Design and Implementation of an Indoor Mobile Navigation System

by

Allen Ka Lun Miu

B.S., Electrical Engineering and Computer Science,
University of California at Berkeley (1999)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 2002

Author . . . .
Department of Electrical Engineering and Computer Science
January 18, 2002

Certified by . . . .
Hari Balakrishnan
Assistant Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Design and Implementation of an Indoor Mobile Navigation System

by

Allen Ka Lun Miu

Submitted to the Department of Electrical Engineering and Computer Science
on January 18, 2002, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

This thesis describes the design and implementation of CricketNav, an indoor mobile navigation system using the Cricket indoor location sensing infrastructure developed at the MIT Laboratory for Computer Science as part of Project Oxygen. CricketNav navigates users to the desired destination by displaying a navigation arrow on a map. Both the direction and the position of the navigation arrow are updated in real-time as CricketNav steers the users through a dynamically computed path. To support CricketNav, we developed a modular data processing architecture for the Cricket location system, and an API for accessing location information. We implemented a least-squares position estimation algorithm in Cricket and evaluated its performance. We also developed a rich and compact representation of spatial information, and an automated process to extract this information from architectural CAD floorplans.

Thesis Supervisor: Hari Balakrishnan
Title: Assistant Professor

Acknowledgments

I would like to deeply thank my advisor, Hari Balakrishnan, for being my personal navigator who always guided me back on track whenever I am lost. I also thank my mentors Victor Bahl and John Ankcorn for sharing their invaluable advice and experiences.

I thank Bodhi Priyantha for his devotion in refining the Cricket hardware to help make CricketNav possible. I thank the rest of the Cricket team (Dorothy Curtis, Ken Steele, Omar Aftab, and Steve Garland), Kalpak Kothhari, "Rafa" Nogueras, and Rui Fan for their feedback about the design and implementation of CricketServer and CricketNav. My gratitude also goes to Seth Teller, who showed me the way to tackle the map extraction problem.

I would like to acknowledge William Leiserson, who wrote the source code for manipulating UniGrafix files and floorplan feature recognition, and Philippe Cheng, who has contributed some of the ideas used in extracting accessible paths and space boundaries from CAD drawings.

My time spent working on this thesis would not have been nearly as fun without my supportive and playful officemates and labmates: Godfrey ("Whassup yo?"), Kyle ("I gotta play hockey."), Michel and Magda ("Hey, you know.", "Hey, you know what?"). I also thank my ex-officemates, Alex, who have shown me the ropes around MIT when I first arrived, and Dave, who always have something interesting to say about anything and everything.

I would like to thank my close friends—Alec Woo, Eugene Shih, Karen Zee, Norman Lee, Ronian Siew, Victor Wen—for their companionship and support.

I dedicate this thesis to my family and my beloved Jie for their unconditional love and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Imagine that it is your first time visiting the Sistine Chapel and you have a strong desire to see Michelangelo's landmark painting, the "Last Judgement." Although the Musei Vaticani is filled with numerous magnificent works of art, the museum is about to close and you would like to head directly to admire your favorite painting. After entering the museum, you arrive at the first gallery and eagerly look for signs that points you to the Last Judgement. No luck. You try to ask a museum attendant for help but she only gives a long sequence of directions so complicated that you cannot possibly hope to follow them. After wandering around for a while, you find a pamphlet that includes a map indicating the location of your favorite painting. Holding the desperately needed navigation aid in your hand, you feel hopeful and excited that you will soon see Michelangelo's masterpiece. But soon, those feelings turn into feelings of defeat and frustration. As you walk through different galleries, you have trouble locating yourself on the map; the landmarks surrounding you do not seem to correspond to those indicated on the map. Moreover, you discover that various galleries are closed for the day and the path you are currently taking will not take you to your eventual goal.

Scenarios such as the one described here happen whenever people try to find a particular place, person, or object in an unfamiliar or complex environment. Examples of this includes airports, conferences, exhibitions, corporate and school campuses, hospitals, museums, shopping malls, train stations, and urban areas. These examples motivate the need for a navigation compass that navigates users in real time using a sequence of evolving directions to the desired destination (see Figure 1-1). A navigation compass is not only useful for guiding users in unfamiliar places, but also has applications in navigating autonomous robots and guiding the visually or physically disabled people through safe paths in an environment.

Recently, a number of location sensing technologies have emerged to allow powerful handheld computing devices pinpoint the physical position [13, 3, 18] and orientation [19] of mobile devices with respect to a predefined coordinate system. The combination of location infrastructure and portable computation power provide a substrate for the implementation of a *digital navigation system*: assuming a suitable map representing the surrounding environment is available, the handheld device can dynamically compute a path between the device's current location and the desired destination, and display an arrow pointing the user towards the next intermediate point along the path.

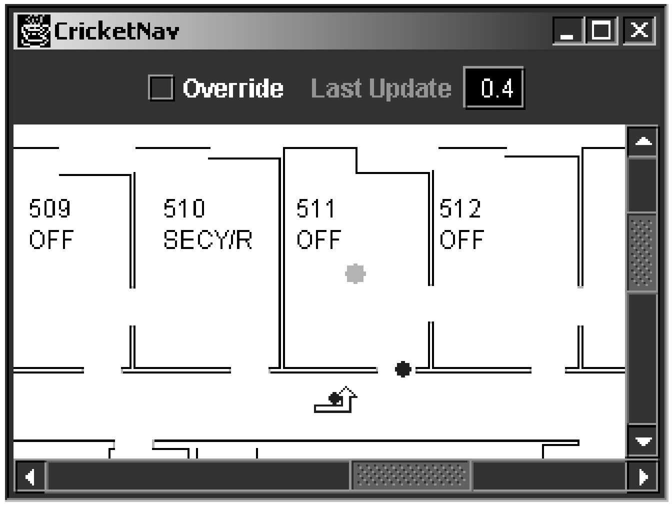


Figure 1-1: Screen shot of a navigation application. It shows an navigation arrow indicating the user's current position. The arrow indicates the direction that leads the user towards the destination.

Navigation systems based on GPS [10] exist today to help guide drivers and pedestrians in open outdoor environments where GPS works. However, no system exists for indoor mobile navigation in the way we envision. This thesis describes the design and implementation of CricketNav, a precise indoor navigation system using the Cricket indoor location sensing infrastructure developed at the MIT Laboratory for Computer Science as part of Project Oxygen.

The design requirements of indoor navigation systems are different from outdoor navigation systems in several ways. First, the coordinate system and the measurement precision supported by the indoor location infrastructure often differs from GPS. For supporting indoor navigation applications, this thesis argues that the detection of *space boundaries* by an indoor location system is more important than accurate coordinate estimates. Second, navigation within a building is different from navigation in the road system. Mobile users in a building do not have a set of well-defined pathways because they can roam about in any free space within the environment whereas cars are mostly restricted to travel only on the road network. Consequently, the techniques used for charting a path between any indoor origin and destination are different from those used in charting paths on the road network. Finally, digitized maps and Geographical Information System (GIS) [8] databases for the road network are generally available from various commercial and public sources whereas maps of buildings and campuses are not. Because the CAD floorplans of buildings and campus are not represented in a form that can be readily processed by indoor navigation systems, it is likely that the deployment of indoor navigation infrastructures will be conducted on a custom basis. Such a deployment model requires a set of tools to automate the process of creating indoor maps.

An indoor navigation system can be generalized to support a variety of location-aware applications. For example, location systems can use information about the physical structure of a floorplan to correct its position estimates. Similarly, applications that track a user's trajectories can use structural information to find the maximum likelihood trajectories taken by the user. Resource discovery applications may augment the navigation map

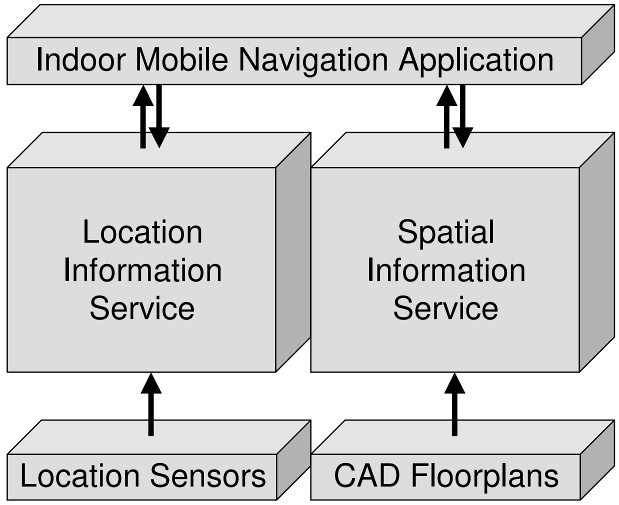


Figure 1-2: Components required for CricketNav.

to display the locations of nearby resources such as printers and other devices of interest. More importantly, the same resource discovery applications may use the navigation system to locate the positions of various resources and calculate the path length cost relative to the user's current position. This allows a user to find, for example, the nearest uncongested printer to the user's current position.

1.2 Design Overview

CricketNav is a mobile navigation application that runs on a handheld computing device connected to a wireless network. CricketNav interacts with two supporting systems to obtain the necessary information for navigation (Figure 1-2). The Spatial Information Service provides maps of the physical structure and the spatial layout of the indoor environment. CricketNav uses this information to perform path planning, to determine the direction of the navigation arrow with respect to the user's current position, and to render a graphical representation of the physical surrounding on the handheld display. The Location Information Service periodically updates CricketNav with the user's current position. CricketNav uses the position updates to anchor the starting point of a path leading to the destination, to prefetch the relevant maps from the spatial information service, to update the current position and direction of the navigation arrow, and to determine when to recalculate a new path for a user who has wandered off the original course.

We use the Cricket location-support system to supply location updates to the application [18, 19]. The details of the original Cricket system, extensions made to Cricket to support CricketNav, and an evaluation of Cricket's positioning performance are described in Chapter 2. For the Spatial Information Service, we convert architectural CAD floorplans into a data representation that can be conveniently used for both path planning and rendering. The process used for automatic data conversion, and the data structures used for representing spatial information are described in Chapter 3. Chapter 4 describes the design and implementation of CricketNav and Chapter 5 concludes this thesis with a discussion of deployment issues and the current status of CricketNav.

1.3 Contributions

This thesis makes three main contributions

- Cricket: A modular data processing architecture for the Cricket location system, and an extensible API for accessing Cricket location information. Implementation and evaluation of a position estimation algorithm for Cricket.
- An automated process of converting architectural CAD floorplans into a spatial data representation that is useful for a variety of location-aware applications.
- An interactive indoor navigation application that adapts to the quality of information supplied by the location and spatial information systems.

Chapter 2

The Cricket Location Infrastructure

CricketNav requires periodic updates from a location information service to determine the user's current position, which is then used as the anchor point for selecting an appropriate path and arrow direction for the user to follow. In this chapter, we discuss the location infrastructure requirements for our indoor navigation system and briefly describe some possible location sensing technologies. We then describe the Cricket location system and the software extensions made to it for CricketNav. We conclude this chapter with an experimental evaluation of Cricket's positioning performance.

2.1 Requirements

There are three requirements for an indoor location support system. The first concerns the types of location information offered by the system. The second concerns its performance in terms of accuracy and latency. The third concerns the practicality of deployment and maintenance of these systems.

2.1.1 Space Location

In practice, it is not sufficient to use only coordinate information for navigation. Although there is enough information to match a reported coordinate to a location point in a map, the uncertainty of coordinate estimates sometimes crosses the boundary between two spaces. The error in the reported coordinate may cause the location point to be mapped into a different space, which in turn causes the system to give a navigation direction at the wrong anchor position (see Figure 2-1). The error in the resulting navigation direction is often large because indoor boundaries fundamentally define the shape and the direction of navigation pathways.

Our experience with CricketNav suggests one way of greatly reducing, and sometimes eliminating, this source of error. We rely on the location infrastructure to report the current *space* in which the user is located. A space is an area defined by some physical or virtual boundary in a floorplan such as a room, a lobby, a section of a large lecture hall, etc. CricketNav uses the reported space information to help keep the location point within the boundary of the current space in which the user is located. For example, a location system may report that the user is inside the office 640A to disambiguate the two possible

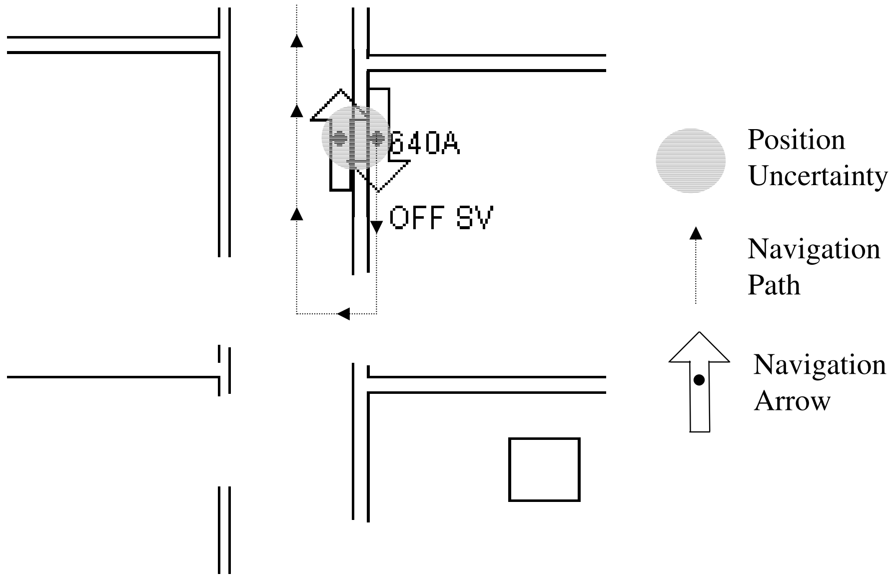


Figure 2-1: A person carrying a CricketNav device is in the office 640A so the correct navigation arrow should point downwards. However, small errors in reported coordinate value may cause the navigation arrow to anchor outside the room and point in the wrong direction.

navigation arrows shown in Figure 2-1.

2.1.2 Accuracy and Latency

This section discusses the accuracy and latency requirements of a location system to support CricketNav. For the purpose our discussion, we define *location accuracy* to be the size of the uncertainty region as illustrated in Figure 2-1. For simplicity, we model the uncertainty region as a circle, with the center representing the mean reported (x, y) position[1] under a normal distribution of independent samples. The radius of the region is set to two standard deviations of the linear distance error from the mean location. Thus the true location should lie within the uncertainty region in 95% of the time. The *update latency* is simply defined to be the delay between successive location updates.

CricketNav uses location information to mark the user's current position on a floor map. Thus, the location system needs to be accurate enough so that CricketNav does not mark the user's current position in the wrong space. As discussed in the previous section, CricketNav uses space information to reduce this type of error. Hence, it is essential that the location system reports space locations with high accuracy. In particular, the location system should accurately detect opaque space boundaries defined by objects such as walls. However, the reported space do not have to be strictly accurate at open space boundaries, such as those located at a doorway. Whether the reported location lies just inside or outside the open boundary, the navigation direction will be similar.

The reported position should also be accurate enough so that it roughly corresponds to the relative positions in the real setting. For example, if a user is standing to the right of one door, it would be confusing to mark the user's position indicating that he/she is standing

[1]For our application, we ignore the z component of the reported location.

to the left of it. In this case, a reasonable position accuracy—in terms of the uncertainty radius—should be between 30 to 50 *cm*, which is half the width of a typical door.

The final performance requirement is the update latency, which is the delay between successive location updates. Ideally, the update rate should be no less than once in one to two seconds, and the update rate should remain fixed whether the user is standing still or walking. The latter requirement implies that the location system is capable of tracking a user *continuously* while she is in motion. Some location system tracks a user *discontinuously*, which means that a location update is given only when the user is stationary. Continuous tracking is very difficult to implement for most location systems because these location systems often requires collecting multiple location measurements from the same location in order to achieve a reasonable accuracy. Therefore, we drop the continuous tracking requirement and assume that the user remains stationary for about five seconds whenever she wants a direction update from CricketNav.

2.1.3 Practicalities

There are a number of practical requirements for deploying an indoor navigation system. First, the indoor environment contains many objects such as computers, monitors, metals, and people that interfere with radio signals and magnetic sensors. Hence, a practical location sensing system should operate robustly in the presence of such interfering sources. Second, the location sensors of mobile devices must have a compact form factor such that it can easily be integrated with small handheld devices. Third, the location system must be scalable with respect to the number of users currently using the system. Finally, the location sensing infrastructure must minimize deployment and maintenance costs. It is undesirable to require excessive manual configuration, wiring, and/or frequent battery changes to deploy and maintain the system.

2.2 Existing Systems

Over the past few years, a number of location sensing technologies have emerged to provide location information needed for navigation. Below is a brief description of some of these systems.

2.2.1 GPS

The Global Positioning System uses a network of time-synchronized satellites that periodically broadcast positioning signals [10]. A mobile GPS sensor receives the signals from four or more satellites simultaneously over orthogonal radio frequency (RF) channels. The positioning signals are coded such that the receiver can infer the offset time between each pair of synchronized signals. Moreover, the orbits of each satellite can be predicted; hence, their positions can be calculated at any given time. From the known position of satellites and three or more offset time values, one can calculate the GPS sensor's current position by trilateration.

Some specialized GPS receivers have integrated inertial sensors to provide continuous position tracking of mobile objects between GPS positioning updates, and offer positioning accuracies to within a few meters in an open outdoor environment. Unfortunately, GPS does not work well indoors as satellite signals are not strong enough to penetrate building walls, or in many urban areas.

2.2.2 HiBall Tracker

The HiBall Tracker uses synchronized infrared LEDs and precision optics to determine the user's position with sub-millimeter accuracy with less than a millisecond latency [23]. It consists of deploying large arrays of hundreds to thousands of LED beacons on ceiling tiles and a sophisticated sensor "ball" that is made up of six tiny infrared sensors and optical lenses. Position and orientation estimations are obtained by sighting the relative angles and positions of the ceiling LEDs. Both the sensor and the LED arrays are centrally synchronized by a computer to control the LED intensity and lighting patterns required to determine the user's position. The system requires extensive wiring, which makes it expensive and difficult to deploy.

2.2.3 RADAR

RADAR uses a standard 802.11 network adapter to measure signal strength values with respect to the base stations that are within range of the network adapter. Because radio signals are susceptible to multi-path and time-varying interference, it is difficult to find a suitable radio propagation model to estimate the user's current position. Instead, RADAR proposes the construction of a database that records the signal strength pattern of every location of interest. A system administrator does this manually during system installation. At run time, a server compares the signal strength pattern measured by a receiver and tries to infer its location by finding the closest matching pattern in the database. A drawback of this location approach is that the signal pattern that is recorded statically in the database may greatly differ from the values measured in the dynamic environment, where the signal quality varies with time and the noise level in the environment. Thus, RADAR may not operate robustly in a highly dynamic indoor environment.

2.2.4 Bat

The Bat System consists of transmitters attached to tracked objects and mobiles, and an array of calibrated receivers deployed at known locations on the ceiling. RF and ultrasonic signals are transmitted simultaneously from the transmitters. The receivers measures the lag of the ultrasonic signal to infer its time-of-flight from the mobile transmitter. By multiplying the measured time-of-flight with the speed of sound, one can calculate the distance between a receiver and the mobile transmitter. A different distance estimate is obtained for each of the receivers deployed on the ceiling. The known positions of the receivers and their estimated distance from the mobile transmitter are used to calculate the mobile transmitter's position.

One disadvantage of the Bat System is the expensive wiring infrastructure used to relay information collected at the ceiling receivers to a central computer for processing. Then the processed location information must be relayed back to the user's handheld device. Since the location updates are transmitted periodically over a wireless network, the handheld pays a significant communication and energy overhead. Also, the Bat architecture does not scale well with the number of objects being located in the system. It places the transmitters at the locatable objects. As the number of locatable object increases, the level of contention among Bat transmitters increases.

Nevertheless, Bat is a pioneering location system that successfully demonstrates the advantages of using ultrasonic-based time-of-flight measurements in the indoor environment. Time-of-flight measurements generally provide more robust distance estimations than techniques that rely on signal strength, which are prone to interference by random noises generated in the environment. Although ultrasound is susceptible to non-line of sight (NLOS) errors caused by deflection and defraction, these errors can be reduced using various filtration techniques. Finally, the timing circuitry required to measure the time of flight at the speeds of sound is simpler and cheaper than the hardware required for measuring the time difference of GPS satellite signals that travel at the speed of light.

2.3 Cricket

CricketNav uses the Cricket location system as a low cost and practical solution to obtain periodic updates about the mobile user's current space, position, and orientation. Like Bat, Cricket uses a combination of RF and ultrasonic signals to determine the time-of-flight between a transmitter and a receiver. However, the architecture is fundamentally different from the Bat system. The Bat system places passive receivers in the infrastructure and an active transmitter on the mobile device. In contrast, Cricket deploys active transmitters (beacons) in the ceiling and passive receivers (listeners) on the locatable objects and devices.

The main advantages of Cricket's architecture are scalability, privacy, and ease of deployment. Because the mobile listener is passive, the level of channel contention does not increase with respect to the number of locatable objects and devices. Having passive listeners on the mobile device also fosters a model that facilitates user privacy; a mobile device can determine its location without having to inject any query messages into the infrastructure, which makes user tracking harder. Cricket is an ad hoc, distributed system. Beacons can be placed almost arbitrarily with little manual configuration and all components work autonomously without synchronization or centralized control, which simplifies system deployment.

Cricket beacons are programmable, which enhances the support for location-aware applications. Currently, Cricket beacons are programmed to broadcast a space descriptor and its coordinate over the RF channel during each periodic beacon. Thus, a listener can offer two types of location information. One of these is the listener's space location, which is a named area defined by some physical or virtual boundary such as a room or a section of a lecture hall. The other is the listener's position expressed in coordinates defined by the beacon coordinate system. Cricket beacons may also be programmed to include bootstrapping information such as the network location of the local navigation map server or resource discovery service.

One subtle, but important, distinction here is that the space information is offered directly by the Cricket location system, in contrast to other techniques that might use a spatial map to map a coordinate estimate to a space location. The space locations reported by the latter systems are not useful because we cannot use them to help correct erroneous coordinate estimates. On the other hand, Cricket offers *independent* and high quality space information. A Cricket listener infers its space location by finding the space descriptor advertised by the nearest beacon; hence, it directly infers the space location without external information such as a map. Moreover, the space location reported by Cricket is near-perfect at space boundaries that are not penetrable by ultrasound (e.g., a wall). Hence, applications can use Cricket's space information and a map of a floorplan to test if a reported position estimate is ambiguous (i.e., test if the point lies in the wrong space as in Figure 2-1). If the reported position is indeed ambiguous, applications can use the map information to help them guess a position that is consistent with the reported space. Therefore, by offering accurate space information, Cricket allows applications to

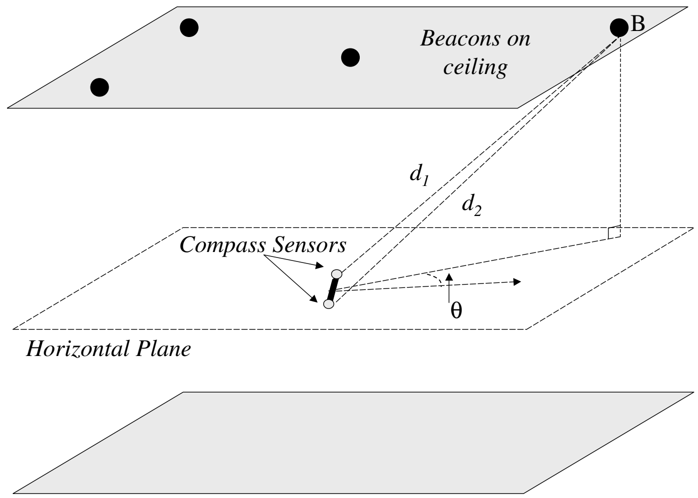


Figure 2-2: The Cricket Compass measures the angle θ with respect to beacon B.

correct ambiguous position estimates, which in turn increases the overall robustness of the location system.

Cricket listeners enhanced with Cricket Compass capabilities can provide orientation information [19]. Multiple ultrasonic sensors are mounted on the enhanced Cricket listener so that when the listener is oriented at an angle with respect to a beacon, the same ultrasonic signal hits each sensor at different times. Assuming that the listener is held relatively flat on the horizontal plane, the timing difference can be translated into an angle θ with respect to the beacon B that transmitted the beacon, as shown in Figure 2-2.

All Cricket beacons run the same distributed, randomized algorithm for avoiding collisions among neighboring beacons. Additional interference protection is implemented at the listener for detecting signal collisions and dropping beacon samples that are corrupted by an interfering beacon. Unlike many other existing location-sensing technologies, no part of the Cricket system require central control or explicit synchronization. This property helps keep the Cricket location system simple and robust.

2.4 Software for Cricket: The CricketServer

On their own, the Cricket listeners output raw measured distance samples to a host device via a serial port. The raw data must be processed by a space inference algorithm to determine the current space containing the host device. The raw data must also be processed by a filtering algorithm and a least-square matrix solver to produce sanitized distance and position estimates.

One contribution of this thesis is the design and implementation of the CricketServer, a service program that processes raw Cricket distance measurements from a particular Cricket listener into location information. Instead of implementing its own routines to process raw data, location-aware applications can simply connect to the CricketServer to obtain periodic location updates. The CricketServer normally runs in the handheld device attached to the Cricket listener but it can also run on the network to free up scarce CPU cycles on the handheld. The following sections discusses the CricketServer in detail.

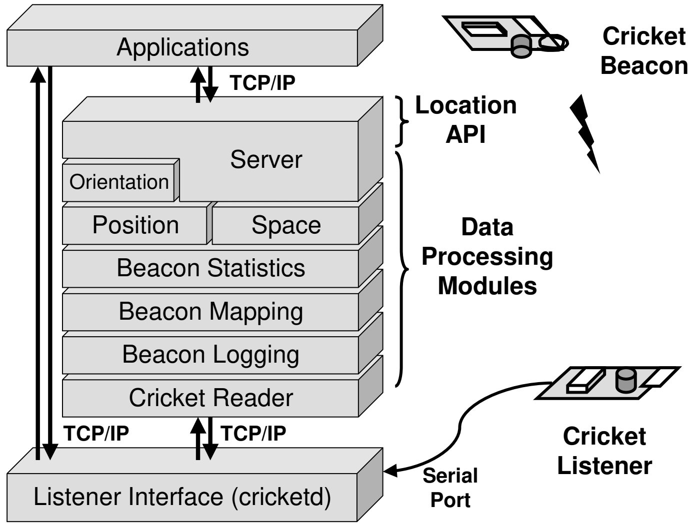


Figure 2-3: Software architecture for CricketServer.

2.4.1 Overview

The CricketServer is composed of a Cricket listener interface called cricketd[2], a stack of data processing modules, and a server that interfaces with location-aware applications. The software architecture for the CricketServer is shown in Figure 2-3.

Whenever a new beacon sample is received, the Cricket listener hardware checks the integrity of the received beacon packet. If not in error, it sends both the beacon and the time-of-flight measurement to cricketd via the serial port. In turn, cricketd pushes this unprocessed beacon via TCP/IP to the data processing stack and to other registered applications soliciting unprocessed data. Hence, applications and data processing modules may run locally on the host machine to which a listener is attached or remotely on a network-reachable server machine.

The data processing modules collect and convert time-of-flight measurements into location information such as space and position. Each module represents a basic data processing function and the output of each module is pushed to the input queue of the next until the data reaches the server, where location information is dispatched to all registered applications. Intermediate results may be multiplexed to different modules so that common data processing functionalities may be consolidated into one module. For example, the position and space inference modules may share a common module that filters out deflected time-of-flight measurements.

The modular data processing architecture has a couple of other advantages. For instance, one can experiment with different space inferencing algorithms by simply replacing the space inferencing module. In addition, new modules may be added later without affecting the existing ones. For example, if accelerometers are added to the basic Cricket listeners, new modules may be added to process the acceleration information without affecting the processing done by the existing modules.

[2] cricketd is based on gpsd, a popular serial port reader program for GPS devices [22].

2.5 Data Processing Modules

This section describes the data processing modules shown in Figure 2-3.

2.5.1 Cricket Reader Module

The Cricket Reader Module (CRM) is responsible for fetching raw beacon readings from a variety of beacon sources, which can be a local or remote instance of `cricketd`, a file containing previously logged beacons, or a simulator that generates beacon readings. Once a beacon reading is obtained from the source, it is parsed and fed into higher level processing modules.

2.5.2 Beacon Mapping Module

During system testing and evaluation, Cricket beacons may be moved frequently into different locations. Reprogramming the beacon coordinates may become a hassle. Instead, a Beacon Mapping Module (BMM) may be used to append beacon coordinates to the incoming beacon messages. By using the BMM, beacons do not advertise its coordinates over RF; thus, energy is saved. The beacon coordinate mapping is conveniently stored in a text file so that the coordinates can be modified easily. The beacon mapping can be downloaded over the network as the user device enters the beacon system.

One caveat about using the BMM is that the beacon coordinates must be kept consistent at all times. If a beacon is moved to a new position, the configuration file must be updated. Currently, the update process needs to be done manually. However, we have ongoing research in enhancing Cricket beacons with networking and auto-configuration capabilities so that the BMM configuration files can be updated automatically.

2.5.3 Beacon Logging Module

The Beacon Logging Module (BLM) logs beacon readings that can be replayed later by the Cricket Reader Module. This facility helps to debug modules and study the effects of various beacon filtering and position estimation algorithms. The logging facility also helps to support applications that wish to track the history of all the locations that a user has visited.

2.5.4 Beacon Statistics Module

The Beacon Statistics Module (BSM) collects beacon readings over a sliding time window. Distance measurements that occur within the sliding window are grouped by the source beacons. Various statistics such as the mean, median and mode distances are computed for the samples collected in the sliding window. The resulting beacon statistics are passed on to the Space Inferencing and Position Modules. They are also passed to the Server Module and dispatched to applications that have been registered to receive the beacon statistics information.

Space Inferencing Module

In Cricket, each beacon advertises a *space descriptor* so that a user device can infer its space location [18]. An example space descriptor is shown below:

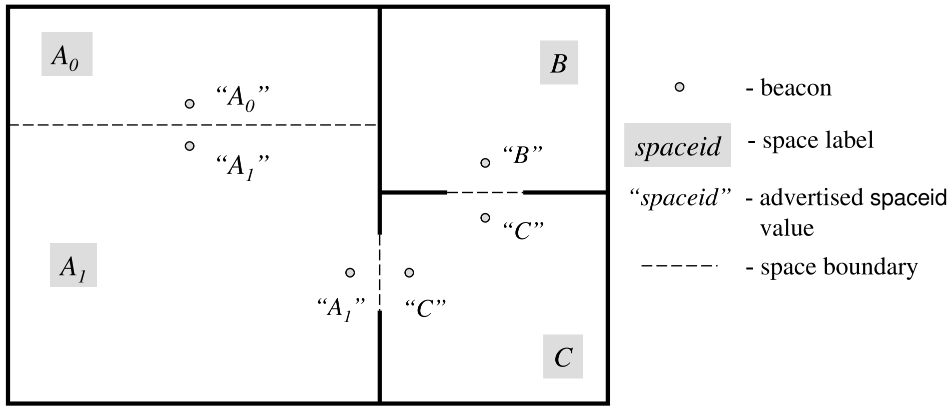


Figure 2-4: A Cricket beacon pair is used to demarcate a space boundary. The beacons in a beacon pair are placed at the opposite sides of and at equal distance away from the common boundary that they share. There is a beacon pair that defines a *virtual* space boundary between A_0 and A_1. Each beacon advertises the space descriptor of the space in which the beacon is situated. This figure does not show the full space descriptor advertised by each beacon.

[building=MIT LCS][floor=5][spaceid=06]

We associate the listener's current space location with the space descriptor advertised by the nearest beacon. In order to support this method of space location, we demarcate every space boundary by placing a pair of beacons at opposite sides of the boundary. In particular, each pair of beacons must be placed at an equal distance away from the boundary as shown in Figure 2-4. Note that a pair of beacons may be used to demarcate *virtual* space boundaries that do not coincide with a physical space-dividing feature such as a wall.

The Space Inferencing Module (SIM) collects the space descriptors advertised by all the near-by beacons and infers the space in which the listener is situated. The SIM first identifies the nearest beacon by taking the minimum of the mode distances of each beacon heard within the sliding window specified in the Beacon Statistics Module. Once the nearest beacon has been identified, its space descriptor is sent to the Server Module to update the user's space location.

It is important to note that the size of the sliding window in the Beacon Statistics Module must be large enough so that it can collect at least one distance sample from every beacon within the vicinity to identify the nearest beacon within range. The window size should be at least as large as the maximum beaconing period. The beaconing period depends on the number of beacons that are within range of the listener: the more beacons there are, the longer the listener has to wait before it collects a sample from every beacon. Hence, the window size defines the latency of space location and that latency depends on the average beacon density.

2.5.5 Position Module

The Position Module (PM) implements multilateration to compute the listener's position in terms of (x, y, z) coordinates. Let v be the speed of sound, d_i be the actual distance to each beacon B_i at known coordinates (x_i, y_i, z_i), and $\hat{t}_i$ be the measured time-of-flight to beacon B_i. The following distance equations hold:

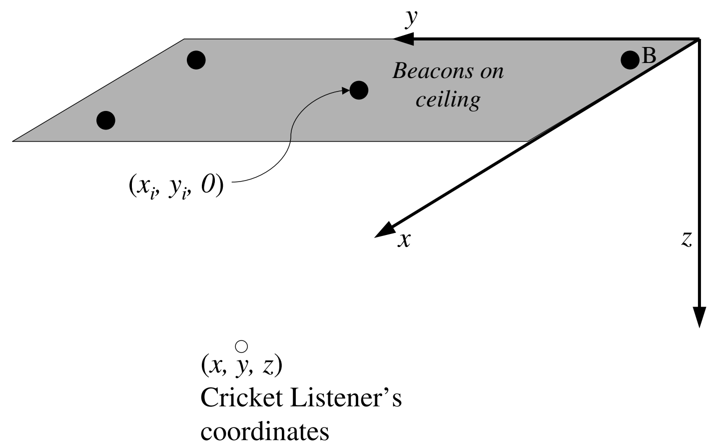


Figure 2-5: The coordinate system used in Cricket; the beacons are configured with their coordinates and disseminate this information on the RF channel.

$$(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2 = d_i^2 = (v\hat{t_i})^2 \tag{2.1}$$

If the beacons are installed on the same x-y plane (e.g. a ceiling), we can set $z_i = 0$.[3] Thus, the coordinate system defined by the beacons has a positive z axis that points downward inside a room, as shown in Figure 2-5. Consider m beacons installed on the ceiling, each broadcasting their known coordinates $(x_i, y_i, 0)$. We can eliminate the z^2 variable in the distance equations and solve the following linear equation for the unknown listener coordinate $P = (x, y, z)$ if the speed of sound v is known:

$$A\vec{x} = \vec{b} \tag{2.2}$$

where the matrix A and vectors $\vec{x}, \vec{b}$ are given by

$$A = \begin{bmatrix} 2(x_1 - x_0) & 2(y_1 - y_0) \\ 2(x_2 - x_0) & 2(y_2 - y_0) \\ \dots & \\ 2(x_{m-1} - x_0) & 2(y_{m-1} - y_0) \end{bmatrix}, \qquad \vec{x} = \begin{bmatrix} x \\ y \end{bmatrix},$$

$$\vec{b} = \begin{bmatrix} x_1^2 - x_0^2 + y_1^2 - y_0^2 - v^2(\hat{t_1^2} - \hat{t_0^2}) \\ x_2^2 - x_0^2 + y_2^2 - y_0^2 - v^2(\hat{t_2^2} - \hat{t_0^2}) \\ \vdots \\ x_{m-1}^2 - x_0^2 + y_{m-1}^2 - y_0^2 - v^2(\hat{t_{m-1}^2} - \hat{t_0^2}) \end{bmatrix}, \qquad m \geq 3$$

When $m = 3$, there are exactly two equations and two unknown and A becomes a square matrix. If the determinant of A is non-zero, then Equation (2.2) can be solved to determine unique values for (x, y). Substituting these values into Equation (2.1) then yields a value for z.

When more than three beacons are present ($m > 3$), we have an over-constrained system.

[3]The results presented here hold as long as z_i is set to some constant for all i.

In the presence of time-of-flight measurement errors, there may not be a unique solution for (x, y). We can still solve for an estimated value (x', y') by applying the least-squares method. For a general system of linear equations, the least-squares method finds a solution $\vec{x'}$ that minimizes the squared error value δ, where

$$\delta = (A\vec{x'} - \vec{b})(A\vec{x'} - \vec{b})^T, \quad \vec{x'} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

In some sense, the least-squares approach gives the "best-fit" approximation to the true position (x, y). However, the notion of "best-fit" here is a solution that minimizes δ, which is the squared error expressed in terms of a set of linearized constraints; it does not correspond to a solution that minimizes the physical distance offset between the true and estimated positions. Nevertheless, our experimental results presented in Section 2.6 show that the linearized least-squares solution produces estimates that are reasonably close to the true values.

We note that there is a variant to the position estimation approach described above. Instead of assuming a nominal value for the speed of sound, we can treat it as an unknown and rearrange the terms in Equation (2.2) to solve for (x, y, z, v^2) as discussed in [19]. The expression for $A, x, \vec{b}$ becomes:

$$A = \begin{bmatrix} 2(x_1 - x_0) & 2(y_1 - y_0) & \hat{t_1^2} - \hat{t_0^2} \\ 2(x_2 - x_0) & 2(y_2 - y_0) & \hat{t_2^2} - \hat{t_0^2} \\ 2(x_1 - x_0) & 2(y_1 - y_0) & \hat{t_3^2} - \hat{t_0^2} \\ \vdots & & \\ 2(x_{m-1} - x_0) & 2(y_{m-1} - y_0) & \hat{t_{m-1}^2} - \hat{t_0^2} \end{bmatrix}, \qquad \vec{x} = \begin{bmatrix} x \\ y \\ v^2 \end{bmatrix},$$

$$\vec{b} = \begin{bmatrix} x_1^2 - x_0^2 + y_1^2 - y_0^2 \\ x_2^2 - x_0^2 + y_2^2 - y_0^2 \\ \vdots \\ x_{m-1}^2 - x_0^2 + y_{m-1}^2 - y_0^2 \end{bmatrix}, \qquad m \geq 4$$

This approach solves for v dynamically using the time-of-flight measurements. Since the speed of sound can vary with environmental factors such as temperature, pressure, and humidity [7], this approach might improve the positioning accuracy when the location system operates in varying environmental conditions. When the listener collects distance samples from five or more beacons ($m \geq 5$), we can further improve the position estimation accuracy by applying least-squares to find a "best-fit" solution for (x, y, z, v^2).

We can still apply other heuristics to use the extra beacon distance measurements whenever they are available. For example, we can choose the three "best" beacon distance estimates out of the set of beacons heard in the sliding window (from BSM) to estimate the listener's position. Alternatively, we can compute a position estimate from all possible combinations of three beacon distances, and then average all the results to obtain the final position estimate [5]. In practice, extra beacons may not always be within range of the listener as the number of redundant beacons deployed may be minimized to reduce cost. The best algorithms should adapt to the information that is available.

We consider the tradeoff between improving the accuracy of position estimates and latency, which can increase as a Cricket listener tries to collect distance samples from more beacons. In our current implementation, we impose a latency bound by specifying a sliding sampling window of five seconds in the BSM. Then we apply the following heuristic based on our experimental results described in Section 2.6: If distance samples are collected from fewer than five beacons within the sliding window, we assume a nominal value for the speed of sound, and apply least-squares to solve for (x', y', z'). When the listener collects distance samples from more than or equal to five beacons within the sampling window, we treat the speed of sound as unknown and apply least-squares on *all* the available constraints to solve for (x', y', z', v'^2). This heuristic improves the accuracy of our position estimates while keeping the latency acceptably low.

Besides collecting samples from multiple beacons, a listener may collect multiple distance samples from the same beacon within a sampling window. In this case, the BSM helps refine the distance estimate to a beacon by taking the mode of the distances sampled for that beacon. Taking the mode distance value effectively filters out the reflected time-of-flight measurements [18, 20].

Finally, the multilateration technique discussed above assumes that the listener is stationary while it is collecting distance samples. More specifically, Equation (2.2) assumes that the $\hat{t}_i$ measurements are consistent with a given listener coordinate so that the equation can be solved *simultaneously* [24]. However, when the listener is mobile, as in our navigation application, the distance samples collected from each beacon are correlated with the listener's velocity. As the user moves, successive distance samples will be collected at a different location; thus, the simultaneity assumption in Equation (2.2) is violated. In Cricket, the time lag for collecting samples from three different beacons takes at least 0.5 to 1 second. Assuming the mobile user travels at 1 m/s, the distance estimates can vary between 0.5 to 1 meter. Thus, we cannot simply ignore how mobility affects position estimation.

To continuously track the position of a mobile user, we need a more sophisticated positioning technique such as integrating inertial sensors to the Cricket listener and/or adopting the Single-Constraint-At-A-Time (SCAAT) tracking algorithm presented in [23, 24]. The original SCAAT algorithm was developed for the HiBall tracking system and must be adapted to Cricket. In this thesis, we implement the stationary positioning technique described above and assume that the listener stays stationary for a few seconds before its position can be reported. Ideally, the listener should remain stationary until all the stale distance samples is flushed from the sliding window in the BSM.

Orientation Module

The Orientation Module (OM) processes data from a Cricket Compass and outputs orientation information in terms of θ and B, which are the relative horizontal angle and the reference beacon shown in Figure 2-2. Currently, we have not implemented the OM module but the algorithm for estimating θ can be easily implemented [19]. The Cricket Compass finds θ by measuring the absolute differential distance $|d_1 - d_2|$ with respect to a reference beacon B. The algorithm for estimating θ involves computing the mode of a sequence of differential distance samples reported by the Cricket Compass, and performing a table lookup for translating the difference into an angle value. The final step of the algorithm uses the output of the Position Module to calculate a number used to scale the final result for θ.

The CricketServer is not limited to process orientation measurements from a Cricket Compass. We can just as easily implement and load an orientation module that processes

Beacon Name	(x, y, z)
506A	427, 183, 0
506B	0, 122, 0
506C	244, 244, 0
506D	61, 61, 0
510A	61, 488, 0
510B	0, 305, 0
510D	427, 305, 0
600LL	366, 61, 0
600LR	244, 366, 0
613B	366, 427, 0
615	122, 366, 0
617	183, 0, 0

Table 2.1: Experimental setup: Beacon coordinates (in cm)

data from a digital magnetic compass, and report θ with respect to the magnetic north pole.

Server Module

The Server Module (SM) implements the CricketServer 1.0 protocol to access various location information produced by the data processing modules. The protocol uses a self-describing format, which makes it extensible. It uses ASCII, which eases the debugging and parsing process during application development. The SM transfers location information over the TCP/IP protocol. Thus applications may fetch processed location information from a specific Cricket listener over the network. The complete specification of the CricketServer 1.0 protocol is in Appendix A.

2.6 Cricket's Positioning Performance

We conducted an experiment to evaluate the positioning performance of the Cricket location system. In our setup, we placed twelve Cricket beacons on the ceiling of an open lounge area as illustrated in Figure 2-6. To prevent degenerate cases that cause $det(A) = 0$, we ensured that no three beacons are aligned on a straight line and that no four beacons lie on the same circle. The beacons were configured with the coordinates shown in Table 2.1. We placed a Cricket listener at coordinate $(122cm, 254cm, 183cm)$ and collected over 20000 distance samples from the twelve beacons in about an hour. For each distance sample, we recorded the time stamp and the source beacon that generated the distance sample. The listener remained stationary for the duration of the experiment.

2.6.1 Sample Distribution

Figure 2-7 plots the distribution of logged samples. The plot shows an uneven distribution of samples collected from the different beacons. In particular, the listener received very few successful transmissions from beacon 510D over the one hour period. We do not know the

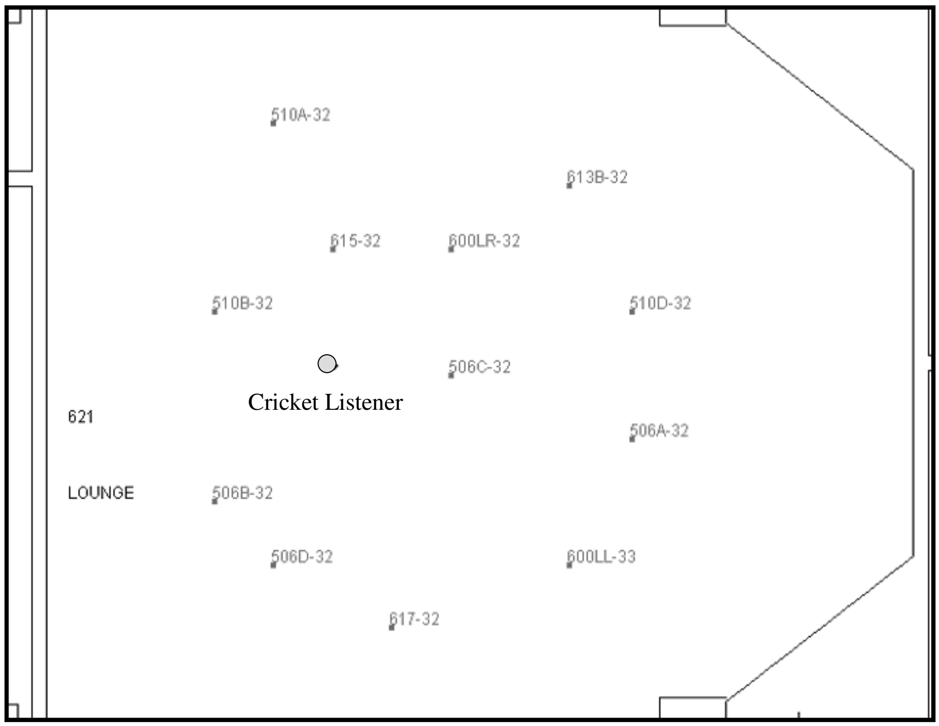


Figure 2-6: Experimental setup: Beacon positions in the ceiling of an open lounge area

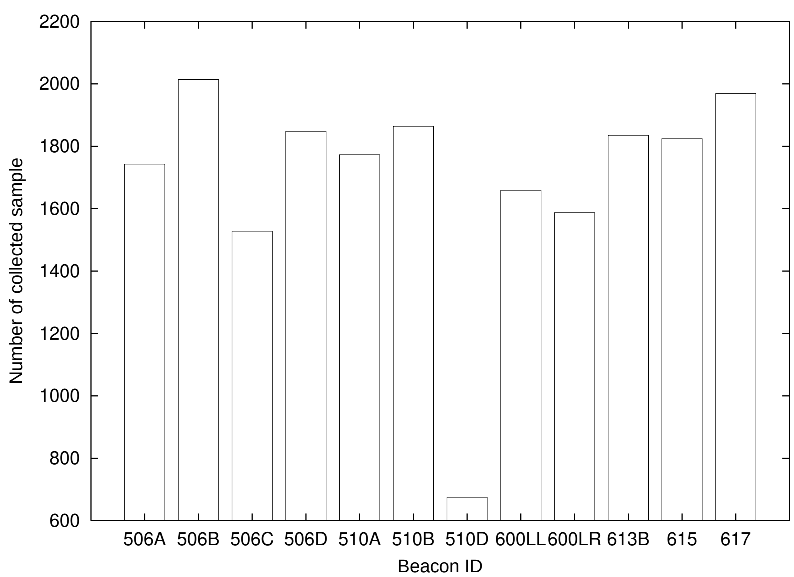


Figure 2-7: Sample distribution for all beacons listed in Table 2.1

exact cause of this behavior but we suspect that beacon 510D may have a faulty antenna that is emitting an unusual RF radiation pattern. When the strength of the RF radiation from a faulty beacon is not evenly distributed in free space, the neighboring beacons may not detect its transmission. Thus, the neighboring beacons incorrectly assume that the channel is clear for transmission. Instead of staying silent to avoid a collision, the neighboring beacons attempt to transmit while the faulty beacon is transmitting. Consequently, the listener receives fewer successful samples from the faulty beacon.

2.6.2 2D Positioning Accuracy

Recall from Section 2.5.5 that the position estimates depend on the distance estimates to the individual beacons. Thus, it is intuitive that if we improve the accuracy of the individual distance estimates, we would improve the accuracy of the position estimates. One way to improve accuracy of distance estimates is to collect multiple distance samples from each beacon and then take either the mean or the mode of these values. Alternatively, we can collect distance samples from a large number of beacons, which increases the number of constraints in the linear system of Equation (2.2). Denote k to be the *sample frequency*, which we define to be the number of samples collected from *each* beacon. Also, we denote m to be the *beacon multiplicity* or the number of distinct beacons from which distance estimates are collected, which is equivalent to the number of constraints in the linear system. We examine how the positioning accuracy varies with k and m from our experimental values.

Because 2D position estimates in the xy plane are sufficient for a large number of applications such as CricketNav, we analyze the 2D positioning accuracy. For our analysis, we define the position estimation error, ϵ, to be the linear distance between the listener's true coordinate (x, y) and the estimated coordinate (x', y') produced by the least-square algorithm:

$$\epsilon = \sqrt{(x - x')^2 + (y - y')^2}$$

Thus, we ignore the z component in our accuracy analyses.

Methodology

We obtain our experimental results by randomly choosing distance samples from our data log and process the selected set of samples to obtain a least-squares position estimate. More precisely, we first group the collected data into twelve sample arrays, each containing distance samples from a single beacon. For each position estimation trial, we randomly pick m beacon arrays, and from each array, we randomly pick k samples. To obtain distance estimates to each beacon, we take either the mode (denoted as "MODE") or the mean (denoted as "MEAN") of the k samples chosen from each beacon. For MODE, the distance estimate is set to the average of all modal values when there is more than one distance value occurring with same maximal frequency. Using either MODE or MEAN, we compute a distance estimate for each of the m beacons. Then we plug the distance estimates into a least-squares solver to obtain a position estimate. As mentioned in Section **??**, the least-squares solver can find a position estimate by treating the speed of sound as a known value or an unknown variable. We compare the position estimates for both and denote them as KNOWN and UNKNOWN respectively. From each position estimate we get, we compute ϵ. For each experiment, we repeat this process with the same parameter values k and m for

500 trials. We take the average position error, $\bar{\epsilon}$, over these 500 trials as our final result for the experiment.

Occasionally, we obtain a set of distance estimates that yield imaginary results, where either $z^2 < 0$ or $v^2 < 0$. We ignore these values in our reported results. We note, however, that no more than 25% of the trials in all our experiments gave imaginary numbers for the range of k and m values we have chosen.

The nominal value we used for the speed of sound, v, is $345m/s$, which is the expected value at $68°F$ [7]. Unless stated otherwise, the speed of sound is treated as a known value when we compute a least-squares estimate.

Our experiment method closely emulates how the CricketServer collects and processes beacon samples from the Cricket listener. In particular, we randomly choose a subset of m beacons out of the twelve beacons. The geometry of beacon placement is known to affect positioning accuracy in other location systems [14]. We do not yet understand how the beacon placement geometry in our setup affects the results in our experiment. But by selecting m beacons randomly in every trial, we obtain position estimates from different beacon configurations. Thus, this strategy minimizes the bias that might arise from using samples from only a fixed set of m beacons in a fix geometric configuration.

Results and Analysis

Figure 2-8 plots $\bar{\epsilon}$ as a function of k using MEAN distance estimates. The different curves represent the results obtained by varying the beacon multiplicity, m. For example, the curve MEAN4-KNOWN shows the results obtained by collecting samples from four beacons ($m = 4$). The figure shows that the positioning accuracy do not improve as the sample frequency, k, increases. To explain why the accuracy does not improve, we note that there are occasional spikes in the distance measurements caused by reflected ultrasonic signal in the environment. Such measurements skews the average of the measured distance samples. Also, the likelihood of sampling a reflected distance reading increases with k. Thus, the average $\bar{\epsilon}$ actually increased slightly with k. However, we note that the positioning error decreases with increasing m. This result is expected because increasing the beacon multiplicity is equivalent to increasing the number of constraints in the linear system to improve the fitting of the position estimate.

Figure 2-9 plots $\bar{\epsilon}$ as a function of k using MODE distance estimates. This figure shows that $\bar{\epsilon}$ decreases with k. When a large number of samples are collected from each beacon ($k > 5$), MODE effectively filters out the reflected distance readings. However, when few samples are collected (i.e., $k \leq 5$), it is difficult to collect enough consistent samples to benefit from MODE. Essentially, the MODE distance estimation becomes a MEAN distance estimation for small values of k. Some evidence of this is shown by comparing the similar values of $\bar{\epsilon}$ measured at $k = 1, 2$ in Figures 2-8 and 2-9.

Also from Figure 2-9, we note that the results for MODE3-KNOWN represents a non-least-square solution because the beacon multiplicity is three, which means that there are exactly three equations and three unknowns. In this case, MODE3-KNOWN does not benefit from the least-squares fitting effect that would be induced by an over-constrained system. Nevertheless, the average error from MODE3-KNOWN is not too much larger than the errors from the other least-square position estimates.

Figures 2-10 and 2-11 compares the results obtained from a linear system that solves the speed of sound, v, as an unknown (UNKNOWN) versus one that assumes a nominal value for the speed of sound (KNOWN). The MODE4-UNKNOWN curves in both figures

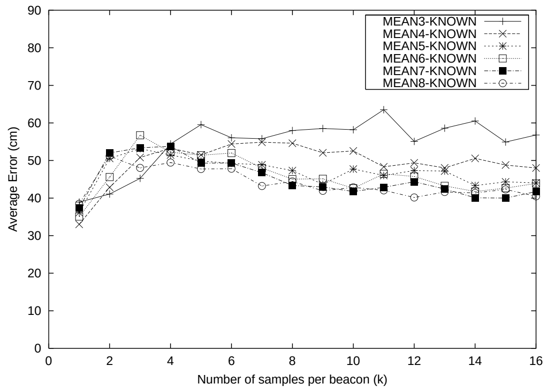


Figure 2-8: Position Error (cm) vs. k using MEAN distance estimates. The speed of sound is assumed known when calculating the position estimate.

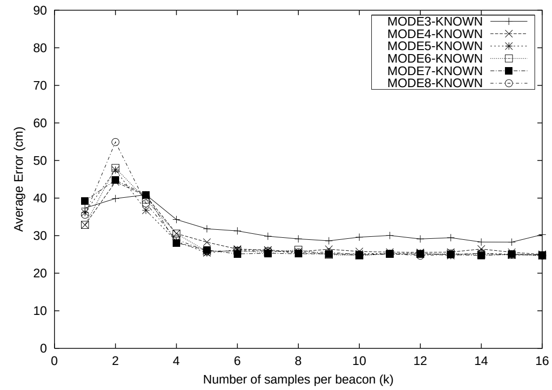


Figure 2-9: Position Error (cm) vs. k using MODE distance estimates. The speed of sound is assumed known when calculating the position estimate.

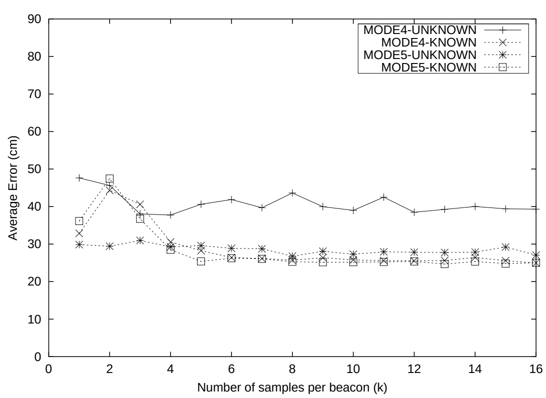


Figure 2-10: Position Error (cm) vs. k using MODE distance estimates and varying m between 4 and 5. The curves labeled UNKNOWN (KNOWN) represent the results obtained from a linear system that assumes an unknown (known) speed of sound.

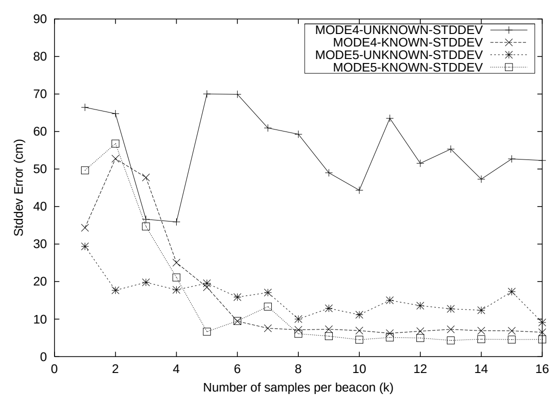


Figure 2-11: Standard deviations of the results reported in Figure2-10.

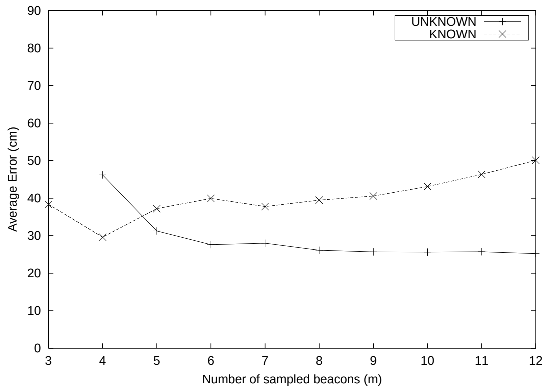


Figure 2-12: Position Error (cm) vs. m for $k = 1$.

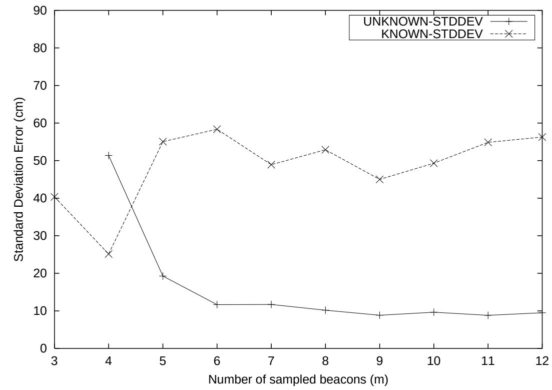


Figure 2-13: Standard deviations of the results reported in Figure2-12.

respectively plot the values for the average error, $\bar{\epsilon}$, and the standard deviation of the position error over 500 trials, $\sigma(\epsilon)$. The results of these curves are obtained by treating v as an unknown. In this case, there are exactly four constraints and four unknowns in the linear system. Thus, there is no redundancy in the system to induce a least-squares-fitted solution. Indeed, the positioning estimates are poor for MODE4-UNKNOWN when compared to MODE4-KNOWN, which has one extra constraint. However, the results are not always poor when we solve v as an unknown. For example, all the MODE5-UNKNOWN curves have generally lower estimation errors and standard deviations than MODE5-KNOWN. Although MODE5-KNOWN has one more constraint in the linear system than MODE5-UNKNOWN, both systems have a beacon multiplicity that is greater than the number of unknowns in the linear system. Hence, both MODE5-KNOWN and MODE5-UNKNOWN are over-constrained and both schemes induce a least-squares-fitted position estimate. One fundamental difference between the two schemes is that MODE5-KNOWN finds a best-fit estimate in the x, y space whereas MODE5-UNKNOWN finds a best-fit estimate in the x, y, v^2 space. We do not understand the theoretical implications of this difference but our experimental results show that for $k \leq 5$, the MODE5-UNKNOWN scheme performs better, while it performs similarly to MODE5-KNOWN when $k > 5$. Although we do not have an explanation for these results, the two figures suggest that the least-squares method is a significant factor in minimizing $\bar{\epsilon}$, especially when k is large. This relationship holds in all our experiments with $m \geq 5$.

We paid particular attention to the results at $k = 1$ because the previously discussed figures 2-10 and 2-11 respectively show that the positioning error and standard deviation decreases as m increases. To verify if the trend holds, we plot $\bar{\epsilon}$ and $\sigma(\epsilon)$ as a function of m at $k = 1$ in Figures 2-12 and 2-13 respectively. The curves labeled KNOWN (UNKNOWN) represents the results obtained from a linear system that assumes the speed of sound is known (unknown). The errors and standard deviations of the UNKNOWN curves decreases with increasing m but an opposite trend occurs for KNOWN. Currently, we do not have an explanation for this. However, our results suggest an interesting way to optimize the beacon placement strategy to improve the accuracy of estimating position: *We should increase the beacon density*[4] *such that the listener can hear from five or more beacons from any location.*

[4]However, we should not increase the beacon density so much that the contention in the system creates

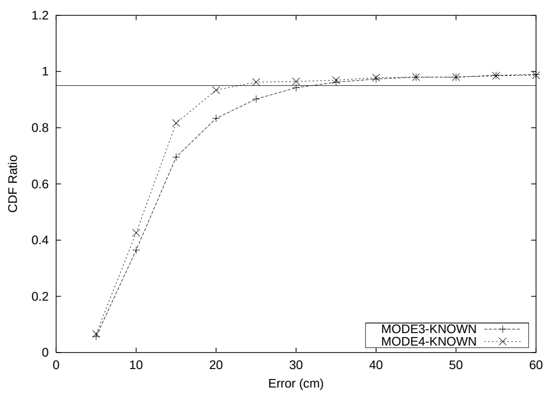


Figure 2-14: Cumulative error distribution for $k = 1$ and $m = 3, 4$. Speed of sound is assumed known. The horizontal line represents the 95% cumulative error.

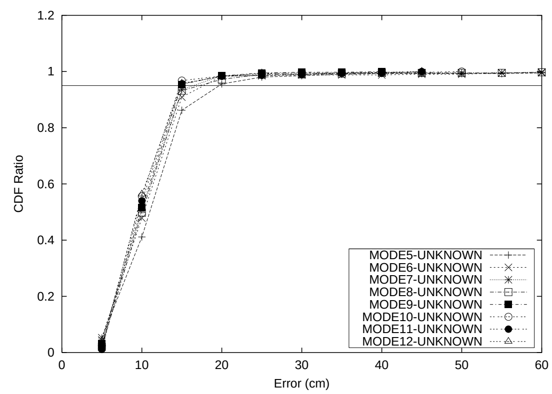


Figure 2-15: Cumulative error distribution for $k = 1$ and $5 \leq m \leq 12$. Speed of sound is assumed unknown. The horizontal line represents the 95% cumulative error.

Assume that the BSM uses a small sliding window to collect only one sample from each beacon (i.e., assume $k = 1$), we now analyze if our positioning algorithm yields the required accuracy as mentioned in Section 2.1.2. To minimize positioning error for all m, our least-squares positioning algorithm crosses-over between UNKNOWN and KNOWN depending on the number of distinct beacons sampled in the BSM's sliding window. Figure 2-12 shows that the cross-over point is at $m = 5$. Thus, to test if our positioning algorithm meets CricketNav's accuracy requirements, we plot the cumulative distribution function (CDF) of the positioning error in two figures. Figure 2-14 plots the CDF for $m \leq 4$, and the speed of sound is KNOWN, and Figure 2-15 plots the CDF for $m > 4$, and the speed of sound is UNKNOWN. The highest error value intersected by the 95% horizontal line in both figures is no greater than $30cm$. Hence, we conclude that our positioning algorithm meets CricketNav's accuracy requirement; our positioning algorithm produces results that have less than $50cm$ position error 95% of the time.

We summarize the results of our positioning accuracy analysis:

- A MODE distance estimation produces more accurate position estimates than MEAN.
- MODE does not begin to take effect until $k >= 5$.
- The least-squares method described in Section 2.5.5 effectively reduces positioning errors.
- For $k < 5$ and $m \geq 5$, it is better to assume speed of sound is unknown. Otherwise, it is better to assume speed of sound is known.
- For $k = 1$, Cricket is accurate to within $30cm$, 95% of the time.

an unacceptable delay for a listener to receive a successful transmission.

2.6.3 Latency

From the previous section, we learn that the positioning accuracy varies with the number of samples collected per beacon, k, and the number of distinct beacons sampled, m. Conceivably, we could specify the values of these parameters in the CricketServer to obtain the desired positioning accuracy. Such a feature would be useful if applications require strict guarantees on positioning accuracy. However, the cost of guaranteeing accuracy is latency, which we define here as the time it takes to collect enough samples to compute a position estimate with a certain accuracy. As in the previous section, we parameterize accuracy by k and m. Thus, for our experiments, we define latency to be the time it takes to collect k distance samples from m different beacons. Since CricketNav have requirements for both positioning accuracy and latency, it is interesting to study how latency varies with k and m. The following sections describes our experimental results on latency.

Methodology

We wish to design an experimental methodology that closely resembles a typical situation that would be faced by the CricketNav application. In a typical situation, Cricket beacons are deployed throughout the building. A Cricket listener receives a successful sample only if it is within range of a beacon's ultrasonic *and* radio transmissions. This requirement has an implicit effect on latency. For example, a listener may not receive a successful sample from a beacon that is situated in a neighboring room; while the radio signal can pass through the wall standing between the beacon and the listener, the ultrasound cannot. Thus, if there are a large number of *interfering beacons* in the neighboring rooms, a large number of received radio transmissions will be dropped by the listener, which then increases the delay to collect a distance sample from every beacon in the same room.

In our experiments, we assume that the Cricket listener is within radio range of twelve beacons. This is a realistic assumption if the beacon density is low and the radio range is short. Next, we assume that the Cricket listener is within ultrasonic range of m beacons. Thus, as we vary m, we model that there are $12 - m$ interfering beacons.

We measure the average latency by sampling our data log. Recall that each distance sample in our log were recorded with a time stamp and that the log entries are sorted in time order. Before each sampling trial, we randomly pick a set of m beacons. To measure the average latency, we randomly choose a starting time in the log file and scan forward until we have read at least k samples each from the m selected beacons. The scan wraps around to the first entry if it continues through the end of the log. After the scan stops, the time difference between the time stamps at the starting and stopping points of the scan becomes the latency result for one trial. The final latency result for an experiment with specific values for k and m is computed by averaging the latency over 500 trials.

Results and Analysis

Figure 2-16 shows the mean and median latency of collecting one sample from varying number of different beacons. The latency stays at around $3.25s$ for different values of m. The latency does not depend on m because the scan stops only after it collects at least one sample from a *randomly-predetermined* subset of m beacons. Note that the average delay for collecting a sample from a specific beacon is half of the beacon's average beaconing period. On average, every beacon gets to transmit once within a beaconing period. Thus, the average waiting time is the same whether we collect one sample each from a few beacons

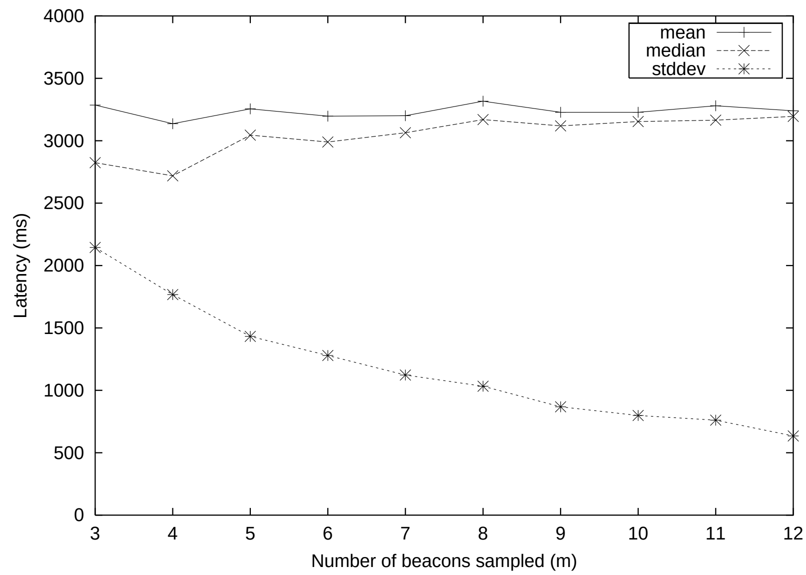


Figure 2-16: Mean, mode, and standard deviation of the time elapsed to collect one sample from m beacons.

(small m) or from a large number of them (large m). From Figure 2-16, the average delay is $3.2s$, which suggest an average beaconing period of $6.4s$.

As mentioned before, we expect every beacon to transmit a signal within a beacon period in the average case. Thus, to collect k samples from the same beacon, we must wait for $k - 0.5$ beaconing periods on average. Hence, we expect a linear relationship between latency and k, where the slope is the average beaconing period. In Figures 2-17 and 2-18, we plot the mean and median latency of collecting k samples each from m beacons. We notice a reasonable linear relationship between latency and k as expected. The slope varies for different beacon multiplicity. This makes intuitive sense because the chances for *all* m beacons to transmit successfully in all the periods of a given series of consecutive periods

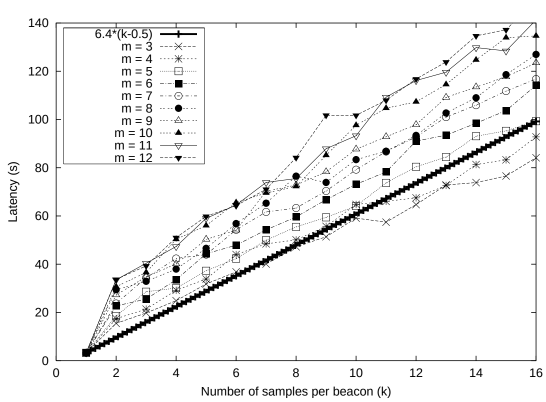


Figure 2-17: Mean latency to collect k samples each from m beacons.

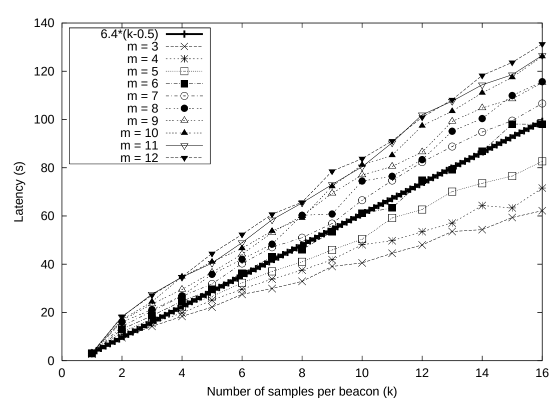


Figure 2-18: Median latency to collect k samples each from m beacons.

decreases as m increases. For reference, we also plot a thick line with a slope equal to the average beaconing period derived from Figure 2-16.

In general, the latency is greater than 10 seconds when the sample frequency is greater than one. Thus, if we set a sampling window of less than $10s$ in the Beacon Statistics Module (Section 2.5.4), it is unlikely that the CricketServer would collect more than two samples from each beacon. For applications that sets a sliding sampling window size of less than $10s$, such as CricketNav, we conclude that the accuracy cannot be improved by increasing sample frequency, k. Fortunately, when $k = 1$, the latency is low ($3.2s$) for all values of m as shown in Figure 2-16. If we set a sliding window size of five seconds, there is a high probability that all twelve beacons within range will be sampled at least once. Recall from Section 2.6.2, the accuracy required for CricketNav can be achieved when $k = 1$. Hence, a window of five seconds will collect enough distance samples to achieve the required accuracy.

2.7 Chapter Summary

We decided to use the Cricket indoor location support system to supply CricketNav with updates about a user's current space location *and* position, which are required for indoor navigation to reduce the chance of locating the user in the wrong space. Cricket is an ad hoc, distributed system that is scalable, robust, and easy to deploy in the indoor environment.

To enhance Cricket's software support for location-aware applications, we developed the CricketServer to process raw Cricket distance measurements and to export an extensible interface that the applications can use to access location information. One of the main features of CricketServer is the implementation of the Cricket positioning algorithm, which uses a least-squares estimator that computes a best-fit position estimate from a set of imperfect distance measurements. The algorithm adapts to the number of different beacons sampled to improve accuracy.

We conducted a series of experiments to evaluate our positioning algorithm. Our results show that the Cricket positioning algorithm produces position estimates that are accurate to within $30cm$, 95% of the time, and meets the accuracy and latency requirements for the CricketNav application.

Chapter 3

Spatial Information Service

CricketNav needs a Spatial Information Service (SIS) to provide *spatial information* about all the accessible paths and space-dividing features within a given building. CricketNav uses the path information to guide a user from an origin to a destination. To navigate users, CricketNav uses the structural information of a given building to render a map showing the location of the user, and to compute the direction of the navigation arrow pointing toward the next step to reach the desired destination.

The SIS service is also useful for other location-aware applications. For example, resource finders need spatial information to determine the resource object (e.g., a printer) that is nearest to the user's current position. The absolute Euclidean distance between the user's coordinate and an object's coordinate provides one metric for processing the query. A more useful metric is the relative path distance, which models the cost of taking a certain path to reach a particular destination. For example, the nearest reachable printer may be located across a hallway from where the user is, even though another printer is located on the next floor directly above the user and is physically closer in terms of Euclidean distance. The relative path distance requires spatial information and its cost model can be enhanced by weighting the path lengths according to the physical features found along the path. For example, a user may prefer to reach a resource on a path that crosses fewer doorways and staircases, or is accessible by wheelchairs. A more sophisticated cost model may incorporate dynamic information such as the state of a smoke detector or a door-lock to determine if the path is safe or accessible.

The SIS service can be extended to provide path feature annotations and dynamic state information to allow location-aware applications apply their own relative path distance metrics. However, we believe that the dynamic state queries and updates should best be handled by a separate service such as COUGAR [4]. We describe the design and implementation of a SIS service that supplies static spatial information to CricketNav and other location-aware applications.

3.1 Requirements

The main challenges in building a Spatial Information Service is the process of generating spatial information and finding a suitable data structure to represent this information. To reduce the cost of generating indoor maps, we would like to avoid manual surveying. Instead, we would like to automate the process by developing tools to convert the original

CAD[1] drawings of architectural floorplans composed of segments, vertices, and text labels into *spatial information maps* that annotate structural features such as walls, doorways, elevators, and staircases.

A spatial information map for navigation and other location-aware applications should provide information to enable the following operations:

1. Compute the shortest path between some origin and destination expressed in coordinates or spaces
2. Find the set of structural features that lie within a given radius of a given coordinate
3. Find a reference coordinate that represents the "center position" of a given space
4. Find the space that contains a given coordinate
5. Given a space and coordinate (S, C) pair, locate a coordinate D such that the distance between C and D is a minimum for all $D \in S$. In particular, if $C \in S$, then $D = C$.

To facilitate the first operation, a spatial information map is required to lay the plan for the accessible paths in the building structure. We represent the network of accessible paths as a graph. In this graph, a node represents an intermediate point on the path called a *waypoint*. An edge represents an unobstructed path segment between two waypoints. To permit applications to apply a custom metric for computing the shortest path, the data structure must not specify the cost of traversing an edge. Instead, each edge should be annotated with a feature (e.g., a door boundary) that lies between the two waypoints, if one is present. Furthermore, each waypoint on the graph should be annotated with a coordinate and a descriptor of the space in which it is located. Thus, applications can perform the shortest path operation with the origins and destinations expressed in terms of either coordinates or spaces.

To indicate the user's current position, applications may use the second operation to extract the physical features of a specified region and render them on a user interface. For simplicity, the query region is represented by a circle specified by a center coordinate and a radius. Sometimes the Cricket location infrastructure may provide the current user's position only in terms of the current space. In this case, applications may use the third operation to convert a given space to a map coordinate representing the center position of that space. The reference coordinate can then be used to specify the query circle. Note that the definition of the center position of a given space is somewhat arbitrary. For convenience, we define the center position of a given space as the position of the corresponding text label in the CAD drawing.

Although the Cricket location infrastructure reports the user's space and position, the fourth operation is useful for testing the consistency of the reported space and coordinate values. If the reported values from the location sensors are different from those found in the map, there is most likely an error in the position estimate (as explained in Section 2.1.1). In this case, applications should assume that the user is located in the reported space. If an application still requires an estimated position, the fifth operation can be used to find a coordinate in the reported space that is nearest to the reported coordinate (see Figure 2-1). Thus, the fifth operation is a powerful primitive that corrects implausible position estimates that are sometimes reported by the Cricket location infrastructure.

The algorithms used to implement these operations are discussed in Chapter 4.

[1] CAD is an abbreviation for Computer Aided Design

3.2 Dividing Spatial Information Maps

The spatial information map of a building may be too large for timely downloads or to fit in the memory of handheld devices. In our implementation, we have subdivided the map into floors so that there is a separate spatial graph (called the *floor map*) for every floor. This is a natural subdivision because each CAD drawing usually contains the structural plan for a single floor. It is also an effective subdivision since we find the average size of a map for a typical floor of our current lab building is between 300 to 400 kilobytes. The desired maps can be downloaded from the SIS Server by specifying a unique map identifier, which can simply be a floor number. If a SIS Server stores floor maps for different buildings on the same campus, the map identifier should contain a building identifier, a campus identifier, and so on.

Subdivision by floors also has the advantage of allowing end applications to perform shortest path computations.[2] If the destination is located on a different floor, we simply fetch the map corresponding to that floor. The end-to-end shortest path can then be broken down into two shortest path computations: one from the origin to either a staircase or an elevator in one floor map, and the other from either a staircase or elevator to the destination in another floor map. Thus, elevators and staircases serve as *linkage points* between two floor maps. If the spatial information map were divided into finer subdivisions such as a section of a floor, end-to-end path planning becomes more complicated; applications will require some external mechanisms to fetch and piece together all of the spatial features that lie between the origin and destination.

We note that a number of interesting optimizations can be implemented to help reduce the overhead of fetching the spatial information maps. For example, the downloaded floor maps can be cached so that they do not have to be re-fetched when the user roams forth and back between visited floors. We leave the design and evaluation of caching mechanisms to future work.

3.3 Space Descriptor

A *space descriptor* is used to identify the location of a space. The descriptor consists of one or more attribute and value pairs, which are arranged in a hierarchical order where each pair successively refines the space location of the previous one. For example,

[building=MIT LCS][floor=5][spaceid=06]

Note that the building field is not necessary when the location domain of the application is contained within the current building (i.e. the application do not refer to spaces outside of the current building). For this thesis, we assume that our navigation application work within a building and do not plan routes between buildings.

The spaceid field represents a generic value used to name a given area within a floor. The spaceid can be a room number, the name of a lecture hall, the label for a particular section of a hallway, or even the name of the person occupying an office cubicle. In our implementation, the spaceid field is assigned the same value as the one given on the floorplan so that the assignment process can be automated. If no label is given for a space in the

[2]It is advantageous for end applications to perform shortest path computations locally because it avoids communication overhead, which can be costly for battery-powered wireless devices. Furthermore, it allows applications to use the desired distance metrics.

floorplan, we can either ignore the space (by assigning a null descriptor value to it) or assign it an arbitrary value.

Although the Cricket location system allows a system administrator to define arbitrary space boundaries, we use the space boundaries that are defined by the floor map generation process (see Section 3.5.1). This assumption is necessary for automation. It is important to note that the automated map generation process can sometimes produce space boundaries that do not correspond exactly to the intended space boundaries drawn on the floor maps. It is very difficult to produce perfect space boundaries when the CAD drawings do not contain semantic information about space boundaries.

3.4 Coordinate System

Points in the spatial information map must be expressed in some coordinate system. But which coordinate system should we use? We devote this section to address this issue.

A coordinate system is defined by a representation (e.g., Cartesian, polar, or longitude-latitude coordinates) and a reference frame (i.e., the point of origin). For example, global navigation systems used in aviation, land and marine transportations use longitudes and latitudes (long-lat) with a reference point at the prime meridian. Conceivably, the Cricket beacons could be configured with the same coordinate system. In this case, mobile applications can keep the same position representation and reference point as the user roams between the indoor and outdoor environment.

On the other hand, it may not be practical to *deploy* Cricket using the long-lat coordinate system. For instance, it may be easier to define a set of local reference points throughout a building (e.g., corner of a floor or a room) and configure the beacon coordinates with respect to the local reference frames. Also, local reference frames may be necessary when the reference frame itself is mobile (e.g., deploying a location system on a cruise ship). Within each local reference frame, we can still express coordinates in terms of longitudes and latitudes. However, for indoor applications, Cartesian coordinates provides more meaningful representation than long-lat coordinates. For instance, most application writers can readily tell that two objects at coordinates $(1m, 1m)$ and $(1m, 2m)$ are one meter apart whereas it may not be so easy when the coordinates are expressed in terms of longitudes and latitudes (unless, of course, you are an expert cartographer).

In our implementation of the SIS service, we chose to divide each floor into its own reference frame. Hence, the Cricket beacon coordinates are configured using the same coordinate system as defined in the corresponding floorplan drawing. This is a natural subdivision because beacons on the same floor are already assumed to have the same z component value (see Section 2.5.5). Also, each floorplan is a two-dimensional representation of the spatial boundaries within a given floor. Dividing reference frames *by* floors allows us to assume a 2D representation for all coordinates in each floor map. Otherwise, if reference frames were divided *across* floors, we would require a 3D coordinate representation to distinguish the floor on which a point is located.

Dividing reference frames by floors has other advantages as well. First, we do not need to find an arbitrary reference point during deployment; we simply follow the convention as laid out in the architectural floorplan. Because the reference frame is subdivided in the same way as the spatial information map, the coordinate system is switched at the same time as a new floor map is downloaded. Furthermore, if the original floorplans adopt a different coordinate system between different floors, consistency between the beacon coordinate and the floorplan coordinate is still maintained. Second, beacon coordinates can be configured by measuring its position with respect to a reference point of a physical feature that appears on the floorplan. Assuming that the physical feature is built according to the floorplan within reasonable accuracy, the absolute coordinate of the beacon can be computed by adding an offset relative to the reference point's coordinate.

We might need to perform coordinate transformations when we introduce different reference frames. For example, augmented-reality applications might compare relative positions of different objects located on different floors to determine which ones are within range from the user's current position. In this case, the coordinates of each object must be converted to a common coordinate system before the range comparison can be made.

Fortunately, no such coordinate transformation is required to support navigation applications. The only operation listed in Section 3.1 that might require position information from two different coordinate systems is the shortest path computation, which consists of computing the shortest reachable path and the associated path cost between any given origin and destination. We claim that all shortest path computations may be performed without a coordinate transformation. This is trivially true when the origin and destination are located on the same floor. When the origin and destination are located on different floors, the shortest path computation is divided into two smaller shortest path computations (see Section 3.2). Each sub-path is computed entirely in its own coordinate system. Furthermore, the two sub-paths meet at the linkage point. Thus, the total path cost can be computed by adding the path cost from each sub-path to the path cost of traversing the linkage point. Hence, we proved that neither the sub-path computations nor the path cost require coordinate transformation.

3.5 Map Generation and Representation

An important part of the SIS service is the automated generation of spatial information. We have leveraged part of the work done by Drury [9] in developing tools to convert MIT architectural floorplans into floor maps. This section describes the process of map generation and the data structure used to represent floor maps.

3.5.1 Map Generation Process

The basic process of generating floor maps is shown in Figure 3-1. It begins by converting the architectural floorplans saved in DXF (Drawing Interchange Format) [2] into UniGrafix [21]. UniGrafix is a convenient format for manipulating graphical elements programmatically. The elements saved in the DXF and UniGrafix consist of vertices and line segments without sufficient semantics information to indicate how they compose objects found inside a building such as walls, doorways, staircases, and elevators. We apply a series of feature-recognition algorithms on the UniGrafix file to extract such objects, which are then used to annotate the floor map. Next, we apply the Constrained Delaunay Triangulation (CDT) [6] to produce a planar graph that binds the spatial relationship between different structural features of the floorplan. Applications can use this planar graph to extract all the accessible paths in a given floorplan. The final process extracts space descriptors from the original floorplan and use them to label each element in the floor map. The result of the map generation process is an annotated floor map that models the spaces, structural features, and accessible paths of a given floor of a building. The following sections describe the details of each map generation process.

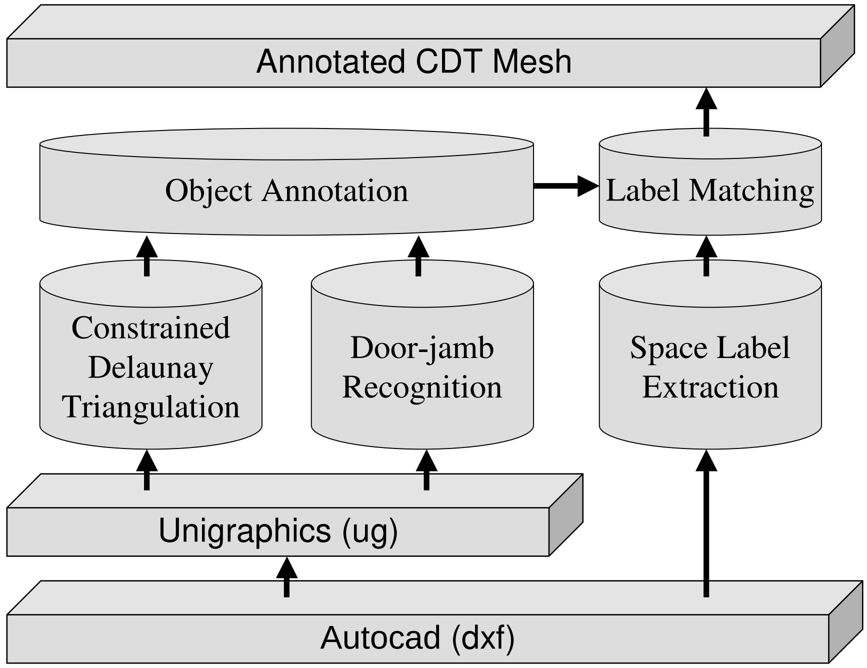


Figure 3-1: Process for floorplan conversion.

DXF to UniGrafix

Most architectural floorplans are drawn using CAD tools such as AutoCAD. Floorplan elements are commonly represented by graphical primitives such as vertices, segments, polylines, circles, arcs, and text. These primitives are organized into different layers. For example, the MIT floorplans have layers named "A-WALL-CORE", "A-WALL", and "A-AREA-IDEN", which respectively name the group of primitives that represents load-bearing structures, ordinary walls, and labels of rooms, staircases, and other objects. Unfortunately, the layering organization depends only on the convention used by the draftsperson, so it is not entirely useful for object extraction. For example, there is no separate layer to identify things like staircase and doorway elements.

A common file format used by CAD tools to save floorplan information is the Data Exchange Format (DXF). DXF files are encoded in ASCII and are grouped into code/value pairs. A group code describes the meaning of a group value. For example, a vertex object is represented by a specific group code followed by a coordinate value. This makes DXF files easy to parse. However, we convert the DXF files into UniGrafix because a set of powerful APIs have been developed to manipulate UniGrafix elements in C [15]. This lets us develop a set of tools for extracting physical features and accessible paths in the later stages of the map generation process.

To convert DXF files into UniGrafix, we use a utility called `acad2ug`, which was originally part of the BMG (Building Model Generator) [16] toolkit developed at UC Berkeley and later modified by Drury [9] to convert MIT floorplans.

Feature Recognition

It is important to recognize objects such as doorways from graphical primitives so that applications can compute a customized shortest path metric. Due to lack of embedded

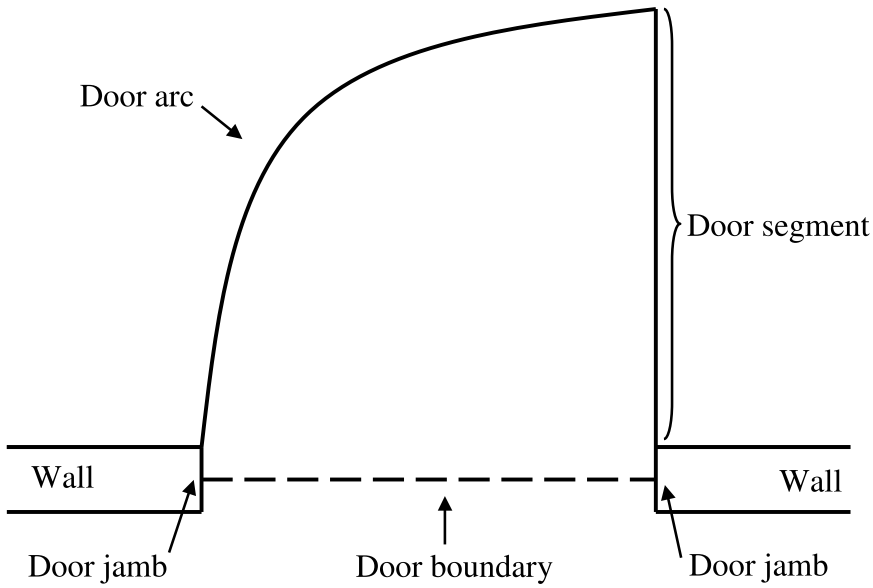


Figure 3-2: Components of a doorway in a typical CAD drawing.

semantics information, we must manually determine a regular pattern as drawn on the floorplans to recognize a class of objects. The pattern for doorways is relatively easy to recognize because it is the only object among almost all MIT floorplans that is represented by an arc in the DXF file. Unfortunately, this is not good enough because certain applications often need to know the exact doorway boundary (e.g., if it wants to determine the space boundary of a room).

To extract a doorway boundary, we need to recognize the surrounding door jambs. Figure 3-2 shows the primitive composition of a doorway in the floorplan. Sometimes, the door jambs may be connected to the door arc or represented as a separate segment that belongs to the A-WALL layer. In this case, we find one of the door jambs by searching the entire A-WALL layer for a segment with an end point that matches the one on the door arc. A similar technique is used to search for the other door jamb. A simple $O(n^2)$ algorithm can accomplish these tasks quite easily.

We currently implement an algorithm to recognize doorways only. Other objects such as staircases and elevators do not have to be recognized when they are labeled. Otherwise, they can be recognized by implementing heuristics similar to doorway recognition.

Path Extraction

To support navigation applications, we must find all the accessible paths between all origins and destinations in a given floorplan. We can divide the floorplan into two types of spaces: *valid* and *invalid* spaces. The valid space represents the area that can be occupied by a person whereas the invalid space is the area occupied by a physical obstacle such as a wall. Hence, the desired accessible path should not intersect any invalid space and should lie approximately in the center of valid space; it should keep a comfortable distance away from the walls and other obstacles.

One simple solution to the problem of extracting accessible paths is to apply the constrained delaunay triangulation (CDT) to subdivide the floorplan into triangles (see Figure 3-3). The CDT algorithm can be applied to a general set of input segments known as

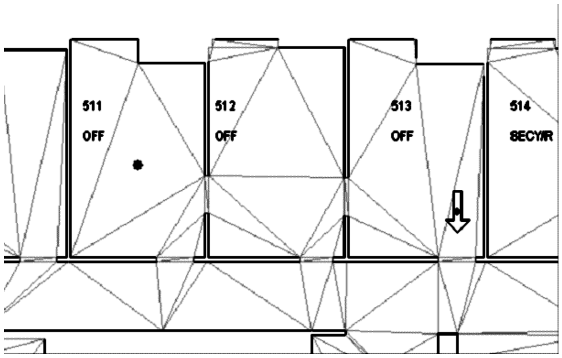


Figure 3-3: The figure shows a triangulated floor map, which shows the constrained (dark) and unconstrained (light) segments. For clarity, the unconstrained segments we show are the ones in the valid space only.

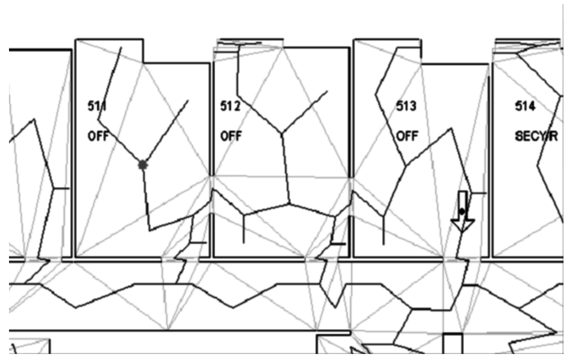


Figure 3-4: The figure shows the extracted accessible paths from the same triangulated floor map. The accessible path connects the midpoint of the triangles in valid space.

the *constraining segments*. The result is a triangulated planar graph that includes all of the input segments, which define the boundary between valid and invalid space. The remaining *non-constraining segments* are added by the CDT algorithm to form triangles, which are entirely contained in either space. We traverse the planar graph starting at a seed point that is in valid space and join the centers of two adjacent triangles that do not share a constraining segment. Hence, the centers of triangles become the waypoints of accessible paths if it is traversed by the algorithm. These waypoints form a desirable path as they lie approximately in the center of free space (see Figure 3-4). The data structure used to represent the CDT planar graph and the accessible paths is described in Section 3.5.2.

We need to address several other issues during path extraction. As mentioned before, coordinates have finite precision. Thus, two segments that are supposed to form a corner for a room may not exactly meet at an end point. The resulting gap can cause a “leakage” into the invalid space from which paths may be inadvertently extracted. We solve this problem by defining a minimum gap length for the planar graph traversal. If the gap is narrower than minimum allowable width occupied by a person, the algorithm is restricted from laying a path across it.

Another problem arises when we extract paths from the ground floors of buildings. The ground floors have portals that lead to the outside of the building. Thus, the path extraction can “leak” outside the building, creating paths that lead to random spaces. We can limit this problem by defining a tight bounding box encompassing the outline of the bundling.

A final problem that needs to be addressed is the imperfections in the acad2ug tool and the feature-recognition algorithms. Some wall segments are sometimes discarded because they were incorrectly recognized as extraneous lines by the acad2ug tool, while some other features may not be recognized due to inconsistencies between drawings. Consequently, invalid paths are extracted or elements are annotated with incorrect values. Handling errors and special cases are very difficult problems and they currently require manual intervention. We hope that future CAD tools will embed standardized semantics information to help us reduce the complexity of this problem.

Labeling

Because UniGrafix does not have an expression to represent textual information, the floorplan space labels are not propagated to UniGrafix. Instead we developed a separate tool based on `acad2ug` for extracting labels and their coordinates from the DXF floorplan.

For operations 4 and 5 described in Section 3.1, it is helpful to annotate each planar graph element with a space label. Since the label assignment depends on the space boundary, we use the following simple heuristic for labeling. First, we use the coordinate C_L of the space label L as the seed for starting a planar graph traversal. We perform either a breadth-first or depth-first traversal, annotating each planar graph element with L along the way. The traversal continues until it hits a wall or doorway boundary. Since almost all rooms are completely bounded by walls and doorways, this heuristic correctly outlines the space boundary and labels each planar graph element. Other spaces may be subdivided along imaginary boundaries (e.g. open lounge areas connected to multiple hallways). Our labeling heuristic does not handle such cases.

3.5.2 Data Representation

Our floor map is represented by a map descriptor, a table of space labels, and an annotated quadedge data structure. The data structures for each floor are stored in a separate *map file*. The specification of the map file is documented in Appendix B.

Map Descriptor

The Map Descriptor contains general information about the floor map such as the version, the unique map identifier, and the unit used in the coordinate values.

Space Label Table

The Space Label Table is used to store all the space labels and their coordinates extracted from a floorplan from the Labeling procedure described in the previous section. Sometimes, a reference arrow is used in the floorplan drawings to point a label into a target space (e.g., a room) that is too small to fit the label. In this case, we insert the target point's coordinate into the table, instead of inserting the label's coordinate.

The coordinate of a label defines the center reference point of a given space (see Section 3.1). Applications may use the space label table to look up a space's reference point.

Quadedge

Recall that the physical structure of a floor is represented by a CDT planar graph. From the CDT planar graph, we extract the accessible paths by walking through centers of triangles.

Define a face to be an empty region in a planar graph G. Hence, a face can be an area bounded by a set of segments or it can be the entire area that is outside of the planar graph. The *dual graph* of G has a vertex for every face in G. A segment joins two vertices in the dual graph if the two corresponding faces share a segment in the primal graph. Figure 3-5 shows the primal and dual graphs for a planar graph that has a single primal triangle. Note that dual vertex 2 is in the outside face of the primal graph. Also, observe the duality property that there is a primal vertex for every face in the dual, and vice versa.

It turns out that the graph of accessible graph is a subgraph of the dual graph of the CDT planar graph. Hence, we can statically represent the CDT planar graph and its dual

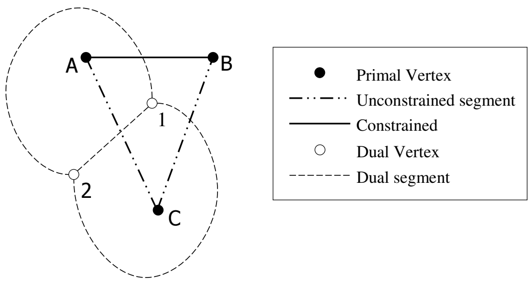


Figure 3-5: The primal and dual planar graph consisting of a single primal triangle.

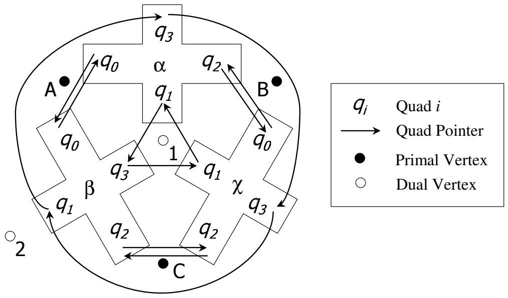


Figure 3-6: The corresponding quad edge representation for the primal triangle.

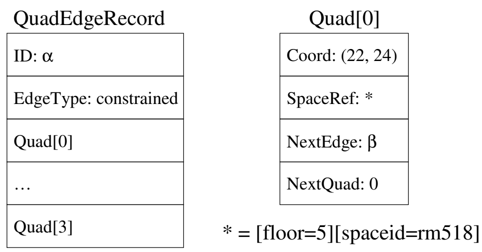


Figure 3-7: The data structure representing quadedge α.

using a single data structure called the *quadedge* [11]. At run time, applications can start at a seed point in valid space within the planar graph and traverse the subgraph of accessible paths to compute the shortest paths. Therefore, the quadedge data structure serves as a compact and elegant representation that supports both map renditions and shortest path computations.

A quadedge representation of the planar graph in Figure 3-5 is shown in Figure 3-6. A quadedge simultaneously represents a primal and a dual segment, whose end points are represented by two opposite quads, q_i. Hence, the primal segment $\overline{AB}$ on the triangle is represented by the opposite quads $\overline{q_0 q_2}$ in quadedge α. Quads that share the same vertex are joined by an *edge ring* of quad pointers. The ring of quad pointers can be viewed as a linked list of quads that share the same vertex. Notice that there is an edge ring $(\alpha : q_3, \quad \beta : q_1, \quad \chi : q_3)$ for dual vertex 2.

The data structure representing quadedge α is shown in Figure 3-7. Each QuadEdgeRecord contains a unique ID, an EdgeType, and a record for each quad. The ID field is used to reference a quadedge. The EdgeType field determines the type of a primal edge, which can be a constrained, non-constrained, or a doorway boundary. Each quad of a quadedge is represented by a Quad record, which contains the coordinate of the planar graph vertex it represents, and the space in which the associated vertex is located. For example, the quad record in Figure 3-7 shows that vertex A is located at $(22, 24)$ in room 518. To save memory, we do not have to store the full space descriptor in each quad record. Instead, the SpaceRef field stores the row number corresponding to the appropriate space descriptor entry in the Space Label Table. Finally, the NextEdge and NextQuad fields are used to reference the next quad in the edge ring link list.

Given this data structure, an algorithm can traverse between quadedges by rotating around edge rings and jump between the primal and dual planar graphs by rotating to the *next* quad in the quadedge. An algorithm traverse from one vertex to another on the same edge by rotating to the *opposite* quad the quadedge.

In our implementation, we used a software package called *cdt* by Lischinski [17] to compute a CDT. It conveniently represents the output of CDT in the quadedge data structure. It also implements a set of powerful primitive functions defined in [11] to help applications traverse quadedge planar graphs. We ported the quadedge data structure and the graph traversal primitives into Java (with slight modifications for handling annotations) to support our navigation application in traversing the floor maps.

3.6 SIS Server API

The SIS Server API is simple. Applications simply send a map identifier, which consists of a building identifier and a floor number, to fetch the desired floor map. The map is transmitted over TCP/IP for reliable delivery. The SIS service may be discovered via a service discovery mechanism such as INS [1]. Alternatively, the SIS server's port and address may be advertised by the Cricket beacons.

The SIS server may also be used to store and serve beacon configuration files for the Beacon Mapping Module (see Section 2.5.2).

3.7 Chapter Summary

We designed and implemented a Spatial Information Service (SIS) to provide CricketNav with the necessary spatial information for finding the shortest path between any pair of origin and destination, rendering a map to show the user's current position, and computing the direction of the navigation arrow. The spatial information of a building is divided into *floor maps*, which contains information about the accessible paths, the space-dividing features, and the coordinate system of a single floor. Certain map elements such as doorway boundaries are annotated so that applications can readily identify them and apply their own metrics for computing the shortest path.

To reduce deployment cost, we developed and implemented an automated process for extracting floor maps out of existing CAD drawings. The process consists of four steps. First, we convert the CAD drawings into a convenient data format for manipulating the graphical objects in the original floor plan. Second, we apply a set of features recognition algorithm to extract and annotate important objects for navigation, such as doorways, staircases, and elevators. Third, we feed all the segments in the original floor plan into the Constrained Delaunay Triangulation (CDT) algorithm to produce a planar graph that resembles the original floor plan except it is filled with triangles. The triangular fillings bind the spatial relationship between each segment, which allows us to extract a network of accessible paths that lay in a comfortable distance away from walls and other obstacles. The final process is to identify the space to which each planar graph vertex and segment belong and label them with the appropriate space descriptor. Using this map extraction recipe, we generate floor maps for every floor of a building and store them in a SIS server so that CricketNav may fetch them as needed over the network.

Chapter 4

Design of CricketNav

With the location and spatial information services in place, CricketNav can now determine the user's current location within a building and navigate the user towards the desired destination. In this chapter, we discuss the design of CricketNav's user interface and the algorithms used for matching the user's position in the floor map, planning the shortest path to the specified destination, and orienting the direction of the navigation arrow. We also describe how CricketNav adapts to the quality of information supplied by the location infrastructure and spatial information service.

4.1 User Interface

We envision a mobile navigation system that is interactive and user-friendly. CricketNav provide users feedback about the delay experienced by the location system and allows users to manually specify their current location. CricketNav also lets users know when they have wandered off course so that they can correct their direction promptly. We have implemented CricketNav using Java 2 and Java Swing components so that it can run on a variety of platforms such as the Compaq iPAQ.

Figure 4-1 shows a screen shot of our CricketNav prototype. It is shown in 1:1 scale to the size of a typical PDA screen (320x240 pixels). The user interface consists of an area that renders the section of the floor map in the proximity of the user's location, an arrow indicating the user's current position and navigation direction, a disc marking an intermediate point in the path and another disc marking the destination. The user may zoom or scroll the rendering area as desired.

CricketNav refreshes the navigation arrow whenever the user changes the destination or whenever the CricketServer provides it with a location update. Because the update rate varies with the rate of receiving beacon samples at the Cricket listener's position, the user will experience varying delays from CricketNav. We use the "Last Update" field to indicate the number of seconds since CricketNav last received a location update from the CricketServer. This provides useful feedback to the user about the delay currently experienced by the location system. For instance, a long location update delay informs the user that the navigation direction being displayed may be stale. In this case, the user may try to find a location that provides better "Cricket reception" or faster location update rate to improve the accuracy of the navigation arrow.

To prepare the user about the upcoming turns in the path, CricketNav gives the next direction change before the user approaches the upcoming turn. However, giving overly-

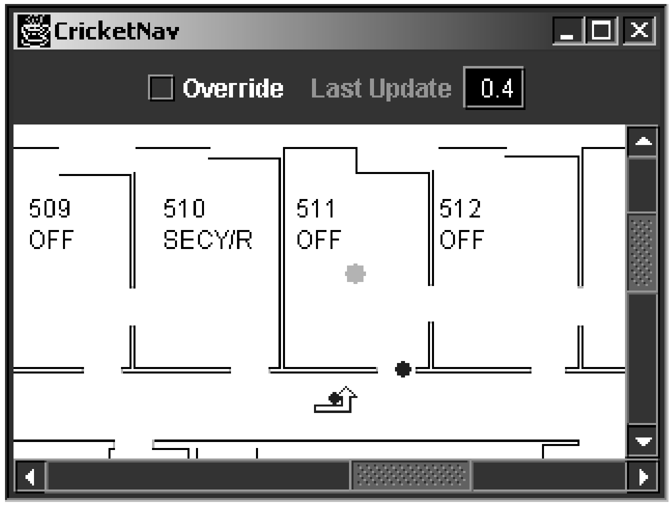


Figure 4-1: CricketNav prototype. The snapshot is shown in 1:1 scale to the size of a typical PDA screen (320x240 pixels).

anticipated directions can be misleading and ambiguous. For example, when asked to make a certain turn into a room, the user might have to decide between two adjacent doorways leading to different rooms. CricketNav disambiguates this by placing a colored disc at an intermediate point in the path to indicate the correct direction.

When CricketNav detects the user has wandered off course, it shows a small, red arrow to indicate the (wrong) direction that the user is currently heading (see Figure 4-2). The navigation arrow is refreshed to point at the direction leading back to the original path. From the two arrows, users can figure out that they have wandered off course and they can compare the two arrows to find out how to correct their heading.

Although not in our current implementation, CricketNav's user interface can be extended to handle the case where a user takes an elevator and exits on the wrong floor. In this case, CricketNav can show a message box to notify the user.

We anticipate that the Cricket location system may occasionally provide erroneous position updates and cause CricketNav to report wrong navigation directions. Hence, we let users specify their correct location manually. In CricketNav, users can check the "Override" box and click on the location of the map that corresponds to what they think is their actual current position.

Finally, to specify destinations for CricketNav, we can implement a menu-based interface to let a user browse through all of the spaces in a given building. However, this can be very cumbersome due to the limited size of the screen in a typical handheld device. Instead, we are working on integrating speech recognition into CricketNav to let users specify a destination by voice.

4.1.1 Fetching Floor Maps

In our implementation, the set of floor maps of a given building is stored in a centralized SIS Server (see Section 3.6). We assume the network location of the SIS Server is known; it can either be pre-configured or advertised by the Cricket beacons. CricketNav extracts the floor value from the space descriptor currently reported by CricketServer and compares it with the value that corresponds to the floor map it currently possesses. If the value differs,

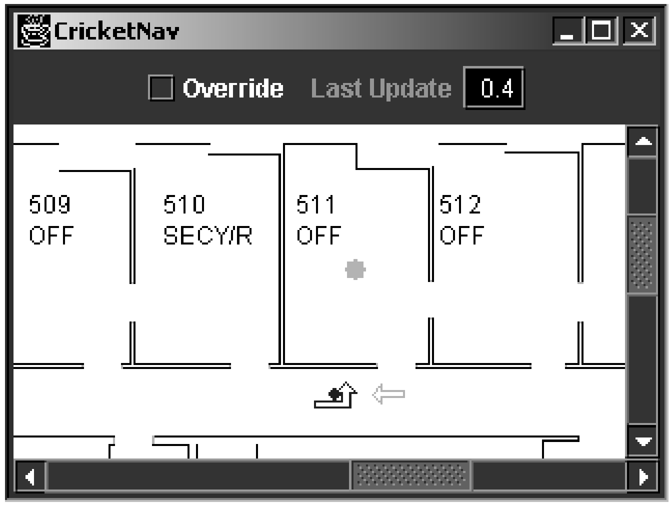


Figure 4-2: A correction arrow appears when the user wanders off the wrong direction.

CricketNav sends a message to the SIS Server to request a new map for the current floor.

4.2 Algorithms

To find the navigation direction, CricketNav locates a point in the floor map that matches the user's current position, finds the shortest path between the current position and the destination, and orients the arrow direction based on the shape of the path at the current position. In the next few sections, we briefly describe the algorithms used by CricketNav to perform these tasks.

4.2.1 Point Location

Point location is the process of finding a vertex in the quadedge graph that corrresponds to a given query coordinate. For CricketNav, the coordinates of interest are the user's current position and her destination. We find the quadedge vertex that has the closest matching coordinate (in Euclidean distance) by performing a breadth-first search from a seed point in the valid space. Alternatively, we can use the quadedge "walking" algorithm suggested in [11], which basically starts at a seed point in the quadedge graph and "walks" towards the query point. The walk steps closer to the query point by traversing over an immediate graph segment that defines a line dividing between the query point and the current point in the traversal. It is easy to see that the point can be located in time proportional to the distance between the seed and the query point, which is at most the diameter of the graph of the floor map.

As mentioned in Section 2.3, the Cricket location system directly reports both space and position estimates. In this case, the position estimate is used as the query point to locate the user's position on the map. The uncertainty of position estimates sometimes put the position estimate in the incorrect space boundary (see Figure 2-1). CricketNav detects this problem by checking if the reported space matches the space located by the point location algorithm.[1] When the problem does occur, CricketNav starts a Euclidean breadth-first search from the reported position until it finds the first quadedge vertex in the reported space.

Finally, we handle the case when CricketNav does not receive position estimates from Cricket. This can happen when the user walks into a location with fewer than three Cricket beacons within the listener's range. However, if Cricket continues to report the user's space location, CricketNav can mark the user's location inside the reported space of the map. The precise marking position within the reported space can be arbitrary. In our implementation, CricketNav chooses the marking position to be the space descriptor's reference coordinate listed in the floor map's Space Label Table. The resulting reference coordinate then becomes the input to the point location algorithm described above.

4.2.2 Path Planning

Once the point location algorithm have located the user's origin and destination on the floor map, CricketNav needs to search for the shortest path between them. If the origin and the destination are on the same floor, we apply the shortest path algorithm directly. Otherwise, the search must be divided into parts: we find one path from the origin to the elevators on that floor, and the other path from the elevators in the destination floor to the destination point.[2]

For the choice of shortest path algorithms, we can either use Dijkstra's shortest path algorithm or the A^* algorithm, which uses a simple heuristic to reduce the search size [12]. Because the A^* algorithm does not always apply to more general distance metrics, we decided not to use it. Hence, CricketNav uses Dijkstra's algorithm.

In our current implementation, we use the simple Euclidean distance metric to perform the shortest path calculation. However, we assign longer distance values for a segment that crosses a doorway boundary dividing between two adjacent rooms. This tends to favor a path of "least resistance". It is preferable to take a longer path that traverses a hallway than a shorter path that traverses adjacent offices (see Figure 4-3).

Finally, we note that the user's position is updated periodically by the CricketServer. It is undesirable to recalculate the shortest path every time an update is received. Instead, we keep the original path when the destination remains the same. As long as the user remains on the original path, we do not have to perform a new search. However, this optimization requires detecting when the user has wandered off the original path. We determine the user has wandered off a path when she is at a threshold distance away from the path. When the user is in between the path and the threshold, we perform a localized path planning, which constructs a path from her current position to the nearest point on the original path (see Figure 4-4). If the user has wandered off the original path, CricketNav recomputes a new route to the specified destination.

4.2.3 Arrow Direction

For simplicity, CricketNav selects a navigation arrow from a set of twelve angled and straight arrows shown in Figure 4-5. There are four groups of three arrows representing the following

[1] As mentioned in Section 3.5.2, every quadedge vertex is annotated with a space descriptor.

[2] We assume that the elevators (or staircases) reach all floors of the building. Otherwise, we need extra information from the floor maps to indicate which set of elevators (or flight of stairs) should be taken to reach a particular floor.

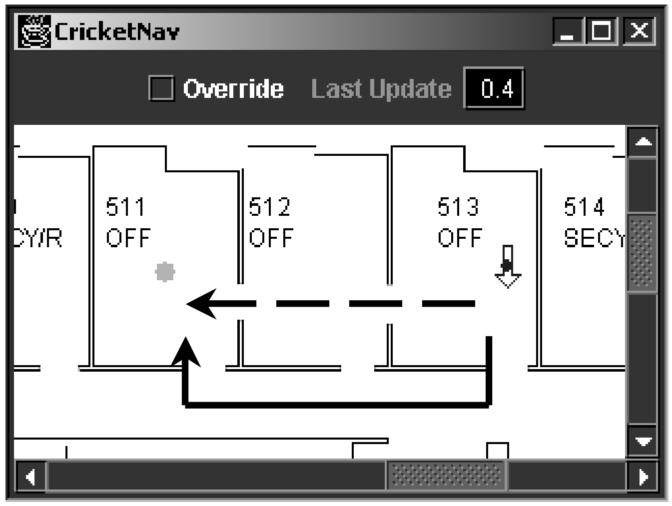
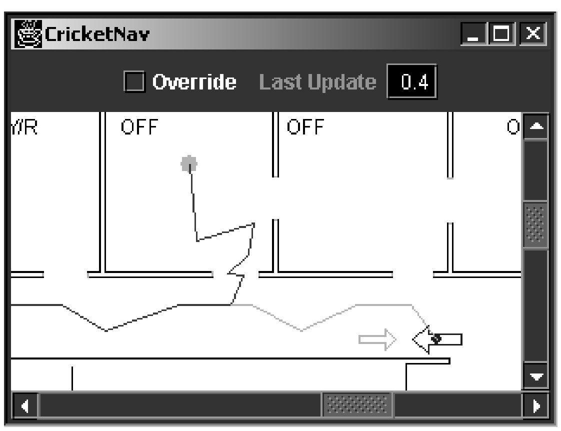


Figure 4-3: Two possible paths leading to destination. The solid path is generally preferred.



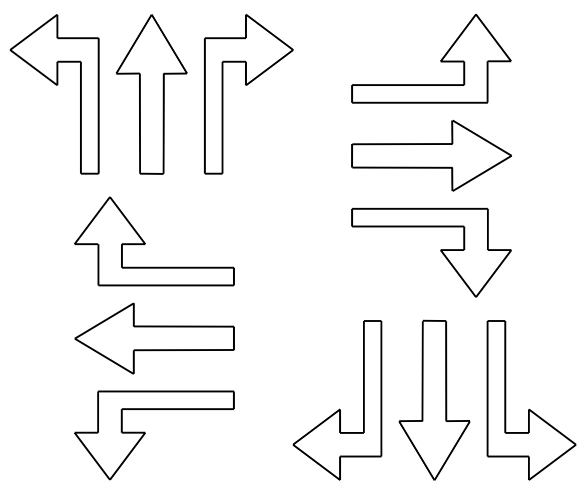


Figure 4-4: A local path (light) is constructed when the user wanders away from the original path (dark).

Figure 4-5: Three arrows pointing upwards. The arrow orientations are left-turn, straight, right-turn.

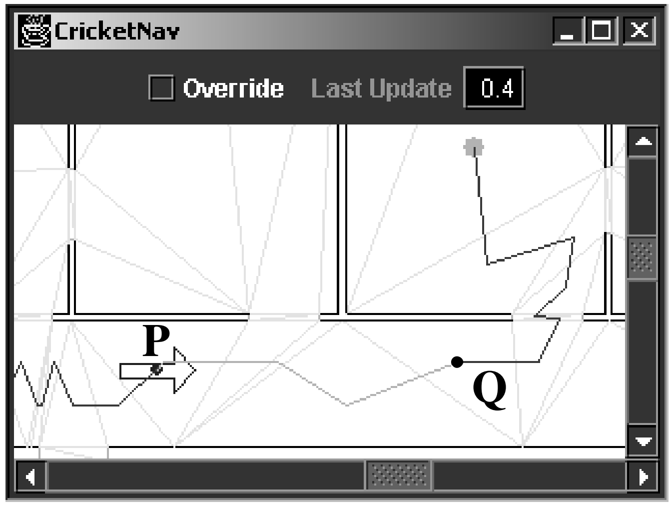


Figure 4-6: The shape and direction of the path between points P and Q are used to determine the orientation and direction of the navigation arrow.

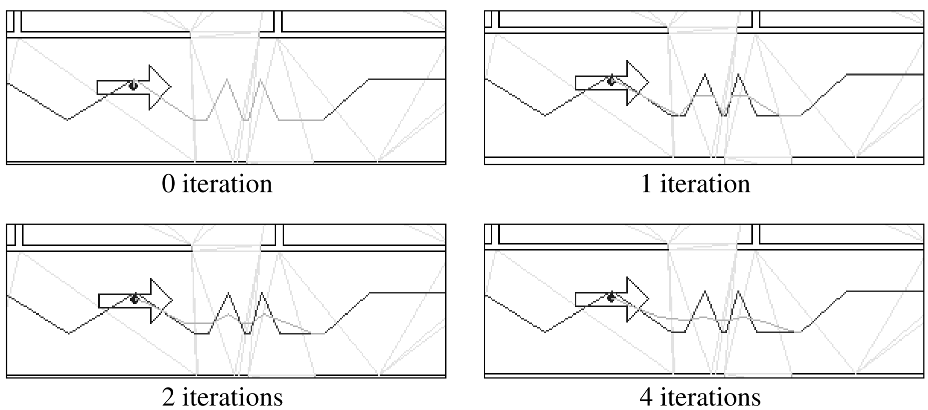


Figure 4-7: The effect of running different iterations of path smoothing.

orientations: *left-turn*, *straight*, and *right-turn*. The rest of the twelve arrows are copies of these three arrow orientations rotated by 90°, 180°, and 270°. CricketNav selects the appropriate navigation arrow based on the path's shape and orientation at the user's current position.

To help users anticipate turns, CricketNav selects an arrow by examining the local path that extends from the user's current position P up to a point Q at a threshold length into the path as shown in Figure 4-6. The threshold length controls the degree of anticipation given to the user. As the threshold length increases, point Q extends further into the path, which in turn allows CricketNav to detect upcoming turns sooner.

Next, CricketNav calculates the slope of the segment joining P and Q. The sign and magnitude of the slope (i.e. if $|slope| > 1$) determines the arrow direction.

Finally, CricketNav determines the arrow orientation by examining the shape of the path between P and Q. It does this by first finding the largest triangle that can be formed by P, Q and a vertex M, where M is a point on the path between P and Q. Then CricketNav selects the arrow orientation based on the direction of the turn made by the angle $\angle PMQ$.

As a consequence of triangulation, accessible paths in a floor map are often jagged (see Figure 4-7). This may cause the arrow selection heuristic to fail. To mitigate this problem, we apply a path-smoothing algorithm. We can smooth a path by replacing every successive pair of path segments with a new segment that joins the mid points of the segments that are being replaced. This procedure can be repeated iteratively to improve the smoothing effect. Figure 4-7 shows a path being smoothed over a number of different iterations.

4.2.4 Rendering

After CricketNav determines the user's navigation direction, it needs to render all the navigation information on the screen. CricketNav needs to render the navigation arrow overlaying the portion of the map in which the user is located. Thus, we need to render only the objects that intersect the region of interest. Since there can be a large number of objects in the floorplan, it is inefficient to use a brute-force approach that performs an intersection test for every object in the floorplan. Instead, we take advantage of the information embedded in quadedge graph. Recall that every segment in the original floorplan induces exactly one constrained edge in the quadedge graph. Thus, all CricketNav needs to do is to walk the quadedge graph from the vertex corresponding the user's current location and render all the constrained edges traversed along the way. The walk expands until all the edges and vertices contained within the rendering region has been traversed. This algorithm is more efficent than the brute-force approach because the total running time is proportional to the size of the map rendering region.

4.3 Chapter Summary

CricketNav navigates users by using location updates from the Cricket location system and floor maps from the Spatial Information Service. We note that both support systems report imperfect data. The position estimates are associated with uncertainty and delay, while the shape of the path provided by the floor maps are prodded with tight twists and turns. Hence, CricketNav must be carefully designed so that it remains usable and robust in the face of flawed data.

In this chapter, we describe a number of designs that help CricketNav to adapt to the quality of information supplied by the location and spatial information support systems. For instance, CricketNav copes with irregular path shapes by applying a path smoothing algorithm. Also, the user interface provides feedback about the current location update delay so that the user learns that the navigation direction being displayed on the screen may not be accurate. When the reported space locations and position estimates are ambiguous, CricketNav makes a best-effort attempt to find the nearest position in the floor map that corresponds to the reported space location. In case this procedure fails to find the correct user location, CricketNav allows the user to override the location updates.

To conclude, CricketNav improves usability and robustness through permitting feedback and user interaction, and through *combining* information from the Cricket location system and the Spatial Information Service. Because it is practically impossible to expect flawless data from any location or spatial information systems, we believe that the design strategies mentioned in this chapter will be applicable for implementing many other location-aware applications that use either of these services.

Chapter 5

Future Work and Conclusion

We have implemented a working system for indoor mobile navigation using the designs described in this thesis. As a result, the Cricket location system now has an extensible, modular software architecture for processing and distributing location information. Through analyzing real data samples, we discovered that the accuracy of Cricket is mostly affected by the beacon multiplicity for delay-sensitive applications. In addition, this thesis outlines an automated process for extracting floor maps out of conventional CAD drawings and a data representation that enables path planning and map rendering. These floor maps are useful not only for CricketNav but also for numerous other location-aware applications such as resource finders. Finally, we describe the design of CricketNav, which demonstrates how applications might handle imperfect data supplied by the location and spatial information services.

Although we have a working indoor navigation system, we still need to resolve at least three challenges before CricketNav can be widely deployed. First, a full-scale deployment will likely involve a large number of Cricket beacons deployed throughout a building or campus. Hence, we need to find a cost-effective solution to monitor and maintain a large collection of Cricket beacons. Second, there is still a significant amount of manual work involved in handling exceptions and errors that occur during map extraction. Although much of this work can be reduced if the future CAD tools will embed the necessary semantic information in their data files, map extraction for existing buildings is still problematic. Finally, an ideal personal navigation system should work everywhere and navigate its user seamlessly through different buildings and outdoor environments. Enabling seamless and global personal navigation might require designing a spatial information service infrastructure that is scalable and universally accessible, and a mechanism that supports seamless handoffs between different types of location sensors such as GPS and Cricket.

CricketNav is one of the first complete, low-cost, and configurable indoor mobile navigation system. We hope the work of thesis will motivate further development in location-aware applications and ubiquitous computing.

Appendix A

CricketServer 1.0 Protocol

The CricketServer1.0 Protocol allows applications to obtain processed information from the Cricket listener software, the CricketServer. The information is packaged in a type-length-value format so that the content of the packet is self-describing. This feature allows future extensions to the packet format while preserving backward compatibility to existing applications.

A.1 Registration

Applications REGISTER with the Server by sending to the Server at port 5001 a TCP byte stream containing the following null terminated string:

```
register
```

The Server uses the source address and port in the TCP packet to return TCP RESPONSEs to the Application. Whenever the Server receives an update from one of the CricketServer processing modules, it will send a packet to each of the registered Applications. An update may span across multiple packets. This is indicated by the fragment bit as described in the next section.

A.2 Response Packet Format

The RESPONSE packet is encoded in UTF-8 (ASCII compatible). A generic packet consists of a cricket server version token, and a payload made up of a sequence of "|"-separated fields. We use the symbol "+" to denote concatenation and "*" as a modifier to denote zero or more instances of a token.

For example, a packet may have the following format:

```
cricket_ver|timestamp||type|length|value||type|length|value|...
```

Note that the character "|" is reserved as a field separator and therefore it must not be within the type or length field. However, this character can be safely used in the value field as the length field can be used to distinguish whether "|" is part of the field value.

Note that nested fields are possible. For example, given a payload:

```
|type1|length1|VALUE||type2|length2|value2||type3|length3|value3...
```

packet = cricket_ver + "|" + timestamp + "|" + fragment + "|" + payload + "\n"
cricket_ver = Specifies the version of the protocol format. The current value for this field is "CricketServer1.0".
timestamp = (ASCII long value) time (ms) when the packet was sent out
fragment = an ASCII 0 or 1, where 1 means this packet is a fragment of a complete message (e.g. part of an array). The client should reassemble all packets with a 1 until a packet with 0 fragment value is received.
payload = field*
field = "|" + type + "|" + length + "|" +value + "|"
type = an UTF-8 string denoting the type of the field value
length = a positive integer (UTF-8 encoded) that denotes the length of the value string (in number of bytes), excluding the field separators "|" around the value field.
value = field* 'or' ASCII-string

The VALUE field can be nested as:

```
|type11|length11|value11|type12|length12|value12|
```

Note that length1 in the payload denotes the entire length of the VALUE field, including the "|" separators in between but excluding the "|" separators around VALUE.

The server and the client MUST agree on how to parse and interpret the ASCII-string in the value field for each field type. The client MAY ignore and skip over any field(s).

A.3 Field Types

Table A.1 lists the definition of various field types used in the CricketServer 1.0 Protocol. The types labelled "(nested)" denotes a type where its value field contains a nested sequence of fields.

Table A.1: Field type specification for CricketServer 1.0 Protocol

Type	Interpreted Value	Description
array:dist_est (nested)	array of dist_est	An array of distance estimate records representing the beacons heard by the listener, sorted in ascending distance order.
array:stat_est (nested)	array of dist_est	Same as above except each record in the array contains statistical information about the distance estimate for each beacon. The array is sorted in ascending distance order.
cur_space (nested)	a dist_est	A distance estimate record representing the closest beacon heard by the listener.
dist_est (nested)	record of { space, id, dist, dist_stat, pos, last_update }	A distance estimate record representing a beacon heard by the listener. NOTE: a dist_est record can contain any combination of listed sub-elements under Interpreted Value.
space	ASCII-string	The string of the space identifier in the beacon.
id	non-negative decimal integer	The beacon identifier within a space.
dist	double	A distance estimate between the beacon and listener.
dist_stat (nested)	record of { dist, median, mean, mode, max, min, stddev, var }	Statistical information about the the distance estimate wrt a beacon.
device_pos (nested)	record of { pos, array:dist_est }	The coordinate of device and the *calculated* distances of each beacon used to solve the device's coordinate.
pos (nested)	record of {x, y, z}	The coordinate of device/beacon.
last_update	64-bit long in ms	In dist_est, the time of last distance sample received for a particular beacon.
median	double	median
mean	double	mean
mode	double	mode
max	double	maximum
min	double	minimum
stddev	double	standard deviation
x	double	x coordinate
y	double	y coordinate
z	double	z coordinate

Appendix B

Spatial Information Map File Format

As mentioned in Section 3.5.2, the Spatial Information Map File contains the data of a single floor map. The map file is formatted in ASCII and it is divided into the three parts in the following order: a Map Descriptor, a Space Label Table, and a Quadedge Graph.

B.1 Map Descriptor

The map file begins with a Map Descriptor, which consists of three data fields recorded in a separate line:

```
version
map_identifier
unit
```

where

version	=	specifies the file version (currently "1.0")
map_identifier	=	unique descriptor for this floor map
unit	=	specifies coordinate unit (either "cm" or "in")

Currently, the map_identifier consists of one attribute-value pair:

```
[floor=floor_num]
```

where floor_num is an integer specifying the floor represented by the current map file.

B.2 Space Label Table

The Space Label Table begins right after the Map Descriptor. A Space Label Table consists of one or more row entries, each of which represents a space label on the original floorplan. Each row entry is recorded as a separate line, which consists of a spaceid/coordinate pair in the following format:

```
(spaceid, x, y)
```

where

spaceid	=	a space label in ASCII (see Section 3.3)
x	=	floating point value for the label's x-coordinate
y	=	floating point value for the label's y-coordinate

B.3 Quadedge Graph

Immediately following the Space Label Table is the Quadedge Graph. The Quadedge Graph consists of QuadEdgeRecord entries, each of which is recorded in a separate line. Each QuadEdgeRecord has the following format:

```
(id,type,quad0,quad1,quad2,quad3)
```

where

id	=	quadedge reference
type	=	integer value representing the type of the primal edge 0=unconstrained, 1=constrained, 2=door boundary
quad0...3	=	a QuadRecord

Each QuadRecord has the following format:

```
x,y,spaceref,next_edge,next_quad
```

where

x	=	floating point value for the x-coordinate of the quad vertex
y	=	floating point value for the y-coordinate of the quad vertex
spaceref	=	the row index of the spaceid identifying the space that contains the quad vertex
next_edge	=	reference of the next quadedge in the edge ring list
next_quad	=	integer in [0..3]; reference to the next quad of the quadedge in the edge ring list

N.B.: The spaceref to the first row entry in the Space Label Table is 0.

Bibliography

[1] Adjie-Winoto, W., Schwartz, E. and Balakrishnan, H. and Lilley, J. The design and implementation of an intentional naming system. In *Proc. ACM Symposium on Operating Systems Principles*, pages 186–201, Kiawah Island, SC, December 1999.

[2] Autocad 2000 dxf reference. http://www.autodesk.com/techpubs/autocad/acad2000/dxf/.

[3] P. Bahl and V. N. Padmanabhan. RADAR: An In-Building RF based User Location and Tracking System. In *Proc. of IEEE INFOCOM 2000*, Tel-Aviv, Israel, March 2000.

[4] Bonnet, P., Gehrke, J., Seshadri, P. Towards Sensor Database Systems. In *Proc. of the 2nd International Conference on Mobile Data Management*, Hong Kong, China, January 2001.

[5] P. Chen and J. Holtzman. Polling Strategies for Mobile Location Estimation in A Cellular System. In *Proceedings of the 48th IEEE Vehicular Technology Conference*, Ottawa, ON, Canada, May 1998.

[6] P. Chew. Constrained delaunay triangulations. In *Symposium on Computational Geometry*, pages 215–222, 1987.

[7] Cramer, O. The Variation of the Specific Heat Ratio and the Speed of Sound in Air with Temperature, Pressure, Humidity, and CO_2 Concentration. *Journal of Acoustical Society of America*, 93(5):2510–2516, May 1993.

[8] DeMers, M. *Fundamentals of Geographic Information Systems*. John Wiley & Sons, Inc., 1997.

[9] T. Drury. Generating a Three-Dimensional Campus Model. MIT Laboratory for Computer Science Undergraduate Project, May 2001.

[10] I. Getting. The Global Positioning System. *IEEE Spectrum*, 30(12):36–47, December 1993.

[11] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985.

[12] Hart, P., Nilsson, N., and Raphael, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transaction on System Science and Cybernetics*, 2:100–107, 1968.

[13] Harter, A. and Hopper, A. A New Location Technique for the Active Office. *IEEE Personal Communications*, 4(5):43–47, October 1997.

[14] Langley, R. B. Dilution of Precision. *GPS World*, 10(5):52–59, 1999.

[15] Leiserson, W. sug.h. C source code, 2001.

[16] Lewis, R. Generating Three-Dimensional Building Models from Two-Dimensional Architectural Plans. Master's thesis, Computer Science Division, UC Berkeley, May 1996.

[17] D. Lischinski. *Incremental Delaunay Triangulation*, pages 51–58. Academic Press, 1994. Available from http://www.cs.huji.ac.il/ danix/.

[18] N. Priyantha, A. Chakraborty, and H. Balakrishnan. The Cricket Location-Support System. In *Proc. 6th ACM MOBICOM Conf.*, Boston, MA, August 2000.

[19] N. Priyantha, A. Miu, H. Balakrishnan, and S. Teller. The Cricket Compass for Context-Aware Mobile Applications. In *Proc. 7th ACM MOBICOM Conf.*, Rome, Italy, July 2001.

[20] A. Schwartz. Experimentation and Simulation of the Cricket Location System. MIT Laboratory for Computer Science RSI Report, August 2001.

[21] C. Séquin. Primer to the Berkeley UniGrafix Language. http://www.cs.berkeley.edu/ sequin/CS284/GEOM/UGprimer.html.

[22] Remco Treffkorn. gpsd-1.06. C source code, 2000.

[23] Welch, G., Bishop, G., Vicci, L., Brumback, S., Keller, K., and Colucci, D. The HiBall Tracker: High-Performance Wide-Area Tracking for Virtual and Augmented Environments. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, London, December 1999.

[24] G. Welsh. Scaat: Incremental tracking with incomplete information. Technical Report TR-96-051, Department of Computer Science, University of North Carolina, Capitol Hill, NC, October 1996.