kokusai Electric Inc. KP-D20A specifications, Jan. 2006. `http://www.hitachikokusai.com/supportingdocs/products/industrial_video_systems/interlace_scan/KPD20A.pdf`.

[21] Howard, A. Mezzanine: An overhead visual object tracker, June 2005. `http://playerstage.sourceforge.net/mezzanine/mezzanine.html`.

[22] Johnson, D., Stack, T., Fish, R., Flickinger, D. M., Stoller, L., Ricci, R., and Lepreau, J. Mobile Emulab: A robotic wireless and sensor network testbed. In *Proceedings of IEEE INFOCOM 2006* (Barcelona, Spain, Apr. 2006).

[23] Kang, H.-D., and Jo, K.-H. Self-localization of autonomous mobile robot from the multiple candidates of landmarks. *SPIE – The International Society for Optical Engineering 4902* (2002), 428–435.

[24] Meltzer, J., Gupta, R., Yang, M.-H., and Soatto, S. Simultaneous localization and mapping using multiple view feature descriptors. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Sendai, Japan, 2004), pp. 1550–1555.

[25] Priyantha, N. B., Miu, A. K., Balakrishnan, H., and Teller, S. The cricket compass for context-aware mobile applications. In *Proceedings of the Annual International Conference on Mobile Computing and Networking (MOBICOM)* (Rome, Italy, 2001), pp. 1–14.

[26] Raychaudhuri, D., Seskar, I., Ott, M., Ganu, S., Ramachandran, K., Kremo, H., Siracusa, R., Liu, H., and Singh, M. Overview of the ORBIT radio grid testbed for evaluation of next-generation wireless network protocols. In *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC)* (Mar. 2005).

[27] Richasse, N. Iperf, Mar. 2008. `http://iperf.sourceforge.net`.

[28] Takai, M., Martin, J., and Bagrodia, R. Effects of wireless physical layer modeling in mobile ad hoc networks. In *Proceedings of the 2nd ACM International Symposium on Mobile Ad Hoc Networking & Computing (MobiHoc '01)* (Long Beach, CA, Oct. 2001), pp. 87–94.

[29] Werner-Allen, G., Swieskowski, P., and Welsh, M. Motelab: a wireless sensor network testbed. In *Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks (IPSN'05), Special Track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS)* (Los Angeles, California, Apr. 2005).

[30] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation* (Boston, MA, Dec. 2002), pp. 255–270.

[31] Woo, A., Tong, T., and Culler, D. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys '03)* (Los Angeles, CA, Nov. 2003), pp. 14–27.

[32] Zhao, J., and Govindan, R. Understanding packet delivery performance in dense wireless sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys '03)* (Los Angeles, CA, Nov. 2003), pp. 1–13.

[33] Zhou, G., He, T., Krishnamurthy, S., and Stankovic, J. A. Impact of radio irregularity on wireless sensor networks. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services (MobiSys '04)* (Boston, MA, June 2004), pp. 125–138.

[34] Zhou, J., Ji, Z., and Bagrodia, R. Twine: A hybrid emulation testbed for wireless networks and applications. In *Proceedings of IEEE INFOCOM 2006* (Barcelona, Spain, Apr. 2006).