2006:283 CIV

MASTER'S THESIS

Design, Implementation and Evaluation of a Self-controlled, Sensor-reading Scorpion Robot using Ekahau Positioning System, Wireless MICA2 Motes and a Bluetooth Mulle

Daniel Larsson
Sebastian Sjödin

Luleå University of Technology

MSc Programmes in Engineering
Computer Science and Engineering
Department of Computer Science and Electrical Engineering
Division of Computer Science

2006:283 CIV - ISSN: 1402-1617 - ISRN: LTU-EX--06/283--SE

Design, implementation and evaluation of a self-controlled, sensor-reading Scorpion robot using Ekahau positioning system, wireless MICA2 motes and a Bluetooth MULLE

Daniel Larsson Sebastian Sjödin

September 13, 2006

Abstract

New state of the art technologies, that revolutionize their areas, emerges unceasingly. To sustain usability and continue evolving, these technologies should be subject to new ideas. This can be achieved in different ways, but one efficacious way is to synthesize technologies to reveal independent information about weaknesses and possible improvements. As an outcome, the need for compatibility can be analyzed, and new usage areas can be ascertained.

In accordance, this thesis is an attempt to combine a robotic device with a positioning system for office spaces, various wireless sensors, and a small embedded system which communicates using Bluetooth. These technologies have their own interfaces and runs on different platforms. The aim for the thesis is to obtain a self-controlled robotic device that reads information from sensors, stores the information, moves to the base area and uploads the information to a client.

Acknowledgments

The master thesis was performed at Monash University in Melbourne, and began in November 2005 and ended in April 2006.

Jointly, we would like to thank our prominent supervisor Arkady Zaslavsky at Monash University. Without him, this project would not been possible.

Furthermore, we are especially grateful to Kåre Synnes, our examiner at Luleå University of Technology, for his exceeding help with organizing everything in Luleå and introducing us to Arkady.

Many thanks also goes to Shonali Krishnaswamy, our cooperative supervisor at Monash University, for her insightful opinions, and Paul Hii, technical advisor at Monash University, for his efforts supplying us with any technical equipment needed.

One must also appreciate the help from Michelle Ketchen, School Manager at the Faculty of Information Technology at Monash University, Inger Niska Ekblom and Christina Hamsch, at the International Office at Luleå University of Technology.

And finally, thanks to Jens Eliasson, for his help with the MULLE.

Melbourne, 20 April 2006.
Cook Islands, 7 May 2006.

Daniel Larsson and Sebastian Sjödin

List of Figures

Contents

Chapter 1

Introduction

1.1 Problem area

This project is an attempt to combine a Scorpion[15] robot with the Ekahau positioning system[9], wireless MICA2 motes[34] and a Bluetooth MULLE[23] embedded system with sensors in order to create a self-controlled robotic device that will read data from the sensors. The sensors will be located at appointed positions.

From Ekahau, a position for the robots location and logical areas for where to read sensors can be retrieved. The sensors will give the current temperature at its location within the logical area.

To allow some form of platform independence, the programming language used throughout the project will be Java.

1.2 Purpose

The main objectives of the project are:

- To attain knowledge in the stipulated areas.
- To integrate areas and educe a software prototype to obtain depicted goal.
- To enhance experience and understanding of the Java programming language.
- To improve software documentation skills.
- To reveal independent weaknesses and improvements for the stipulated areas.
- To appraise the educed software prototype.
- To ruminate the results and ascertain improvements.

Furthermore, two important questions will be answered:

- Is there a need for current software applications and its developers to think more about compatibility?
- Will the method of combining technologies uncover improvement information more concisely than looking at a single technology independently?

1.3 Delimitations

The disposable hardware and software will limit the thesis from becoming too comprehensive. However, the educed software prototype will, as far as possible, be designed to allow extensions of other areas or technologies.

As a combination of this kind has never been conducted, the support is limited, and will influence the extent.

The focus of the thesis will be in combining technologies, and for that reason there will be no harder attempt in making the code for the prototype either neat, efficient or secure.

1.4 Outline

The thesis is outlined in the following manner:

Chapter 2 gives background details about the Scorpion robot, ER1 robot, Evolution Robotics Software Platform, Ekahau, Mica sensors and the Bluetooth MULLE. It also briefly describes all the software and tools used during the project.

Chapter 3 outlines the design and procedure for each technology. The interconnection is structured in a Client/Server approach.

Chapter 4 discusses weaknesses and improvements for the independent areas. Usage areas are covered and the questions will be answered.

Chapter 5 summarizes the project, constitutes the problems that occurred and gives an evaluation.

1.5 Contribution

During the thesis the work load has not been completely divided. The main focus for Daniel was the Mica motes, the MULLE and Ekahau while Sebastian worked on the robot. However, the work has overlapped, and both of the authors have been involved in each step of the development process.

Chapter 2

Background

This section describes the different technologies in order to give a more in-depth view of what they, and their capabilities, are.

2.1 Scorpion robot

The Scorpion robot is maintained by Evolution Robotics and is equipped with the following features [13]:

- 10 IR range finder sensors.
- Contact sensing bumper mounted in front of the robot.
- A Robot Control Module. The module provides motion control, auxiliary digital and analog I/O capabilities, and sensor I/O.
- Wide angle, calibrated video camera with precision tilt adjustment.
- Servo powered differential drive.
- Rechargeable 12V battery and charger.
- USB interface.
- Cushioned cradle for laptop computer.

The Scorpion robot can be seen in figure 2.1 on page 9.

2.2 ER1 robot

The ER1 robot, also maintained by Evolution Robotics, is equipped with the following features:

- 3 IR range finder sensors.
- A Robot Control Module similar to the Scorpion.
- Rechargeable 12V battery and charger.
- USB interface.
- Cushioned cradle for laptop computer.
- Wide angle, calibrated video camera with precision tilt adjustment.

The use of the ER1 robots was not initially planned. It was only thought of as a comparison robot, to the more complex Scorpion robot, but due to some mechanical errors, it became the robot of use half through the project. The ER1 robot is shown in figure 2.2 on page 9.

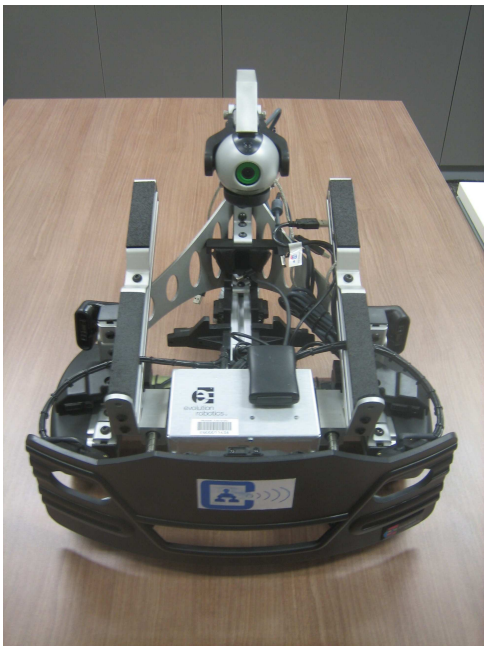

Figure 2.1: Evolution Robotics Scorpion robot.

Figure 2.2: Evolution Robotics ER1 robot.

2.3 Evolution Robotics Software Platform

Evolution Robotics Software Platform (ERSP) is a trademark of Evolution Robotics, and is made up by a number of software modules. These modules are components containing a broad range of drivers, tasks, and behaviors.

The ERSP Application Programming Interface (API) is written in C++ and contains a substantial number of libraries, behaviors and tasks. They are all divided into parts that describe their functionality. Some of the more extensive libraries are base, behavior, GUI, math, navigation, resource, task and vision, while some of the more extensive behaviors are emotion, GUI, navigation, resource, speech and vision. There are tasks available for navigation, networking, resource, speech, utility and vision. Obviously, navigation and vision are important parts of the architecture, and they will be described later on.

2.3.1 ERSP architecture

For simplicity, the ERSP consists of three layers, the Hardware Abstraction Layer, the Behavior Execution Layer and the Task Execution Layer.

Hardware Abstraction Layer

Abbreviated HAL, is the interface between the applications and the hardware. Using physical devices, like a camera and range sensors, the HAL receives information about the surrounding environment. The interaction between the physical world and the low-level operating system dependencies are done by the HAL drivers.

Using resource drivers, which are software modules that allow access to a resource, one can interact with the underlying resources, like physical devices such as sensors, microphones, actuators, network interfaces, battery and speech recognition systems. This can be done from the operating system or other API function calls. The resource functions implemented by the driver are dependent on the operating system and the device specifics.

A number of well-defined interfaces are provided by the HAL. These interfaces, which are a set of public C++ abstract classes, protect higher-level modules from OS dependencies.

The resources are made up of three subtypes:

Bus: A transport layer to access external devices. Examples are USB, serial and Ethernet.

Device: Resource type representing a single physical device, like an IR sensor, a motor or a camera.

Device group: Multiple devices handled as a single resource, such as a drive system composed of several motors.

The HAL has a resource manager to handle allocation, instantiation and activation of resources. It is also responsible for deallocation and to free memory when the resource is not needed anymore. There are configuration files needed for the resource manager to be able to load resources correctly. The resource description files are usually done in XML. Evolution Robotics supplies predefined configuration files for both the Scorpion and the ER1 robot.

Behavior Execution Layer

Short BEL, is a framework for building autonomous robotic systems.[14] The main part of the BEL is the behavior. It is defined as a computational unit that maps a set of inputs to a set of outputs. The inputs and outputs are conveniently named ports. Ports are characterized by their data type, data size and semantic type. Output ports can have connections to an arbitrary number of input ports.

Behavior networks can be formed by chains of connection behaviors. They are executed sequentially and at the same rate, in a FIFO (first in, first out) manner.

The implementation of behaviors and behavior networks can be done using either C++ objects and files or using XML. Since it is easier and more standardized to use XML, the ERSP provides an interpreter in form of the **behave** application, which instantiates and runs behavior networks.

Task Execution Layer

The Task Execution Layer (TEL) is a high-level, task-oriented interface to the Behavior Execution Layer. A flexible plan, to be executed by the robot, can be created by combining tasks and sequence them using functional composition. TEL is by so a way to express higher-level execution knowledge and coordinate the actions of multiple behavior.

Advantages with the TEL is familiarity (defining tasks is comparable to writing standard C++ functions) and ease of scripting (including languages like Python).

Due to copyright and restricted access, exact details about the classes and types that make up the TEL can not be given. What can be mentioned is that they are divided into these resembling parts:

- Code that runs when a task is executed.
- A registry for keeping track of tasks and indeces them.
- A way of handling arguments to tasks.
- Information about task status.
- Return values from tasks.
- A way of handling asynchronous tasks and supply a solution for how tasks communicate with each other.
- Running multiple tasks simultaneously.

Core technologies

Navigation and **Vision** are two technologies that are more significant.

Vision is divided into two parts, object recognition in form of ViPR (Visual Pattern Recognition) and image processing. The system for object recognition focuses on recognizing planar, textured objects. Objects with three dimensions can also be recognized reliable, but the objects should not be deformable in order for the system to manage the recognition correctly. Characteristics that make the object recognition work properly are variations to [14]:

- Affine transformations and rotation. Rotated objects, and objects placed at a different angle with respect to the optical axis, are recognized.
- Scale changes. Objects at different distances from the camera are recognized.
- Lightning changes. Recognizes illumination changes from natural to artificial indoor lightning. Also insensitive to objects caused by backlights and reflections.
- Occlusions. Objects that are partially blocked, in an amount between 50 to 90 percent, are easily recognized.

If N is the number of objects in the database, the recognition speed is proportional to $log(N)$ [14].

The object recognition algorithm has three main steps:

Training The number of views needed per object depends on the object type (planar or 3-D). One or more images are taken for each view.

Recognition For each object, the algorithm extracts up to 1,000 local and unique features. When recognizing an object, a subset of this feature is identified.

Estimation The correct object (name and full pose) is provided by the identification.

Image processing is used for optical flow computation, color detection, and color tracking. One feature with color tracking is skin detection, so the robot can recognize the presence of a person and follow that person's movement. This can also be done with an ordinary object.

The motion flow module works by computing the motion between two images on a block-by-block basis, with a customizable block size. By computing the cross-correlation of pixel brightness with the first image with all the blocks of the second image, the algorithm can estimate the flow for each block with pixel accuracy.

A Gaussian mixture model [31] is used to represent the color that is to be trained. The color is represented in HSV [32] color space since it is more stable to illumination brightness levels than RGB.

Navigation modules are made up by mapping, localization, exploration, path planning, obstacle avoidance, occupancy grid mapping, target following, and tele operation control [14].

Mapping and localization: The visual Simultaneous Localization and Mapping (vSLAM) algorithm is the core of Navigation. It works in four steps. In the first step, it traverses the area, collects odometry, and acquires an image. Next, the image is processed to extract visual features. Step three consists of three parts, and is dependent on if the number of visual features is large and distinct enough.

Part (a) of step three determines if the features correspond to a stored landmark. A landmark is a collection of unique features that are extracted from an image [14]. If the landmark is stored, the algorithm continues to part (b), otherwise part (c).

A computation of visual measurement is done in part (b). The computation makes a measurement of where the robot is now, and where it was located when the landmark was created. The information gathered from the measurement are used to improve the localization of the robot and of the location of the landmark.

In part (c), where the landmark could not be found, a new landmark, with its own unique number added to the features, is being created. The landmark is then initialized.

Step four returns the algorithm to the first step.

Path planning: By using the vSLAM generated map, path planning is done by computing a safe path between a start and a goal position. Waypoints are then placed between the landmarks in the map. Dynamic environment changes are not taken into account and are relied on the obstacle avoidance routines.

Obstacle and Hazard Avoidance: Using the sensors, the robot detects obstacles and generates the correct movement to avoid them.

Exploration: A slowly increasing imaginary circle, where the robot travels back and forth, is used in the initial phase of the exploration. The robot goes in a straight line towards the boundary of the circle and when it reaches the boundary it goes back to the interior in a random way. This procedure continues until the preset radius of the circle or the preset time has been reached. The next step for the robot is to improve the landmarks. That is achieved by returning to previous landmarks and exploring every unexplored space of it.

Occupancy grid mapping: Combining vSLAM with obstacle avoidance will create an occupancy grid map. The map is an overhead picture of objects in the robots' view. In order to keep the map up to date, vSLAM provides information about where in the map the robot is and obstacle avoidance detects obstacles and then adds them to the map.

2.4 Ekahau positioning system

Ekahau Inc.[9] is the sole proprietor of a software based real-time location system. The approach of Ekahau Positioning Engine (EPE) is measuring signal strengths from a wireless network. This software based approach offers the highest positioning accuracy available and, according to the Ekahau homepage[9], outperforms competing technologies such as signal propagation and triangulation. EPE can track various 802.11 enabled devices such as laptops, handhelds, and the Ekahau T201 Wireless-Fidelity (Wi-Fi) tag. Ekahau Client must be installed and run on each laptop or handheld that should be found by EPE. The network reader component is currently only available for Windows XP, Windows 2000 and Pocket PC 2002/2003.

EPE software based technology is not just accurate, it facilitates a fast implementation with an affordable cost. It can make use of existing wireless network infrastructure and is compatible with all standard 802.11a/b/g Wi-Fi access points. For EPE to function optimal at least three Wireless LAN (WLAN) access point should be "heard" at any location, and, if possible, placed in the corners of the intended coverage area. The positioning engine works with just one access point, but will have inferior accuracy. If less than three access points are audible, the average error will exceed five meters. Increasing the number of overlapping access point signals generates a higher veracity, with three or more audible access points an average accuracy of one meter can be achieved. The competing tracking systems require additional network hardware overlay which cost time and resources.

Due to varying hardware and software used by WLAN card manufactures, the radio reception can differ significantly. This can cause positioning inaccuracy if the tracked devices use another WLAN adapter than the positioning model was configured for. Without using any mitigation technology, the typical positioning inaccuracy caused by different cards is 1-4 meters, depending on the network card. The Ekahau Manager uses a system called Ekahau Card Models to read information about the network cards and mitigate the Received Signal Strength Intensity (RSSI) measurement differences.

EPE is capable of tracking thousands of devices simultaneously, even if they are in different buildings. Information about the tracked device includes:

- X and Y coordinates.
- Building.
- Floor level.
- Logical area (created by user).
- Heading.
- Time.
- Speed.

There are two ways to connect to the engine and retrieve the information; either through a Java API or by making use of the Ekahau Yax protocol. The latter is a character-based Transmission Control Protocol (TCP) and can be used from any programming language that supports TCP sockets. It is recommended to use the Ekahau Software Development Kit (SDK) if the development environment supports Java 1.4.1, seeing that it is the easiest way to implement a location-aware application.

2.4.1 Ekahau manager

Ekahau Manager is an application that is used to create, manage, and store a positioning model in the positioning engine. This positioning model requires a map over the building in one of the supported formats BMP, JPG, or PNG. Some functions provided by EPE also needs a correct map scale to calculate impeccable answers.

Audible access points might belong to another company or a private resident, therefore it is important to restrict the access points that should be used. Otherwise, the model might easily become erroneous if an access point is moved or taken down.

The positioning model requires the user to draw Tracking Rails on the map, indicating possible travel paths for the tracked object. EPE also needs a calibration to pinpoint Wi-Fi enabled devices. This is done by recording sample points with the manager laptop. Sample points should be recorded:

- Every 3-5 meters.
- In all Tracking Rail intersections.
- Start and End position of Tracking Rail.
- Where the radio environment suddenly changes, for example on both sides of a door.
- In large open areas, construct a grid of Tracking Rails and record sample points in the corners of each square.

An immense advantage with EPE is that minor environment changes do not require the user to reiterate to calibration process.

Ekahau Manager is also used to draw logical areas on the map, which can be used to check if a client is inside a specific area. Furthermore, it provides tools for carrying out positioning accuracy analysis and display the LAN coverage. The latter can be viewed from selected access points and is very useful to find areas with poor network coverage. Figure 2.3 on page 15 displays the network coverage of the 7th floor in the H building at Monash University, Caulfield, with four selected access points. It also shows the robots possible travel paths together with three logical areas:

- Base.
- Printer.
- Corridor.

2.4.2 Ekahau Wi-Fi tag T201

The Ekahau Wi-Fi tags are the core of the Real-Time Location System (RTLS). They can be attached to any mobile object allowing EPE to continuously track the target within the coverage of an 802.11 network. The typical operating range is approximately 40-60 meters indoors, depending on the network speed. It has an intelligent power consumption that is based upon motion sensors and gives battery life up to several years. The Ekahau Wi-Fi tag is illustrated in figure 2.4 on page 15.

2.5 Wireless sensors

To make sure that an optimum temperature is maintained in all areas of for example an office, the robotic device moves to predefined locations and receives the current temperature from a wireless sensor. If the temperature should deviate from the normality, various counteractions will be taken. Under optimal circumstances the office has a computer controlled air conditioning and ERSP can immediately amend the situation.

Two different wireless sensors are used to read surrounding temperatures.

- MPR410 MICA2 motes [36] with attached MTS310CA sensor boards [37] operating at 433 MHz.
- MULLE [23] - a wireless sensor communicating over Bluetooth.

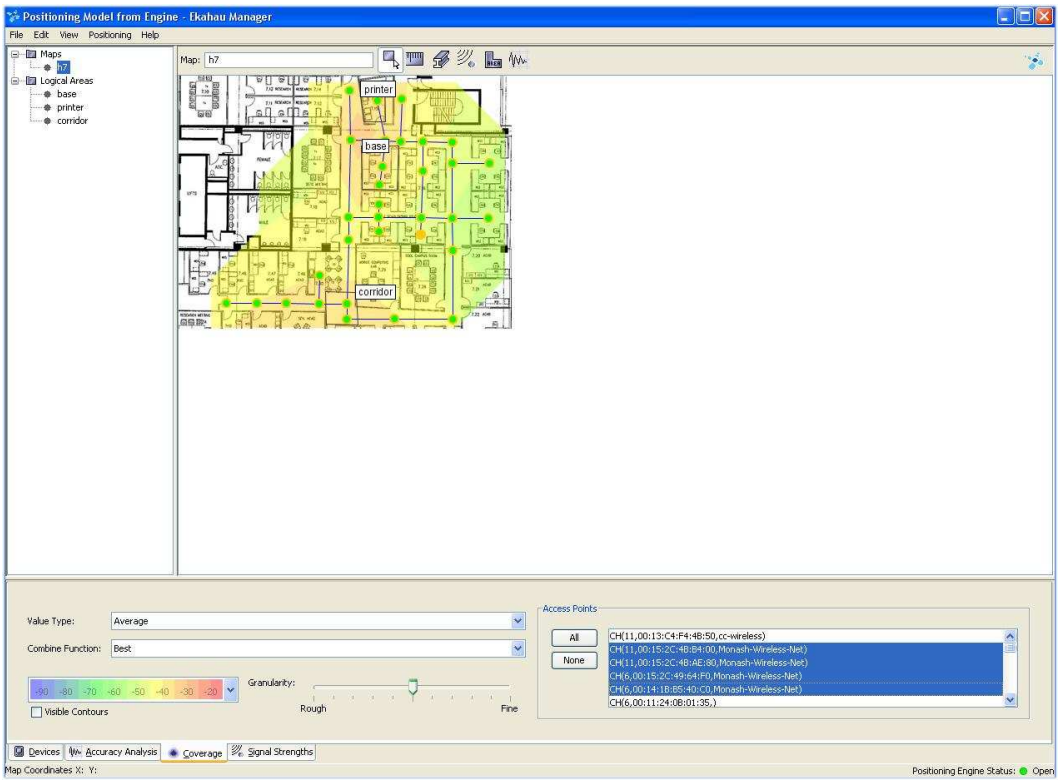

Figure 2.3: Ekahau Manger displaying network coverage for selected access points.

Figure 2.4: Ekahau Wi-Fi Tag T201.

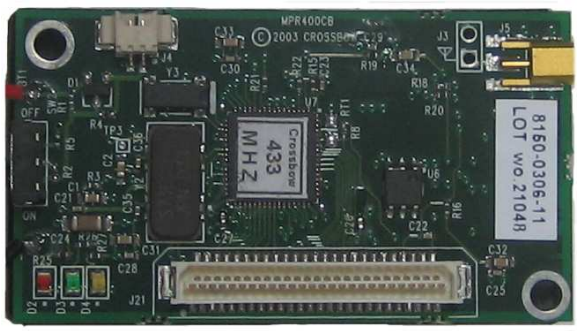

Figure 2.5: MPR410 MICA2 mote.

Figure 2.6: MTS310CA sensor board.

2.5.1 MPR410

The MPR410 MICA2 motes have two separate tasks. At first, when connected to an interface board, they function as a base station. Secondly, equipped with a sensor board, they act as sensors transmitting data to the base station.

The battery-powered MICA2 mote is based on the low-power Atmel ATmega128L microcontroller, which runs TinyOS (TOS) from its internal flash memory. Equipped with two 1.5V AA batteries it has a lifetime that exceeds one year. External power can be provided through the standard 51-pin expansion connector or the 2-pin Molex connector. Indoors, the tested range was on average 16 meters in a fairly open space. The MPR410 mote is illustrated in figure 2.5 on page 16.

2.5.2 MTS310CA

The MTS310CA sensor board, illustrated in figure 2.6 on page 16, is used to read the temperature at desirable locations. It is equipped with a surface mounted Panasonic thermistor with a nominal mid-scale reading at 25 degrees Celsius. This sensor board is also equipped with a light sensor, accelerometer, magnetometer, microphone and a sounder.

2.5.3 MICA2DOT

If physical size and low weight is important, the robotic device can also communicate with the small quarter-sized MICA2DOT [36] motes, see figure 2.7 on page 17. However, small size implies deteriorated battery capacity and reduced functionality. A wireless network installation can

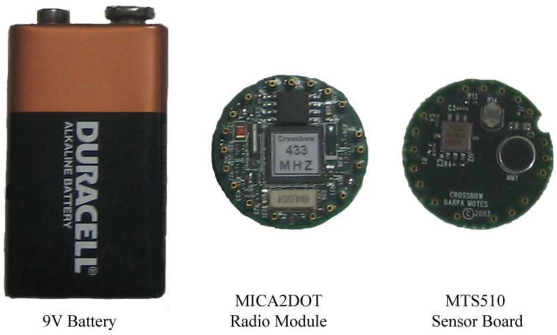

Figure 2.7: MICA2DOT motes.

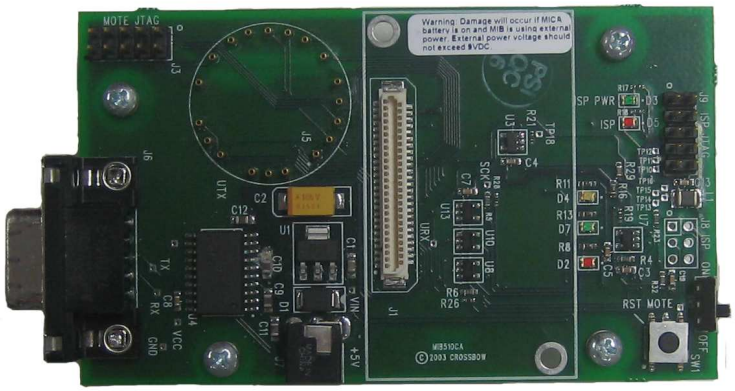

Figure 2.8: MIB510CA interface board.

contain both MICA2 and MICA2DOT motes as they are compatible with each other.

2.5.4 MIB510CA

The MIB510CA [36] interface board allows for the aggregation of sensor network data on a computer and provides an RS-232 serial programming interface. As shown in figure 2.8 on page 17, it has connectors for MICAz/MICA2 and MICA2DOT motes. To program the motes an installation of TinyOS [30] is required, instructions can be found in the getting started guide [35]. The Windows operating systems also requires an installation of Cygwin [5].

2.5.5 MULLE

The MULLE is a small Embedded Internet System (EIS) platform which was developed at Luleå University of Technology. It is based on a Mitsubishi M16C microprocessor and uses Bluetooth for wireless communication. With low power consumption and a small physical size the MULLE is excellent for mobile applications. Its small size (25 x 23 x 5 mm) is visualised in figure 2.9 on page 18.

The model used throughout this project utilizes a LM61 temperature sensor [24] to receive temperature information. Other available sensors include accelerometers and light sensors.

The MULLE software platform consists of the following parts:

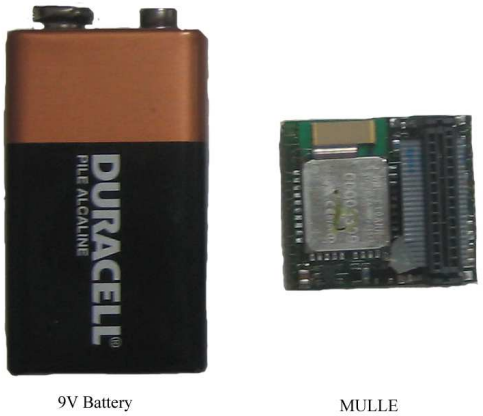

Figure 2.9: The MULLE wireless sensor networking platform.

- lwIP [21] - a small open source implementation of the TCP/IP protocol suite focusing on reducing resource usage. This makes lwIP suitable for embedded systems with scarce memory and processor resources.
- lwBT [19] - implements the upper layers of the Bluetooth protocol suite, designed to transport data from lwIP over Bluetooth. Similar to lwIP, lwBT was developed for small embedded systems with scarce resources.
- Network Address Translation (NAT) [29] - A small open source implementation of NAT for lwIP enabled devices. NAT is used to create an ad-hoc network which shares a single connection to an external network.
- Web and sensor server.

The MULLE is capable of sending data via external networks since it uses standardized Bluetooth and TCP/IP protocols. This requires either a Bluetooth LAN access point or other Bluetooth enabled devices with appropriate software installed.

2.6 Languages and software

2.6.1 Java Native Interface

The Java Native Interface (JNI) provides a way for Java applications to incorporate native code written in programming languages C or C++. This can be done in two ways: using either native libraries or native applications. With native libraries, native methods are written within the Java application, and for native applications, a Java Virtual Machine will be embedded into a native application. Since Java will be the programming language used throughout this project, native libraries will be utilised. See Appendix A for details on how to write native libraries.

2.6.2 Eclipse

Eclipse [8] is an open source framework for delivering rich-client applications. The rich client platform, that forms the basis for Eclipse, contains the following components:

- A core platform to boot Eclipse and load plugins.
- Framework that follows the Open Services Gateway initiative (OSGi).

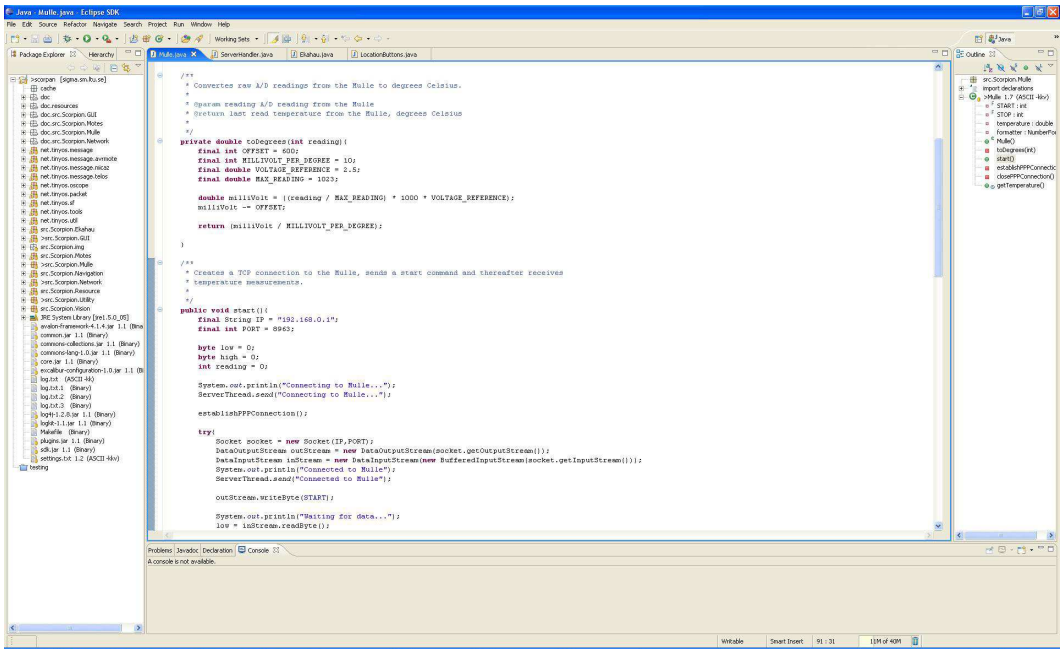

Figure 2.10: The Eclipse framework.

- JFace for handling text and file buffers.
- Eclipse Workbench that contains editors, wizards, views and more.

See figure 2.10 for a screenshot of the Eclipse application.

2.6.3 TinyOS

In a wireless sensor network, where the motes have severe memory constraints, the necessity of a suitable operating system is high. TinyOS[30] is an open-source operating system designed for sensors with diminutive hardware. Its component-based architecture minimizes program size while enabling rapid innovation and implementation. TinyOS has an event-driven execution model that enables exhaustive power management while providing scheduling flexibility. There are two threads of execution:

Tasks: Functions whose execution is postponed until they are scheduled. Tasks run to completion and do not preempt each other.

Hardware event handlers: Executed as a reaction to a hardware interrupt, like a task it runs to completion. However, it may preempt the execution of a task or other hardware event handler.

The TinyOS system, libraries, and applications are written in nesC[25], an extension to the C programming language primarily designed for embedded systems. Both TinyOS and nesC was developed by a consortium led by the University of California, Berkeley.

2.6.4 Cygwin

The free software Cygwin is a porting of GNU is not UNIX (GNU) for Microsoft Windows. Cygwin creates a UNIX like environment with many of its standard utilities, including development tools such as GCC and GDB. Programs can make use of both Microsoft Windows API and the

Cygwin API. This facilitates porting of other UNIX programs without radical changes to the source code.

Cygwin is developed by Cygnus Solutions and supports all modern 32-bit versions of Windows, with the exception of Windows CE. However, Cygwin can not do more than the underlying operating system supports and will thereby sustain different limitations depending on which version of Windows is used.

2.6.5 M-16 Flasher

The M16C-Flasher is a Graphical User Interface (GUI) for programming Renesas M16C microcontrollers. In order to flash the microcontroller it must be connected to a PC via a serial cable. However, be cautious about the different voltages of the COM-port and the microcontroller. The standard PC COM-port delivers +/-12VDC while the microcontroller expects +5VDC and 0VDC. M16C-Flasher supports Microsoft Windows 95/98/ME/2000/XP operating systems and is free for non-commercial use.

2.7 Hardware and software requirements

Besides all previously mentioned technologies this project necessitates two laptops. The first one, located at the base station, has the following demands in order to run Ekahau Positioning Engine:

- At least 1Ghz x86 CPU.
- 256 MB of RAM.
- 500 MB of free hard disk space.
- Wi-Fi support.
- Java 1.5 SDK.
- Microsoft Windows 2000, 2003, or XP.

The other laptop resides on the Scorpion or ER1 robot and requires:

- 800Mhz Intel Pentium III CPU or higher.
- 256 MB of RAM.
- 2 GB of free hard disk space for installations.
- Wi-Fi support to communicate with other laptop.
- Bluetooth support to communicate with MULLE.
- RS232 serial port to communicate with MICA motes.
- Three USB ports to connect robot.
- Java 1.5 SDK.
- Fedora Core 3 for ERSP to work.

Chapter 3

Implementation

This section presents the design and implementation phase of the thesis. It is under the assumption that the reader prepossesses adequate Linux and programming skills in order to fully understand what is covered.

3.1 Scorpion

To be able to use the Scorpion robot, the ERSP needs to be installed. The installation is available for several operating systems and during this project, ERSP was installed and run under Linux, more specifically the distribution Fedora Core 3. One can question this choice of operating system, and a further discussion can be found under section 4.10 on page 35.

3.1.1 Installing and configuring the ERSP

After installing Fedora Core 3, the installation of the ERSP is quick and easy. An install shell script is supplied and will use the Advanced Packaging Tool [1] (APT) to retrieve and install the required packages. The installation will place everything related to the ERSP under the location `/opt/evolution-robotics/`.

For the purpose of making the ERSP working correctly with the Scorpion robot, some modules needs to be loaded into the Linux kernel, and configuration files must be changed. There is a module named `ftdi_sio.o` that is used to communicate with the Scorpion's Robot Control Module. Loading this module will attach the device to the `/dev/ttyUSB0` file descriptor. The IR sensors, that are connected using USB, depend on a module named `legousbtower.o`. One must make sure that this module is loaded before the `usbhid.o` kernel module, otherwise the sensors will not be found. A file descriptor named `/dev/usb/evodev0` is used for communcating with the sensors. To make the camera (a Logitech Quickcam in the case of the Scorpion robot) work, the `ov511.o` module needs to be loaded. The file descriptor `/dev/video0` will be used for getting data from the camera.

3.1.2 Testing the Scorpion robot

To get an overview of how the robot works, the test programs that comes with the ERSP is useful. The Client GUI application provides means to navigate the Scorpion and to get a video stream from the camera. Some important results from the testing:

- The time between action for movement and the actual robot movement is notable.
- Stream delay from the camera is approximately half a second.
- Robot movement is slightly arrhythmic.

- Turning and moving the robot at the same time seems tricky.

These observations will in some way affect the implementation of the Scorpion robot.

3.1.3 Analyzing necessities

What is needed from the Scorpion robot? From the depicted goal, it is obvious that it should be able to:

1. Move around. Forward, backward, left and right. This includes turning.
2. Get information about the surrounding environment for use when navigating.
3. Find a path between two points in the surrounding environment.
4. Get its current position.
5. Avoid stationary objects.
6. Avoid moving objects.
7. Receive a video stream that is of importance when manually helping the robot out in hard situations.
8. Be aware of where sensors are located.
9. Wait for sensor data to be read.
10. Run several methods simultaneously.

Since the applications supplied with the Scorpion can not be extended and used, the ERSP API must be used to its full extent.

3.1.4 The HAL, the BEL or the TEL?

What layer is most suitable for this project? Since the HAL provides interfaces, in form of C++ abstract classes, for interacting with the Scorpion, its usefulness is not questionable. HAL also allows configuration of resources. As a high level, task-oriented interface to the BEL, the TEL gives the advantages of actions made as easy programmable C++ tasks. However, using TEL, the configuration becomes limited. Since BEL uses XML, and the integration of more languages is not encouraged, HAL and TEL will be the layers of use.

3.1.5 Design

There are four major parts of the ERSP that are needed for this project:

- Navigation
- Path finding
- Obstacle avoidance
- Camera

The navigation will contain all the movement and some functionality to avoid objects, since this is hard to implement anywhere else. Path finding will consist of classes to find a path between specific waypoints in a map. Obstacle avoidance will be done by in a class that uses the IR sensors and the camera will be used in a single class to retrieve images.

3.1.6 Navigation

Both HAL and TEL provides ways of navigating the Scorpion. Here are the available classes for moving and stopping:

MoveTo Moves the robot to a set of absolute coordinates. Specifications in the form of a timeout, x and y coordinates, velocity, acceleration and a distance from coordinates can be set.

MoveRelative Moves the robot to a set of relative coordinates. Specifications are the same as for MoveTo.

Stop Stops the robot and will cancel any Move method that is in progress. The hardness of the stop can be set.

DriveMoveDelta Will command the drive system to move a specified distance. Velocity and acceleration can also be specified.

DriveStop Will command the drive system to stop. For stopping DriveMoveDelta.

Accordingly, the most useful method for moving the robot forward would be using relative coordinates, since it is under the assumption that Ekahau will provide the robot with its current location. However, the `MoveRelative` method results in an unexplainable exception when trying to execute it two times consecutively. The same occurs with the `MoveTo` method. Gladly, the `DriveMoveDelta` method is error free.

The navigation needs to be independent in order to run several methods simultaneously. One way is to use the `Parallel` class that comes with the ERSP. This will allow tasks to run asynchronously. Consequently, a lot of the communication between tasks must be done within the C++ native methods, and not within Java, which will complicate the communication process. However, Java has thread support, and threading is a much more convenient solution for this project. A possible drawback though, is a loss in efficiency.

`DriveMoveDelta` does not comprise in any turning of the robot. There are however two methods in the ERSP that can manage this:

TurnRelative Will turn the robot to a relative angle. The angle, and an optional angular velocity, can be specified.

TurnTo Will turn the robot to an absolute angle with the same specifications as above.

A relative turn to the robots' position fulfils the demands of turning. Moving and turning is what is needed for moving around. The Java implementation of the navigation is done in the class `Move`. This class consists of native methods for handling moving and turning. A thread is used for setting up a continuously running native method that, at a specified interval, reads static variables from the `Move` Java class. The static variables consist of the position to move to and the angle to turn to. There is also a possibility to change the linear velocity of the movement. A complete description of the `Move` class can be found in Appendix B.

But how does the robot know when to turn, and in what direction? Before that can be answered, the robot needs some form of instructions on how to get to a position from its current position. It needs to find a path, and more explicitly, find a path in the surrounding environment.

3.1.7 Path finding

The ERSP supplies three ways of finding a path, but removing the ways that rely on the BEL, only one way is available, as stated in the ERSP API [12]:

- PathPlanner. This method builds a graph of waypoints, attempts to connect them, and returns shortest length paths between given waypoints.

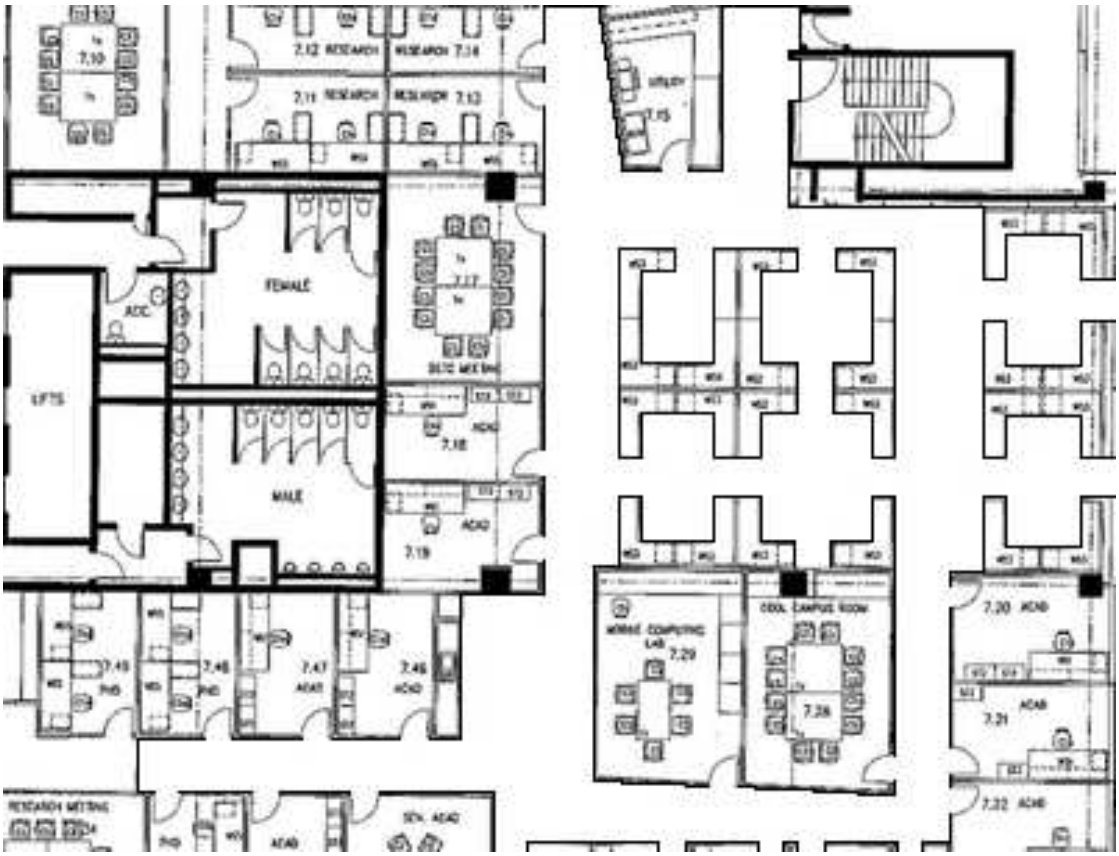

Figure 3.1: Map of the surroundings used during this project

This method will give an already complete way of generating a path between a start position and a goal. But due to extent amount of waypoints needed to render a real world representation of the area where the robot can move around, and since the `vSLAM` application can not be used, another way of accomplish the path planning is necessary.

The ERSP has a class named `OccupancyGrid` that can be used to represent a grid structure containing information about, for example, the surroundings. In order to do that, a map of the surroundings is needed. Figure 3.1 on page 30 represent the image map used throughout this project for generating the grid. A grid based on this image map will be created, and this grid can be used by the robot to find out where obstacles are located. Obstacles in this case can be walls or other real world, stationary objects. Since Ekahau also relies on a map of the environment to point out positions, the same map will correspondentlyy be of use for the robot. See section 3.2 on page 26 for a description of how Ekahau utilizes the map. As an extension to the `OccupancyGrid`, a class named `ExplorationGrid` is present. This class can be used to find the shortest path between two x and y positions in the complete grid.

To load an image map of the surroundings, the `ImageIO` class, available in the ERSP, is used. The width of the map is necessary in order to create an `OccupancyGrid` object. In order for the `OccupancyGrid` to generate an appropriate complete grid of the map, every pixel in the map where the robot can move, needs to have RGB [33] values of 255. With RGB values of 255, the corresponding grid square of that pixel will be set as empty. For RGB values of 0, the grid square will be set as full. Other RGB values will set the grid square as unknown. An `ExplorationGrid` object will be created based on the `OccupancyGrid` object.

The Java implementation of the path planning consists of two classes, `Exploration.java` and `FindPath.java`. In `Exploration.java` there is a native method `doExplore()`, that takes start and goal position as arguments, and is used to find the shortest path. In this native method, another method from the `ExplorationGrid` class is used to get the shortest path between the points. An integer array of the path is returned by the `doExplore()` method.

Since the method for finding the shortest path gives coordinates as close as possible to stationary objects like walls, in order to reduce the path length, it is necessary to ensure that the robot moves via specific waypoints in order to reach a goal. If not, an excessive amount of obstacle avoidance is needed and will result in gratuitous program execution. The `FindPath` class contains methods to retrieve waypoints (in any order), connect them together, and use the `Exploration` to find the path to go. A more detailed description of how the most important parts of the `Exploration` and `FindPath` classes works can be found in Appendix C.

For the robot to move around, and avoid moveable objects that can occur, some form of obstacle avoidance is needed.

3.1.8 Obstacle avoidance

There are two more concrete methods to avoid getting too close to stationary and moving obstacles:

- Using the camera and object recognition.
- Using infrared sensors.

Since the camera is stated to be used for retrieving a video stream, necessary when helping out the robot in stuck situations, it can not be used for obstacle avoidance simultaneously. However, the object recognition for the ERSP relies on images, and a possibility to make object recognition useful is to ensure that the video stream is made up of a sequence of images. But due to the amount of CPU power used by this solution for the camera (see section 3.1.9 on page 25), and the limited need of object recognition (there is only a need to find objects that are in the way), the IR sensors seemed like a more convenient method for this aim.

The process to obtain the IR sensors, and to read data from them, is very straightforward. Logically, the sensors are named after their location on the physical robot. An array of the sensors is created to be used when obtaining the interface to each of the sensors. Every sensor will give data on how far away an obstacle is, and a maximum distance can be set. Defining an explicit distance value for each sensor is necessary and if an obstacle comes within this range, countermeasures must be taken. In a first, simple method, any obstacle appearing on the sides of the robot will turn the robot in a small angle in the opposite direction, just to avoid a hit by the side. A second, recursive method, tries to go around an obstacle or break out from a stuck situation. It should be noted that these methods does not work correctly, due to the unexpected results that the `DriveStop` and `Stop` methods does not fully comply with what they are supposed to achieve and due to time constraints. For a more complete description of the IR sensors, see Appendix D.

3.1.9 Camera

One way to implement the camera is by using the HAL, which allows configuration of the camera. However, using this method resulted in the strange error that an image could not be captured. The TEL, and the available task `GetImage`, worked directly, but the configuration possibilities are lost. In order to reduce image size, and by so the size of data that needs to be sent to the GUI, the image is downscaled before it is stored. This is all done in the native method `takePicture` that is called from the `Camera` Java class. To create a streaming effect, a picture is taken every tenth of a second. In the `Camera` class, a `JPGSender` object is constructed and is used to send the images to the GUI. The `JPGSender` uses the Java NIO to effectively send data via file channels.

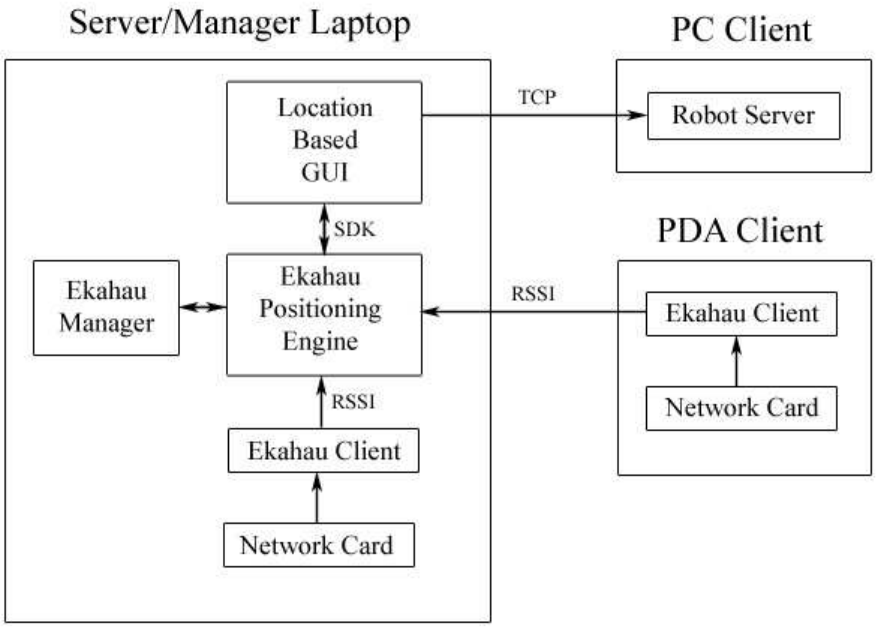

Figure 3.2: Ekahau conceptual model

3.2 Ekahau

3.2.1 Hardware setup

Because of Ekahau Clients limited operating system support, as briefly discussed in section 2.4, the EPE can not directly track the robotic device since it is controlled by a laptop running Fedora Core. Attaching an Ekahau Wi-Fi tag to the robot would be the natural solution to circumvent this. However, the available tag malfunctioned and another solution had to be used. The selected alternative was to place a Wi-Fi enabled PDA, running Pocket PC 2003, on the robot. Both Ekahau Positioning Engine and Ekahau Manager are installed on a laptop running Microsoft Windows XP.

3.2.2 Software

The location based client establishes a connection to the positioning engine through the Java API and registers listeners that receive location information about the desired wireless device. The implementation supports tracking of multiple wireless devices even though this project only tracks the ER1 robot, thereby making it more applicable. Detailed information about how to connect and communicate with the positioning engine is described in Appendix E. Figure 3.2 illustrates the Ekahau concept used throughout this project. More information about how the location based GUI communicates with the robot server can be found in section 3.5 on page 29.

3.3 MICA sensors

The components used in the wireless sensor network setup are:

- One MIB510 interface board.

- Three MPR410 processor/radio modules.
- Two MTS310 sensor boards.

3.3.1 Software

Before the MICA2 motes can read and wirelessly transmit temperature measurements a TinyOS application has to be installed into the mote. Appendix F elucidates how to compile and install an application using Cygwin. The two motes that operate as sensors have a modified version of `OscilloscopeRF` installed. In its standard design `OscilloscopeRF` periodically takes readings from the photo sensor and sends them over the radio channel. The modified version functions much in the same way besides utilizing the temperature sensor. It is of the outmost importance that two different group ID's are assigned when programming the motes as discerning them would be impossible otherwise. The third mote, which is connected to the MIB510 interface board, is running an application called TOSBase that acts as a bridge between the wireless and the serial channel. Once the data is made available for the serial port, another program, `SerialForwarder`, takes over. `SerialForwarder` is a Java program that reads TinyOS messages from the serial port and forwards them over a TCP connection to connected clients. This means that other programs, even running at another machine, can communicate with the sensor network via a sensor network gateway.

In this project, the robot server connects to the `SerialForwarder` program and registers a `MesssageListener` object that will be invoked when TinyOS messages arrives. The received TinyOS message contains the raw ADC output from the MTS310 sensor board, an approximation from [37] is used to convert it into degrees Kelvin. This approximation is only valid if the temperature is within the interval [0, 50] degrees Celsius.

$$\frac{1}{T(K)} = a + b \cdot \ln(R_{thr}) + c \cdot (\ln(R_{thr}))^3 \tag{3.1}$$

where:

$$R_{thr} = \frac{R1(ADC\_FS - ADC)}{ADC}$$

$$a = 0.00130705$$

$$b = 0.000214381$$

$$c = 0.000000093$$

$$R1 = 10\ k\Omega$$

$$ADC\_FS = 1023$$

$$ADC = \texttt{output value from mote's ADC measurement.}$$

$$T(K) = \texttt{temperature in degrees Kelvin}$$

Once T(K) has been calculated, it is a straightforward task to calculate degrees Celsius.

$$\texttt{Temperature Celsius,} T(C) = T(K) - 273.15$$

Furthermore, the received TinyOS message includes the unique group ID that is used to determine which sensor made the transmission. The corresponding local variable is updated and can later be retrieved through its `get` method. Detailed information about how the temperatures, received from the mica sensors, are sent to the client can be found in section 3.5 on page 29.

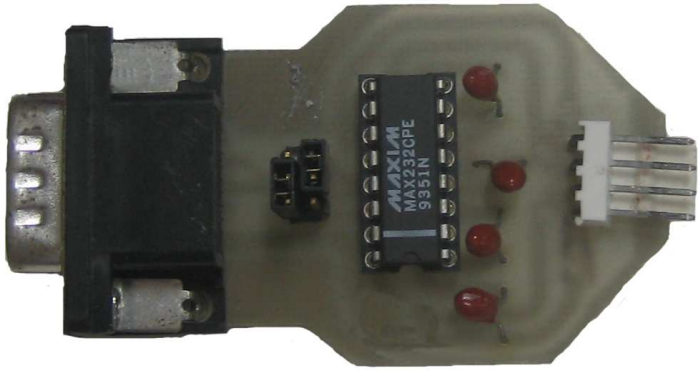

Figure 3.3: RS232C Level Converter.

3.4 MULLE

3.4.1 Hardware setup

The first step is to furnish the MULLE with a suitable power supply. Since the current varies vigorously the MULLE should be powered either by a battery that delivers 3.7VDC or through a voltage regulator, using a resistor to reduce the voltage is indecorous. Unfortunately neither a suitable battery nor a voltage regulator where available at Caulfield campus, consequently a voltage regulator kit was bought and soldered together. Once the MULLE receives an appropriate voltage it starts up and executes its firmware, by default this meant searching for a Bluetooth LAN access point and start uploading data to an Embedded Internet System Laboratory (EISLAB) server at Luleå University of Technology. Sending data between Australia and Sweden is unpractical and therefore a new firmware was necessary.

Re-programming the flash memory of the MULLE requires numerous preparations. Firstly, a programming adapter has to be attached to the MULLE allowing it to receive data from a PC. Secondly, short-circuit pin 5 and 15 to set the MULLE in its programming mode. Thereafter should pin 2 (RxD), 3 (GND), 4 (TxD), and 15 (VCC) be connected with corresponding pins from the computers serial port. The voltage difference between the computers serial port and the MULLE is managed by a RS232C level converter [11]. This level converter consists of a MAX232 circuit together with four capacitors, as illustrated in figure 3.3. Connecting and uploading the new firmware to MULLE is carried out by the M16C-Flasher program. Finally, remove the short-circuit between pins 5 and 15 to set the MULLE in its run mode again.

3.4.2 Connect and receive temperature from the MULLE

With the current firmware the MULLE waits for a Point to Point Protocol (PPP) connection over the LAN Access Point (LAP) profile. One way to establish such a connection is using the two UNIX commands hcitool and dund. The latter is used to set up the PPP connection over LAP using rfcomm whereas hcitool is used to scan for the Bluetooth address needed by dund. Appendix G illustrates the procedure in Java.

Once a connection is properly set up it should be possible to send a ping request to the MULLE on the Internet Protocol (IP) address 192.168.0.1. The best method to receive the raw readings from the LM61 temperature sensor is to create a TCP connection to the MULLE. When the MULLE receives a START command over the created TCP connection it will begin trans-

mitting data, in packets of two bytes, containing the 16-bit number from the Analog to Digital Converter (ADC). Since the bytes are sent in little endian format while Java Virtual Machine (JVM) uses big endian format some byte arithmetic is necessary to receive a faultless reading.

- **Little endian:** the least significant byte of any multi-byte data field is stored at the lowest memory address.
- **Big endian:** the most significant byte of any multi-byte data field is stored at the lowest memory address.

After the byte arithmetic operations the received value should be in the interval [0, 1023]. Since the temperature sensor has a nominal output voltage ranging from 0 to 2.5 volt, a corresponding value can be calculated. The LM61's output voltage is linearly proportional to the temperature which means that an increase with 10 milli Volt (mV) is analogous with a raise of one degree Celsius. According to [9] a voltage offset (600mv) must be subtracted before determining the temperature. The complete course of action can be viewed in Appendix H. For information about how the temperature is sent to the client see next section.

3.5 Client/Server

The Java application created for this project is divided into a common client-server model. In the vicinity of the base station should a computer, running Microsoft Windows, have the client installed and running. The server resides on the mobile robotic device that moves around and collects sensors readings.

3.5.1 Server

The server is principally a continuous loop that reads and parses objects sent over the network. How the server is set up and sends and receives messages can be viewed in Appendix I. The received messages can be of the types:

SteerPanelPacket A packet that contains information about how the user wants the robot to move, generated from the steering panel in the GUI.

EkahauPacket This packet is sent when EPE update the location information about a wireless device. It contains information about current location, map scale, and which logical area the device is currently visiting.

ImagePanelPacket Generated from the ImagePanel in the GUI that displays the floor map, it contains all waypoints, landmarks, and goals that were added to the map.

DisconnectPacket Sent when the client desires to disconnect from the server.

The server handles all communication with the wireless sensors and holds the data until it is sent to the client when visiting the base station.

3.5.2 Client

The GUI provided by the client, illustrated in figure 3.4, is the actor of the system and facilitates moving control over the robotic device. It has a steering panel which can make the robot move in all direction and at different velocities. Moreover, waypoints and goals can be uploaded to the robot server by marking them on the map. Once the robot server receives them it calculates the best route towards the goal. Unnecessary waypoints are left out. The video feed from the robots camera is convenient when steering the robot away from the user's field of sight or getting the robot out of locked situation. In the GUI, the temperature measurement from all three motes are

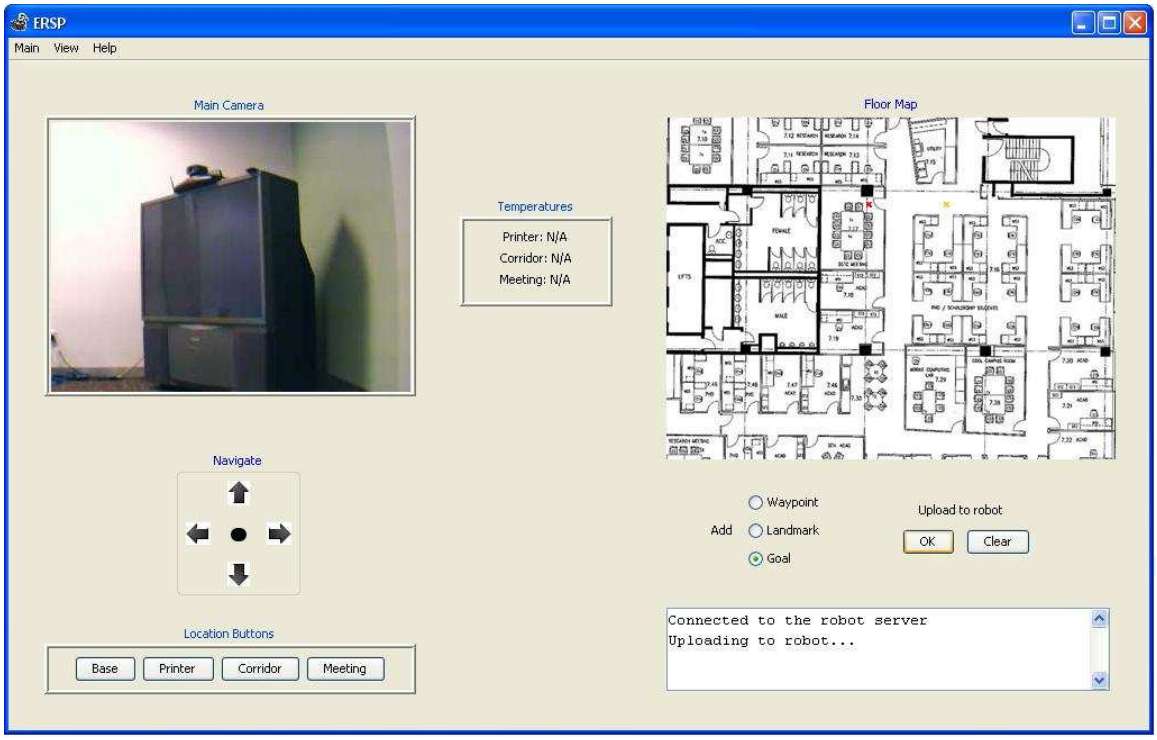

Figure 3.4: Client GUI.

also shown. Important output and status messages will be shown in the console in the right-hand bottom corner. Finally, the user can read a log of the temperatures that deviated from the normality under the view - log menu. The log includes which sensor read the erroneous temperature and at what time. More detailed information about the client's functionality can be found in Appendix J.

Chapter 4

Discussion

This chapter discusses the problems that occured and investigates the stipulated areas independently, revealing weaknesses and possible improvements. A couple of considerable usage areas will be stated and the questions asked in the introduction will be answered. The software prototype is then evaluated.

4.1 Problems

These are the major problems that occurred throughout the project. Several minor problems that appeared will not be mentioned, since a solution surfaced almost instantly.

4.1.1 Scorpion robot

A major, decisive issue with the Scorpion robot was the mechanical problem that occurred with the drive system. The robot could not move forward or turn left, but move backward and turn right. After a week of efforts trying to make it work, the Scorpion was discarded and the ER1 robot became the primary robotic device. To make the ER1 work with the ERSP, different software modules were needed. Some of the modules supplied with the ERSP were not working, and new modules had to be downloaded.

The drive system for the ER1 is, however, not moving the robot accurately. If the robot is supposed to move along a straight line, it will almost always make a small turn. A longer movement will then result in a more deviant final position. Furthermore, the ER1 is not fully compatible with the ERSP API. Some methods are not working as stated in the API.

4.1.2 Ekahau

Due to the inaccuracy of the Ekahau Positioning Engine, the robot can not solely rely on this software-based tracking system for navigation. In the best case scenario the positioning engine has an average error of one meter. This is not good enough for navigating a robot in a closed office space, since a door-opening is approximately of that dimension. Two factors contributed to an even higher average error. Firstly, an old version of EPE was used. Secondly, the network card used by the PDA was not officially supported by Ekahau. Thereby no mitigation technology could be used and this typically causes a positioning inaccuracy of about 1-4 meters.

4.1.3 MULLE

The first real problem with the MULLE was the lack of documentation. Furthermore, not a single person at Monash University had previously even got it to start up. The next problem was to find a power supply that delivers 3.7VDC. Since a voltage divider circuit could not be used a voltage

regulator was bought and soldered together. Once the MULLE had an impeccable connection with a suitable power supply the next problem was detected, it tries to upload data to an EISLAB server in Sweden. This is both unpractical and requires an additional component, a Bluetooth LAN access point. Consequently, a new firmware was needed for the MULLE. Loading a new firmware requires communication over the serial channel. Since the serial port of a computer operates with 12VDC a level converter was needed, otherwise the MULLE would risk being perished.

4.2 Enhancements for Scorpion, ER1 and ERSP

4.2.1 A smallar robotic device

Since the physical size of both the Scorpion and ER1 robot are designed to fit a laptop, this will limit access to small locations, and the need for obstacle avoidance will be greater. Combining a smaller size of the robot with possibilities to use built in applications would certainly enhance the robots passability.

4.2.2 Environment insensitivity

Throughout this project the surrounding environment has consisted of an office space. This area is neither wet, humid, heavy trafficked nor very narrow and the probability for the robot to work correctly under such circumstances is small. As a result, the number of usage areas will be limited.

4.2.3 Improve the ERSP API

In the ERSP API there are some incomplete classes and methods. Some of these classes and methods would have been useful at certain times. There was also some problems with the API when using the `MoveTo` task and when using the HAL to get an image from the camera. Tracing the system calls, it was shown that the problem lied within thread scheduling in the Linux kernel, and the identified cause was the Java Native Interface. Furthermore, some changes were done to one header file in the API, to make it compile cleanly.

4.2.4 Add laser sensors

The IR sensors are not updated fast and accurate enough, hardware based, to allow good avoidance of moving objects like a walking person. Since the camera also has a delay, real-time obstacle avoidance will be hard to achieve with the Scorpion and ER1. Adding laser sensors would improve the time for detection of obstacles.

4.2.5 Built in application

Since the Scorpion robot and the ER1 robot needs a laptop mounted in order to run applications, a lot of flexibility is lost. A possible solution is to upload an application to the robot that stores it, for example, on a harddrive. However, this approach will result in a loss of extensibility, so some way of adding extensions is necessary. By adding USB outlets this could partly be avoided.

4.3 Enhancements for Ekahau

4.3.1 Operating system independency

One of the major problems in combining different technologies was managing the demands they put on the underlying operating system. It has also been noted that operating system such as

Linux and Mac OS X are continuously increasing their number of users. Attaining a broader user group and making their products more adaptable ought to be in the companies' best interest and hopefully something that will change at their earliest convenience.

4.3.2 Better accuracy

The original thought was that EPE should continuously send which logical area the robot is visiting. Supposing that it was inside the logical area where the Bluetooth MULLE is located, the robot server activates and connects to the MULLE. All necessary Java code is written to support this but due to the inaccuracy of the positioning system, user generated events had to replace the initial idea. In the best case scenario the average error from the positioning engine is one meter; this is not good enough for a closed office space. Without an impeccable object avoidance algorithm the robot would frequently collide with the office furniture. The positioning engine can provide a rough estimate but are not something that the robot can solely rely on. It is in our opinion that this is the most important factor that should be ameliorated for an evident success. If EPE could solely be used for pinpoint and navigate objects even in narrow passages it would be insurmountable.

4.3.3 Combine software-based with hardware-based

One thing that ought to attain a better accuracy is if the software-based location system could collaborate with a hardware-based system. This would be more expensive and lay claim to a careful hardware setup, still if high accuracy is invaluable this might be the solution. How the different approaches should cooperate is not clear but might be something to bear in mind.

4.4 Enhancements for MICA motes

4.4.1 Reduce energy consumption

Due to the lack of time, the application used to transmit temperature measurements from the MICA motes wastes energy. It periodically sends data readings in spite of absence from a receiver. The optimum solution would be to make the MICA motes transmit sensor data merely when a receiver is present. However, this requires an extensive revision of the already modified code. Furthermore, rewriting code for the MICA motes requires comprehensive knowledge of the programming language nesC and the TinyOS model. Hence, it has been left out as an improvement.

4.5 Enhancements for MULLE

4.5.1 Battery

The first problem that revealed itself was finding a suitable power supply. In our case the MULLE is powered with a voltage regulator. However, a voltage regulator is lumbering and devastates the otherwise small and mobile MULLE. It would be convenient if the MULLE had a small built-in battery.

4.5.2 Update firmware via Bluetooth

Changing the behavior of the MULLE requires an update of its firmware. This is an arduous task that should be made more user friendly. Firstly, a connection has to be established over the serial channel. Secondly, a level converter has to be used to shun burning the MULLE. Finally, the MULLE has to be connected according to a special wiring diagram. It would spare the user a

great amount of time if the firmware could be uploaded via Bluetooth. This would require a few additional circuits but should not be impossible.

4.5.3 Documentation

The most important improvement for the MULLE is the documentation. This is something that has been made clear to the developers and should be arranged at their earliest convenience. Without an exhaustive conversation with Jens Eliasson, the MULLE would not have functioned.

4.6 Enhancements for Mica

4.6.1 A programmable interface

Mica has no programmable interface and this has resulted in a prototype that runs the enclosed program via system commands in order to obtain data. A better solution would be to provide a library that could be used instead.

4.7 Software prototype usage areas

4.7.1 Air-condition controller

As the goal for this project was to create a self-controlled robotic device that moves to specific areas, in order to read temperature sensors, a possible usage area is an air-condition controller. Once a value is read from the sensor the robot assures that it is within the specified range. If, contrary to expectation, the temperature should deviate from the normality, counteractions will be taken to ameliorate the situation. A mobile robotic device of this kind might be necessary when the sensors can not directly communicate with the base station. This might be the case if the sensors are to far away or if there is distortion in the network. The created prototype can move to numerous sensors and store information from them and deliver that information to the base station once it is in range.

4.7.2 Surveillance

The temperature sensor on the MULLE can be exchanged for a motion sensor, an IR sensor or other types of sensors. With motion sensors and IR sensors, the robotic device could act as an analyst of the data received from these sensors. If something suspicious appears, an alarm could be triggered and the robot will try to move to the area where the alarm was set off. However, since the Bluetooth version currently on the MULLE has a limited range of ten meters, only a small area can be covered. A newer version of Bluetooth on the MULLE would allow a greater area. With the help of the positioning system the robotic device can also superintend the movement and location of all individuals and material that carries a Wi-Fi tag. The Wi-Fi tag is also equipped with a panic button that works as an assault alarm and sends the location to the robot. Additionally, the MICA motes can gauge the intensity of the light and listen after sounds.

4.8 Is there a need for extensive analysis of compatibility?

In today's information technology society, when the seemingly escalating number of software applications takes the software developing market into a highly competetive level, the need for a product with well analyzed compatibility is almost indisputable. Failing in this aspect results in fewer users, and by so testers. This adds up to lesser usage areas and lesser improvements. Good and useful feedback is important for every product and its developers.

What is meant by compatibility? Compatibility in this context, when combining different technologies, means, in our sense:

- Degree of operating system independence.
- Flexibility of SDK; is the programming language of choice limiting what can be achieved?
- Source code and libraries; is it written so that it is easy to extend?

Analyzing the technologies used in this project, there are some parts that could be improved. For the products by Evolution Robotics, the main thing that is missing is a complete API in Java. This would have removed the use of the Java Native Interface that has caused a lot of time consuming problems. Moreover, it would have allowed the use of more methods. For Ekahau, not supplying a client for Linux is limiting. Furthermore, TinyOS uses `nesC` and will limit any further extensions.

How can improved compatibility be achieved? Since the common life cycle of software, which consists of nine phases as specified in [2], is exercised by many developers, an inevitable addition to this life cycle would be compatibility. Adding compatibility as a part of the specification and design phases would reveal in a product that is more prepared for extensions.

Another way of improving the compatibility is by allowing targeted user groups of an upcoming product to give reflections about their intended usage. More usage areas will result in better specifications and a more concrete design.

4.9 Combining technologies - a concise method for uncovering improvements?

Analyzing a specific technology, by using it and testing it during a precise period, will probably uncover more individual improvements then combining a technology with other technologies for a period of the same length. However, the importances of the improvements uncovered when the technology is combined are, in our sense, greater. These improvements are often major and would result in extensive changes of the technology, changes that often could give the advantage of more users, and all of its positive side effects. Conclusively, it is important for a developer or a company to retain a good way of receiving feedback and make sure that the feedback is thoroughly analyzed.

4.10 Evaluation

4.10.1 Educed software prototype

What has been achieved is that a Scorpion or ER1 robot can move to waypoints and goals (consisting of x and y positions), received by a GUI. The shortest path to the goal is calculated, and waypoints on this path will be used when navigating. This means that waypoints that are not along the path will not be taken into consideration when navigating. When the robot is at a start position, there is no way of knowing the direction the robot is facing, except for looking at the video stream that is sent to the GUI. There is a possibility to move the robot by a steer panel, so that it is facing in the direction of the upcoming waypoint.

Ekahau is used for getting the current position for the robot. However, Ekahau supplies no client for the Linux operating system, so a PDA with the Ekahau client installed needs to be placed on the robot. Due to the inaccuracy of Ekahau, a start position must, for now, be set by hand. With Ekahau, logical areas can be created, and when the robot is within such an area, data will be read from the sensors that have been placed at these areas.

The stipulated technologies are all working together, but more effort could have been made in trying to integrate more of the features of each technology. Without time as a constraint, a try to achieve the following would have been done:

- Obstacle avoidance for the robot, working with both IR sensors and object recognition.
- A more path reducing method for the navigation, for example turning and moving at the same time.
- Some form of streaming methodology for the camera instead of sending pictures.
- Speech interaction to give commands to the robot.
- Object recognition to recognize sensors.
- Making the source code secure, fail-safe, efficient, and easy to understand.

4.10.2 Is the robot self-controlled?

It is, in some sense. Getting from start to goal is all done by the code for the robot. However, the robot does not avoid obstacles at the moment, so manual interaction might be needed, and Ekahau is not reliable enough to provide a current position for the robot.

4.10.3 Alternatives to Java?

The programming language used for each technology are the following:

ERSP	C++
TinyOS	nesC
Mulle	C
Ekahau	Java or YAX

As with the ERSP, it would have been easier to use the C++ API directly then to use JNI. However, using C++ strictly for the whole project would have aggravated the process of creating network code and a graphical interface. Furthermore, the authors were more familiar with Java, so the choice of programming language was not soundly discussed.

Chapter 5

Conclusion

Throughout this thesis, the following work has been performed:

- A research of areas concerning Evolution Robotics Scorpion robot, Evolution Robotics Software Platform, Ekahau positioning system, wireless MICA sensors and Bluetooth MULLE.
- Scrutinize the possibility of joining existing technology into something new and useful.
- Design and derivation of a software application that utilizes these stipulated areas in order to assemble a self-controlled robotic device with sensor reading capabilities.
- Solder electronic components and updating hardware via circuit boards.
- Investigation of the viability of acquiring related weaknesses and improvements.
- Evaluation of the educed software application, bringing forth strengths and weaknesses.

After the project came to an end innumerable things were learned, and these are some of the most important findings:

- It is a nuisance having to reflect over system dependencies, programming languages and hardware requirements.
- Documentation can never be exhaustive enough.
- It is feasible to combine the described technologies into a functional prototype.
- This way of working is an excellent method for evaluating the modularity of a product or technology.
- The time spent on planning and documentation in an early phase of the project is invaluable.

Bibliography

[1] Advanced packing tool description.
http://en.wikipedia.org/wiki/Advanced_Packaging_Tool.
Address valid as of April 26 2006.

[2] Frank M. Carrano and Janet J. Prichard. *Data Abstraction and Problem Solving in Java*. Addison Wesley, 1 edition, 2001.

[3] Gary Cornell Cay S. Horstmann. *Core Java 2*, volume Volume 1 - Fundamentals. Sun Microsystems Press, 7 edition, 2005.

[4] Gary Cornell Cay S. Horstmann. *Core Java 2*, volume Volume 2 - Advanced Features. Sun Microsystems Press, 7 edition, 2005.

[5] Cygwin homepage. http://www.cygwin.com/. Address valid as of April 4 2006.

[6] Bruce Eckel. *Thinking in C++*. Prentice Hall, 1995.

[7] Bruce Eckel. *Thinking in Java*. Prentice Hall, 3 edition, 2003.

[8] Eclipse homepage. http://www.eclipse.org/. Address valid as of April 17 2006.

[9] Ekahau homepage. http://www.ekahau.com/. Address valid as of April 4 2006.

[10] Ekahau developer guide. Not available for public access.

[11] Ekahau user guide. Not available for public access.

[12] Evolution robotics software platform api. Not available for public access.

[13] Ersp scorpion manual. Not available for public access.

[14] Ersp user's guide. Not available for public access.

[15] Evolution robotics homepage. http://www.evolution.com. Address valid as of April 15 2006.

[16] Elliote Rusty Harold. *Java Network Programming*. O'Reilly, 3 edition, 2004.

[17] Java sun homepage. http://java.sun.com/. Address valid as of April 24 2006.

[18] Sheng Liang. *The Java Native Interface - Programmer's guide and specification*. Addison Wesley, http://java.sun.com/docs/books/jni/download/jni.pdf, 1 edition, 1999. Address valid as of April 13 2006.

[19] lwbt homepage. http://www.sm.luth.se/~ conny/lwbt/. Address valid as of April 4 2006.

[20] lwbt - a lightweight bluetooth stack for lwip.
Address valid as of April 4 2006.

[21] lwip homepage. http://savannah.nongnu.org/projects/lwip/. Address valid as of April 4 2006.

[22] M16c-flasher homepage. http://m16c.cco-ev.de/M16C-Flasher.19.0.html. Address valid as of April 12 2006.

[23] MULLE homepage. http://www.ltu.se/web/pub/jsp/polopoly.jsp?d=4111&a=14037. Address valid as of April 4 2006.

[24] *LM61 Temperature Sensor datasheet*. http://www.national.com/ds.cgi/LM/LM61.pdf. Address valid as of April 4 2006.

[25] nesc homepage. http://nescc.sourceforge.net/. Address valid as of April 4 2006.

[26] Scott Oaks. *Java threads*. O'Reilly, 3 edition, 2004.

[27] *J2SE 1.5.0 API*. http://java.sun.com/j2se/1.5.0/docs/api/. Address valid as of April 4 2006.

[28] *The Java Tutorial*. http://java.sun.com/docs/books/tutorial/index.html. Address valid as of April 4 2006.

[29] Nat homepage. http://www.sm.luth.se/~ conny/nat/. Address valid as of April 4 2006.

[30] Tinyos homepage. http://www.tinyos.net/. Address valid as of April 4 2006.

[31] Gaussian mixture model description. http://en.wikipedia.org/wiki/Gaussian_mixture_model. Address valid as of April 17 2006.

[32] Hsv color space description. http://en.wikipedia.org/wiki/HSV_color_space. Address valid as of April 17 2006.

[33] Description of rgb att wikipedia.org. http://en.wikipedia.org/wiki/RGB. Address valid as of May 3 2006.

[34] Crossbow technology inc homepage. http://www.xbow.com/. Address valid as of April 4 2006.

[35] *Getting Started Guide*. http://www.xbow.com/Support/Support_pdf_files/Getting_Started_Guide.pdf. Address valid as of April 4 2006.

[36] *MPR/MIB User's Manual*. http://www.xbow.com/Support/Support_pdf_files/MPR-MIB_Series_Users_Manual.pdf. Address valid as of April 4 2006.

[37] *MTS/MDA Sensor and Data Acquisition Board User's Manual*. http://www.xbow.com/Support/Support_pdf_files/MTS-MDA_Series_Users_Manual.pdf. Address valid as of April 4 2006.

Glossary

ADC Analog to Digital Converter

API Application Programming Interface

BEL Behaviour Execution Layer

Bluetooth Short range wireless technology

EIS Embedded Internet System

EISLAB Embedded Internet System Laboratory

Ekahau Positioning system for office spaces

EPE Ekahau Positioning Engine

ER1 A robotic device maintained by Evolution Robotics

ERSP Evolution Robotics Software Platform

GNU GNU is Not Unix

GUI Graphical User Interface

HAL Hardware Abstraction Layer

IP Internet Protocol

IR Infrared

JNI Java Native Interface

JVM Java Virtual Machine

LAN Local Area Network

LAP LAN Access Point

NAT Network Address Translation

PDA Personal Digital Assistant

PPP Point to Point Protocol

Python An interpreted programming language.

RGB Red, green and blue. A color model.

RSSI Received Signal Strength Intensity

RTLS Real Time Location System

Scorpion A robotic device maintained by Evolution Robotics

SDK Software Development Kit

TCP Transmission Control Protocol

TEL Task Execution Layer

USB Universal Serial Bus.

ViPR Visual Pattern Recognition

vSLAM Visual Simultaneous Localization and Mapping

Wi-Fi Wireless-Fidelity

WLAN Wireless Local Area Network

XML Extended Markup Language

Appendix A

JNI and native libraries

Writing native libraries can be done step by step.

1. A Java class for declaring the native methods is needed. In the thesis, a class Move is present, and this will be the Java class, named Move.java. In this class, a native method named startUpMove is present and declared as follows:

   ```
   public native void startUpMove(int updateFreqMicroSec);
   ```

   In the Mova.java file, there needs to be a line that tells Java to load the native library that will be created.

   ```
   static { System.loadLibrary("Move"); }
   ```

   Under Linux, this will load the shared library `libMove.so` and under Windows, `Move.dll`.

2. The Move.java file needs to be compiled, and javac can be used for this.

3. To generate the needed header file, `javah -jni` can be used. This header file will contain the function prototype for the native method, and needs to be included in the C++ source file. For the Move example it might look something like this:

   ```
   /* DO NOT EDIT THIS FILE - it is machine generated */
   #include <jni.h>
   /* Header for class src_Scorpion_Navigation_Move */

   #ifndef _Included_src_Scorpion_Navigation_Move
   #define _Included_src_Scorpion_Navigation_Move
   #ifdef __cplusplus
   extern "C" {
   #endif

   /*
    * Class:     src_Scorpion_Navigation_Move
    * Method:    startUpMove
    * Signature: (I)V
    */
   JNIEXPORT void JNICALL Java_src_Scorpion_Navigation_Move_startUpMove
     (JNIEnv *, jobject, jint);

   #ifdef __cplusplus
   }
   #endif
   #endif
   ```

4. The C++ implementation of the native method must be written. Again, for the Move example.

   ```
   #include "src_Scorpion_Navigation_Move.h"

   JNIEXPORT void JNICALL Java_src_Scorpion_Navigation_Move_startUpMove
     (JNIEnv *env, jobject jobj, jint updateFreqMicroSec) {
   ... C++ code goes here ...
   }
   ```

5. To create a native library, the C++ implementation needs to be compiled. Under Linux, this can be done by the following command:

   ```
   # g++ -c -o Move.o -Wall Move.cpp
   ```

   There might be a need to add arguments to `g++` about where to find the java header files and java Linux header files. The above command will create an object that needs to be linked as a shared library, which can be done with gcc:

   ```
   # gcc -shared -levobase -levotask -o libMove.so Move.o
   ```

   In this case, some ERSP libraries needed to be linked with the shared library. It should be noted that the path to `libMove.so` needs to be in the LD_LIBRARY_PATH environment variable in order for Java to find it.

6. To run the application, use the java runtime interpreter. Both the `Move.class` and the `libMove.so` native library will be loaded at runtime.

Appendix B

Move class

Since most of the navigation is handled by the native method `startUpMove()` found in the `Move.java` class, this appendix will concentrate on describing this method. This method will run until it is stopped, so a while loop is used. The first thing that happens in this loop is that static variables in the `Move.java` class file will be read.

```
 while(isRunning) {
    linear_velocity = env->GetStaticDoubleField(cls, lvid);
    x = env->GetStaticDoubleField(cls, xid);
    y = env->GetStaticDoubleField(cls, yid);
    readangle = env->GetStaticIntField(cls, aid);
```

When this is done, a check for a value not equal to zero for the `readangle` is made. If that is the case, the arguments for the task `TurnRelative` are set, and then the task is run. The value of the static variable where `readangle` got its value is set to 0, to allow new values. Sure this can mean that values set in the static variable might be skipped.

```
if(readangle != 0) {
   TaskArg argen[] = { readangle, 90 };
   context2 = TaskContext::task_args(2, argen);
   turnrelative->run(context2);
   env->SetStaticIntField(cls, aid, 0);
   delete context2;
}
```

Next step is to look for changes in the `x` and `y` variables. Bear in mind that the x and y values are from the waypoints set in the client GUI, and the `MoveTo` method is not working when executed sequentially. Therefore, a distance to move and an angle to turn to is needed. When x and y are greater than zero, we are in the first quadrant, and turning right is, according to the ERSP, a negative turn. The angle is easily calculated using the arctan function. The same thing is done for to other three quadrants. Then the angle is calculated into degrees, and the distance is calculated.

```
if(x != xold && y != yold && x != 0) {
   double angle = 0;
   if(x > 0 && y > 0) {
      angle = -atan(y/x);
   }
   else if(x < 0 && y > 0) {
      angle = -(M_PI/2) + (-atan((x*(-1))/y));
   }
   else if(x < 0 && y < 0) {
      angle = (M_PI/2) + atan((y*(-1))/(x*(-1)));
   }
   else if(x > 0 && y < 0) {
      angle = atan(x/(y*(-1)));
   }
   angle *= (360/(2*M_PI));
   double distance = sqrt((x)*(x)+(y)*(y));
}
```

Now the arguments are set and can be used for the `TurnRelative` and `DriveMoveDelta` methods. First, a turn to the right angle is needed. Then a move can be done. The total time for moving is needed when the move is interrupted by a detection of an obstacle done by the IR

sensors, in order to calculate the distance moved, and how long is left. The boost library is used for this. Continuing, the static variables from where the `x` and `y` variables are read, is set to 0, and a Boolean value, that indicates that we are done with the moving, is set to true. Contexts are deleted to get memory allocation back.

```
   TaskArg args2[] = { distance, linear_velocity};
   TaskArg args3[] = { angle, 90 };

   context = TaskContext::task_args(2, args2);
   context2 = TaskContext::task_args(2, args3);

   turnrelative->run(context2);
   timer t0;
   movedelta->run(context);
   double timerTime = t0.elapsed();
   env->SetStaticDoubleField(cls, xid, 0.0);
   env->SetStaticDoubleField(cls, yid, 0.0);
   env->SetStaticBooleanField(cls, readyid, readyValue);
   delete context;
   delete context2;
```

Next, the same procedure is done, but now when only the `x` value has changed. However, the difference is that here there is a try to use pthreads in order to get the thread id, and to kill the thread when an obstacle is found by the IR sensors. This is not finished due to time constrains.

```
else if(x != xold && y == yold && x != 0) {
   TaskArg args[] = { x, linear_velocity};
   context = TaskContext::task_args(2, args);

   int rc, t;
   void* status;
   pthread_t thread;
   pthread_attr_t attr;
   pthread_attr_init(&attr);
   timer t0;
   //movedelta->run(context);
   pthread_create(&thread, &attr, ThreadRun, (void *)t);
   pthread_attr_destroy(&attr);

   rc = pthread_join(thread, &status);
   double timerTime = t0.elapsed_min();
   env->SetStaticDoubleField(cls, xid, 0.0);
   env->SetStaticBooleanField(cls, readyid, readyValue);
   delete context;
}
```

The same thing is then done for a change in the `y` value. But the angle to turn to, either 90 or -90 degrees, is needed. After this the `xold` and `yold` variables are set to the `x` and `y` variables correspondingly. Then a `usleep(100000)` is used so that a unnecessary use of resources are not done.

```
xold = x;
yold = y;
usleep(100000);
```

In `Move.java` there is an instance of the `IRSensor` class, since this is the only place where it can interfere with any movement. Furthermore, there are two methods, `breakFree()` and

`breakFreeLight()`, implemented in `Move.java`. None of them are working fully. The method `breakFreeLight()` reads the sensors on the side of the robot, and if the values obtained from these sensors are lower than a specified value, a small turn in the other direction is done. However, this method is not completed, and the robot can get stuck.

```
private void breakFreeLight() {
   boolean sensorNE = (sensor_ne <= NE_VAL);
   boolean sensorNW = (sensor_nw <= NW_VAL);

   if(sensorNE && sensorNW) {
      if(sensor_ne < sensor_nw)
         setAngle(10);
      else
         setAngle(-10);
   }
   else if(sensorNE) {
      setAngle(10);
   }
   else if(sensorNW) {
      setAngle(-10);
   }
}
```

Appendix C

Exploration and FindPath classes

The `Exploration` class contains native methods for accessing the `OccupancyGrid` class and the `ExplorationGrid` class. One native method that is used extensively, is the `doExplore()` method. Before this method is run, the `initializeGrid()` should be run, in order to set up the `OccupancyGrid`, which is used by the `ExplorationGrid`.

There are some global variables and pointers defined in the `Exploration.cpp` file that needs to be mentioned.

```
ExplorationGrid *eg = NULL;
OccupancyGrid *grid = NULL;
GridPointList path;
bool isRunning = false;
```

As for `doExplore()`, it takes a start and goal position in form of x and y coordinates and returns an array of integers. The array will contain the pixel positions in the map of where the robot can move. First in `doExplore()`, some variables are declared. Then the `eg` pointer is instantiated, using the global `grid` pointer. The method `find_shortest_path()` is then used on the supplied arguments, and the result is stored in the `path` list.

```
JNIEXPORT jintArray JNICALL
Java_src_Scorpion_Navigation_Exploration_doExplore(
JNIEnv *env, jobject jobj, jint start_x, jint start_y,
jint goal_x, jint goal_y)
{
  int y = 0;
  jintArray jArray;
  jclass cls = env->GetObjectClass(jobj);

  jfieldID startxID = env->GetFieldID(cls, "startX", "I");
  jfieldID startyID = env->GetFieldID(cls, "startY", "I");

  bool first = true;
  eg = new ExplorationGrid(grid);
  bool success = eg->find_shortest_path(start_x, start_y, goal_x,
                                        goal_y, &path);
```

If a shortest path is found, the success variable will be true and the path will be in the `path` list. Since JNI does not have any good way of returning specific objects, the path list will be converted to an integer array by looping through the list, obtaining the positions and setting them. The start position will be set in static variables in the `Exploration` class.

```
  if(success) {
    int buf[1];
    jclass intArrayClass = env->FindClass("[I");

    if(intArrayClass == NULL)
      return NULL;

    jArray = env->NewIntArray((path.size()*2));
    if(jArray == NULL)
      return NULL;

    for (GridPointList::iterator i = path.begin();
```

```
         i != path.end(); i++)
      {
        if(first == true) {
          env->SetIntField(jobj, startxID, i->x);
          env->SetIntField(jobj, startyID, i->y);
          first = false;
        }
        else {
          buf[0] = i->x;
          env->SetIntArrayRegion(jArray, y, 1, buf);
          ++y;
          buf[0] = i->y;
          env->SetIntArrayRegion(jArray, y, 1, buf);
          ++y;
        }
      }
  }
  else {
    cerr << "Could not generate a path from start to goal" << endl;
    return NULL;
  }

  return jArray;
}
```

When the integer array is returned, it is used by the `nextMove()` method in the `Exploration` class. This method will make the robot move, and the process in doing so starts out with turning the integer array into separated arrays for the x and y coordinates. In order for the robot to move as long as possible without turning, a while loop is used to find out how many pixels the robot will move in the same direction. This is done for the whole integer array. The `x` and `y` variables will be multiplied with the number of pixels the robot can move in the same direction.

```
for(i = 0, j = 0; i < (intArray.length/2)-1 ; i++, j++) {
   Xcords[i] = intArray[j];
   Ycords[i] = intArray[++j];
}

i = 0;
x = Xcords[i] - startX;
y = Ycords[i] - startY;

i++;
count += 1;
xfirst = x;
yfirst = y;

while(i <= (size/2)-1) {
   while(i < (size/2) && Xcords[i] - Xcords[i-1] == xfirst &&
         Ycords[i] - Ycords[i-1] == yfirst) {
      count += 1;
      i++;
   }

   x = xfirst * count;
   y = yfirst * count;
```

Then, the coordinates needs to be switched, since x coordinates on the map represents y coordinates for the robot and vice versa. There is also a check to see if we need to turn before we move. The `x` and `y` variables are multiplied by the length of a pixel, in this case 8.5 centimetres. These values should be set from the GUI or somewhere else. When this is done, the static `Move` instance is used and the method `setNextWP(x,y)` is called.

```
tmp = y;
y = x;

if(tmp < 0)
   x = -tmp;
else
   x = tmp;

if(x > 0 && ylast < 0) {
   move.setAngle(90);
}
else if(x < 0 && ylast < 0) {
   move.setAngle(-90);
}

x *= 8.5; y *= 8.5;

move.setNextWP(x, y);
ylast = y;
```

The movement must be finished before new coordinates can be set, otherwise they will be skipped. This is achieved by having a Boolean variable in the `Move` class that is true when moving is in progress, and false otherwise. Its state is then returned to false, after they wait has been done.

```
try {
   while(move.isReady() == false)
      sleep(200);
}
catch(Exception e) {
   System.out.println(e);
}

move.setReady(false);
```

To know in what direction the robot should move next, the x position for the next move is stored in the variable `xnext`.

```
if(i == (size/2)-1) {
   break;
}
else {
   xnext = Xcords[i] - Xcords[i-1];
}
```

Analyzing the differences between the `xnext` and `xfirst` variables tells in what direction the robot must turn. After this, the `xfirst` variable is updated and the `count` variable is reset.

```
if(xnext != xfirst) {
   if(xfirst - xnext == 1) {
      move.setAngle(45);
```

```
      }
      else if(xfirst - xnext == 2) {
         move.setAngle(90);
      }
      else if(xfirst - xnext == -1) {
         move.setAngle(-45);
      }
      else if(xfirst - xnext == -2) {
         move.setAngle(-90);
      }
   }
   xfirst = xnext;
   count = 0;
}
```

So the `Exploration` class handles the movement between two waypoints, but there needs to be something that handles the waypoints. This is done by the `FindPath` class. The public methods in the `FindPath` class are the following:

- void makeMove(int[][] WPs)
- int[] getCurrentGoal()
- int setGoalWP(int goalx, int goaly)
- void setCurrentGoal(int index)
- void run()
- void addWayPoint(ArrayList<int[]> array)
- void addWayPoint(int x, int y)
- int[][] getConnectedPairs()
- void addLandMark(int x, int y)
- void go()

The main method in this class is `getConnectedPairs()`. This method should be called when all waypoints and goals are added. First thing that happens is that the current goal is added to the end of the `waypointList`.

```
int way[] = new int[2];
way[0] = goal_x;
way[1] = goal_y;
waypointList.add(WPCount, way);
WPCount++;
```

A loop will go through the whole `waypointList`, and use the static `Exploration` class instance `eg` and its method `gotPath` to examine if a path between the waypoints is possible. If that is the case, the size (length) between the waypoints is stored in the `sizer` integer array.

```
int sizer[] = new int[waypointList.size()];
for(int count = 0 ; count < waypointList.size() ; count++) {
   way = waypointList.get(count);
   boolean test = ex.gotPath(start_x, start_y, way[0], way[1]);
   if(test == true) {
      sizer[count] = ex.getSize();
```

```
   }
   else {
      if(way[0] == goal_x && way[1] == goal_y) {
         System.err.println("Can not get path from start to goal!");
      }
      System.err.println("Waypoint ("+way[0]+","+way[1]+")
                          not possible");
   }
}
```

Next, the order of the waypoints needs to be set. The waypoint that is closest to the start position should be moved to first. The one closest to the second waypoint should be moved to next, and so on, until the goal is reached. To achieve this, a method named `getMinIndex` is used to get the index of which waypoint has the shortest length from the start position. This waypoint is then added to the `retArray` integer doublearray and the value for this index is set to a maximum, so it will not be used again. The returned array will by so contain the waypoints in the order the robot should move.

```
int[][] retArray = new int[sizer.length+1][2];
retArray[0][0] = start_x;
retArray[0][1] = start_y;

for(int i = 0 ; i < sizer.length ; i++) {
   int index = getMinIndex(sizer);
   way = waypointList.get(index);
   retArray[i+1] = way;
   sizer[index] = 999999999;
}
return retArray;
```

Appendix D

IR sensors class

To obtain the IR sensors are pretty straightforward. There is a thread running continuously, that sets the readings from the sensors in static variables in the `IRSensor` java class. It looks like this in the case of the ER1 robot:

```
while(isRunning) {
   for (int i = 0; i < NUM_SENSORS; ++i)
   {
      double distance;
      Timestamp temp_time;
      result = ir_sensors[i]->get_distance_reading(NO_TICKET,
               &temp_time, &distance);

      if (result != RESULT_SUCCESS) {
         cout << "Failed to obtain distance reading for sensor "
              << SENSOR_ID[i] << endl;
         return;
      }

      if(SENSOR_ID[i] == "IR_tne") {
         env->SetStaticIntField(cls, ne_sensor_id, (int)distance);
      }
      else if(SENSOR_ID[i] == "IR_tn") {
         env->SetStaticIntField(cls, n_sensor_id, (int)distance);
      }
      else if(SENSOR_ID[i] == "IR_tnw") {
         env->SetStaticIntField(cls, nw_sensor_id, (int)distance);
      }
   }
   usleep(500000);
}
```

The `IRSensor` class is instantiatied as a static variable in the `Move` class, and in `Move` the methods for avoiding obstacles are implemented, as described in Appendix B.

Appendix E

Communicating with Ekahau

Firstly, ensure that an instance of EPE is running on the computer that the application should connect to. Then, call the `connect()` method of the `PositioningEngine` class to establish a connection to the desired positioning engine.

```
PositioningEngine.connect("127.0.0.1");
```

Once connected to the positioning engine the API is ready for use, call the `getDevices()` method to receive all trackable devices.

```
Map devices = PositioningEngine.getDevices();
```

Before looping through all devices a `LinkedList` and a `Handler` is created to store all tracked devices and handle their input.

```
List trackedDevices = new LinkedList();
Handler handler = new Handler();
```

In the sequel for-loop an `Iterator` is used to go through all the devices in the `Map` and creates a `TrackedDevice` instance, which represents a physical wireless device, for every one. The `TrackedDevice` class is the base class for Ekahau Positioning functionality. Firstly, it is used to set various tracking options via its `setTrackingParameters()` method, in this case the number of logical area that should be received. Secondly, it registers listeners that receive information about where the wireless device is located, which logical area that contains it and possible status messages. Finally, the `setTracking()` method is called with the argument `true` to start tracking the wireless device.

```
for(Iterator i = devices.values().iterator(); i.hasNext();) {
   Device device = (Device) i.next();
   TrackedDevice t = new TrackedDevice(device);
   t.setTrackingParameter("numberOfAreas", "2");

   trackedDevices.add(t);

   t.addLocationEstimateListener(handler);
   t.addLogicalAreaListener(handler);
   t.addStatusListener(handler);

   t.setTracking(true);
}
```

When automatic location information updates are no longer required for a device, tracking should be stopped by passing the `false` argument to the `setTracking()` method.

```
for(Iterator i = trackedDevices.iterator(); i.hasNext();) {
   TrackedDevice t = (TrackedDevice) i.next();
   t.setTracking(false);
}
```

In the location estimate handler much information can be found about the tracked device. The essential point is the current X and Y coordinates, other important information subsumes speed, heading and mapscale. There are two approaches for receiving location information:

getLatestLocation() The latest X and Y coordinates for the tracked device.

getAccurateLocation() A more accurate location, but with a short delay.

For this project the latter is chosen since an accurate position is of more importance than a fast and inaccurate one.

```
Location accurate = locationEstimate.getAccurateLocation();
xPosition = (int)accurate.getX();
yPosition = (int)accurate.getY();

LocationContext lc  = accurate.getLocationContext();
mapScale = lc.getMapScale();
```

The logical area handler is called with two arguments where one is an array that holds the logical areas, `area`. From this variable it is possible to extract useful information such as the logical areas name and with which probability the device is located within that specific area.

```
String logicalArea = areas[i].getName();
LogicalArea area = areas[i];
Double probability = (area.getProbability() * 100);
```

Ultimately, status messages are received in the status handler. The handler receives both the tracked device and its belonging status message. Here, the status message is just printed to the standard output with the IP-address of the tracked device.

```
String ipAddress = (String) device.getAttribute("IPADDRESS");
System.out.println("Status for device " + ipAddress + ": " + status);
```

Appendix F

Installing a TinyOS application

The syntax for compiling an application in Cygwin is of the form:

```
make <platform>
```

Since MICA2 processor/radio platforms, MPR410, are used the syntax should be:

```
make mica2
```

This will compile the application that is located in your current working directory. However, this does not install the compiled application into a mote. There are two commands for executing the installation, `install` and `reinstall`. The difference is that the latter only downloads the pre-compiled program into the mote and does not recompile the application, thus being significantly faster. The syntax is accordingly:

```
make <platform> re|install
```

Moreover, there are several programming boards available. Hence, the one in use must be specified either at the command line or in a Makefile. Since a MIB510 serial interface board is used, connected to the COM1 port, <programmer> and <port> should be `mib510` and `com1` respectively. Instructions on how to use a Makefile can be found in [3]. The definitive syntax is:

```
make <platform> re|install,<n> <programmer>,<port>
```

Here <n> is a unique node address which allows multiple motes to share the same radio channel. Each TinyOS message contains a group ID in the header which allows the system to filter out messages and separate motes. Setting the node address/ID during program load is not mandatory. The following line illustrates how to install an application into a mote, with group ID set to 10, using a MIB510 programming board connected to the COM1 port.

```
make mica2 install,10 mib510,com1
```

Appendix G

Establishing a PPP connection to the MULLE

Dynamically setting up PPP connections to the Bluetooth MULLE necessitates interaction with the environment in which the Java application is running. This is accomplished by obtaining a unique instance of the class *Runtime* from the *getRuntime* method.

```
Runtime rt = Runtime.getRuntime();
```

The *exec* method executes the command, given as argument, in a separate process and returns a *Process* object for managing the subprocess. Here *hcitool* is used to search for Bluetooth devices and their respective addresses.

```
Process p = rt.exec("hcitool scan");
```

This will however return all reachable Bluetooth devices, not just the desired MULLE. That is where the subprocess p comes to assistance, *getInputStream* obtains data from the standard output stream of this process.

```
BufferedReader in = new BufferedReader(
                    new InputStreamReader(p.getInputStream()));
```

The MULLE's Bluetooth address can be found by searching through the character stream and check whether the current line contains the name of the Bluetooth device. Calling the trim function assures that the string object contains no leading or trailing whitespaces. Finally, the Bluetooth address is extracted via the substring method which creates a new string object beginning at START and extends to the character at index END - 1.

```
while((line = in.readLine()) != null){
   if(line.contains(SEQUENCE)){
      line = line.trim();
      address = line.substring(START, END);
      break;
   }
}
```

Once the Bluetooth address is know it is used in conjunction with *dund* to create the PPP connection to the MULLE.

```
rt.exec("dund -c " + address);
```

Appendix H

Connect and receive temperature from the MULLE

Creating a TCP client socket, that is connected to the MULLE, is done via the Java *Socket* class.

```
Socket socket = new Socket(IP,PORT);
```

Communication with the MULLE is handled by two data streams. The output stream lets the application write data across the network whereas the input stream reads information sent from the MULLE.

```
DataOutputStream outStream = new DataOutputStream(
                                socket.getOutputStream());
DataInputStream inStream = new DataInputStream(
                           new BufferedInputStream(
                               socket.getInputStream()));
```

Before the Bluetooth MULLE start transmitting readings from its temperature sensor it must receive a start command over the network connection.

```
outStream.writeByte(START);
```

Thereafter the data packets, containing the 16-bit number from the ADC, are received by reading two subsequent bytes from the input stream.

```
low = inStream.readByte();
high = inStream.readByte();
```

Changing the endianness necessitates some byte arithmetic's. Firstly, shift the bits of *high* 8 steps to the left, the right hand side is then filled with 0 bits. Secondly, merge the *high* and *low* bytes together with the bitwise OR function. Finally, the recently merged bytes are passed through the bitwise AND function together with the hexadecimal number 0x00000FFF.

```
reading = (low | (high << 8) & 0x00000FFF);
```

Table X and X explains the result of using bitwise OR and bitwise AND functions respectively whereas table X illustrates how the decimal numbers 0 to 15 is represented in hexadecimal and binary form.

FIXME. Tables!

Appendix I

Server

The first step in creating the server is to set up a `ServerSocket` that clients can connect to, the `accept()` method blocks until a connection is made.

```
ServerSocket serverSocket = new ServerSocket(8189);
Socket incoming = serverSocket.accept();
```

After a client connects two object streams are associated with the socket allowing object to be sent between the client and the server.

```
ObjectOutputStream outStream = new ObjectOutputStream(
                                  incoming.getOutputStream())
ObjectInputStream inStream = new ObjectInputStream(new
                                BufferedInputStream(
                                incoming.getInputStream( )));
```

Sending and receiving objects are accomplished via the read and write methods of their respective stream class.

```
outStream.writeObject(packet);
object = inStream.readObject();
```

The newly received object is then passed along to a handler which uses the Java operator `instanceof` to determine the type of the object. Depending on the packet type different action are taken.

```
handler.parse(object);
```

If the object is a `SteerPanelPacket` the user wants to move the robot and has activated the steering panel in the GUI. The relative X and Y movement can be found by calling its respective get method.

```
SteerPanelPacket packet = (SteerPanelPacket)object;
int x = packet.getX();
int y = packet.getY();
```

If the object is an `ImagePanelPacket` the user uploaded specific locations that the robot should visit. They are represented as three different `ArrayLists` that contains points and can be received via their get functions. Scanning through the `ArrayLists`, in this case just to display the coordinates on the standard output, is easily done by the new for loops introduced in Java 1.5.

```
ImagePanelPacket packet = (ImagePanelPacket)object;
ArrayList<Point> waypoints = packet.getWaypoints();
ArrayList<Point> landmarks = packet.getLandmarks();
ArrayList<Point> goals = packet.getGoals();

for(Point p: waypoints) {
   System.out.println("(" + p.getX() + "," + p.getY() + ")");
}
```

In case an `EkahauPacket` is received the logical area visited by the robot has to be determined before taking appropriate actions.

```
EkahauPacket packet = (EkahauPacket)object;
String logicalArea = packet.getLogicalArea();
```

If the robot is near one of the mica motes, that is inside the logical area surrounding either the printer or the corridor, their respective temperature are updated by calling the appropriate `getTemperature()` method provided by the `moteHandler` class.

```
if(logicalArea.equalsIgnoreCase("printer"))
   temperature[PRINTER] = moteHandler.getPrinterTemp();

if(logicalArea.equalsIgnoreCase("corridor"))
   temperature[CORRIDOR] = moteHandler.getCorridorTemp();
```

Should the robot be in the vicinity of the Bluetooth MULLE, which is located in the meeting area, some additional rows of codes are required. This is due to that it is impossible to directly call the MULLE's get method. Firstly, a new thread is created, this circumvent the application from stalling. Secondly, the MULLE's start method is called which sets up a PPP connection to the MULLE and creates a TCP connection that is used to update the variable holding the temperature data. Finally, the get method can be used to receive the new temperature.

```
if(logicalArea.equalsIgnoreCase("meeting")) {
   if(!busy) {
      new Thread(new Runnable() {
         public void run() {
            busy = true;

            Mulle mulle = new Mulle();
            mulle.start();
            temperature[MEETING] = mulle.getTemperature();
            busy = false;
         }
      }).start();
   }
}
```

When the robot approaches the proximity of the base station the array containing all three temperatures are sent to the client which in turn updates the GUI.

```
if(ServerThread.isConnected())
   ServerThread.send(new SensorPacket(temperature));
```

Appendix J

Client

Before connecting to the server the logger is initialized, it outputs its data to the file `log.txt`. The data can latter be scrutinized either from the GUI or from a normal file reader.

```
Logger logger = Logger.getLogger("src.Scorpion.Network.Client");
try{
   FileHandler fileHandler = new FileHandler("log.txt", true);
   fileHandler.setFormatter(new SimpleFormatter());
   logger.addHandler(fileHandler);
}
catch(IOException e) {
   e.printStackTrace();
}
```

The IP address that narrates the server's location is read from the file written by the Main - Settings menu.

```
BufferedReader reader = new BufferedReader(new FileReader("settings.txt"));
ip = reader.readLine();
reader.close();
```

In much the same way as the server the client associates two object streams with the socket allowing objects to be sent between the client and the server.

```
Socket socket = new Socket(ip, 8189);
ObjectOutputStream outStream = new ObjectOutputStream(
                                 socket.getOutputStream());
ObjectInputStream inStream = new ObjectInputStream(
                             new BufferedInputStream(
                             socket.getInputStream()));
```

The client consists principally of two things, a loop that parses messages from the server and a send method that consigns messages to the server. Firstly, the loop determines the type of the received object then parses it accordingly and finally waits for a new object to arrive. The send function simply tries to send the object passed along as an argument to the server, it uses a `ReentrantLock` to prevent race conditions.

```
while(true) {
   Object object = inStream.readObject();

   if(object instanceof String) {
      ConsolePanel.addLine((String)object);
   }
   else if(object instanceof DisconnectPacket) {
      jpgReceiver.close();
      ConsolePanel.addLine("Disconnected from the robot server");
      break;
   }
   else if(object instanceof PictureSizePacket) {
      PictureSizePacket packet = (PictureSizePacket)object;
      long pictureSize = packet.getPictureSize();
      jpgReceiver.setPictureSize(pictureSize);
      jpgReceiver.read();
   }
```

```
   else if(object instanceof SensorPacket) {
      SensorPacket sensorPacket = (SensorPacket)object;
      double[] temperature = sensorPacket.getTemperature();
      TempLabelPanel.setTemp(temperature);

      for(int i=0; i<temperature.length; i++) {
         if(temperature[i] < 20 && temperature[i] > 0)
            logger.warning(name[i] + " temp low (" +
                           formatter.format(temperature[i]) + ")
                           degrees Celsius.");
 else if(temperature[i] > 25)
            logger.warning(name[i] + " temp high (" +
                           formatter.format(temperature[i]) + ")
                           degrees Celsius.");
         }
}

public static void send(Object packet) {
   sendLock.lock();
   if(isConnected) {
      try {
         outStream.reset();
         outStream.writeObject(packet);
      }
      catch(IOException e) {
         e.printStackTrace();
      }
   }
   sendLock.unlock();
}
```