MASTER THESIS IN COMPUTER SCIENCE
INF-3990

A Distributed and Parallell Robot Environment for Competitive Search and Rescue using a Display Wall for Visualization

Marte Karidatter Skadsem

May 15, 2006

Department of Computer Science
Faculty of Science
University of Tromsø, N-9037 Tromsø, Norway

MASTER THESIS IN COMPUTER SCIENCE
INF-3990

A Distributed and Parallell Robot Environment for Competitive Search and Rescue using a Display Wall for Visualization

Marte Karidatter Skadsem

May 15, 2006

Department of Computer Science
Faculty of Science
University of Tromsø, N-9037 Tromsø, Norway

Abstract

In this thesis we present the development of a distributed and parallel environment which offers functionality to robots to support them in their task performance. We want the environment to be a framework where students can experiment with robots, and in which we can arrange robot competitions.

Our motivation for this thesis is an earlier developed robot system at our department. It was used as an instrument to demonstrate the principles and practice of distributed and high performance parallel computing. The system was used by students on advanced courses on cluster architecture and programming, and popular competitions were held in it. The old system had many infrastructure demands and had to be closed. We want to make a new system that has less infrastructure demands and more functionality.

The environment has control over a certain amount of work space where the robots can operate. Within this work space, the environment offers functionality that includes context awareness, location, mapping, naming, and structured interfaces for interaction between the different components. Users can control the robots through the environment, and they assign tasks to the robots. Users can also download extensions to robots through the environment, and robots can upload data to the environment.

The state of the environment, the robots and the work space is visualized on a display wall. Users can interact with this visualization and assign simple tasks through it.

Acknowledgements

This thesis concludes my education in computer science at the university of Tromø. I would like to express my gratitude to all the people that have supported me during this time.

Fist and foremost I would like to thank my supervisors Otto Anshus and John Markus Bjørndalen for their encouragement, ideas and support.

I would also like to thank the technical staff at the department of computer science for support and new hard disks.

Thanks to my fellow students for all study related and non study related conversations and for all the breaks in the climbing wall.

Thanks to my family for believing in me. Special thanks to my mother, Kari Skadsem, whom I always can call, and always pushes me on.

Last, but not at least I would like to thank my boyfriend Åsmund Grammeltvedt. Thank you for your moral support, feedback and for your patience with me in the hardest periods.

TromsøMay 15, 2006

Marte Karidatter Skadsem

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and motivation

During the years 2000 to 2004 the department of computer science at the university of Tromsø worked on a robot system, called ROBO. ROBO was used as an instrument to demonstrate the principles and practice of distributed and high performance parallel computing. The system developed was used by students on advanced courses on cluster architecture and programming. The goal was that through the system they would get good understanding of distributed and parallel systems. Also they would learn how to utilize the resources in a distributed system in practice. The goal was also to make the system portable so that it could be used both in university courses, on exhibitions and other demonstration arenas [1].

The system consisted of several Lego robots with small on-board computers which operated inside an arena after some given rules. The arena was a physical frame on the floor that prevented the robots to drive out of reach. The robots had low processing, memory and I/O performance and

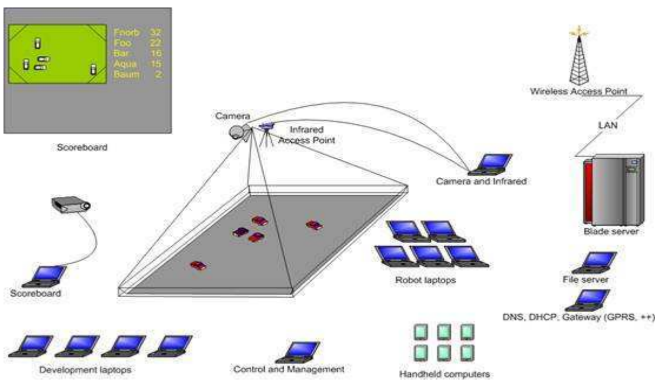


Figure 1.1: The technical solution of the old robot system.

were therefore supported by a computer each. They were also supported by a cluster and a file server. These were used to some heavy computing that the robots should perform. The location data came from a positioning computer with a video camera attached. The camera was mounted over the arena. A scoreboard computer monitored the state and progress of the system. This dat was showed with a projector. A control and management computer started and stopped the system and manipulated it according to user input. To make the system independent of other infrastructures and networks, an infrastructure computer provided network services [1]. All this is showed in figure 1.1.

As seen from the description above, the system had high infrastructure demands. It demanded a large room for the robot arena and lots of installation before it could be used. When everything were installed, the room could not be used for other purposes. All this installation work made the system less portable than intendent.

Another drawback was the nature of the Lego robots. They had low battery capacity and were hard to debug. The experience was that the students used much of their time understanding how the robots worked and less time on the distributed and parallel computing concepts.

In the end the project was put on ice. The infrastructure demands were too big and when the room situation changed, there were nowhere place it.

This thesis documents the development of a new environment for robots which demand less equipment and can be installed practically everywhere. The goal is to reduce the physical infrastructure and make the system more virtual. In addition to a replacement for the old system, we want this new system to support more than one type of robots. We also want that parts or the whole system easily can be replaced or extended. This is because we want to have a system that we can change as our demands change. In this way we hope that there will be continuing work with robots on the department.

One other motivation is the students' wishes to have robot competitions. The sudent assignments in the old ROBO system were made as competitions. The competitions drew lot of attention from other computer science students who also wanted to participate. Our intention is to make an environment in which students can experiment with robots on their own, and in which it can be held competitions.

1.2 Problem definition

The main purpose of this thesis is to develop a distributed and parallel environment that supports a dynamic number of robots with functionality. This functionality will aid the robots in their tasks.

The nature of the tasks can vary. They can be as complex as a “search end rescue” setting ([8]) or as simple as moving from one point to another.

This means that the environment must be able to handle all kinds of tasks.

Single robots or groups of robots enter and leave the environment dynamically. When robots enter the environment, they receive downloads that enable them to use the functionality offered.

The environment exists on servers and on each robot. Functionality and services offered by the environment include:

- Context awareness. The environment has information about the area in which the robots operate. This information is given to the robots so that they can react accordingly. Such information can be obstacles to avoid, difficult surface to drive on or air temperatures.
- Location of each robot. The environment keeps track of the location of each robot. The location data helps the robots in their navigation.
- A map over the area which the environment covers. This map contains all the location based information the environment knows about included robots' positions, known obstacles and objects discovered by robots. The knowledge is shared with the robots. They can either know the whole map, or just a piece of it.
- Naming of each robot. Each robot gets a unique name to be used whenever the robot contacts the environment or the other way round.
- Structured interfaces for interaction between the environment and the robots. These interfaces include reporting, monitoring and visualization of state and sensor data from the robots.

1.3 Requirements

We have a few more requirements to the environment. One thing the old ROBO system lacked was an easy way handle input to and output from robots while they were operating. This is a useful feature in most operations except when the robots are supposed to operate fully autonomously. Examples on output are a robot's sensor state and data found during execution of a task. Input can for example be new code or new tasks for the robot to execute. The Lego robots could be loaded with simple byte code, and the sensor state could be read. But this code is specialized for the Lego robts and can not be used by other robots.

We want the new environment to support this feature. It should be possible for the environment or a user controlling some robots to download data and code to robots. It should also be possible for robots to upload data to the environment.

The state of the environment and the robots operating within should be visualized on a display wall, or a computer screen if no display wall is available. This visualization is both for demonstration purposes and for users controlling robots to get an overview of the whole situation. New information that the robots find is reflected in the visualization. When for example a new obstacle is found, this will be reported to the environment and shown on the display wall.

It should be possible to interact with the visualization where this is appropriate. Following the earlier example; When a new object is shown, a user can get all known information of it by marking it with the mouse. It can also be possible to give tasks to robots via the visualization.

With these two features robots will be able to succeed in their tasks with feedback from users. The robots will in other words not be fully autonomous.

1.4 Limitations

The environment is meant as an expansible platform. It offers a few services, but there exist an infinite number of services that can be offered. In stead of supporting all possible services, we give the opportunity to add support for new services as they are needed.

There are no limitations to robot types that can operate inside the environment. The only requirement to robots is that they can use the existing interfaces. They must be able to contact the environment and use the interfaces it offers, and they must be able to receive connections from the environment and offer a interface the environment knows. Except for these, there are no specific requirements to the robots.

Because of this limitation to the robots, we have not focused on robot AI or robotics. But in order to test the environment, robots are needed. The developed robots are not optimized. They are only made for demonstration and testing purposes.

1.5 Method

The primary purpose of this thesis is to investigate how to make the environment described above. We have had an experimental approach. We have made a prototype that includes all features we needed and wanted. Then we have tested the prototype to see how it works, how it can be used and what it can not be used for.

1.6 Outline

The rest of this thesis is outlined as follows:

Chapter 2 describes the architecture of the environment.

Chapter 3 presents the design.

Chapter 4 describes the implementation.

Chapter 5 explains the tests made.
Chapter 6 presents and discusses the results of the tests.
Chapter 7 evaluates the thesis.
Chapter 8 gives an overview over related work.
Chapter 9 concludes the thesis.

Chapter 2

Architecture

2.1 Overview

There are four main components in the system; the infrastructure, the robots, the visualization module and a user. The infrastructure, the robots and the visualization module make the environment. See figure 2.1. All components communicate through a network. The user can only reach the robots by going through the infrastructure. He can interact with the infrastructure either by a direct connection or through the visualization module. The robots communicate with each other through the infrastructure. The visualization module shows a graphical output of the robots and their work space based on information from the infrastructure.

In this chapter we will describe the architecture of the infrastructure, the robots, the visualization module and the user.

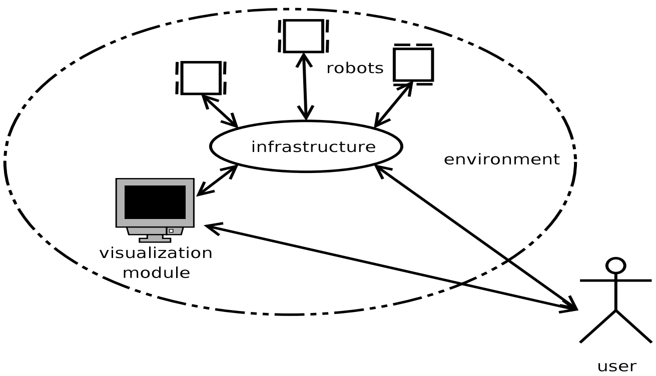


Figure 2.1: The system architecture. A user communicates to the infrastructure either directly or through the visualization module. Robots communicate with each other through the infrastructure.

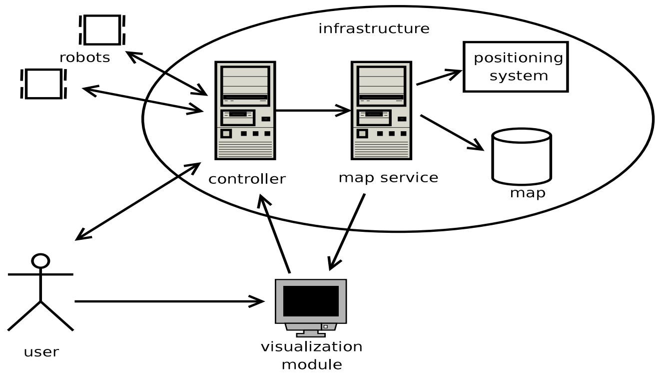


Figure 2.2: The architecture model. The infrastructure is divided in a controller and a map service. Users and robots communicate with the controller. Localization and visualization data are sent to the map service.

2.2 Infrastructure

We have divided the infrastructure in two parts; a controller and a map service. This can be seen in figure 2.2.

The map service takes care of localization functionality. The localization functionality includes storing a map and locating robots. The map contains all known, relevant information about the area in which the robots are operating, i.e. the robots' work space. Location of robots is done through a positioning system.

The controller takes care of communication with the robots and the user. It also takes care of task conveying and events coming from the visualization module. Following is a description of the two parts.

2.2.1 Controller

The controller offers structured interfaces for interaction with map service, robots and user.The robots contact it to get information of, for example, where other robots are or where they are themselves. A user contacts the controller to deliver tasks to robots or to get some information about the state of the environment.

The controller also takes care of conveying tasks and assisting in execution of tasks. A user tells the controller what task to be done by which robot. The controller then forwards the task to the desired robot. The robot will need some information during task execution, such as an updated map or the robot's position. The robot requests this information from the controller. The controller will then get this information from the map service and send it to the robot.

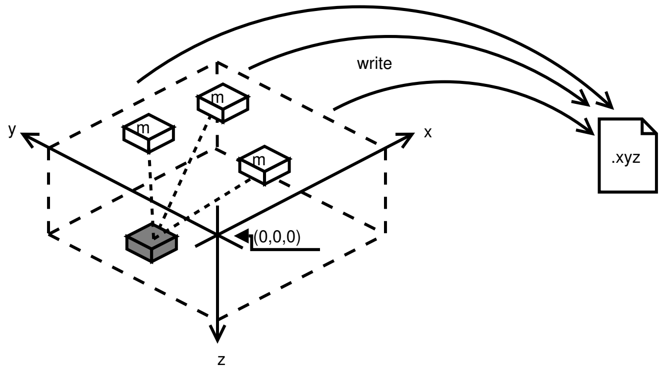


Figure 2.3: The positioning system. Monitors in the roof receive ultrasonic signals from tags mounted on moving objects. The positions are written to a *.xyz-file.

A user can also retrieve information from the infrastructure. This request will also go to the controller that gets the information from the right place and sends it back to user. The visualization module takes contact when a user has initiated some action through the graphical output (see section 2.3).

2.2.2 Map service

The map service gets localization data from a positioning system described under. The map is updated according to the positioning data. The map service is also responsible for sending data to the visualization module. All events that cause some changes in the map, and the the movements of the robots are sent to the visualization module to be shown graphically.

Localization

The map service uses a positioning system to get the location of the robots. The positioning system available for this project is HX5 ultrasonic positioning system, delivered by Hexamite. It works as follows (See also figure 2.3):

Signal receivers, or monitors, are mounted in the roof of a room in such a way that they cover the whole area which they shall monitor. Signal transmitters, or tags, are mounted on the moving objects to be monitored, in our case: the robots. The tags send out ultrasound signals which are registered by the monitors. The monitors are connected together in a network with a 4 conductor telephone cable. When they receive a signal, they forward it on this network. A network controller controls the network and communication with a personal computer. A program, called xyz.exe, reads the data from the network controller and calculates the position of the transmitters detected. The program stores the position in a file together with tag-id and the time the signal was received. A user or other programs can read this file

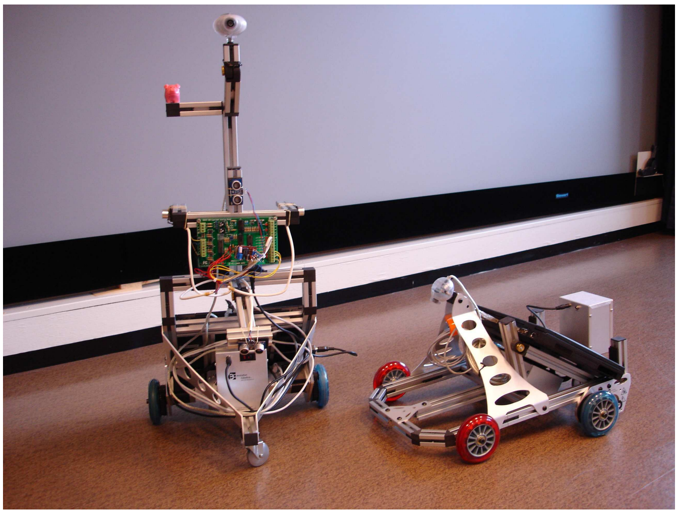


Figure 2.4: A few of the ER1 robots that we have used.

either in real time as it is written, or later. In our case it is the map service that reads the position data.

2.3 Visualization module

The visualization module uses a display wall to produce a graphical output of the robot's work space with all its contents. The graphical output is made based on data sent from the map service. We also want to produce graphical output of monitor data from robots. Monitor data from different robots must be clearly separated from each other. Which robots to monitor at a given time is decided by user. We want to see both the map and the monitor data at the same time and clearly separated from each other, so that it is easy for users to get an overview of the current situation.

2.4 Robots

The robots used are ER1s delivered by Evolution Robotics. Figure 2.4 shows a few of the ER1s we have available. They are delivered as kits that can be assembled in many different ways, but the hardware and software is the same on every one, so they are homogeneous.

The ER1 robots consists of three hardware components; a notebook on which the software is running, a robot control module (RCM), which controls the motors that drives the robot, and a web camera used for obstacle avoidance and object recognition. The components are shown in figure 2.5. The camera and the RCM are both connected to the notebook with USB.

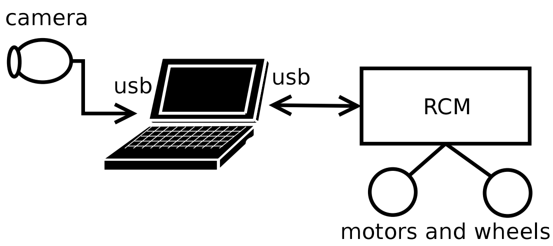


Figure 2.5: The robot architecture. The robot consists of the Robot Control Module (RCM), two motors with wheels, a notebook and a USB web camera.

The RCM, and thus the ER1, is controlled from a robot control center (RCC). The RCC is a program running on the notebook and can be accessed from a graphical user interface or a command line interface. The graphical user interface is used to make simple behaviors for the robot. For more "brain", we write our own program that accesses the RCC from the command line interface. The communication with the command line interface is done via a telnet connection. Commands and responses are sent as strings over the telnet connection. From the RCC's point of view it does not matter where the telnet connection comes from. It can be from the robot's notebook or a remote host.

2.5 User

A user wants to retrieve monitor data from robots, he wants to get a full overview of the situation, and he wants to give tasks to robots.

The full overview of the situation with all the robots and their work space is got from the visualization module. The user can interact with the map and do simple tasks through it, like choosing a robot to monitor. Monitor data can also be retrieved by telling the controller part of the infrastructure which robot or robots to get such data from. Task assignment is done through the controller. The user tells which robot to perform which task, and the controller convey this to the right robot.

The user interacts with a user application. This application takes care of the communication with the controller. When a user gives a command, the user application sends it to the controller, and deliver responses back to the user.

In a competition situation there may be more than one team operating in the work space. This means that more than one user can be connected to the interface concurrently, so the controller must be able handle more than one user at the same time. The controller must also assure that members of one team can not control robots of an other team.

Chapter 3

Design

3.1 Overview

In this chapter we will look at the design of the environment. First we will present the design of the infrastructure introduced in chapter 2. Then we will present the design of the visualization module, the robot design, the user application design, how the navigation is designed, the different interfaces, and the different states the environment can be in.

3.2 Infrastructure

Figure 3.1 shows the design of the infrastructure. The arrows indicates the communication lines. The dotted arrows show the communication lines going over the network. These communication lines use XML-RPC.

XML-RPC is a remote procedure call protocol that uses HTTP as the transport and XML as the encoding. It is a simple protocol that allows complex data structures to be sent across the network, and it can be used across different platforms. We have chosen XML-RPC because of its simplicity and the transparency it gives. It supports few but quite complex data structures as parameters for the procedure calls [13].

In this section we will describe the design of the two parts of the infrastructure, the controller and the map service, presented in the previous chapter.

3.2.1 Controller

The controller must handle events from the user, the visualization module, the map service and the robots. The interaction with the different actors is done through three interfaces; the user interface, the map service interface and the robot interface. An event handler acts upon the events coming from

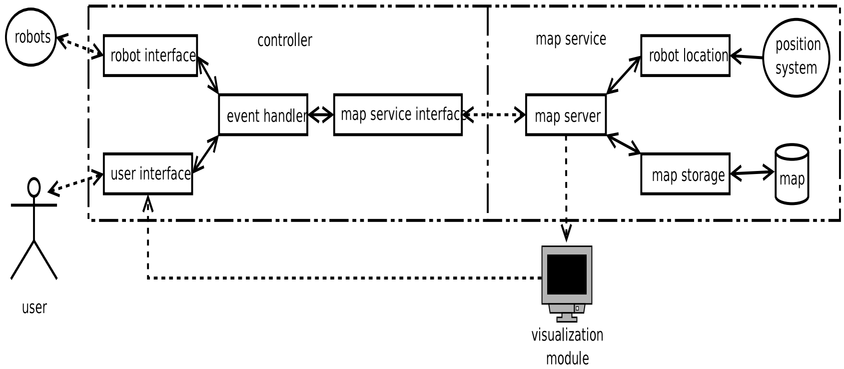


Figure 3.1: The infrastructure design. Arrows indicate communication lines. The dotted arrows shows communication lines that use XML-RPC

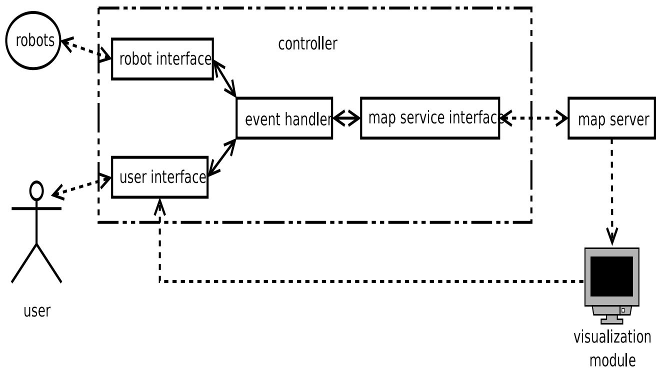


Figure 3.2: The controller design. The map service, robots and users connects to different interfaces. An event handler handles events from the interfaces and redirects them to the right place.

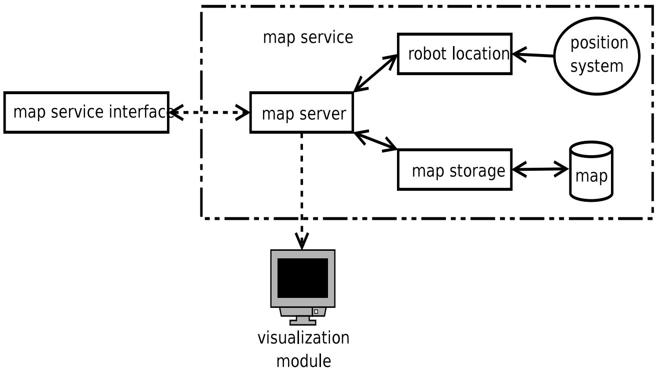


Figure 3.3: The map service design. A map server module handles events coming from the controller and the visualization module. A robot location module gets data from the positioning system. The map is stored in a map storage module.

the interfaces. See figure 3.2. How the components work together is best described with an example:

Assume that the user wants to move a specific robot to point A. He uses the controller's user interface to express this task. The user interface makes the event handler aware of the task. The event handler gives the task to the robot interface, which sends it to the robot. The robot navigates to the point with help from a map. To get a map, or to update the existing, the robot must send a request to the controller's robot interface. The interface sends the request to the event handler which redirect it to the map service interface. The response, i.e. the map, is sent from the map service interface, through the event handler and to the robot interface which sends it back to the robot. The robot executes the task and when that is done, it reports this to the robot interface. The event handler sends a message to the user interface that the task is completed.

The user can also give some commands through the visualization module. He uses a mouse to initiate these commands. The mouse actions are sent from the visualization module to the user interface.

It is not only the user that can initiate some action. Both the robots and the map service can trigger events in their interfaces that the event handler needs to react upon. What these events might be is described later in this chapter.

3.2.2 Map service

The map service is responsible for localization and storing the map. More precisely its responsibilities are:

- Locating the robots by using the positioning system.
- Making and maintaining the map (i.e. the data structure representing the area the robots are operating in). The map must be able to store all the different objects that can be in the area (i.e. robots, walls,things that the robots can find etc), where they are and what they are.
- Give data to the visualization module.
- Response on events coming from the controller.

How the map service is designed can be seen in figure 3.3. A map server module oversees and controls all that happens. This includes to take care of the communication with the controller and the visualization module. Information about the work space, known objects and obstacles are given by the controller. The storing module stores this information and creates a map from it.

The map is basically a system of coordinates, where the coordinates correspond to real world positions. When an object, an obstacle or a robot is registered at a position, it is stored at the corresponding coordinates in the map. It is the map that is visualized in the visualization module. The visualization module uses the information stored in the map and shows it graphically.

The positions from the positioning system are handled by the robot location module. When a robot wants to know its position, the map server asks the robot locating module for it. The robot location module returns the position, and the map server send it back to the controller. For more description on how localization is done, see section 3.6.

3.3 Visualization module

The visualization module is responsible to make a graphical output of the robot's workspace with all its contents. The data comes from the map service as it stores new information. The visualization module also handles mouse events from a user in the graphical output. It makes an appropriate choice on how to act upon the event, and tells this to the controller.

For and example of how the graphical output looks like, see figure 4.4 on page 38. How a robot's monitor data is presented can be seen in figure 4.5 on page 40.

The visualization module's design is shown in figure 3.4. The communication with the map service uses XML-RPC. The user uses a mouse to trigger events in the graphical output on the display wall.

The visualization module does not have to run for the rest of the environment to work. The graphical output is only meant as a support for the users so they easily can get the full overview.

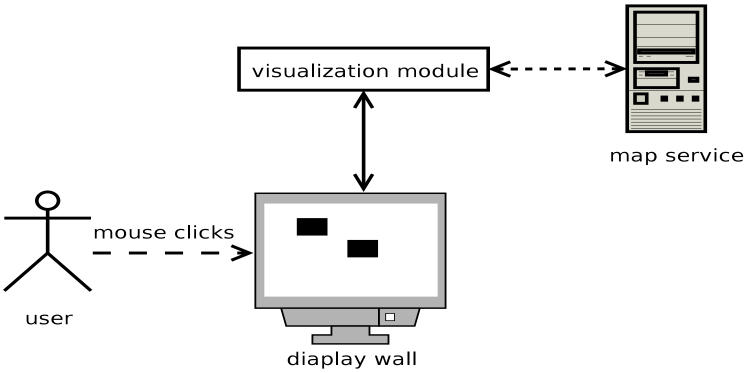


Figure 3.4: The visualization module gets data from the map service and makes a graphical output of it. Users can interact with the graphical output by using a mouse.

3.4 Robots

Figure 3.5 shows a high level view of the robot design. As mentioned in the architecture chapter, if a program wants to control the robot, it needs to communicate with the command line interface of the robot control center (RCC). This communication is done via telnet, and the RCC does not care if the program is run on the same notebook as itself or on a remote host. We have chosen to run the program on the robot's notebook because of fault tolerance. If the program is run on a remote host, it is dependent on having a reliable network connection all the time. If the network connection is lost, the robot is lost. A connection between two processes on the same computer is not lost. The robot is still dependent on a network connection on order to keep in touch with the rest of the world, but it is possible to save the robot even if the network connection is lost for a short period. If, for example, a robot discovers it has lost its network contact, it stops all its task execution and movement, and tries to get contact again.

A ER1 has two sensors; a rotation sensor that is used to control how much the wheels rotate, and a web camera that can be used for object recognition and obstacle avoidance. These are two important features for a robot to operate autonomously. From earlier work with the robots ([12]) we know that the obstacle avoidance feature is not working very well. The object recognition feature is not very good, but it can be used. In addition to object recognition, we want to be able to take pictures and video streams of the surroundings. Picture taking is not supported by the RCC. After some testing, which are described in the discussion and evaluation chapter (chapter 7), we ended up using two cameras. The RCC uses one for object recognition and our robot program uses the other for video making.

Figure 3.6 shows how the robot program is designed. From now on we

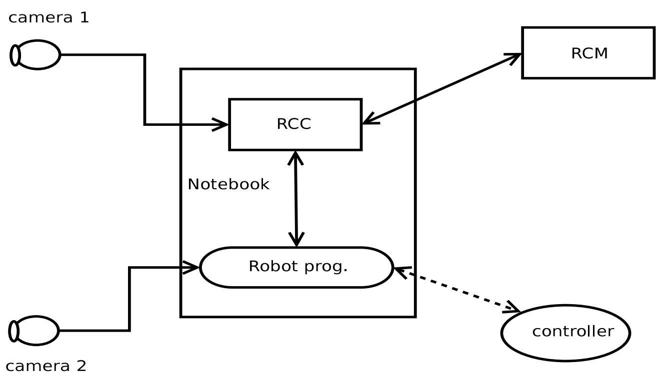


Figure 3.5: High level of robot design. The RCC and the robot program runs on the notebook. The RCC communicates with the RCM. The robot program communicates with the controller. One web camera is used by the RCC, the second by the robot program.

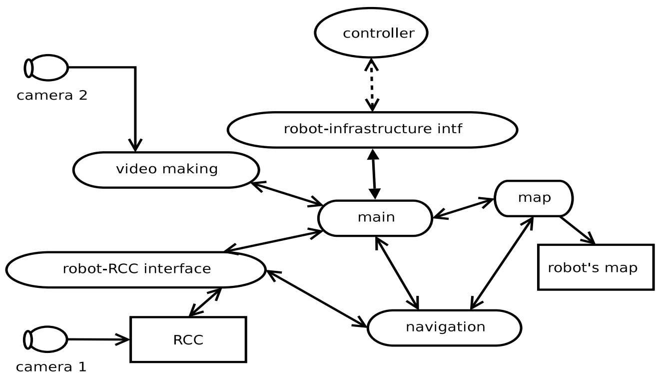


Figure 3.6: The different parts that makes up the robot program. The robot-infrastructure interface is downloaded from the infrastructure during start up.

denote this program the "robot" because it is here all the functionality lies that makes our robot. The communication with the RCC is done through an interface module, the robot-RCC interface. This module takes care of all the extra overhead with the telnet communication. The extra overhead is for example to make sure that all commands are received by the RCC, and to make a new sending if it was not received.

Accessing the second camera is done through a video making module. This module takes care of all picture taking and video making. The map module makes the robot's map and maintains the its state. It is also responsible for finding safe paths that the robot can follow.

The navigation module is told where the robot is and where it is going. The module accesses the map module to get a path to follow. Then it calculates distances and directions to move and uses the robot-RCC interface module to move the robot.

This design is developed for the ER1 robots we have used. Other types of robots with different design than this should also be able to operate in the environment. In order to get this work as smooth as possible, all robots must download an interface from the controller that they use in communication with the controller. This interface is the same for all types of robots and it is each robot's responsibility to integrate with this interface. How this interface is downloaded is described in subsection 3.4.2.

The last module is a main module that is in charge for everything. It is responsible for the communication with the controller through the interface, handling downloads and uploads and executing tasks. This is the module that students will replace in later projects and competitions.

3.4.1 Task Execution

A task is some action performed by a robot. It can be as simple as to move from A to B, or more complex like to explore a big area and report all objects found.

The robots can have some predefined tasks they can perform without receiving new code on how to do it each time. Such tasks can be tasks that we expect to occur often, like moving to a given point. Each robot reports its predefined tasks to the controller. They are announced to the user through the user interface so he will know how to use them. Our ER1 robots have three predefined tasks, namely:

1. Move to a given point. The robot will find a path to the given point and move to it.

2. Follow a given path. The user will give a path which the robot shall follow.

3. Examine a small, given area. The robot will move to the given point

and take a video of the surroundings of this point. The video is then sent to the controller to be stored.

Robots are given new tasks by the controller. When a user has assigned a new task to a robot, the controller checks if the robot is executing some task already. If so, the controller stores the task so that it can be given later. When a robot reports that it is finished with one task, a new one is given to it if there are pending tasks for this robot.

Sometimes we want to interrupt a robot in its task execution. The reason for this can for example be when we see that a robot is on its way towards some stairs or when a new and more important task is ready to be executed. Because of this, when a task is started, we must assure that the robot can be interrupted. And we must be able to stop the robot from moving with immediate effect. How this is done is described in the implementation chapter, section 4.4.3.

3.4.2 Downloads and uploads

There are basically two types of downloads a robot can receive from the infrastructure; code to be executed and new extensions. The two types need different treatment. To execute some given code which we have no idea of what contains is a bit risky. For all we know, the code can take completely control of the robot and the robot can be lost for the environment. We assume that the code sent to robots is of the "kind" type, sticking to the rules. The reason we allow given code to do what it wants is that we want to have the opportunity to change the whole robot's software on the fly. This cannot be done if we set some restrictions to what the code can and cannot do.

Extensions are code that extend a robot's functionality. An example is a set of new tasks the robot can perform. When an extension is received, it will be imported in the existing code so that it is ready to be used. The existing code can not use the new features unless it receives code for doing so.

The most important download is the interface used in communication with infrastructure's controller. This download is done during the start up phase described in section 3.8.1. This is a standard interface that ensures all robots look the same to the controller. The robot program interfere with the interface when it communicates with the controller. The interface hides the controllers complexity from the robot and the other way round. A more detailed description of the interface's contents can be found in section 3.7.1.

Since the main module is responsible for the communication with the robot-controller interface, it is also responsible for the downloads and uploads. The transmission and receiving of data is done in the interface, but what to be sent and what to do with the data received is decided by the control module.

In the previous section we mentioned that in one of the predefined tasks the robot should take a video of its surroundings and send it to the controller. This is an example on data upload from robots to the infrastructure. Other data that can be uploaded is a robot's state. In order to do an upload, a robot needs to know the destination, so uploads are never initiated by a robot. They are told by the controller or the user what to upload where.

3.5 User-application

The user-application is the connection between the user and the infrastructure. It gets input from the user and send requests to the infrastructure's controller based on this input. The user-application's goal is to make an interface to the user that is easy to use.

The best would be if this interface were graphical. We have not managed this because of time constraints. The interface we have developed is text based. Users meet a set of menus and make choices from these by typing some number or some text. Through the menus, the application decides what the user wants and sends the request over to the controller.

The user application is made so that it can determine what a user wants before sending the request to the controller. This implicates that the menus and decision making are made based on the infrastructure's basic functionality. It also implicates that the menus are updated all time. When we here speak of the menus we mean both the text menus presented to user and the functionality that translates user choices to valid requests to be sent to the controller (i.e. the decision making).

When new functionality is added to the infrastructure, the menus must be updated so that they reflect the changes. If they are not updated, no user will know that new functionality is added or how to use it.

Following is a description of the initial menu offered to users.

3.5.1 Main menu

The main menu looks like this:

```
****  WELCOME!  ****
********************
What do you want to do? (Make a choice)
1. Get List of robots
2. Monitor robot
3. Give task to a robot
4. Give new code or module to a robot
5. STOP ROBOT
6. Quit
```

This is what the user meets after the different choices:

1. **Get List of robots.** This will return a list of all the robot tags and their last known positions.

2. **Monitor robot.** The user will be asked to write the tag of the robot he wants to monitor. Then a text based monitor data from that robot will be shown. (If the user wants to see the monitor data graphically, he must use the visualization module.)

3. **Give task to a robot.** After this choice the user will be asked to write the tag of the robot he wants to give a task to. Then a list of this robot's predefined tasks will appear and the user is asked to write in the name of the task he wants preformed. After this, the user is asked to write the arguments to the task. Arguments can for example be the coordinates to the point where the user wants the robot to move. The user must also say if the robot should be given the new task right away, i.e. interrupt the robot, or if it can be executed when robot asks for a new task. Then the task is sent to the controller. The user can make other choices and do other things while the task is executed. A notice will show when the task is done.

4. **Give new code or module to a robot.** The user will be asked to type the tag of the robot the code or module should be given to. Then he must write the name of the file which contains the code. And before it is sent, he must tell if the it is a module extension he sends or code or be executed.

5. **STOP ROBOT.** User must give the tag of the robot he wants to stop.

6. **Quit.** Quits the user application.

3.6 Navigation

The Hexamite positioning system turned out to not work. It arrived approximately in the middle of the project period and after some weeks of testing, we concluded that it could not be used. The design and implementation were at this point based on that the positioning system would work the way it was supposed to, and the development process had evolved so far that it was hard to turn around and do it all over. We decided to keep the design the way it was, and simulate data coming from a positioning system.

The reason for this decision is that we do not have any other way of doing positioning. We tested one alternative navigation method. One of the technical staff in our department made some ultrasonic sensors that consist of a transmitter and receiver, and that measure the distance between the sensors and objects around them. These sensors were mounted on one of the robots and used to avoid driving into walls. The sensors were made very late in the project period, and we did not manage to incorporate this method properly into the system. A description of what we did is found in appendix B.

The rest of this section describes the navigation design made based on a working positioning system like Hexamite.

3.6.1 Tracking vs. guidance

A positioning system like Hexamite gives two possibilities of finding positions; tracking and guidance. Tracking refers to when a central unit, for example a computer, monitors the positions of moving objects relative to fixed points. That is, putting the monitors at fixed positions and the tags on the robots as shown in figure 2.3. The moving objects are not aware of their positions unless they are told by the central unit. Guidance refers to when moving objects calculate their own positions relative to satellite objects. That is, putting the monitors on the robots and the tags at fixed locations. We are using tracking. This is because of the competition aspect. For example, the position data can be encrypted. In order to get their positions, the robots must decrypt this data. Fast decryption methods will then be one of the challenges in the competition.

3.6.2 Navigation method

As described in section 2.2.2 the positioning system calculates positions in a program called xyz.exe and writes the positions to a file called *.xyz. This file is read by the robot location module in the map service. If the positions need to be recalculated to fit the map, this is done in this module.

Because of the not working position system, we ended up giving the robots their positions when they ask for them. Between these requests, the robots navigate by using dead reckoning. This navigation method is best described as "walking blindfolded". The only known information about where to go is on the form: "three steps forward, turn to the left and go two steps forward". If we take the slightest smaller steps than required and turn a little bit less than 90 degrees, which is very easy done, we will not end exactly where we wanted. With regular updates from the positioning system, the deviation will quickly be corrected.

3.6.3 Path planning

Except for getting their positions on request, all the navigation is done by the robots. The robots have their own copy of the map. When some new object is detected, this is reported to the map service and propagated to all the robots. The robots use the map for path planning. Path planning is the task of planning the motions of a robot so that it avoids collisions with objects in the workspace.

Path planning is not a part of this thesis. But it is hard to work with robots that should operate freely within an area without any planning of their motions. During resent robot projects in our department there has not been developed any algorithm that we can use in this thesis. The RoMo project [12] solved this problem by using a central map server. All robots contacted this to get paths to follow. In this thesis we want to move the path planning to the robots. This is because different robots with different sensors can plan their movements and paths differently. And path planning is a good theme for robot competitions.

We have used a very easy and far from optimized path planning method. The basic thought is that every robot get their own copy of the map. This map is two dimensional. This is because it is easier to work in two dimensions than in three, and because our robots drive on the floor in very stable environments. All furniture and other objects that are placed on the floor are marked in the map so that the robot will avoid driving into them. This map is then split in equal sized squares. Then we have a two dimensional array and we can use common search algorithms to find paths between two points.

We have chosen to use the A* search algorithm for our path planning. A short description of this algorithm can be found in appendix A. A* is one of the most used path finding algorithm in AI game development today ([3]). A* will always find the best way between two points, if a path exist between them. It is also relative effective and easy to implement.

3.6.4 Obstacle avoidance

As mentioned in section 3.4, the obstacle avoidance software delivered with the ER1s is not working. This means that there is no way a robot can avoid running into walls, objects like tables and chairs, or other robots. Combining this with dead reckoning navigation can cause chaos as robots drive into each other and getting out off track.

We tried to solve this by giving each robot their own piece of the map where they could operate freely without risking a crash with another robots. But this solution raised many questions, and because of the limited time and because path planning is not a part of this thesis, we decided not to use much time on developing this solution. The solution and the questions we met are described in the discussion and evaluation chapter (chapter 7).

We have chosen to give each robot the whole map and assume that the user will make sure no robots crash. This is of course far from an optimal solution. We find it sufficient here because finding a better solution can be part of the competition challenges.

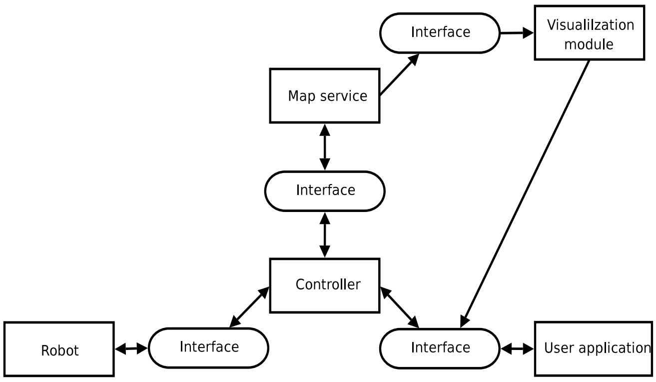


Figure 3.7: The interfaces between the controller and the map service, the robots and the user application, and between the map service and the visualization module.

3.7 Interfaces

According to the problem definition, the environment should provide "Structured interfaces for interaction between the environment and the robots. These interfaces include reporting, monitoring and visualization of state and sensor data from the robots." In this section we will describe these interfaces in addition to the interface between the user and the controller, the interface between the controller and the map service, and the interface between the map service and the visualization module. See figure 3.7.

3.7.1 Controller - robots

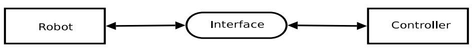


Figure 3.8: Controller - robot interface.

The infrastructure, or more precisely the controller, must be able to contact the robots in order to get monitor data and to give tasks. The robots must be able to contact the controller in order to use the services offered. Both must be able to initiate contact with the other because the controller does not know when the robots need it and the other way round.

We can look at this communication as two separate channels. One controller-to-robot and one robot-to-controller. Each channel can be modeled in two ways; As a client-server model or as a pull based model.

In a pull based model the controller has a queue for each robot where it can put tasks and commands. The controller must be able to reorder and remove tasks and commands pending in the queue. The robots will access the queue only when they are finished executing their current tasks. But what if we want to abort a current task? This will be a problem when the controller only can give commands through the queue.

In a client-server model, the robots will act as servers and the controller, as a client. The controller can at any time send a new command or task to a robot, and the robot will execute it upon arrival. Tasks can be executed concurrently with serving new commands. Because of this flexibility, the client-server model is chosen for the controller to robot communication.

For the robots-to-controller channel, the client-server model is also chosen. In a pull based model there is no guarantee when a request will be executed only that it eventually will. The robots depend on responses on their requests before they can continue execution. A pull based model will delay them. Thus, in this communication channel the controller acts as a server and the robots as clients.

I will now describe the two communication channels in more detail. As we are using XML-RPC as the communication mechanism, are the interfaces described with the callable functions.

Controller to robots channel

Each robot offers the following interface to the controller:

- executeTask() - Execute a task given by the controller.
- runCode() - Run some code given by the controller. The robot does not know what the code does.
- importModule() - Import some module that extends the robot's functionality.
- getMonitorData() - Return relevant sensor data and context information.
- updateMap() - Update the robot's map because some new information has been registered.
- stop() - Stop the robot's movement immediately.

Robots to controller channel

The controller offers the following interface to the robots:

- welcome() - Welcome and register new robots to the environment. A robot gives information about itself, like its ip-address, and gets in return it's position and a map.
- taskOptions() - Register a robot's predefined tasks.
- reportPosition() - Robot reports a new position.

- getNewMap() - Return a new, updated map to the robot who requested it.
- storeFile() - Store uploaded data from robot.
- taskDone() - Register that a robot is done executing its task. Returns a new task if any is pending.
- robotDisconnect() - A robot reports it leaves the environment.

3.7.2 Controller - user

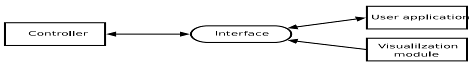


Figure 3.9: Controller - user interface.

The communication between the controller and the user-application uses the client-server model where the controller acts as the server and the user-application as the client. The same goes for communication between the visualization module and the controller. Following is a description on the interface offered to the user-application by the controller. The visualization module uses only two of these, namely monitor() and giveTask().

- getRobotList() - The controller returns a list of all the robots it knows about. More precisely their tags.
- monitor() - Takes a robot tag as input and returns this robot's monitor data in text format.
- giveTask() - Gives a task to a robot.
- sendCode() - The user send some code to be executed or a module to import by a robot.
- checkDone() - Checks if a robot has reported that its task execution is done.

3.7.3 Controller - map service

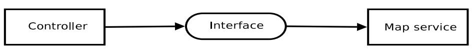


Figure 3.10: Controller - map service interface.

In the communication between the controller and the map service, the controller gives new information to the map service, or it needs some information from the map service. The client-server model is used in this communication, where the controller acts as the client and the map service as the server. Following is a description of the interface offered to the controller by the map service

Controller to map service channel

- makeMap() - Make the map. The controller must provide all the information needed to make and manage the map: the size of the map, all known information about the area and the size of the robots map pieces.
- getMap() - Return a robot's map, and where the robot is placed within it.
- moveRobot() - Reports a robot's new position to the visualization module. Also verifies the reported position with the positioning system. Returns the correct position.
- markNewObject() - Store information of a new object in the map.
- noContactRobot() - Mark in the map that a robot is lost.

3.7.4 Map service - visualization module

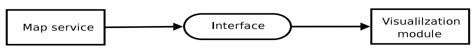


Figure 3.11: Map service - visualization module interface.

This interface is based on the client-server model since the visualization module can be started after the rest of the infrastructure has been running for a while. Following is a description on the interface offered to the visualization module by the map service.

- startUp() - Returns all data needed to get a graphical output up and running. This data includes size of workspace and known object, walls and other obstacles within it. Map service registers that the visualization module is running and starts to store new events to be displayed.
- getKnownRobots() - If the map service has been running for a while, this will return a list of all the robots in the work space and their positions. This will always be the newest information about the robots. The map service will not store the whole history of the robots movements
- getNewEvent() - Returns new events to be shown in the graphical output.

- newUserEvent() - Visualization Module sends the command initiated by a user via the graphical output.
- disconnect() - Visualization Module calls this when it shuts down. Map service registers this and stops storing events.

3.8 Phases

We will now describe the three main states the environment can be in; start up, execution and stopping.

3.8.1 Start up

Controller: The main purpose for the controller during start up is to get the event handler and all the interfaces up and running. All the interfaces must run concurrently because they act as front ends to the different actors which can connect to the controller at all times.

Map service: The first thing that happens during the map service's start up is that the map is made. In order to do this, the map service needs information about the map's size and all known walls, obstacles and objects that have known positions. This information is gotten from the controller.

When the map is made, data needed for the graphical output is sent to the visualization module. The map service also makes contact with the positioning system before it is ready to enter the execution phase.

Visualization Module: The visualization module can start independent from the other parts. This means that it can be started during execution phase of the other parts.

During start up it gets all needed information form the map service. The map service will store this information during its execution so that it can be retrieved at any time.

If the map is small or very big, a "zoom factor" must be calculated. The zoom factor denotes the amount of pixels needed to draw one length-unit. Assume that the robot's work space has the size $200x200$ cm. In a 1:1 mapping, 1 cm in real world would correspond to one pixel. In this case we would get a picture with $200x200$ pixels. This may be too small on the display wall or even on a computer screen. In other cases the map may be too large to fit the screen it is shown on. In these cases, the zoom factor is used to scale down the map. The zoom factor is used to get a workable size of the graphical output.

Robots: When started, the robots will call the infrastructure's controller to get the interface downloaded. When this is loaded, it starts the server that will accept events from the controller. The start up phase is finished when the first task is received.

User application: The user application does not have a start up phase. It is however not allowed to start before the task scheduler has entered the execution phase. If a user tries to initiate some action before that, he will be asked to wait.

3.8.2 Execution

Controller: During execution, the controller receives events from the different actors and responds to them. The event handler handles all the incoming events from the different interfaces. The controller stores all known information of all robots operating within the work space. This information is stored here because the task scheduler is the one that most frequently accesses and uses this data.

Map service: The map service updates the map according to data from the controller, sends updates to the visualization module, and delivers localization data to the task scheduler.

Visualization module: The visualization module will continue to receive new data from the map service and show them graphically. It also registers mouse events from the user and send the user's requests to the map service

Robots: Robots are occupied with task execution during execution phase. Task execution runs concurrently with the server that handles events from the controller. When the robot has no task to perform, it will ask the controller for a new task and stay idle until one is received.

User application: The user application serves the user as best it can. The user expresses his wishes by using menus. And based on these will the user application send events to the infrastructure's controller. It contains no data or state except for a small history log of the users actions. This history log is only used for the user application itself in order to remember which menu to show and which requests correspond to the responses from the controller.

Concurrently with serving the user is the user application listening on events from the controller. When a user initiated task is done, it is told the user application, which tells it to the user.

3.8.3 Stopping

Controller: If the controller is stopped while robots are executing tasks, it is considered as a system crash and recovery must start. Only when all robots are idle and the user application is stopped, can the controller stop running without lots of fuzz.

Map service: The map service can not stop execution before after or at the same time as the controller. The user can choose to stop the visualization during execution phase without stopping the whole map service.

Visualization module: The visualization can be stopped at any time independent from the other parts. It is done by closing the window to the graphical output.

Robots: How to stop the robots totally and not only stop their movement can vary. One way to stop them is to make all go to the same point where they are turned off manually. Another way is to send some code that stops the robots.

Robots can stop whenever they want. If they stop during task execution they are considered as missed.

User application: When the user is finished with all he wants he stops the user application. The user application can be stopped and started again many tomes during the infrastructure's execution phase.

Chapter 4

Implementation

4.1 Overview

Before we look at the implementation of the system's different parts, we will discuss some general implementation requirements and choices.

There is one important requirement to the implementation. It must support that different components run on different platforms. The robot control center (RCC) runs on Microsoft Windows. The display wall and the computers connected to the display wall lab all run on Linux or Mac OS. So at least the visualization module must run on Linux. Because most of the machines we use run on Linux, the rest of the infrastructure and the user application also run on Linux.

The code is written in Python. We made this choice because Python is highly portable among different platforms. Also, XML-RPC is easily done in Python. Python has a library module called `xmlrpclib`. This library hides all the details of connecting to a server, sending a request, and receiving a response. For setting up a server, Python offers a basic server framework in the library module `SimpleXMLRPCServer`. This library hides the details of setting up the server, handling requests and sending responses.

There is one drawback with XML-RPC which the programmer must remember. XML-RPC has no support for `None`. Since all Python methods return `None` as default, some transparency is lost. The programmer must make sure that all methods that can be called by RPC must have a return value.

In the rest of this chapter we will describe the most important implementation issues of the system. First we look at the infrastructure, then we will look at the visualization module, the robots, and at last we will describe the user application.

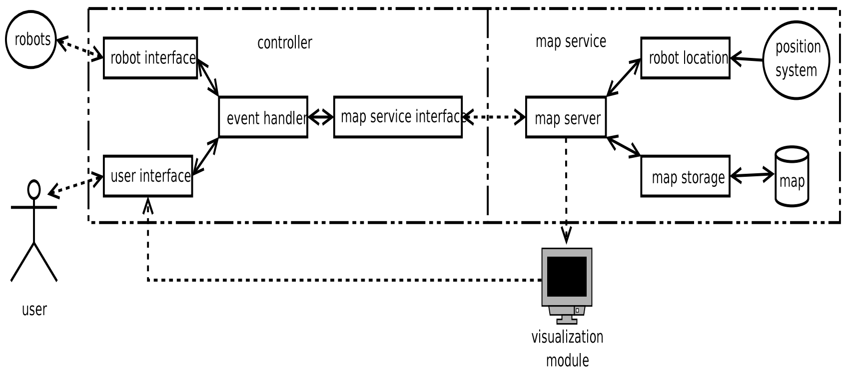


Figure 4.1: The infrastructure design. Arrows indicate communication lines. The dotted arrows shows communication lines that use XML-RPC

4.2 Infrastructure

In this section we will look at the implementation of the infrastructure. The section is divided in a description of the controller and a description of the map service.

A copy of figure 3.1, the infrastructure design is put here as a reference. See figure 4.1.

4.2.1 Controller

Roughly we can say that the controller is a server handling requests from several different types of clients. It offers interfaces special made for each client type in order to serve them the best way. The interface's job is to redirect requests from clients to the event handler, and from the event handler to clients. The event handler is the boss that decides what to do with different requests and where to send responses. The map service interface is the only one not running a server. This is because all communication between the controller and map server is initiated from the controller.

Each interface server runs in its own thread. They get the address and port number to use in initialization of their servers upon initialization. Each interface server use different port numbers. The address and port numbers are hard coded into the controller. The interfaces also get a pointer to the event handler, so that they know where to redirect requests.

The robot interface must be able to handle more than one concurrent robot connection. So this server is made asynchronous. The user interface should also be able to handle more than one user at a time, but the current implementation supports only one.

Two of the interfaces acts as clients, namely the map service interface and the robot interface. This is so they can be able to redirect requests from the event handler to the map service or robots. The map service address and port number are hard coded into the controller. All robots use the same port number on their servers. The different addresses are stored at the event handler together with the other known information of each robot. Each robot has to tell its address when it calls the *welcome()* method and registers for the first time.

The connection to the map service is established during initialization of the map service interface. Connections to the robots are made on the fly as needed. This means that each time a request is sent to a robot, a new client is set up.

The list of all known robots is stored at the controller. For each robot is the following information stored:

- A tag unique for each robot.
- The robot's ip address. Used to connect to the robot.
- The robot's last reported position.
- Whether the robot is connected to the controller or considered lost.
- The current task that the robot executes.
- A list of tasks that the robot has executed and how execution went.
- A string containing the robot's possible tasks.

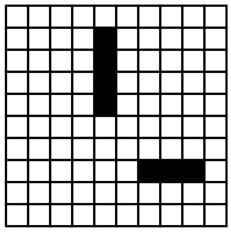
Also, the controller stores a list of tasks not given to a robot yet. It exists one such list for each robot.

As mentioned earlier, the controller is not intelligent when it comes to task assignment and following up task execution. It does not store any state and if it crashes, all data is lost. The design is so that this can be implemented, but due to time constraints, again, this is not implemented. We considered it not to be a crucial part to test and demonstrate the use of the infrastructure.

4.2.2 Map service

The map service's main function is to make, store and maintain the map. The map is implemented as a list where every element is again a list of empty strings. In figure 4.2 we can see how a map consisting of $10x10$ units is implemented. Walls are marked in the representation with a `w`.

Objects discovered by robots are stored at the map service, but not in the map. They are stored in a dictionary indexed by an object's unique identification assigned upon registration. Information stored about each object includes its position and the tag of the robot that discovered it. If pictures



```
[ ['', '', '', '', '', '', '', '', '', ''],
['', '', '', '', 'w', '', '', '', '', ''],
['', '', '', '', 'w', '', '', '', '', ''],
['', '', '', '', 'w', '', '', '', '', ''],
['', '', '', '', 'w', '', '', '', '', ''],
['', '', '', '', '', '', '', '', '', ''],
['', '', '', '', '', '', '', '', '', ''],
['', '', '', '', '', '', 'w', 'w', 'w', ''],
['', '', '', '', '', '', '', '', '', ''],
['', '', '', '', '', '', '', '', '', ''] ]
```

(a) (b)

Figure 4.2: The map is implemented as a list where every element is again a list of empty strings. Figure b) shows the data representation of the map in figure a). Walls are marked with a `w` in the implementation.

were taken of the object or area around, this is also stored here. It is this list that is sent to robots so that they can update their map.

We wanted to simulate the data from the positioning system, but this turned out to be a lot of work and having it should not prove or show any more than not having it. So when robots wants to know their position, the map service returns the same positions the robots think they have.

4.3 Visualization Module

The implementation of the visualization module does not correspond to the design. This is due to short of time. The current implementation is based on an earlier design where the visualization module was part of the infrastructure. Before we describe how the implementation is done, we will look at this older design version. We will see that even though the design is different, the implementation is not so different from what it would be with the newer design version.

4.3.1 Old design version

In the design version that the current implementation is based on, the map service is responsible for visualization. See figure 4.3. As the map server handles events from the controller and updates the map, it also updates the graphical output. The graphical output can be closed during run time, but it can not be started again.

The map server tells the visualization module what to display. The visualization module gets information about the map from the storing module. Mouse events form the user are registered in the visualization module and handled in the map server.

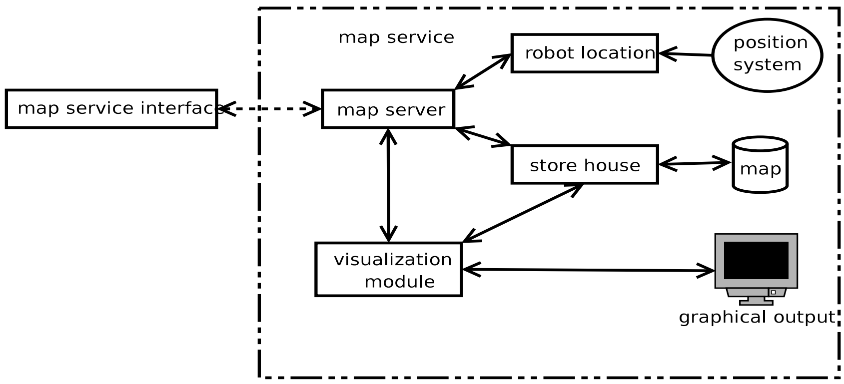


Figure 4.3: In the old design of the visualization module, the map service is responsible for visualization. The visualization module is here part of the map service.

The reason why we did not end up with this design is simplicity. The graphical output will always start, a user has to manually close it if he does not want it. Also it is not possible to run more than one visualization module. There may be occasions when we want to have a graphical output on more than one screen or display wall.

4.3.2 Current implementation

The visualization module run as a thread from the map service. The map server and the visualization module use queues to communicate. One queue is used for messages to the visualization module one for messages to the map server.

There are three types of messages going to the visualization module. These are: *newObj, move* and *lost*. The first is used to mark a new object in the map. The object can either be a robot or an object a robot has discovered. The *move* message is used when an object, or more precisely a robot, has a new position. The last message, *lost*, is used to mark that contact with a robot is lost. Included in these messages is an instance of a class containing all information needed to draw the object in question in the graphical output. The messages going to the map server is more simple. They consist of either a tag or a position. These indicates what was clicked by a user. We will describe them more in the next section.

We use Pygame for making the graphical output. Pygame is "a set of Python modules designed for writing games"[1]. Pygame is also excellent for other graphical representations. To display an object graphically, we need to have an image of it. Currently, we use three images, one for the robots when the infrastructure has contact with them, one for when the contact with a

[1] http://www.pygame.org/wiki/about

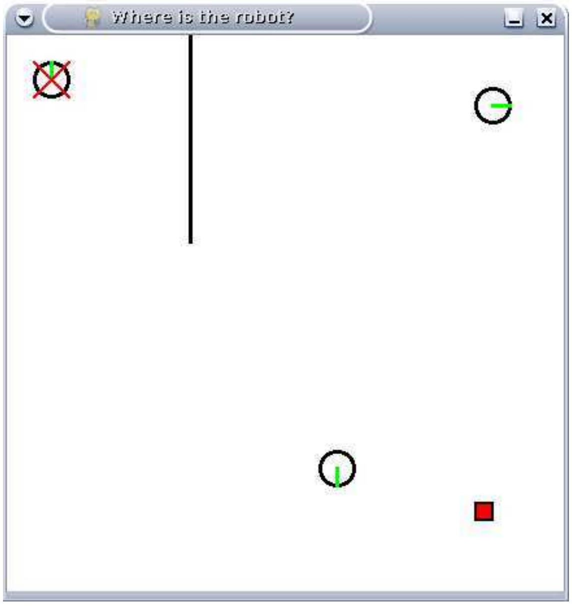


Figure 4.4: The graphical output. This situation shows on lost robot and two operational robots. The red square marks a object discovered by a robot.

robot is lost and one for discovered objects. These images can all be seen in figure 4.4.

Figure 4.4 is a picture of how the graphical output looks like. The work space is very simple with only a small wall to watch out for (the black line). There are marked three robots, but only two of them are operational. The robot with the red cross is marked as lost. Down in the right corner is an object discovered by a robot marked. For the two operational robots we can see their directions through the green line.

We will not describe the details of Pygame here. This can be found at the Pygame project's pages on the Internet ([11]). We will now describe how a user can interact with the graphical output.

4.3.3 User interaction

A user can do two things in the graphical output. He can get information about the objects and robots displayed, and he can give two types of tasks to a robot. Monitor data for a specific robot will pop up if a robot image is double-clicked. If an object image is clicked once, all known information of this object will appear. The user can initiate a goTo-task and a followPath-task. A goTo-task is initiated by clicking once on the robot to move and once at the point where the robot should go to. In a followPath-task, the user clicks the robot and then one click for each point to pass in the path.

So how do we distinct between the different mouse actions? Pygame registers many types of events. Among these is if a mouse button is pressed down. This is the event we use to register a user's mouse actions. Each mouse events is registered by the visualization module. The module checks if the the mouse's position is within any image of the objects displayed, i.e. if the user clicked any of the objects. If so, the tag of this object is sent to the map server. If not so, the mouse position is sent.

When receiving a mouse event message, the map server does the following:

1. If the message is a tag of any robot, it checks if it get an equal message within a second. If such a message is received, i.e. a new message with the same tag, it means that the user has double clicked the image. So the map server calls the *showMonitorData()* method at the controller. The controller will then display monitor data for the robot that was clicked. If the message does not contain the a tag, it contains a position. This means that the user wants to give a task. The position is registered. To figure out if the task is a goTo-task or a followPath-task , the map server checks if a new position is received within a second. If not, it is a goTo-task, and the *newRobotTask()* method at the controller is called with the goTo-task and the position as an argument. If a new position is received, it is a followPath-task. New positions are then registered as long as new positions are received within one and a half second after the previous. One and a half second may not be long enough in big graphical outputs. In the current implementation this waiting time is hard coded. For big graphical outputs, is should be configurable. The registered positions are saved in a list. When no new position is received, the whole path is sent to the *newRobotTask()* method at the controller.

2. If the message is a tag of a discovered object, all known data of this object is displayed to the user. If there is a video saved for this object, this is also shown. This data is, as mentioned earlier, saved at the map server and not at the controller.

3. If the message is a position, nothing happens.

The user will know from the updates in the graphical output when a task is executed.

Figure 4.5 shows monitor data for a robot that the user double clicked. We can see that this robot has the tag 101, its last known position and that it is connected. We can also see that it does not have a task at the moment, and that it has executed a goTo-task to its current position. At last there is a list of the predefined tasks that this robot has.

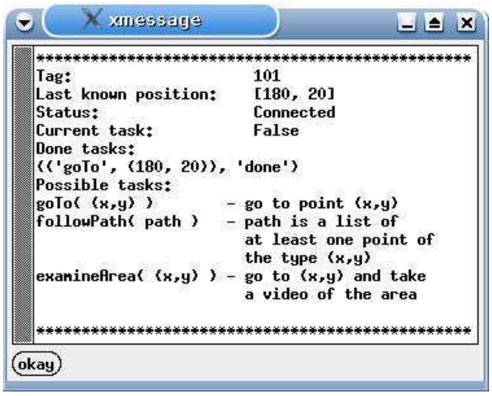


Figure 4.5: The window shows monitor data for a robot that the user double clicked.

4.3.4 Display wall

The display wall on the department of computer science at the university of Tromsø is $6x2.5$ meters and illuminated from behind by a tiled cluster of 28 projectors. The resolution is $7168x3072$ pixels and it is because of this very high resolution we want to use the display wall for visualization and not an ordinary screen. But because of portability is it possible to use an ordinary screen if no display wall is available. The data sent from the map service is independent of the resolution of the screen.

With low resolution on the display it is hard to see the details in the map, especially if there are many details in a small area. And with a small display, different robot's monitor data and the map must fight for the space to be visible.

4.4 Robots

In this section we will look at the implementation issues for the robots. First we will look at the start up phase. Then, how threads are used, and how the path planning is done. After that we will describe the implementation of the video making module and task execution. Last we describe how code execution is done.

Remember from the design chapter and figure 3.1 that when the robots communicate with the infrastructure, they talk to the controller. Throughout this section when we say "infrastructure", we mean "controller". We use "infrastructure" because it is more easy to to keep the different participants from each other then.

4.4.1 Start up phase

When started, the robot's control module invokes the RCC through the ER1 command line interface. Communication to this interface goes over a telnet connection. The implementation of the communication is done with Python's `telnetlib` module.

After connected to the RCC, the robot contacts the infrastructure in order to download the interface to use between them. The robot sets up a client to the XML-RPC server at the infrastructure. To do so, it needs the server's address and port number. These are hard coded in the robots. We wanted to make a start up phase where the robots entered some new area and sent out a "is anyone out there?"-message on a common known address. The controller would constantly listen to this address and respond to such messages with its address and port number. Then in the next step, the robot could set up a client with this information. We did however not manage to implement this.

When the connection is set, the robot calls the *initPhase()* method. This will return the interface to use in communication with the infrastructure. More precisely it returns a object of the `Binary` class, and a string. The `Binary` object contains the code for the interface class. The string is the name of this class. The code is written to a file and the class is imported with the `__import__` command.

The interface contains code for setting up a server that the infrastructure can use. The robot's control module must give the interface a reference to itself. This so that the interface can call the robots methods when the infrastructure calls them.

When the interface is imported and ready to use, the robot calls the infrastructure's *welcome()* function. *welcome()* takes as an argument the robot's address. The port number should be the same for all robots and are known to the infrastructure from its start. In return the robot gets information about its map, like the size and objects known to be in it. The robot also gets it position and a tag to identify it from the other robots. This tag should be used every time the robot make a request to the infrastructure.

4.4.2 Execution threads

When a robot is idle, it runs two concurrent threads; one server thread responsible for receiving events from the infrastructure, and one main thread. The main thread is in the current implementation not doing anything. It is in this thread we can put functionality that for instance stops the robot if the network connection is lost.

Tasks are assigned to the robot by the controller. The server thread is the one receiving the tasks. When is receives a task, it will start a new thread, called the task thread, that executes the task. This is because the task is received in a remote procedure call which must return so that the infrastructure can continue its execution. Execution of one task can take long time, and the infrastructure must be able to do other things in the meanwhile. When the task thread is started, the procedure call can return.

When the task thread is started it will start a fourth thread called the task-kill thread before the actual task execution is started. This thread is constantly checking if a global interrupt variable is set. When this variable is set, it means that the server thread has received a new task and that the robot must stop its current task execution.

When the task is executed, the global interrupt variable is deliberately set by the main thread in order to kill the task-kill thread. The two threads are joined and task execution finished.

So, there will at least be two running threads all the time. During task execution there will four threads running concurrently.

4.4.3 Task execution

We define a task as some assignment a robot is ordered to do. The initiator for every task is a user. The user decides what should be done by whom. The user can give a task either by selecting one of a robot's predefined tasks, or by writing some code to make up a new task and make the robot execute that. We will now describe how a task is given and executed.

As mentioned the robots can have predefined tasks. What predefined tasks that exist can vary from robot to robot. One robot may have many tasks and another none. The robots must therefore tell the infrastructure what predefined tasks they have.

When a robot has received its map and is ready to start execution, it sends a string describing its predefined tasks to the infrastructure. This string is then stored at the infrastructure together with all the other information of that robot. The predefined tasks our ER1 robots have are described in section 3.4.1. The string that these robots send to the infrastructure looks like this:

```
''goTo( (x,y) )         - go to point (x,y)
  followPath( path )    - path is a list of
                          at least one point of
                          the type (x,y)
  examineArea( (x,y) ) - go to (x,y) and take
                          a video of the area''
```

A task is given to a robot as a Python type `tuple` in the following form: (*task*, *arguments*) where the Python type of *task* is `string` and the Python type of *arguments* is `list`. When a user wants to get a task done, he first asks for what predefined task the robot he wants to use has. Then this string is displayed for him and he can choose if he wants to give some of these tasks or if he wants to do something else. Assume the user wants to give robot A a goTo-task. The user application asks for which robot the user wants to give a task to. Then the user is prompted to write the name of the task. In this case, the user writes *goTo*. Then the application asks for the arguments and the user writes the coordinates he want the robot to move to. The user application will then put the task and argument in the form just described, and send this together with the robot tag to the infrastructure.

The infrastructure will look up the robot tag in its list of known robots to find the right ip address. Then it sets up a connection with the robot and sends the task.

It is first when the robot receives the task that it is checked if the user has written the task right and if everything is all right. If not so, it will send a return message that describes the problem. This message is sent as a remote procedure call. The robot calls the same function as it does when a task is successfully performed. The procedure take as an argument a status message in addition to the robot's tag. This message is a string which tells how task execution went. This string is shown to the user, who will decide what to do.

The reason why the check is not done at the user application or the infrastructure is that neither of these know what tasks are supported by which robots. We have done it this way because of the competition aspect. In other settings, like a real search and rescue setting, it could be better if the infrastructure knew all the tasks. The whole system would thus be more autonomous. An other reason is that different tasks have different status messages to send in return. Take for example the examineArea-task. This task is dependent on having a second camera. If this is not mounted properly, a status message *no camera* is sent in return from the robot. This message will never be returned from a followPath-task because this task does not use the camera. In order for the infrastructure to be able to react upon an error message, it needs to know all types of messages that can occur. As new tasks are added to a robot, types of error messages will increase. If we want the infrastructure to handle error messages, we must add support for this when we add new tasks to a robot. In other words, we must add extensions two places instead of only one.

If a user wants to interrupt the robot in its current task execution to give it a new task, the global interrupt variable is set. This will make the task execution stop, but before that, the robot will finish its current movement. This is because of the dead reckoning navigation method. If we interrupt the robot's movement, we will not know how much of the movement was completed. The new position can be sent from the positioning system. But if the robot was turning, there is no way to tell what its direction might be. There is of course a stop method that stops the robot's movement immediately. This function is used for example if the robot is about to go down some stairs. But unless it is an emergency, we will avoid using it. Instead we use the interrupt variable to stop the task execution.

We will now describe the three predefined tasks in more detail.

The goTo-task

The goTo-task is a simple task moving the robot from its current position to a position given as an argument. The task execution goes as follows:

1. Use the A* algorithm to find a path to the given position.

2. Follow the path if one exists. Return an error message if no path is found.

3. Send a message to the infrastructure that the task is executed.

The followPath-task

The followPath-task differs from the goTo-task in that the argument is a list of points to visit. For every point in the given path the action performed is the same as for the goTo-task.

The examineArea-task

The examineArea-task is the most complex of the predefined tasks. It takes as an argument a point where the user wants the robot to go. At this point the robot shall take a video of the surroundings and save the movie on the place given as a second argument. The task execution goes as follows:

1. Find a path to the point.

2. Move to the last point on the path before the goal point.

3. Start a thread which moves the robot the last part of the path and turns the robot 360 degrees. Concurrently with this thread start taking pictures.

4. When movement is over, stop taking pictures and put them together in a movie. Save the movie at the given location.

5. Send a message to the infrastructure that the task is executed.

4.4.4 Navigation

Navigation consists of two parts; path planning and path following.

A robot's map is stored in the same way as at the map service. The A* search algorithm is implemented as described in appendix A. A straight forward implementation is not done because of performance. We will now describe an optimization we have done to get better performance.

The node with the lowest F cost in the OPEN list becomes the new current node. If we add new nodes to the end of the OPEN list without sorting them on the F cost, we have to search through the whole list every time we must find a new current node. Such search takes long time. So a good point in increasing performance is to sort the OPEN list on the F cost. We sort the list so that the node with the lowest F cost is the first element in the list. Then we only have to do a cheap pop operation to get the new current node. One problem however is to insert new nodes. This can take very long time if the F cost of the node to be added is high. We may have to search through big parts of the list before we find the place to insert the new element. We have chosen to sort the OPEN list as a binary heap. There are two reasons for this choice. First, heap sort is a very efficient sorting algorithm [4]. And second, Python has a library module called `heapq` which provides an implementation of heap sort.

Because of the ER1 robots lack of sensors we have decided that they only can turn in 90 degrees angels. This means that the robots can only drive in four directions; north, south, east and west. If the robots could turn in all angles, the possibility of a robot far off track would be higher than with this restriction. It is also easier to make the map implementation and path planning this way. It is by no means the best solution, but with the restrictions in lack of sensors and a not working position system, this solution is the easiest.

A* returns a safe path that the robot can follow to get to its goal. The path is a list of all the point the robot should drive trough. The path for getting from (1,1) to (5,1) will look like this, assuming no objects between the points: [(1,1), (2,1), (3,1), (4,1), (5,1)]. There are two ways of following this part. The first is to move from one step in the path to the next. The ER1's move command is on the form:

'move <distance> <units>'

Units can be *cm* or *foot*. To move backwards, a "-" must be added before the distance. (Turns are done with the same move command where the unit is *degree* and distance is the number of degrees to turn.)

If we follow the path above by moving from point to point, we would send four move commands to the robot, which again will drive the same amount of distance four times. This is clearly a unpractical way to do it. The second way to follow a path is to find the distance to move in one direction before the goal is reached or a turn is to be made. By doing it this way, the robot will move four units instead of one unit four times. This is more natural and faster.

To make sure the robot is not driving out off course, it periodically sends a position report to the infrastructure. This report is forwarded to the map service. The map service will check the reported position against the positioning system. The position the positioning system says the robot has is sent back to the robot. If this position differ from the one it reported, it has to correct its position and drive to the position where it is supposed to be. Because of the lack of a positioning system, we have not implemented it this way. The robots report their positions, but they will always get the same position in return. The robots work space is a stable laboratory environment. As long as we know the robot turns nearly or exactly 90 degrees and drives nearly or exactly the distance it is supposed to, it will not cause seriously trouble to assume it is always at the position it thinks it is.

4.4.5 Video making module

To access a camera in order to get the raw video stream from it, is not easily done on Windows. Luckily there exists a Python extension for Win32 that does this; the VideoCapture extension[2]. VideoCapture consist of two module levels. The low level native module (vidcap.pyd) uses DirectShow, which is included in DirectX 8.0 and higher. The RCC also uses DirectX, so there is no need to install this after installing the RCC. The high level module uses Python Image Library (PIL)[3] to produce images of the pixel data form the camera.

VideoCapture and PIL give us pictures in any file format we wish, but not the raw video stream. The best thing would be if we could get the raw stream and do analysis on that. The second best option is then to take lots of pictures for a period and make a video of them. For this purpose we use ffmpeg. ffmpeg is one of several components of the FFmpeg project[4] which is a complete solution to record, convert and stream audio and video. FFmpeg is developed under Linux, but it can be compiled under Windows. To do this, Msys and MinGW[5] are used. We use ffmpeg to merge a set of jpeg files to a mp4 file.

All the latest versions of this software is included on the CD. See appendix C for a short installation guide.

During implementation we came across a problem with the video making. As described under the thread execution section, all tasks, except for the ones the robot has asked for, run in a thread. VideoCapture is not able to access

[2]http://videocapture.sourceforge.net
[3]http://www.pythonware.com/products/pil/index.htm
[4]http://ffmpeg.sourceforge.net/index.php
[5]http://www.mingw.org

the camera properly unless in runs in the originally parent thread. This is, as discussed over, not possible. Our solution is to run the video making module in an own process. Variables such as duration of picture taking and the frame rate is set through a file. The thread writes the variables to a file which the video making module reads when it starts.

An other problem has also been that ffmpeg on Windows has been a bit unstable. Sometimes it works just well, and other times it crashes. We have not managed to find the reason for this, and because of time restrictions, we made a "work around". Instead of sending the movie to the infrastructure, we send all the taken pictures and then the infrastructure, which runs on Linux, uses ffmpeg to merge the pictures to a movie. This is not the best solution because the infrastructure should not do work that could be done at the robots, but it works.

4.4.6 Downloads and uploads

In order to upload the content of a file, we must wrap the content in an instance of the Binary wrapper class. Binary data is a type which can be marshaled through XML. It is also possible to send it as a string, as this also is a type that XML supports. But the string has to be free of characters that are not allowed in XML. These include $<$ and $>$ which are very common in code written in any language. This is why we use the Binary wrapper class.

Upload of a file to the robots or to the infrastructure is done through RPC with the file name and the Binary object as arguments. When received, the content of the Binary object, e.g. the data, is written to a file named as the file name-argument specifies.

4.4.7 Code execution

The code to be executed by the robots must be implemented as a Python module. As mentioned earlier, Python can import new modules during run time, and this is what we do with the code. All new modules, e.g. modules that are imported during run time, are stored in a dictionary where the key is the file name of the module. The file name is needed in the **`__import__`** command and is given as an argument to the function (see 4.4.6).

After the module is imported, a thread is started. This thread executes a function in the new module called *init()* and takes as arguments the robot object, so that the new module can access the variables and function in the robot object, and a flag. When the *init()* method has initiated and started whatever it is the code is suppose to do, it raises this flag. When the flag is raised, the RPC returns. Again, the code execution task is initiated by a RPC from the infrastructure. This procedure call must return so that the infrastructure can continue with its work. However, since we do not know what the code will do, we cannot just start it in a thread and then leave it to itself. For all we know the writer of the code does not want it to be started like this. So we give the control to the *init()* method and when this says it is okay to return, the RPC returns.

In other words, we set some requirements to the code to be executed. It must be a Python module that is possible to import during run time. Also it must contain an *init()* method which takes as arguments the robot object and the flag. The *init()* method is responsible for raising the flag when the RPC can return.

4.5 User application

The user application is implemented as a client that connects to the infrastructure's controller. More precisely, it communicates with the controller's user interface. It can not be started unless the controller is running. A simple RPC is done to test if the controller is running. The address and port number to the controller, is given as arguments during start up. This means that the user starting up the application needs to know these data before starting the application.

A thread is periodically polling the controller for tasks that is done, i.e. executed. When a task is done, the controller puts information about it and how the execution went, in a queue at the user interface. It is this queue that the user application checks. Elements taken from the queue is shown to the user.

There is currently no logging of a user's actions. Nor is it possible for more than one user to be connected at a time. Because of lack of time, we have not thought about how to prevent users of one robot team to interfere with other teams. Our focus has been making a prototype of the infrastructure and experiment with it to see if it can be used the way we want. This can be done with only one user.

Chapter 5

Testing

We have done three different tests. In this chapter we will describe these. First we have tested the robot's predefined tasks. Second we have tested code execution. And third the implemented A* searching algorithm is tested.

Tests results are presented and discussed in the next chapter (chapter 6 Results).

5.1 The predefined tasks

We have tested that the predefined tasks work the way they are described in 4.4.3. The goTo-task and the followPath-task are started from both the graphical output and the user application. The examineArea-task can only be started from the user application. We check that the task is transferred to the robot, that the robot executes the task the way it should, and that when the task is done, the robot sends a message to the infrastructure that the user gets.

We also tests how long time it takes for the robot to stop its movement when a stop command is sent from the user interface. This is the most critical command since it can be a question about loosing a robot if it drives down some stairs or something like that.

The last thing we test is if the robot acts the way we described in subsection 4.4.3 when we interrupt the robot with a new task. The right behavior is to stop task execution after the current robot movement is done.

5.2 Code execution

We have tested that files containing Python modules can be loaded from the user, through the infrastructure and to the robots. The Python modules are both code to be executed and new extensions to be stored and used later.

We have made two Python modules for this testing; `squareCode` and `executeCode`. The first module consists of a single function that makes a robot drive in a square. This module shall only be imported by the robot so that it can be used later. We want this function to be a task equal with the other predefined tasks. This new task is called the square-task. But the existing code does not support this new task. So if a user sends a square-task command to the robot where the `squareCode` module is imported, this robot will not recognize this task. The second module fixes this problem.

The `executeCode` module consists of code to be executed. During code execution, the robot's method *doTask()* is replaced with a new version. The *doTask()* method decides what task to be performed based on the command received (see subsection 4.4.3 for how a task command looks). The original *doTask()* method supports only the predefined tasks. The new version supports the previous imported square-task in addition to the predefined tasks. The new version of the *doTask()* method replaces the original version permanently as long as the robot code executes. When the robot is restarted, the new code must be added again.

5.3 A*

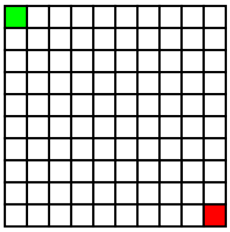
To test the implemented A* algorithm we search for a path from the top-left corner to the bottom-right corner of a map. This is, except for the other diagonal, the longest path in a map with no obstacles. We have tested A* in five different maps with different special cases. Figure 5.1 shows these five different maps. The first map consists of no obstacles. The second has a wall that parts the map in two pieces. There exists no path between the two point in this map. The third and forth maps have openings in one of the ends of the wall from map two. The reason for these two rather similar cases is that we want to see if the order in which we visit the neighbors of the current node, is relevant for the execution time (which neighbor is visited when, is described more in the results in chapter 6). The last map has an opening in the middle of the wall.

For each map we test with different map sizes. We take the time from when the search is started and till when it returns.

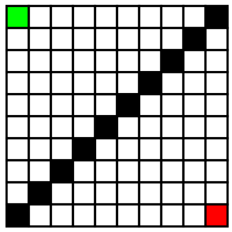
5.3.1 Different map levels

Some initial testing of the A* algorithm indicates that it is very slow in big graphs, i.e. when the map size is big. We also know from [3] that if there are no obstacles between the starting point and the goal, A* will be too slow. Our solution to this has been to make a "high-level-map" of the original map. This high-level-map has a higher scale than the original map, so that in stead of searching in a map of many coordinates, we search in a map with few coordinates.

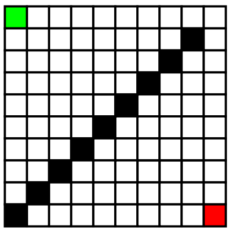
The size of the high-level-map is decided from the size of the original map and experiences made from results of the A* testing. This means that



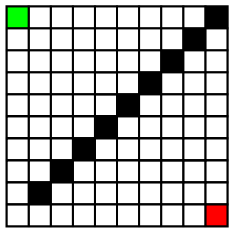
(a) With no obstacles.



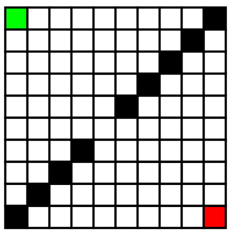
(b) With no path.



(c) Opening at the top-right corner.

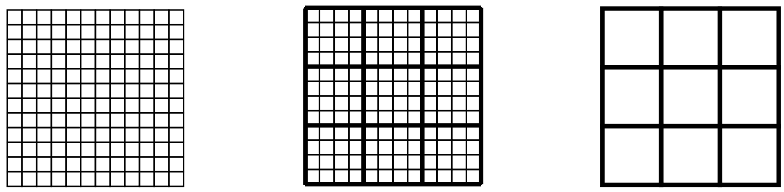


(d) Opening at the bottom-left corner.



(e) Opening in the middle of the wall.

Figure 5.1: The five different maps in which we tested the A* algorithm. We want to find a path between the green coordinate in the top left corner to the red coordinate at the bottom right corner.



(a) The original map. (b) Finding high-level-map. (c) The high-level-map.

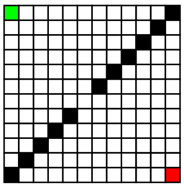
Figure 5.2: How to make a high-level-map from a map with no obstacles. Instead of searching through a map with the size $12x12$ we can do a faster search in a map with the size $3x3$.

we can not say at this point how big the high-level-map will be. Let us assume the tests show that execution is going too slow when the map size is bigger than $200x200$. This will then be the size of the high-level-map of all original maps with size bigger than $200x200$. How the high-level-map is made is easier to explain through the figure 5.2. In this example, the high-level-map size is $3x3$ as can be seen in figure 5.2(c). When the original map in figure 5.2(a) has the size $12x12$, each square, or coordinate, in the high-level-map will correspond to $4x4$ squares of the original map (see figure 5.2(b)). When there are no obstacles in any of the original map coordinates that corresponds to one coordinate in the high-level-map, it means that a robot safely can drive straight through this area.

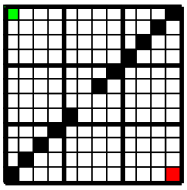
Figure 5.3(a) shows an original map situation equal to the one in figure 5.1(e). The difference is that this map's size is $12x12$ and the map's size in figure 5.1(e) is $10x10$. We want to find a path between the green upper-left corner to the red bottom-right corner.

When an original map coordinate is marked with an obstacle, the high-level-map coordinate covering this original map coordinate is also marked with an obstacle. As we can see in figure 5.3(b), the high-level-map coordinate (1,1), the one in the middle, must be marked with an obstacle. We can see in the original map in figure 5.3(a) that it is possible to find a path between the start and end points. We can not see this in the high-level-map in figure 5.3(c). This means that if a coordinate in the high-level-map is marked with and obstacle, we do not know if there exist a path between the two points.

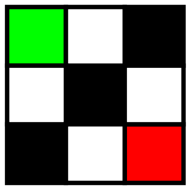
A path search with a high-level-map is performed as follows: First we will search for a path between the start and end points in the high-level-map. If we find a path here, the coordinates are "translated" to original map coordinates so the path can be followed. If we do not find a path, which



(a) The original map.



(b) Finding high-level-map.



(c) The high-level-map.

Figure 5.3: We can see in (a) that there exist a path between the green upper-left corner and the red bottom-right corner. This is not reflected in the high-level-map in (c). In order to be sure if a path exists between the two points, we have to do a search in the original map after a search in the high-level-map fails.

would be the case with the high-level-map in figure 5.3(c), we have to do a search in the original map before we can conclude if there exist a path or not.

We will test to see if the solution with a high-level-map will reduce the search time in the five cases earlier tested. We will do the same tests with the five different maps again, using a high-level-map when the map size grows big.

Chapter 6

Results

In this chapter we will discuss the results of the testings described in the previous chapter. First we will discuss the results of the predefined tasks tests, then the code execution tests, and last the results of the A* testing.

6.1 The predefined tasks

This section is divided in two. First we will describe the results of the goTo- and followPath-task testings, and then we will describe the results of the examineArea-task testing.

6.1.1 goTo- and followPath-tasks

Assignment of these tasks was correct done both through the graphical output and the user application. As long as the robot could finish its current task before staring a new, everything works fine. The task is correct sent to the robot, the robot performs its task, and sends a status message back to the infrastructure. This status message is forwarded to the user application. When the robot has moved to a new position, it reports this to the infrastructure which updates the map. The graphical output is also updated.

Small problems arise when we try to stop the robots movement and when we try interrupt its current task execution. We will look at the movement stopping first, and then the task interruption.

Stopping robots

When we here talk about to stop the robot, we mean to stop its movement. After a robot has been stopped, it should be able to receive and execute new tasks. A stop command can only be given through the user application. We took the time from when the stop command was sent via a RPC and till the RPC returned. In addition to this time comes the time it takes for the user to type the command of choice, and the robot tag to stop. But this time we did not measure because it can vary from user to user.

First we tried to stop the robot during a goTo-task. The average time it took to stop the robot was 0.55797 seconds or 557.97 milliseconds. This is a good reaction time. The only catch is that the user must be able to see if a robot should be stopped a few seconds before it goes wrong. This so he gets the time to type all that is needed to send the command.

Second we tried to stop the robot during a followPath-task. The command was sent correctly and the robot stopped as quick as with the first tests, but the RPC did not return. Hence, we did not get a time of how long it took to stop the robot, and the user application hanged, i.e. waited for the RPC to return and could not do anything else. This is a bad design of the user application. It should be possible to send a stop command and not wait for a RPC to return before the user can type new commands.

Problems occur at the robots also. After a goTo-task is stopped, the robot returns that it has stopped, but then it can not do anything else. A quick look in the code shows that this is due to a stopping variable not being set to the right value after a stop. A programming error, in other words. It is also the robot that is the reason for why the RPC from the user application does not return in the followPath-task stopping. It turns out that this is also due to an error in the code. In this case, we have forgotten to adapt some early written code to the latest version.

Interrupting robots

Interruption of a task is done by sending a new task to the robot. We can interrupt a robot both through the user application and the graphical output. The tests showed no difference in where from the new task was given. The results were the same.

The behavior we want when a robot is interrupted, is that the robot finishes its current movement, reports its position and then stops executing its current task and starts executing the new task. This works just the way we want when a goTo-task is interrupted. Interruption of a followPath-task has variable results. If the robot is in the middle of a movement forwards, the interruption goes well, meaning that the current task is stopped the way we want, and the new task is executed correctly. However, if the robot got the interruption signal in the middle of a turn, the robot stops its movement and hangs. This is the same reaction as when a stop signal is received in a followPath-task, and is due to the same program error.

6.1.2 examineArea

The examineArea-task is the task where we use the second camera to take pictures and make a video that is sent and stored at the infrastructure.

We have not been able to test this task when it comes to stopping and interruption. This is because of trouble with the video making.

As described in subsection 4.4.5 on page 46, the pictures are merged to a movie by using ffmpeg. We chose ffmpeg because it worked the way we wanted when we tested it. When we first made the examineArea-task, and did the initial testings, everything went the way we wanted. However when we did the last testings, ffmpeg stopped working. It will not merge the taken pictures into a movie. We have not been able to find out the reasons for this. And because of little time, this task is currently not working. The robot does everything but the merging.

6.2 Code execution

The code execution feature works the way we want. If the user, before he has given the two python modules described in the testing chapter (section 5.2 on page 49), tries to give a square-task to a robot, he gets a return message saying that this task is not supported. Then the user gives the `squareCode` module to the robot. The robot imports this module correctly. If the user now tries to give a square-task, he gets the same message as before.

Then the user gives the robot the `executeCode` module. After this module is imported, the robot can execute the square-task. It also executes the other predefined tasks the same way as before.

These results show that we are able to: 1) download code to the robots, 2) get the robots import the new modules, and 3) replace code at the robots.

6.3 A*

The results of the first test of the A* algorithm can be seen in table 6.1. In the far left column we can see the map sizes we tested with. When we from here on speak of map sizes of 10 and 100 and so forth, we mean map sizes $10x10$ and $100x100$. The rest of the columns in table 6.1 shows the search time in seconds for each of the five different maps in figure 5.1.

As we can see, for three of the maps we did not test with all the map sizes. We interrupted these tests because they took too long time.

The results are visualized in figure 6.1. We can see that, as map sizes grow, the algorithm performs more badly. Since a run time over 1 second is very bad, we made figure 6.2 to see the results better.

In general we can say that the performance is not very good.

When there are no obstacles in the map, i.e. map 1, the search time passes 1 second somewhere between map size 2000 and 3000. When there are no path between the two point, i.e. map 2, the search time passes 1 second somewhere between map size 100 and 200.

Table 6.1: Results first A* testing.

map size	map 1	map 2	map 3	map 4	map 5
10	0.0	0.0	0.0	0.01000	0.0
100	0.03000	0.68100	0.02000	0.52100	0.15000
200	0.06000	2.21300	0.06000	2.24300	0.63100
300	0.09000	5.64800	0.09000	2.65400	0.62100
400	0.12000	11.25600	0.1400	2.874000	0.69100
500	0.16100	19.83900	0.16000	2.67300	0.70100
600	0.19000	35.27100	0.19100	2.88400	0.79100
700	0.24100	62.04900	0.24000	2.85400	0.75100
800	0.28100		0.27000	2.88400	0.80100
900	0.32100		0.32100	2.94400	0.87200
1000	0.36100		0.36100	2.90400	0.93100
2000	0.81100		0.73100		93.5540
3000	1.46200		1.37200		
4000	1.66200		1.85300		
5000	2.50300		2.84500		

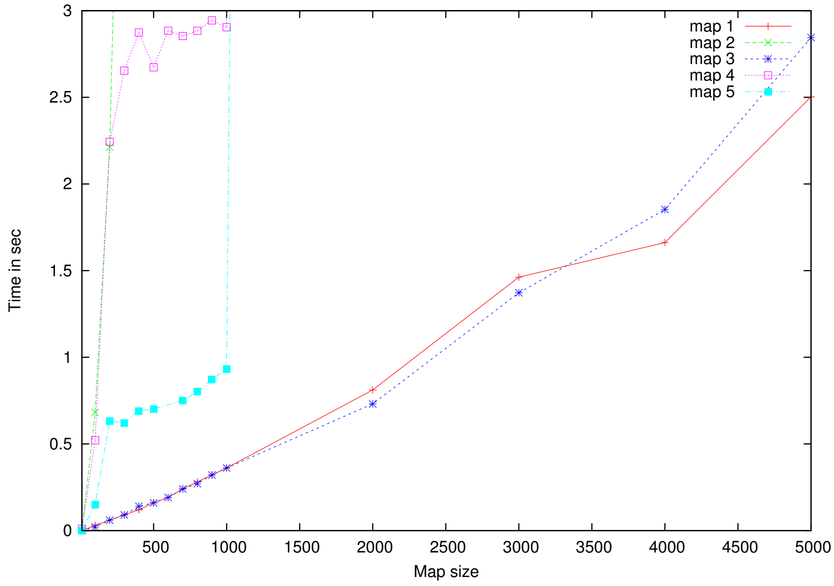


Figure 6.1: Graph showing the results of table 6.1

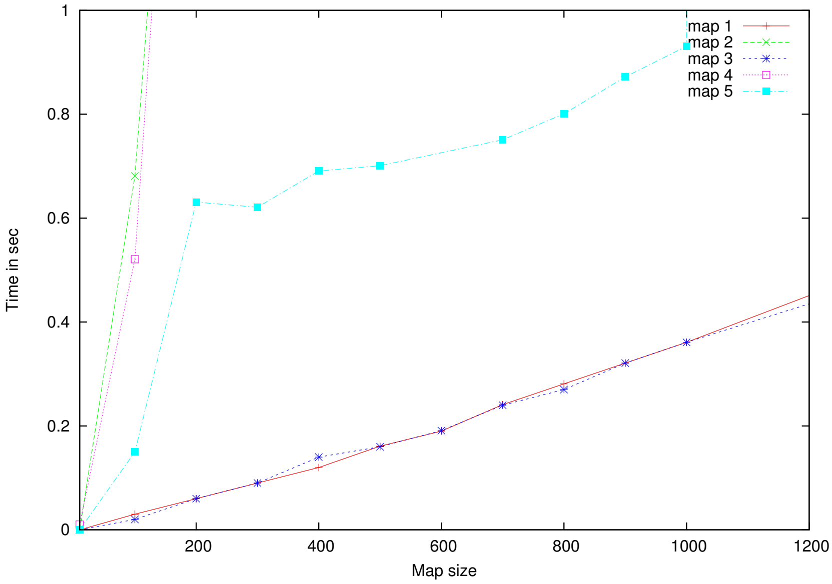


Figure 6.2: Higher resolution of graph in figure 6.1

Remember from figure 5.1 that map 3 and 4 are very similar. Map 3 has an opening at the top-right corner, map 4 in the bottom-left corner. We wanted to see if the order in which we visited the neighbors could have something to say for the searching time. We can conclude that it has. Each node has four neighbors to check. Four because the robot only drives in four directions. The order in which the neighbors are visited is: east, south, west, north. East corresponds to the neighbor on the right, south to the neighbor behind/under, and so fourth. Explained more graphically (C stands for the current node):

```
  4
3 C 1
  2
```

This means that the search works its way towards east first and then towards south, which is the reason why the search times in map 3 are better than the ones in map 4. In fact as we can see in figure 6.1, search time in map 4 starts out as bad as when no path exists (map 2). Then the time stabilizes between 2.5 sec and 3 sec until map size passes 1000. The search times for map 3 almost equals the ones for map 1. This is because they find the same path (top-left corner to top-right corner and then the goal bottom-right corner).

The search times for map 5 is better than for map 2 and 4, but it is not quite good. We have not managed to find out why the A* algorithm has so bad performance. If it is because of a bad implementation or if it is because the A* algorithm is a bad choice for big maps. Because of its popularity amongst AI game developers, it most likely is our implementation of it that is not optimized the way it can be. We have chosen not investigate this more or to test with other algorithms because it is not an essential part of this thesis to have an totally optimized path searching algorithm. Also, the maps we have used during testing has not been bigger than 300. The algorithm works good enough for our testing purposes.

6.3.1 With a high level map

Our expectations for the tests with high-level-maps are that the search time in original map sizes bigger than high-level-map sizes always will be the same as the search time for an original map with high-level-map size when the "obstacle situation" is the same for both maps. In other words: assume a high-level-map size of 10 and an original map size of 100. We expect the search time for this map to be the same as if the original map size was 10. This depends on that the "obstacle situation", i.e. where the obstacles are, is the same for the high-level-map as in an original map with the same size. When in a high-level-map there exist no path between the points, we expect the total search time to be a bit higher when using a high-level-map. This is because the search time in the high-level-map corresponds to the search time of an original map on the same size with no path. The total search time we expect to be the search time in the high-level-map plus the search time in the original map.

Table 6.2 shows the results of the testing of A* with a high-level map when the high-level map size is $100x100$. This choice of size is based on the results in table 6.1. The search time for a map with no obstacles (map 1) in table 6.1 is 30 milliseconds. This is an OK search time. As we can see in table 6.2, the search time for map 1, when original map size is 5000, is the same as when original map size is 200 in table 6.2. Figure 6.3 shows the results in table 6.2 graphically.

If we compare the two tables and figure 6.3 and figure 6.1, we can see that when it comes to map 1 the results agree with our expectations. With a high-level-map size of 100, the search times are the same for all map sizes up to 1000. After that, the search time increases a bit, but not so much as without a high-level-map. When it comes to the other maps, which all have a high-level-map with no existing path, our expectation agree less. The search time with a high-level-map is not higher than without. There are so small differences that we can not say that one is better than the other.

We also did tests with high-level-map sizes 200 and 300, but these tests failed. The reason for this is that our code can not handle the cases where the original map size divided by the high-level-map size does not yield accurate results. This division is important because it gives how many coordinates of the original map will correspond to on high-level-map coordinate.

Table 6.2: Results of A* testing with high-level map. High-level map size is 100x100

map size	map 1	map 2	map 3	map 4	map 5
10	0.01000	0.0	0.0	0.0	0.01000
100	0.03000	0.51100	0.02000	0.52100	0.14000
200	0.03000	2.70400	0.06000	2.24300	0.68100
300	0.03000	6.30900	0.11000	2.62400	0.63100
400	0.03000	12.3780	0.12100	2.86400	0.69100
500	0.03000	21.3810	0.16000	2.73400	0.70100
600	0.03000	34.6500	0.19000	2.89500	0.78100
700	0.02000	61.9890	0.24000	2.83400	0.76100
800	0.02000	82.9490	0.27100	2.89500	0.80100
900	0.03000		0.31100	2.94400	0.86200
1000	0.03100		0.37100	2.95400	0.93100
2000	0.04000		0.77100		92.1319
3000	0.04000		1.38200		
4000	0.07000		1.86300		
5000	0.06100		2.44400		

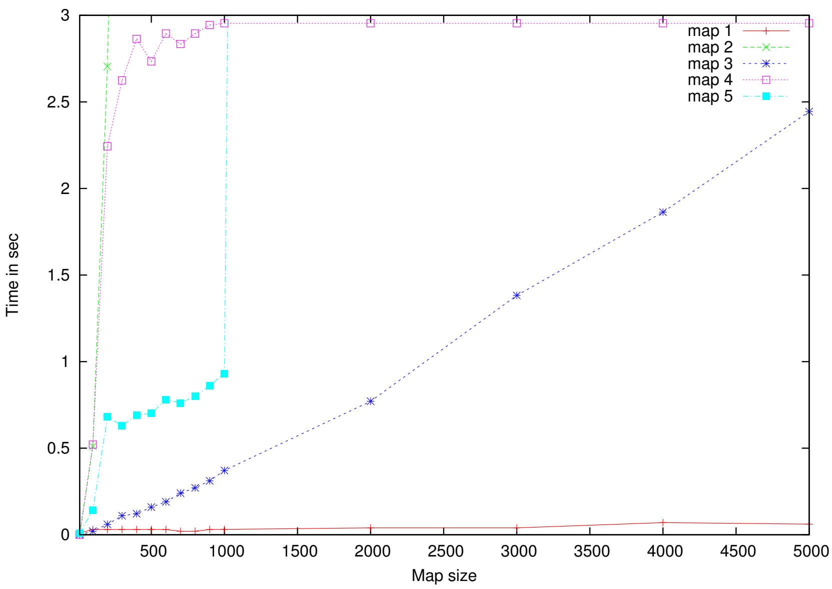


Figure 6.3: Graph showing the results of table 6.2

We do not get the best impression of how the search time is when a high-level-map is used. It is only in map 1 a high-level-map will make a difference because in all the other maps, a path will not be found in a high-level-map. But we can conclude that using a high-level-map will reduce the search time when a path exists in the high-level-map.

Chapter 7

Evaluation

7.1 Overview

In this chapter we will evaluate the developed environment. We will also discuss some of the points where we met problems or why we chose the solution we ended up with.

First we will evaluate how we have met the requirements we stated in section 1.3. Then we will discuss fault tolerance, why we ended up with two cameras on the robots and an obstacle avoidance method we tried. We will also discuss some navigation issues and our experiences with the ER1 robots.

7.2 System requirements

In the problem definition (section 1.2, page 2) we said that: "The main purpose of this thesis is to developed a distributed and parallel environment that supports a dynamic number of robots with functionality. This functionality will aid the robots in their tasks." And then we stated some requirements (section 1.3, page 3). We will now evaluate how the problem definition and the requirements have been met.

Task assignment and execution

Because we do not know all kinds of tasks the robots will be performing in the future, the environment must be able to handle all kinds of tasks. This point has been partly met.

As it is now, all work with task performing is done at the robots. The infrastructure is only forwarding task assignments to them, and reporting status messages back when a task is done. During task execution, the infrastructure offers some functionality to the robots, but it can not help them if they meet problems with a specific task. The infrastructure, as it is now, is not intelligent. The robots, on the other hand, are able to handle new tasks as long as they get to know how to execute them (i.e. gets the code that says how to do it).

It is possible to assign tasks to robots through the user application and through the graphical output. The tasks assigned through the graphical output are only simple move tasks, but through the user application, all task-types can be assigned.

Support for a dynamic number of robots

The infrastructure will not be able to recognize teams of robots. From the infrastructure's point of view, all robots operate on its own. Except from this small drawback, the requirement is fully met. Robots can enter the work space at any time, get the required downloads, and leave the work space again as they wish. The only thing is that they must know the address and port number to connect to in order to communicate with the infrastructure.

Offered functionality

Functionality and services offered by the environment should include: context awareness, location, mapping, naming, and structured interfaces for interaction between the different components. This requirement is almost fully met.

The robots get all known information of the work space they are operating in. They also get their positions upon request. But we do not have a way to keep track of the robots' locations because of the missing positioning system. A map covering the work space exists and it contains all known information. This includes robot positions, obstacles and other objects. Each robot gets its own tag which uniquely identifies it. The tag is used in all communication between a robot and the infrastructure.

The interfaces provide layers between the different parts that communicates over the network. Both sides of an interfaces can be changed without the other side knowing it because they always use the interface.

Downloads and uploads

The requirements state that it should be possible for the infrastructure and a user to download data and code to the robots and the robots should be able to upload data to the infrastructure. This requirement is fully met. We have proved through the tests that data can be successfully loaded both ways.

Visualization

The visualization module makes a graphical output of the map and all the information it stores. This graphical output is shown on a display wall. We

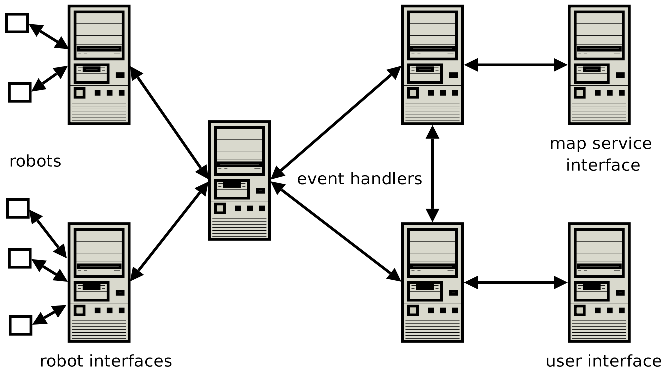


Figure 7.1: An example of how the event handler and robot interface of the infrastructure's controller can be distributed.

have also tested that it can be shown on a ordinary computer screen as long as the scaling is correct.

A user can interact with the graphical output by using a mouse. He can get information about all the robots and objects discovered by robots through it (the graphical output). Two simple tasks can also be assigned through it, namely a goTo- and a followPath-task.

7.3 Fault tolerance

We have not implemented a fault tolerant system. This has to do with our goal with this thesis. We wanted to make a prototype that we could test and investigate to see if this is the right way to go. The infrastructure is however designed so that it can be distributed and then be more able to handle faults. In this section we will describe how this distribution is thought done.

As we can see from figure 3.1 on page 14, the controller' s event handler is a potential bottleneck. All traffic is directed through it and when heavy loaded, it can slow down the performance. We can also see from the figure that with many robots, the controller's robot interface can become a bottle neck as well because of all the traffic it must handle. These are the two most obvious parts that can be distributed.

The event handler can be distributed so that there are one for each interface. This is illustrated in figure 7.1. The different event handlers know about each other and communicate so that each one can reach all the others. In this way, when, for example, a task is forwarded from the user interface to a robot, the task is going directly between the two event handlers involved, not the other. From the interfaces point of view it looks as if there is only one event handler.

Figure 7.1 also illustrates a way to distribute the robot interface. Some robots are communicating with one interface, and others with an other interface. If there are very many robots and robot interfaces, we can have more than one event handler handling the interfaces.

The event handler logs all events. The log is used in recovery from a execution stop and for debugging. It is stored as a file that can be read during or after execution. The log is cleared after the controller is started, but before new events are received. In this way, no data is lost during recovery, and the order of events can be read after execution for debugging.

7.4 The two cameras problem

As mentioned in the description of the robot design in section 3.4, page 17, we ended up with two cameras on the robots in order to use both the RCC's object recognition and take pictures of the surroundings. We will here explain why we ended up with two cameras.

The best way to solve the problem would be to let our robot program control the camera and develop our own object recognition feature. Making a whole new object recognition module is out of scope for this thesis, so we have tried to use a module developed before.

In the old robot system in the department, obstacle avoidance and object recognition were given through the attached video camera. Remember that a camera was mounted above the arena (see figure 1.1). The camera could recognize some graphical tags that the robots had on top of themselves. Through these tags the camera could tell the robots where they were and where the other robots were in order to avoid collisions. Also an algorithm was developed that recognized tennis balls that were spread around the arena. This algorithm used the raw video stream from the camera and analyzed this to find forms that could be tennis balls.

We tried to use this algorithm as our object recognition module. If we could get it work, the robots could recognize tennis balls, and no other objects. But even this would be better than the RCC's object recognition feature. Adapting the algorithm for our robots, turned out difficult. One of the main problems was that the algorithm was developed under Linux and used libraries developed for Linux. We did not find good enough replacements to use the algorithm under Windows without changing most of the code. Considering the time limits we decided not to follow this through.

The next thing we tried was to share the camera between the RCC and our robot program so it could be used for both object recognition and picture taking. This solution did not work either. This was because the RCC will not let any other process access the camera after it, the RCC, has taken control of it. Also, if another process has taken the camera first, the RCC is not able to use it.

In order to get robots that supported both features, and not use enormous amount of time developing object recognition code, the solution became to

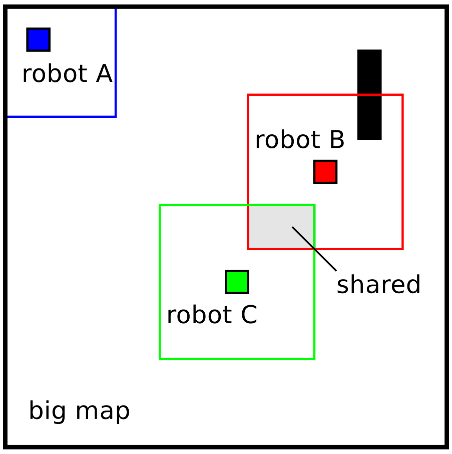


Figure 7.2: Each robot get their own area where they can operate freely without risking a crash with another robot. The figure shows an example situation with three robots. Robots B and C shares an area. Robot A has a smaller area than the other two because its position is close to the work space's borders.

use two cameras. One that the RCC could use for object recognition and one that our robot program could use to take pictures with.

7.5 Obstacle avoidance

In subsection 3.6.4, page 24 we mentioned that we have tried to give each robot their own piece of the map. Within the area this map piece covers they could operate freely without risking a crash with other robots. This would solve the problem of robots driving into each other. We will now describe this solution and the reason for why we decided not to use it. See also figure 7.2.

The size of a robot's area is decided by the robot's position and a predefined maximum size. The robot's position becomes the middle of the area. Based on the predefined maximum size, the borders of the area are drawn around the robot. Robots close to an edge of the work space will thus get a smaller area than robots closer to the middle. See robot A and B in figure 7.2. Robot B has an area with the maximum size. Robot A has a smaller area because its position is close to the border of the work space.

When a robot gets the map of it's area, it also gets all the known information about that area. As we can see in figure 7.2, robot B has a part of a wall inside it's area. If it didn't knew about this, it could drive right into it.

Areas are distributed as new robots arrive. When robot C entered the scene in figure 7.2, it's position was so close to robot B that the areas over-

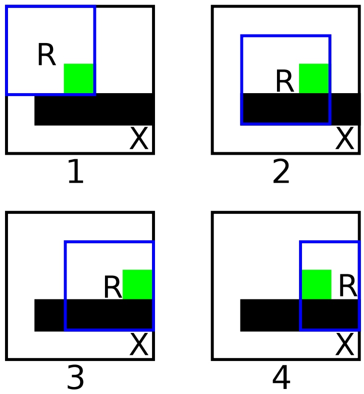


Figure 7.3: The figures show a situation where a robot is not able to find a way to the goal point X. The reason is that it does not know about the situation outside its own area, except for its global position.

lapped. The overlapped area is marked as “shared” in robot C’s map. Robot B’s map is left unchanged. When a robot has some “shared” areas in it’s map it means that it cannot go inside this area because the robot that “owns” the area can be there. This way we avoid robot collisions.

Depending on the settings, a robot can know that it is operating in a bigger area and where it is placed within it, or it can be unaware of this and only know about it’s own little world. Both approaches will cause some problems when a robot is ordered to move to a position outside its own map. When a robot does not know that there exist a bigger world outside it’s map, it can hardly calculate a path to a point far out in the big map.

Assume the first approach is used, and the robot knows where it is in the big world. To reach a position outside its own map, it can first find a path to the position inside its area which is closest to the goal. Then the robot can ask for a new map and do the same over again until the goal is within its area and can be reached. Figure 7.3 shows an example of how it can go with this method.

Robot R is going to position X. Between them is a wall, but there is room to get around it. R’s area is marked with blue. The closest point to X inside the area is marked with green. The robot goes to this position and asks for a new map (picture 1). The closest point to X within the area is calculated from X’s position. The robot does not know where walls and objects are outside it’s area. This is the reason for why the robot in picture 2 and 3 goes to the right and not towards the opening in the wall. Finally in picture 4 the robot reaches the edge of the work space. When it now goes to the closest point within its area and asks for a new map, we will end up in the same situation as picture 3. As we can see, with this method, the robot will not reach the goal point.

There are ways around this problem. One way is to let the map service do the path planning and the send whole path to the robot. It is also possible to make smart algorithms that remember where the robot has been and try other ways when situations like the one described occurs. As we explained in subsection 3.6.4, we have not solved this problems. Both because path planning is not a part of this thesis, but also because this is a good topic for future student assignments and competitions.

7.6 Localization

The lack of a working and reliable localization method has been a problem for several projects using the ER1 robots. It is hard working with the robots when they can not navigate properly. There are two problems with not having localization.

First, the robots have no guarantee that they can move safely around because they can not be sure that they have the position told by the location system. Second, the robots themselves do not have sensors that can be used in obstacle avoidance. If they had this, and/or other sensors that could say something about the surroundings, they could be able to navigate and correct their positions by themselves.

It is the combination of no usable sensors and no localization that gives the problems. If one of these is eliminated, further work with the ER1s will be much easier and more fun.

7.7 ER1 experiences

What we experience as the biggest problems in the work with the ER1s, is the lack of sensors. However, work on this problem has been started during the end of the work on this thesis.

In appendix B we describe a side track of this thesis which investigated the use of ultrasonic sensors that we mounted on one of the robots. With these sensors the robot became much more safe in the sense that it did not run into walls or objects unless they were not discovered by the sensors. There have also been made a ultrasonic sonar which we did not have time to test on the robots. Our expectations to these sensors are that the robots will be more navigable and as a result, we can do more interesting investigations with them.

One other big problem with the ER1s is that we can not control the rotation sensors, and thus control the robots, through the command line interface of the RCC. The command line interface sets many restrictions of what we can do with the robots. What would be the best is to have direct access to the rotation sensors. To do this, we must get the stream coming from the USB cable connected to the Robot Control Module (RCM), that is connected to the rotation sensors. See figure 2.5 on page 11 for a reminder of the ER1 architecture.

Work on this has also been started. Already there has been made a small program running on Linux that accesses the RCM and control the wheels. Data coming from the RCM has also been caught. Unfortunately, also this work started too late to be used in the work with this thesis.

Chapter 8

Related work

8.1 Overview

What we have been working with in this thesis is in literature called a multiple mobile robot system. A multiple mobile robot system is a system of a team or teams of mobile robots that cooperate to reach a goal. Robot teams can consist of homogeneous or heterogeneous robots, or a mixture. The thought is that a team of robots can perform bigger and more advanced operations than one single robot. Also economy can be a factor. A big advanced multifunctional robot may be more expensive than many smaller and less advanced. A third factor is fault tolerance. A team of robots is more likely to be able to adapt failures and unexpected changes.

We can easily see that many of the challenges in a multiple mobile robot system are the same as for distributed and parallel systems [14]. In addition to these challenges comes the aspect that the robots should be autonomous.

In this chapter we will first give an introduction to multiple mobile robot systems. Then we will give a short description of three different, and a bit elder, multiple mobile robot systems that to some extent are similar to our system. We will look at similarities and differences and why we made our choices. Then we will look at the state of the art.

8.2 Multiple mobile robot systems

Typical applications for multiple mobile robot systems are search and rescue situations, and war and terrorist situations. Common for these application areas are that the robots must be able to cooperate in exploration of unknown areas and environments. One important force that drives the research on multiple mobile robot systems is the prospects of reducing the need of humans in dangerous situations [10]. By sending in mobile robots more lives can be saved both victims and rescue forces. Victims can be found faster and analysis on how to get them safe out can be done quicker. Rescue forces will not have to take too big risks in saving others.

Multiple mobile robot systems can be placed on an line from deliberative to reactive systems [5]. Fully deliberative systems have centralized control and a detailed world model. Fully reactive systems expect that the system will emerge through the different robots' behaviors. Most multiple mobile robot systems, including ours, are hybrids of these models. Fully deliberative systems suffer from not having the ability to smoothly adapt to changes in the environment. Fully reactive models are more dynamic when it comes to adapt to changes, but they can be hard to make from "off-the-shelf"-robots like ours.

Most multiple mobile robot systems are focusing on architectures that get heterogeneous robots to cooperate on motions and mission performing. We have been focusing on developing a general framework that offers basic functionality. The framework can be used by a variety of multiple mobile robot system architectures.

8.3 RAVE

RAVE is "a software framework that provides a Real And Virtual Environment for running and managing multiple heterogeneous mobile-robot systems" [5]. It is developed at the institute for complex engineered systems at Carnegie Mellon University. They recognize that "developing multiple mobile robot systems requires many supporting capabilities such as communications, user interfaces and support for simulation." To keep their focus on algorithms and architecture for collaborative behavior, they have developed RAVE that provides these capabilities.

RAVE allows multi robot systems to be developed and tested in simulation. When a system is ready to be deployed, it can seamlessly be transferred to real robots. Any robot program can be run on either a simulated or real robot. This simulation capability is extensively used for testing sensors, both real and virtual. Virtual sensors can be used on real robots, and investigating sensor configuration can be done in software rather than hardware. The advantages of this are that they can determine if sensors types are useful and where the best placement on a robot is. This feature is helpful because it saves a lot of time. Also RAVE allows to place virtual obstacles in the world model so that real robots operating in real world avoids them. Real robots and virtual robots can operate simultaneously in the world model and interact with each other.

The RAVE framework consists of robot libraries, information servers and user interfaces. The robot library's provides a standard interface to real and virtual robots. The interfaces form a layer between the high level robot programs and the low level robot interaction. The information servers consists of an environment manager and a graphical user interface (GUI) server. These controls the system's state. The system state is defined as the robots' positions and virtual obstacles. The robots report periodically their state to the environment manager. The environment manager distributes information to the real and virtual robots. The GUI server sends updates to the different user interfaces. The user interfaces show the system's current state. There are three different types of users; observers, commanders and super-users. The different types have different limitations to what they can do in their GUI.

The ideas of RAVE are the same as ours; development of a framework that offers basic functionality to different robot architectures. The main difference is the simulation. We do not yet have the need for this type of simulation. RAVE was constructed for investigating collaborative behavior in multiple mobile robot systems. We want a frame work for supporting multiple mobile robot systems in their missions. Then later we can expand the frame work to what the needs might be, based on experiences. Development towards a system like RAVE can be the natural way to go if we want to do research on collaborative behavior in multiple mobile robot systems.

8.4 Dynamite

Dynamite is a multiple mobile robots system made in 1993, and is interesting because of its age. It is a testbed for multiple mobile robots, based on off-board vision and off-board computation [2]. Although their focus has been on robot soccer games, Dynamite can be used by other robot applications as well.

Dynamite is a part of a project called Dynamo (Dynamics and Mobile Robots) at the laboratory for computational intelligence, department of computer science, university of British Columbia. Dynamo is an on going project still, that "makes use of multiple radio-controlled vehicles to investigate problems in multi-agent robotics" [1]. Their goal is to "generate the appropriate cooperative and competitive behaviors for complex tasks such as playing soccer."

Dynamite has a very similar architecture as the old robot system in our department (see chapter 1). Their soccer field has a wall around it which prevent the ball and players to drive off the field. Then there is a single color camera mounted above the soccer field. The video output is transmitted to a vision engine which produces an absolute position of all objects in the soccer field. Movement of all vehicles is controlled through a radio link. The robots used were so small that the computation had to be done off broad.

Dynamite was a successful testbed for experiments with multiple radio controlled robots. They showed that off-broad vision and computation could be used for real time control of mobile robots. But this was 13 years ago

[1] http://www.cs.ubc.ca/nest/lci/about.html

and the hardware available today together with the enormous development in technology has made this approach unsuitable for our demands. We want the robots operating in our system to be as autonomously as possible. Hence we want the computation to be done at the robots and not off board. Off board computation demands more infrastructure just like the old robot system in our department, which is one of the things we wanted to avoid.

8.5 ALLIANCE

ALLIANCE is a fully distributed behavior-based architecture that offers fault-tolerant control of heterogeneous multi robot teams [10]. The developers want to solve the problem of "multi robot cooperation for small- to medium-sized teams of heterogeneous robots performing missions composed of independent sub-tasks that may have ordering dependencies."

The developers of ALLIANCE recognizes that robots will fail and that unexpected events will occur. Robot teams in ALLIANCE will be able to perform their mission even with failing robots and unexpected changes.

There exists no centralized unit to do task allocation. The robots are able to determine their own actions themselves. A robot's decision on what action to perform is based on its current situation. What actions a robot can perform is dependent on its behavior sets. A behavior set is the actions that the robot must perform in order to get a task done. The robots have several behavior sets, but only one can be active at a time. I.e. the robots can only perform one task at a time.

Which behavior set to activate is decided through use of motivational behaviors. There exist two motivational behaviors - robot impatience and robot acquiescence. The impatience behavior helps a robot to handle situations where other robots than itself fail to perform a task. The acquiescence behavior helps a robot to handle situations where itself, is failing to perform its task.

How are these motivational behaviors used? A robot team has a mission to perform. The mission consists of several sub-tasks. Each robot has an increasing impatience to get the different sub-tasks done. If a robot is currently working on a sub-task, the other robots' impatience for this sub-task will increase at a slower speed than for sub-tasks that no robot is working on. If robot A notes that robot B is no longer working on the sub-task it says it works on, and A's patience is gone, robot A will take over B's sub-task. The reason why B is no longer working on what it says can be malfunctioning sensors.

Robots fail. Hence the acquiescence motivation. Each robot has some degree of acquiescence to get the sub-task it is working on done. If its sensors are telling it that it fails in its sub-task performing, it will stop the performance by itself and find some other sub-task to do.

ALLIANCE assumes that the robots with help from their sensors will discover if no robot is working on a sub-task or if a robot is failing in the sub-task it is working on.

ALLIANCE is a fully reactive system. The robots work autonomously and can adapt to changes in the environment by them selves. This approach is very good when no sorts of network or other infrastructure are available. This is the reality of most environments where we want multiple mobile robot systems to be used. The main drawback for fully deliberative systems and hybrids is that they depend on some infrastructure in order to work. For a real world application, ALLIANCE is much better suited than our system, because of the few infrastructure demands.

ALLIANCE has a totally different approach than us. It is more a multiple mobile robot system *architecture* than an *infrastructure* supporting different multiple mobile robot systems. Also, robots using ALLIANCE must have a lot of computational power and advanced sensors. An important requirement for us is to see what we can do with "of-the-shelf" robots. These are not so advanced yet. But with an infrastructure like ours, we are able to do research on what teams of such robots can do.

8.6 State of the art

There are few systems like ours out there, that offer an infrastructure with services to single robots or robot teams. Most multiple mobile robot systems today are reactive models with no centralized unit controlling the robots.

[9] realizes that even though robots have become quite advanced and can do advanced missions, humans still do better evaluation of a situation and take better decisions. They have developed a multiple mobile robot system that is supervised by a human operator. Here the robots are controlled by the operator in stead of operating autonomously. This is almost the same approach as we have taken. We say that the robots will success in their missions with feedback from users. Not controlled bu users.

[6] gives an introduction to research on small robots, each of them specialized to sense or do one thing. When these robots are operating in teams, they can get a good impression of the conditions in an environment. Usage for such teams is for example, in a building taken by terrorists. A team of small robots can be sent into the building and report back useful information. This information can then be used to plan an attack. These robots demonstrates what can be done when appropriate and different sensors are available.

Much ongoing research in multiple mobile robot systems today investigate swarms of robots. The idea behind a swarm of robots is taken from animal swarms. A good example is ants. A swarm of ants can in cooperation build amazing things. One single ant is not good for much (unless it is the queen of course). In robotics this is called swarm intelligence. A robot swarm consists of an army of small, mostly homogeneous robots. iRobot[2] works with the world's largest swarm, consisting of over one hundred robots[3]. Their distributed algorithms function with groups of 10 or 10,000 robots. An closed European project that worked with swarms was SWARM-BOTS [7]. They studied self-organizing and self-assembling robots. Amongst their achievements they managed to get a swarm of 20 robots to self-assemble into four smaller swarms and pull a child on the floor.

[2] http://www.irobot.com
[3] http://www.irobot.com/sp.cfm?pageid=149

Chapter 9

Concluding remarks

This chapter concludes this thesis by describing our achievements and outlining directions for future work.

9.1 Achievements

In this thesis we have developed a distributed and parallel environment which offers functionality to robots to support them in their task performance. All types of robots are supported, as long as they can use an interface provided by the environment. All communication between robots and the environment goes through this interface. The environment has control over a certain amount of work space where the robots can operate. Within this work space, the environment offers functionality that includes:

- Information about the work space. This information can be obstacles to avoid and other robots.
- Location of each robot. This is not working properly because we do not have a positioning system that can give location data. But the environment can give the last reported position from each robot known to be in the work space.
- A map over the work space which contains all information the environment knows about it.
- Naming of each robot. Each robot gets an unique name used in all communication with the environment.
- Structured interfaces for interaction with robots and users.

Users can control the robots through the environment. The users provide tasks to the robots. What tasks a robot can perform vary from robot to robot. Each robot is equipped with some predefined tasks they can perform upon orders from a user. It is possible to provide the robots with new functionality during run time. Users can download extensions to robots through the environment, and replace parts or the whole code running on the robots during run time.

Robots can upload data to the environment. Data that is currently uploaded is pictures taken with the robots' camera.

All known information about the work space, including robots, is shown graphically on a display wall. It is possible to interact with this graphical output by using a mouse. A user can assign simple moving tasks to robots through the graphical output. The graphical output can also show monitor data of the robots and objects discovered by robots.

Our intention with this thesis was to make an environment in which students could experiment with robots on their own, and in which it could be held competitions. This we have almost achieved. The biggest problem is that we do not have a proper way to do localization. This brings us over to future work.

9.2 Future work

We will now outline some of the areas that can be elaborated in future work. These include:

- Improvement of the environment. The environment developed is only a first prototype. There are many areas to improve, such as a possibility to extend the infrastructure during run time just the way we do it with the robots. Other improvements are to prevent users of one competing team from access another team's robots, and to make the environment support more than one concurrent user.

- Fault tolerance. If the environment shall be used for student experiments and competitions, it must be able to handle failures. Logging of all actions for recovery and debugging purposes is one way to do this. The user application can log the user actions. The infrastructure can log all incoming and outgoing events. One possible solution in making the environment more fault tolerant is outlined in section 7.3 , page 65.

- Improvement of user application. The user application should be presented with a graphical user interface (GUI) in stead of a text based user interface like it is now. In a GUI, commands can be given faster because the mouse can be used. To type commands in more than one step is not very user friendly, it takes more time than using a mouse, and misspellings that leads to wrong commands happens often.

- Improvements of the ER1s. If more big projects shall include the ER1s, we recommend that more work on the Linux patch that accesses the robot control module (RCM), is done first (see section 7.7, page 69). We believe that the ER1s will be more easier to work with if we can control the rotation sensors without going through the robot control module (RCC).

- A solution to the localization problem. Work with robots, that are supposed to be autonomous, without reliable localization is very hard. There are many ways to go when it comes to choosing a localization method. We believe in the work started with the ultrasonic sensors (see appendix B). The few tests we have done with these sensors are very promising.

Bibliography

[1] O. J. Anshus, J. M. Bjørndalen, O. M. Bjørndalen, D. Stødle, and K. A. Jensen. ROBO: Implementation of a portable robot arena for demonstration of a parallel high performance computing environment within a distributed system. http://www.cs.uit.no/forskning/DOS/hpdc/robots/dprobots_lego/-dprobots.html, February 2003.

[2] R. A. Barman, S. J. Kingdon, A. K. Mackworth, D. K. Pai, M. K. Sahota, H. Wilkinson, and Y. Zhang. Dynamite: A testbed for multiple mobile robots. In *Proceedings of the IJCAI-93 Workshop on Dynamically Interacting Robots*, pages 38–44, 1993.

[3] D. M. Bourg and G. Seeman. *AI for game developers.* O'Reilly, 2004.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms.* The MIT Press, second edition, 2001.

[5] K. Dixon, J. Dolan, W. Huang, C. Paredis, and P. Khosla. RAVE: A real and viritual environment for multiple mobile robot systems. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS '99*, volume 3, pages 1360–1367, October 1999.

[6] R. Grabowski, L. E. Navarro-Serment, and P. K. Khosla. An army of small robots. *Scientific American*, 289(5):62–67, November 2003.

[7] F. Mondada, L. M. Gambardella, D. Floreano, S. Nolfi, J.-L. Deneuborg, and M. Dorigo. The cooperation of swarm-bots: physical interactions in collective robotics. *IEEE Robotics and Automation Magazine*, 12(2):21–28, June 2005.

[8] Robin.R. Murphy. Marsupial and shape-shifting robots for urban search and rescue. *Intelligent Systems, IEEE [see also IEEE Expert]*, 15:14–19, 2000.

[9] A. Nakamura, J. Ota, and T. Arai. Human-supervised multiple mobile robot system. *Robotics and Automation, IEEE Transactions on*, 18(5):728–743, October 2002.

[10] Lynne E. Parker. ALLIANCE: An architecture for fault-tolerant multi-robot cooperation. *IEEE Transactions on Robotics and Automation*, 14(2):220–240, 1998.

[11] Pygame. An open source community project. http://www.pygame.org/news.html.

[12] Marte Karidatter Skadsem. Robotmoderskip (ROMO). Spesialpensum, Institutt for informatikk, Det matematisk-naturvitenskaplige fakultet, Universitetet i Tromsø, June 2005.

[13] S. St.Laurent, J. Johnston, and E. Dumbill. *Programming Web Services with XML-RPC*. O'Reily, 2001.

[14] Katia P. Sycara. Multiagent systems. *AI Magazine*, 19(2), 1998.

Appendix A

The A* search algorithm

This appendix is a short introduction to the A* search algorithm (pronounced A-star). It is a graph searching algorithm that uses heuristic information to find minimum cost paths.

A graph is a set of nodes connected with edges. The edges may have costs associated with them, and in some cases they can be directed. This means for example that we can go from node A to node B but not from B to A. A* is used on graphs where the edges have costs, but are not directed. If we translate this to this thesis, the graph is the map, the nodes are the squares or coordinates in the map, and the edges are the movement from one node to an adjacent node.

A* uses two lists in its search. The OPEN list contains all nodes that need further exploration. The CLOSED list contains all nodes that have been explored.

The path is generated by repeatedly going through the OPEN list and choosing the node with the lowest F score. $F = G + H$ where G is the movement cost to move from the starting node to a given node using the path generated to get there, and H is the estimated movement cost from that given node to the final destination.

The G cost to a given node correspond to the sum of all edges of the graph used to get there from the starting node. We can assign different costs to the nodes to indicate that they are undesirable to be used in a path. Assume there is a table standing in the robots workspace. It is possible for the robots to move around it very close without crashing, but it is safer if they keep some distance from it. To make the robots keep some distance we add more cost to the nodes surrounding the table in the map. The A* algorithm will then avoid using those high cost nodes if it can. We operate with a normal cost of 1 and a high cost of 10.

H is the heuristic. It can be estimated in many ways. We estimate it by calculating the total number of nodes or map coordinates moved horizontally and vertically from the current node to the destination node, ignoring any obstacle that might be in the way. Then we multiply the sum by the normal cost of moving from one node to an adjacent, that is 1.

For each square in the map there are eight adjacent squares. As mentioned in section 4.4.4, the robots we have been working with will only make 90 degrees turns. This means we are operating with only four adjacent squares.

We will no go through the A* search step by step.

The search

Each explored node is given a parent node. The parent node is part of the shortest path to this node from the start node, and is used in the end of the search to find the path.

1. Mark the starting node as the current node.
2. Add the current node to the CLOSED list
3. For each of the current node's four adjacent nodes do:
    - If the node is part of a wall or some object that makes the node unwalkable, or if it is in the CLOSED list, ignore it.
    - If the node is not in the OPENED list, add it to this list. Make the current node the parent of this node and set the F, G and H costs.
    - If the node is in the OPEN list, use the G cost to calculate if this path to the node is better than the one found before. If so, make the current node the parent of this node and recalculate the F, G and H costs.
4. Search the OPEN list for the node with the lowest F cost. Mark this node as the current node.
5. Repeat step 2 to 4 until:
    - the destination node is added to the CLOSED list or
    - the OPEN list is empty which means that there exists no path between the starting and destination nodes.

The next part is to save the path. Start by working backwards from the destination node, searching the CLOSED list for it's parent node, and continue finding the parents until the starting node is reached.

Appendix B

Ultrasonic sensor navigation

B.1 Introduction

In this appendix we will describe a navigation method we tested during the project period. This work started late, and we did not manage to incorporate this method properly into the system within the required time. The results of this testing are very promising when it comes to making the robots able to move more freely and become more autonomous. This is the reason why we have this description here.

One of the technical staff in our department, Ken-Arne Jensen, has made a ultrasonic sensor kit that we have attached to one of the robots. The ultrasonic sensors are used to measure distances between the robot and objects around it. Our goal is to make the robot move by itself through a corridor and safely pass some stairs.

B.2 Design and implementation

The ultrasonic sensors used are BASIC stamp 2 Devantech SRF04 ultrasonic Range Finder #28015. They are attached to a Velleman USB Interface Experiment Board K8055 (after this called the K8055). The USB is connected to the notebook on the ER1 robot. For getting contact with the K8055, we use `ctype`. `ctype` is an advanced Foreign Function Interface package for Python. It "allows to call functions exposed from dlls/shared libraries and has extensive facilities to create, access and manipulate simple and complicated C data types in Python - in other words: wrap libraries in pure Python."[1] The ultrasonic sensors can measure distances from approximately 10 cm to approximately 300 cm from itself.

The four ultrasonic sensors are attached to the robot as can bee seen in the picture B.1. One in the front, one on each side, and one pointing down. This last sensor is meant to measure if the robot is driving toward an

[1] http://starship.python.net/crew/theller/ctypes/

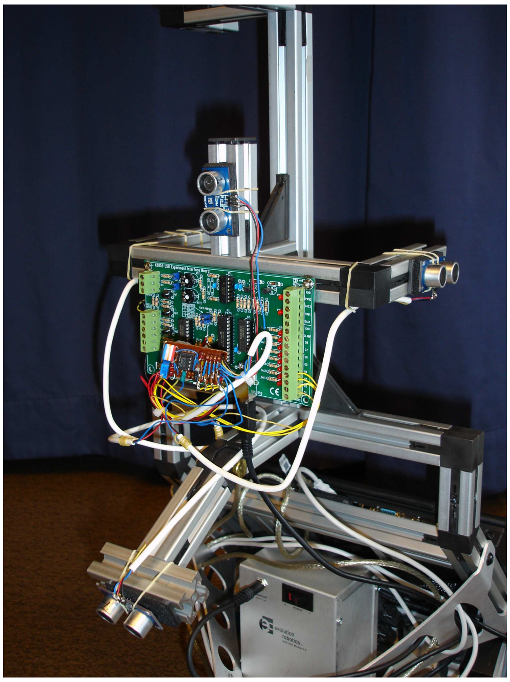


Figure B.1: Ultrasonic sensors attached to a ER1.

edge, like stairs. The one in front measures if the robot is about to run into something ahead. The two on the sides measures the distances to walls and objects on the sides.

We read the sensors in a round robin fashion. The senors a re numbered 1-4 and we read them in the sequence 1 2 3 4 and the we start in 1again. There has to be a delay of at least 10 milliseconds between each time a sensor is sending out a signal. If a smaller delay, the readings may be wrong.

The program we have made is just a simple testing program. Some places we have just chosen a value for how much to turn or how long distance to drive. When we here say that the robot turns or drives “a bit”, it means that it is one of these places.

The robot is programmed to drive on “the right hand side of the road”. By this we mean that it keeps close to walls on the right hand side of itself. As long as the sensor on the right hand side of the robot shows distances between 30 cm and 60 cm to the wall, the robot continues to drive forward. If the distance is smaller than 30 cm, the robot turns a bit to the left. If the distance is bigger than 60 cm, the robot turns a bit to the right. If the sensor on the left hand side of the robot shows a distance less than 30 cm, the robot turns a bit to the right.

If the sensor on the right hand side shows a distance bigger than 100 cm, it assumes that it has reached a corner. In order to turn this corner safely, it drives 30 cm forward (so that it gets room to turn) and turns 90 degrees to the right. Then it continues forward.

When the sensor in front of the robot shows a distance of less than 50 cm ahead, it means that the robot is about to run into something. Assuming this is a wall, the robot backs a bit and turns left. Then it continues forward.

When the sensor pointing down shows a distance bigger than 35 cm, it indicates an edge. The reason for 35 cm is that the robot drives with a speed of 10 cm/sec and that there can be more than one second between each time the sensor is read. 35 cm is the minimum safe stop distance.

B.3 Evaluation

Our goal with testing these sensors was to make the robot move by itself through a corridor and safely pass some stairs. This we almost achieved. Keeping the distance to the wall on the right hand side, went very well. Also avoid driving into something in front of the robot went very well. The tricky part was to turn corners and not driving down the stairs.

The trouble with turning corners was that the robot thought it saw a corner when it actually saw down the corridor. For example, the robot may stand in the middle of a corridor facing the wall and have free sight both upwards and downwards the corridor on the sides. We assume here that the wall in front of the robot is more than 50 cm away. In this situation, the right hand side sensor will correctly show a distance bigger than 100 cm. But it is wrong of the robot to then assume it must turn a corner. This problem is easily fixed by making the robot not assume it is always facing a corner in these cases.

We have not managed to get the robot avoid the stairs. This is due to bad readings from the sensor pointing down. The floor in the hall where the stairs are is the reason for these bad readings. The floor is made of small stones. Between these stones are small spaces that are one to two mm lower than the stones. Our theory is that the ultrasonic signal gets a wrong angle when it hits these spaces. The sensor readings will then show wrong distances. We faced two problems on this floor. The robot could detect an edge in the middle of the floor, and it could miss to detect the stairs.

A solution we did not have time to test, is to make the sensor point straight down and not in an angle like it points now. If we would do this we also would have to place it on a longer arm in front of the robot. This so that the robot could have a safe stop distance.

Another problem has been that the sensors does not discover legs of tables and chairs. This makes the robot run into these if they are standing in its way. How to avoid this, we have not had time to look at.

Despite these small problems, which can be solved, we find that this is a big step towards getting fully self-navigating robots. It was fun working with it, and it can easily be extended to be more dynamic and handle more situations.

Appendix C

Installation guide

This is an installation guide for all the extra software needed on the robots in this project.

In the implementation chapter (chapter 4) we introduced all the software we have used on the robots in order to implement the video making module. These are: the VideoCapture extension to get the video stream from the camera, PIL that VideoCapture uses to produce pictures, and ffmpeg to merge the pictures to a movie. In addition we need Msys and MinGW to compile the ffmpeg under Windows. All this software is included on the CD.

In this section, we will give information on how to install the software. The information is collected from the homepages of the different software. This guide is only meant as a short help. If any problems should occur during installation, please see the respective software's homepage.

C.1 VideoCapture

Packages on CD:	VideoCapture-0.9.zip
Homepage:	http://videocapture.sourceforge.net/

How to install:

1. Unzip the file.

2. Copy the files from the "PythonXX" folder to the corresponding folders of your "PythonXX" installation, where XX must match with the version of Python you have installed on your system.

C.2 Python Image Library (PIL)

Packages on CD:	PIL-1.1.5.win32-py2.4
Homepage:	http://www.pythonware.com/products/pil/index.htm
How to install:	

- Execute the PIL-1.1.5.win32-py2.4 file.

C.3 FFmpeg

Packages on CD:	ffmpeg-0.4.9-pre.1.tar
Homepage:	http://ffmpeg.sourceforge.net/index.php

The following instructions are copied directly from the FFmpeg documentation (see homepage), point 6.3.1 Native Windows compilation. For this project we do not need FFplay. The needed packages for Msys and MinGW are also included on the CD. See next section.

- Install the current versions of MSYS and MinGW from http://www.mingw.org/. You can find detailed installation instructions in the download section and the FAQ.
- If you want to test the FFplay, also download the MinGW development library of SDL 1.2.x ('SDL-devel-1.2.x-mingw32.tar.gz') from http://www.libsdl.org. Unpack it in a temporary directory, and unpack the archive 'i386-mingw32msvc.tar.gz' in the MinGW tool directory. Edit the 'sdl-config' script so that it gives the correct SDL directory when invoked.
- Extract the current version of FFmpeg.
- Start the MSYS shell (file 'msys.bat').
- Change to the FFmpeg directory and follow the instructions of how to compile FFmpeg (file 'INSTALL'). Usually, launching './configure' and 'make' suffices. If you have problems using SDL, verify that 'sdl-config' can be launched from the MSYS command line.
- You can install FFmpeg in 'Program Files/FFmpeg' by typing 'make install'. Don't forget to copy 'SDL.dll' to the place you launch 'ffplay' from.

Notes:

- The target 'make wininstaller' can be used to create a Nullsoft based Windows installer for FFmpeg and FFplay. 'SDL.dll' must be copied to the FFmpeg directory in order to build the installer.
- By using ./configure –enable-shared when configuring FFmpeg, you can build 'avcodec.dll' and 'avformat.dll'. With make install you install the FFmpeg DLLs and the associated headers in 'Program Files/FFmpeg'.
- Visual C++ compatibility: If you used ./configure –enable-shared when configuring FFmpeg, FFmpeg tries to use the Microsoft Visual C++ lib tool to build avcodec.lib and avformat.lib. With these libraries you can link your Visual C++ code directly with the FFmpeg DLLs (see below).

C.4 MinGW and Msys

Packages on CD:	MinGW-5.0.0.exe, MSYS-1.0.10.exe, msysDTK-1.0.1.exe
Homepage:	http://www.mingw.org

1. Execute the MinGW-5.0.0.exe file. Install MinGW in for example c:
2. Execute the MSYS-1.0.10.exe file. Install MSYS in for example
3. Execute the msysDTK-1.0.1.exe. This package gives autoconf, automake, libtool, cvs, etc.

MinGW is "a collection of freely available and freely distributable Windows specific header files and import libraries combined with GNU toolsets that allow one to produce native Windows programs that do not rely on any 3rd-party C runtime DLLs."[1]

MSYS is "a Minimal SYStem to provide POSIX/Bourne configure scripts the ability to execute and create a Makefile used by make."[2]

We need MinGW and MSYS in order to install ffmpeg because ffmpeg is developed under Linux.

[1] http://www.mingw.org/
[2] http://www.mingw.org/

Appendix D

Source code

The rest of this document contains the source code. It is organized as follows:

controller/

- controller.py
- eventhandler.py
- mapServiceIntf.py
- robotIntf.py
- userIntf.py
- robotInfrastructIntf.py
- objects.py

map/

- mapService.py
- storeHouse.py
- visualize.py
- objects.py

therobot/

- robot.py
- robotERintf.py
- robotMapping.py
- astar.py

- navigation.py
- video.py
- common.py
- ultradistRobot.py
- ultradist.py

user/

- user-app.py
- longstrings.py
- squareCode.py
- executeCode.py

```python
################################################################################
#
# The controller part of the infrastructure.
# Starts up all the interfaces and the eventhandler and gives the
# control to the eventhandler.
#
# Written by Marte K Skasdem, 2005/2006
#
################################################################################

# python library modules
import threading

# own written modules
from mapServiceIntf import MapServiceInterface
from robotIntf import RobotInterface
from userIntf import UserInterface
from eventHandler import EventHandler

class Controller:
    def __init__(self):
        # ipaddr to the maschine the controller runs on
        addr = '129.242.19.46'      #addr to rocksvv where tests has been done

        # initializes the different interfaces
        map_intf_port = 8080
        self.mapService_interface = MapServiceInterface(addr, map_intf_port)

        rob_intf_port = 8081
        self.robot_interface = RobotInterface(addr, rob_intf_port)

        usr_intf_port = 8082
        self.user_interface = UserInterface(addr, usr_intf_port)

        # initializes the eventhandler
        self.event_handler = EventHandler(self.robot_interface,
                                          self.user_interface,
                                          self.mapService_interface)


    def main(self):
        # make map
        self.initMap()

        self.mapService_interface.getEventHandler(self.event_handler)
        self.robot_interface.getEventHandler(self.event_handler)
        self.user_interface.getEventHandler(self.event_handler)

        # start interfaces in threads
        self.initThreads()

        # give control to the event handler
        self.event_handler.main()


    def initMap(self):
        '''Gives the map service the information in needs to mak
    the map.
    '''
        # the ipaddr to the computer the map service runs on
        map_addr = '129.242.19.46'

        # the port number to use to connect to map service
        map_port = 8091

        map_size_x = 200       # length of map
        map_size_y = 200       # height of map
        map_factor = 10        # the display factor
        map_piece_size = 200   # size of map piece to give to robots
```



```python
        known_obj = []         # coordinates to all known objects

        data = map_size_x, map_size_y, map_factor, map_piece_size, known_obj

        self.mapService_interface.initMap(map_addr, map_port, data)


    def initThreads(self):
        '''Starts of a thread for each interface. The threads handles
    inncomming calls from the other partisipants.
    '''
        mapService_intf_thread = threading.Thread(target=
                                          self.mapService_interface.serv
e)
        mapService_intf_thread.setDaemon(True)

        robot_intf_thread = threading.Thread(target=self.robot_interface.serve)
        robot_intf_thread.setDaemon(True)

        user_intf_thread = threading.Thread(target=self.user_interface.serve)
        user_intf_thread.setDaemon(True)

        mapService_intf_thread.start()
        robot_intf_thread.start()
        user_intf_thread.start()


if __name__ == '__main__':
    exSrv = Controller()
    exSrv.main()
```

```
################################################################################
#
# The event handler module.
# Handles incomming events from the interfaces.
#
# Written by Marte K Skasdem, 2005/2006
#
################################################################################

import threading, time

from objects import RobotObj

class EventHandler:
    def __init__(self, robotIntf, userIntf, mapServiceIntf):
        self.robotIntf = robotIntf
        self.userIntf = userIntf
        self.mapServiceIntf = mapServiceIntf

        self.robot_list = {}     # all known robots key=tag, value=robot object
        self.task_list = {}      # tasks not waiting to be executed.
                                 # key=robot tag, value=waiting tasks
        self.tag_list = {}       # key=ipaddr, value=tag
        self.tag = 100           # tag given to robots (incremental)


    def main(self):
        print 'Controller ready to start'
        while True:
            time.sleep(5)


#--------------------------------------------------
# functions called from map manipulator interface
#--------------------------------------------------
    #def getMonitorData(self, tag):
    #     declared together with functions
    #     called from user interface

    # def giveRobotTask(self, task, tag):
    #     declared together with functions
    #     called from user interface


#----------------------------------------
# functions called from robot interface
#----------------------------------------
    def appendRobot(self, ipaddr):
        '''Registers a new robot. The new robot is assigned a unique
    tag stored in a list with all known data about it data. Then
    the map service is contacted to get a map for the robot.
    The tag and map sata is returned to robot.
    '''
        # check if robot has been registered before
        if self.tag_list.has_key(ipaddr):
            # robot was lost, but is now found
            _tag = self.tag_list[ipaddr]
            self.robot_list[_tag].status = 'Connected'
        else:
            # robot new to infrastructure
            self.tag += 1
            self.tag_list[ipaddr] = self.tag
            _tag = self.tag
            self.robot_list[_tag] = RobotObj(_tag, ipaddr)

        self.task_list[_tag] = []

        # get map data
        map_data = self.getMapPiece(_tag)
```



```
        return _tag, map_data


    def getMapPiece(self, tag):
        '''Contacts the map service to get a map piece and the
    position of the robot.
    '''
        map_piece, map_size, rob_pos, data = self.mapServiceIntf.getMapPiece(tag
)

        # store the position
        self.robot_list[tag].pos = rob_pos

        return map_piece, map_size, rob_pos, data


    def taskOptions(self, tag, possible_tasks):
        '''A robot reports its possible tasks. This information is
    stored together with the rest information of this robot.
    '''
        self.robot_list[tag].possible_tasks = possible_tasks

        return True


    def requestNewTask(self, tag):
        '''Returns the next task to be executed by the calling
    robot if any is waiting.
    '''
        if not len(self.task_list[tag]) == 0:
            # task is waiting to be executed
            task = self.task_list[tag][0]
            self.task_list[tag].remove(task)
            self.robot_list[tag].task = task
            return task

        # no task available
        return False


    def reportPosition(self, tag, pos, direction):
        '''Robots call this to report a new position. The reported
    position is told to the map service in order for it to
    store it. Real position is returned from map service and
    returned to robot.
    '''
        pos = self.mapServiceIntf.moveRobot(tag, pos, direction)

        self.robot_list[tag].pos = pos
        self.robot_list[tag].direction = direction

        return pos


    def markNewObject(self, tag, imageid):
        '''Reports to the map service that a new object is discovered.
    The object is discovered by the calling robot.
    '''
        pos = self.robot_list[tag].pos

        return self.mapServiceIntf.markNewObject(pos,
                                                 tag,
                                                 imageid)


    def done(self, tag, task, msg):
        '''A robot reports that it has executed a task. This
    is stored and reported to the user.
```

```
    '''
        self.robot_list[tag].done_tasks.append( (task, msg) )
        self.robot_list[tag].task = False

        # report to user
        self.userIntf.robotDone(tag, task, msg)

        return True


    def robotDisconnect(self, tag):
        '''Called by a robot that leaves the work space area.
    Registers thet the robot is lost.
    '''
        self.robot_list[tag].status = 'Lost contact'

        self.mapServiceIntf.noContactRobot(tag)
        return True


#----------------------------------------
# functions called from user interface
#----------------------------------------
    def getRobotList(self):
        '''Returns a list of all robot tags known to be in thea area.
    '''
        if len(self.robot_list) == 0:
            return False

        data = []
        for rob in self.robot_list:
            data.append(rob)

        return data


    def getMonitorData(self, tag):
        '''Returns all known information of the robot requested.
    '''
        if not self.robot_list.has_key(tag):
             return False

        data = self.robot_list[tag].makeMonitorData()

        return data


    def getPossibleTasks(self, tag):
        '''Returns the possibel tasks for the robot requested.
    '''
        if not self.robot_list.has_key(tag):
            return False

        data = self.robot_list[tag].possible_tasks

        return data


    def giveRobotTask(self, tag, task, wait=None):
        '''Gives the given task to the requested robot. If the wait
    parameter is set, it means that the task can be stored and
    given to robot upon request form it.
    '''
        if wait:
            # add task to robot's task list
            self.task_list[tag].append(task)

        else:
            ### OBS! This will interrupt the robot in the task it is doing
```



```
            self.robot_list[tag].task = task
            ipaddr = self.robot_list[tag].ipaddr

            self.robotIntf.giveRobotTask(ipaddr, task)

        return True


    def giveRobotModule(self, tag, filename, binary_obj):
        '''Gives a module to the requested robot.
    '''
        ipaddr = self.robot_list[tag].ipaddr

        return self.robotIntf.exportModule(ipaddr,
                                           filename,
                                           binary_obj)


    def giveRobotCode(self, tag, filename, binary_obj):
        '''Gives some code to the requested robot
    '''
        ipaddr = self.robot_list[tag].ipaddr

        return self.robotIntf.runCode(ipaddr,
                                      filename,
                                      binary_obj)


    def stopRobot(self, tag):
        '''Stops the requested robot
    '''
        ipaddr = self.robot_list[tag].ipaddr

        return self.robotIntf.stopRobot(ipaddr)
```

```
##############################################################################
#
# The Interface to the map service.
# It is run as a XML RPC server, and uses a xmlrpclib client to
# call the map service.
#
# Written by Marte K Skasdem, 2005/2006
#
##############################################################################

import Queue

from xmlrpclib import ServerProxy
from SimpleXMLRPCServer import SimpleXMLRPCServer

class MapServiceInterface:
    def __init__(self, server_addr, server_port):
        self.server_addr = server_addr
        self.server_port = server_port

        # initiate server
        self.my_server = SimpleXMLRPCServer( (server_addr, server_port) )

        # register functions
        self.my_server.register_function(self.getMonitorData)
        self.my_server.register_function(self.newRobotTask)


    def initMap(self, map_addr, map_port, data):
        '''Initiates contact with map service and gives it
   information to make the map of.
   '''
        self.map = ServerProxy('http://' + map_addr + ':' + str(map_port))

        self.map.makeMap(self.server_addr, self.server_port, data)


    def getEventHandler(self, event_handler):
        '''Gets a pointer to the event handler
   '''
        self.eventHandler = event_handler


    def serve(self):
        '''Starts the server
   '''
        self.my_server.serve_forever()


#-----------------------------------------
# functions started from map manipulator
#-----------------------------------------
    def getMonitorData(self, tag):
        '''Gets monitor data for a requested robot
   '''
        print 'Get monitor data for map service'
        return self.eventHandler.getMonitorData(tag)


    def newRobotTask(self, tag, task):
        '''Gives task to requested robot
   '''
        print 'Give task to robot. Given by map service'
        return self.eventHandler.giveRobotTask(tag, task)


#----------------------------------------
# functions called from event handler
#----------------------------------------
```



```
    def markNewObject(self, tag, pos, data):
        '''Marks new object, discovered by the given
   robot, in the map.
   '''
        return self.map.markNewObject(tag, pos, data)


    def getMapPiece(self, tag):
        '''Gets a map piece for the given robot.
   '''
        return self.map.getMapPiece(tag)


    def moveRobot(self, tag, pos, direction):
        '''Registers new position of robot.
   '''
        return self.map.moveRobot(tag, pos, direction)


    def noContactRobot(self, tag):
        '''Marks that a robot is lost
   '''
        return self.map.noContactRobot(tag)
```

```
##############################################################################
#
# The controller's interface to the robots.
# It is run as an asyncron XML RPC server that can serve
# several clients concurrently.
#
# Written by Marte K Skasdem, 2005/2006
#
##############################################################################

import os
import SocketServer
import socket

from SimpleXMLRPCServer import SimpleXMLRPCServer,SimpleXMLRPCRequestHandler
from xmlrpclib import ServerProxy, Binary

class AsyncXMLRPCServer(SocketServer.ThreadingMixIn,SimpleXMLRPCServer):
    def server_bind(self):
        self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
        self.socket.bind(self.server_address)


class RobotInterface:
    def __init__(self, addr, port):
        # initialize server
        self.initServer(addr, port)

        # register server functions
        self.my_server.register_function(self.initphase)
        self.my_server.register_function(self.welcome)
        self.my_server.register_function(self.taskOptions)
        self.my_server.register_function(self.requestNewTask)
        self.my_server.register_function(self.reportPosition)
        self.my_server.register_function(self.newMap)
        self.my_server.register_function(self.storeFile)
        self.my_server.register_function(self.taskDone)
        self.my_server.register_function(self.robotDisconnect)


    def initServer(self, addr, port):
        '''Initiates the xml rpc server
    '''
        self.my_server = AsyncXMLRPCServer((addr, port),
                                           SimpleXMLRPCRequestHandler)


    def getEventHandler(self, event_handler):
        '''Gets a pointer to the event handler
    '''
        self.eventHandler = event_handler


    def serve(self):
        '''Starts the server
    '''
        self.my_server.serve_forever()

#--------------------------------------------
# FROM ROBOT
#--------------------------------------------
    def initphase(self):
        '''Sends the robotInfrastructIntf to the calling robot.
    '''
        intf = open("robotInfrastructIntf.py", "r")
        data = intf.read()
        intf.close()

        binary_obj = Binary(data)
```



```
        return "robotInfrastructIntf.py", binary_obj


    def welcome(self, ipaddr):
        '''Welcomes a robot. Returns map information and a tag.
    '''
        return self.eventHandler.appendRobot(ipaddr)


    def taskOptions(self, tag, possible_tasks):
        '''Receives a robots possible tasks.
    '''
        return self.eventHandler.taskOptions(tag, possible_tasks)


    def requestNewTask(self, tag):
        '''Returns a new task if one exist.
    '''
        return self.eventHandler.requestNewTask(tag)


    def reportPosition(self, tag, pos, direction):
        '''Robots call this to report a new position.
    '''
        return self.eventHandler.reportPosition(tag, pos, direction)


    def newMap(self, tag):
        '''
    '''
        return self.eventHandler.getMapPiece(tag)


    def storeFile(self, tag, binary_obj, filename):
        '''Stores file given by a robot.
    '''
        outfile = open("files/"+filename, "w")
        outfile.write(binary_obj.data)
        outfile.close()

        self.eventHandler.markNewObject(tag, 'files/'+filename)

        return True


    def taskDone(self, tag, task, msg):
        '''A robot is done executing a task.
    '''
        return self.eventHandler.done(tag, task, msg)


    def robotDisconnect(self, tag):
        '''A robot is leaving the area.
    '''
        self.eventHandler.robotDisconnect(tag)
        return True


#--------------------------------------------
# TO ROBOT
#--------------------------------------------
    def giveRobotTask(self, ipaddr, task):
        '''Gives new task to robot
    '''
        print "newTask: " + str(task)
        robot = self.connectRobot(ipaddr)
        return robot.giveRobotTask(task)
```

```
    def exportModule(self, ipaddr, modulename, binary_obj):
        '''Sends a new module to robot.
   '''
        print "new module: " + modulename
        robot = self.connectRobot(ipaddr)

        return robot.importModule(modulename, binary_obj)


    def runCode(self, ipaddr, codefilename, binary_obj):
        '''Sends a file with code to execute to robot and asks
   it to run it.
   '''
        print "run code: " + codefilename
        robot = self.connectRobot(ipaddr)

        return robot.runCode(codefilename, binary_obj)


    def getMonitorData(self, ipaddr):
        '''Gets monitor data from robot.
   '''
        robot = self.connectRobot(ipaddr)
        return robot.getMonitorData()


    def uppdateMap(self, ipaddr, object_list):
        '''Sends a new list of objects to robot so that it
   can update its map.
   '''
        robot = self.connectRobot()
        return robot.uppdateMap(object_list)


    def stopRobot(self, ipaddr):
        '''Stops robots movement
   '''
        print "stop robot " + ipaddr
        robot = self.connectRobot(ipaddr)
        return robot.stop()


#---------------------------------------------------------------------------
# LOCALE FUNCTIONS
#----------------------------------------------------------------------------
    def connectRobot(self, ipaddr):
        '''Opens a connection to the given robot.
   '''
        robot_connection = ServerProxy('http://' +\
                                       ipaddr +\
                                       ':8090')
        return robot_connection
```

```
################################################################################
#
# The controller’s interface to the user.
# It is run as a XML RPC server.
#
# Written by Marte K Skasdem, 2005/2006
#
################################################################################

import Queue
from SimpleXMLRPCServer import SimpleXMLRPCServer

class UserInterface:
    def __init__(self, addr, port):
        # a queue to hold done robot tasks
        self.doneQ = Queue.Queue()

        # initiate server
        self.my_server = SimpleXMLRPCServer( (addr,
                                              port) )

        # register server functions
        self.my_server.register_function(self.testConnection)
        self.my_server.register_function(self.getRobotList)
        self.my_server.register_function(self.monitorRobot)
        self.my_server.register_function(self.giveTask)
        self.my_server.register_function(self.getPossibleTasks)
        self.my_server.register_function(self.sendCode)
        self.my_server.register_function(self.checkDone)
        self.my_server.register_function(self.stopRobot)


    def getEventHandler(self, event_handler):
        '''Gets a pointer to the event handler
    '''
        self.eventHandler = event_handler


    def serve(self):
        '''Starts the server
    '''
        self.my_server.serve_forever()

#---------------------------------------------
    def testConnection(self):
        '''User tests its connection
    '''
        return True


    def getRobotList(self):
        '''Returns a list with tags of all known robots.
    '''
        return self.eventHandler.getRobotList()


    def monitorRobot(self, tag):
        '''Returns a robots monitor data
    '''
        return self.eventHandler.getMonitorData(tag)


    def giveTask(self, tag, task, wait=None):
        '''Gives a new task to a robot.
    '''
        if wait:
            # do not interrupt robot
            return self.eventHandler.giveRobotTask(tag, task, True)
        else:
```



```
            # interrup robot in its current task
            return self.eventHandler.giveRobotTask(tag, task)


    def getPossibleTasks(self, tag):
        '''Returns a robots possible tasks.
    '''
        return self.eventHandler.getPossibleTasks(tag)


    def sendCode(self, tag, codefilename, binary_obj, code=None):
        '''Sends a file to a robot.
    '''
        if code:
            # the file is code to execute
            return self.eventHandler.giveRobotCode(tag, codefilename,
                                                   binary_obj)
        else:
            # the file is a module
            return self.eventHandler.giveRobotModule(tag, codefilename,
                                                     binary_obj)

    def checkDone(self):
        '''Checks the queue if a robot is finished with a
    task execution.
    '''
        try:
            result = self.doneQ.get(False)
            return result
        except:
            return False


    def stopRobot(self, tag):
        '''Returns when requested robot has stopped
    '''
        return self.eventHandler.stopRobot(tag)

#---------------------------------------------
# From eventhandler
#---------------------------------------------
    def robotDone(self, tag, task, msg):
        '''A robot is done with a task execution. Puts
    information about in the queue
    '''
        self.doneQ.put( (tag, task, msg) )
        return True
```

```
################################################################################
#
# The robot's interface to the infrastructure.
# It is run as a XML RPC server, and uses a xmlrpclib client to
# call the infrastructure.
#
# Written by Marte K Skasdem, 2005/2006
#
################################################################################

import time, os

from xmlrpclib import ServerProxy, Binary
from SimpleXMLRPCServer import SimpleXMLRPCServer


class RobotInfrastructureIntf:
    def __init__(self, my_robot, infrastruct_addr, infrastruct_port, my_port):
        self.my_robot = my_robot    # pointer to robot class

        # set up a client to the infrastructure server
        self.infrastructure = ServerProxy('http://' + infrastruct_addr + \
                                          ':' + str(infrastruct_port))

        # set up robot's server
        self.my_server = SimpleXMLRPCServer( (my_robot.addr, my_port) )

        # register server functions
        self.my_server.register_function(self.giveRobotTask)
        self.my_server.register_function(self.runCode)
        self.my_server.register_function(self.importModule)
        self.my_server.register_function(self.getMonitorData)
        self.my_server.register_function(self.uppdateMap)
        self.my_server.register_function(self.stop)


    def serve(self):
        ''' Starts server.
    '''
        self.my_server.serve_forever()


#-------------------------------
# FUNCTIONS CALLED BY ROBOT
#-------------------------------
    def callWelcome(self, addr):
        '''Returns information about the map of the robot.
    '''
        return self.infrastructure.welcome(addr)


    def taskOptions(self, tag, possible_tasks):
        '''Sends a string containing a description of
    possible tasks the robot can execute.
    '''
        return self.infrastructure.taskOptions(tag, possible_tasks)


    def requestNewTask(self, tag):
        '''Requests a new task from the infrastructure.
    '''
        return self.infrastructure.requestNewTask(tag)


    def reportPosition(self, tag, pos, direction):
        '''Reports a new position.
    '''
        return self.infrastructure.reportPosition(tag, pos, direction)
```



```
    def newMap(self, tag):
        '''Requests a new map piece.
    '''
        return self.infrastructure.newMap(tag)


    def streamFile(self, tag, filedirectory, filename):
        '''Streams a file to the infrastructure.
    '''
        print 'streamFile(): ' + filedirectory+filename
        f = open(filedirectory + filename, 'r')
        stream = f.read()
        f.close()

        binary_obj = Binary(stream)
        return self.infrastructure.storeFile(tag, binary_obj, filename)


    def done(self, tag, task, msg):
        '''Reports that a task is done and how it went.
    '''
        return self.infrastructure.taskDone(tag, task, msg)


    def disconnect(self, tag):
        '''Reports that robot leaves the area
    '''
        self.infrastructure.robotDisconnect(tag)

#--------------------------------------
# FUNCTIONS CALLED BY INFRASTRUCTURE
#--------------------------------------
    def giveRobotTask(self, task):
        '''Interrupts the robot in its current tasks and
    gives it a new.
    '''
        self.interrup()
        return self.my_robot.executeTask(task)


    def runCode(self, codefilename, code):
        '''Stores the file containing code to execute. Interrupts
    the robot in its current task execution and starts the
    code execution.
    '''
        outFile = open(codefilename, "w")
        outFile.write(code.data)
        outFile.close()

        self.interrup()

        return self.my_robot.runCode(codefilename)


    def importModule(self, modulename, code):
        '''Stores the file containing the module to be imported.
    '''
        outFile = open(modulename, "w")
        outFile.write(code.data)
        outFile.close()

        modulename = modulename.split('.')[0]

        return self.my_robot.importModule(modulename)


    def getMonitorData(self):
        '''Returns information that can be shown in monitoring of
```

```
    the robot.
    '''
        return self.my_robot.getMonitorData()


    def uppdateMap(self, list_of_objects):
        '''Uppdates the map according to new list of objects.
    '''
        return self.my_robot.uppdateMap(list_of_objects)


    def stop(self):
        '''Stops robot movement.
    '''
        self.my_robot.stop()
        return True


#-----------------------------------
# LOCALE FUNCTIONS
#-----------------------------------
    def interrup(self):
        '''Interrupts the robot in its current task execution.
    Returns when robot is ready for new task execution.
    '''
        print 'Got interruption signal'
        self.my_robot.interrupt = True
        while not self.my_robot.stopped:
            time.sleep(1)
        return True
```

```
################################################################################
#
# Robot object class.
#
# Written by Marte K Skasdem, 2005/2006
#
################################################################################

class RobotObj:
    def __init__(self, tag, ipaddr):
        '''This is the robot object. Here all information
    about a robot should be saved.
    tag = the robot s unique id
    '''
        self.tag = tag
        self.ipaddr = ipaddr
        self.pos = (0,0)
        self.status = 'Connected'
        self.task = False
        self.done_tasks = []
        self.possible_tasks = ''


    def makeMonitorData(self):
        '''Returns a string containig the current situation of the
    robot.
    '''
        txt_done_tasks = '\n'
        for task in self.done_tasks:
            txt_done_tasks = txt_done_tasks + str(task) + '\n'
        data = '**************************************************\n' +\
               'Tag:            \t' + str(self.tag) + '\n' +\
               'Last known position:\t' + str(self.pos) + '\n' +\
               'Status:         \t' + str(self.status) + '\n' +\
               'Current task:   \t' + str(self.task) + '\n' +\
               'Done tasks:     \t' + txt_done_tasks +\
               'Possible tasks:\n' + self.possible_tasks +'\n' +\
               '**************************************************\n'
        return data
```

```
################################################################################
#
# The map service.
# It runs a XML RPC server that the controller uses to get data
# form the map. It also starts off a visuaization thread that
# makes graphical output of the map, and handles moouse events reported
# by thhis visualization thread.
#
# Written by Marte Karidatter Skadsem, 2005/2006
#
################################################################################

import Queue, threading, os, time

from xmlrpclib import ServerProxy
from SimpleXMLRPCServer import SimpleXMLRPCServer

from visualize import Visualize
from objects import Drawing
from storeHouse import StoreHouse

class MapService:
    def __init__(self):
        self.robots = {}            # all robots, key=tag, value=drawing object
        self.controller = False     # pointer to controller server
        self.list_of_objects = []   # list of known objects
        self.ready = False      # set when ready to handle data from controller
        self.connected = False      # set when controller is connected


    def main(self):
        # init server and start server thread
        self.initServer()
        server_thread = threading.Thread(target=self.serve)
        server_thread.setDaemon(True)
        server_thread.start()

        # wait for connection from task manager
        print 'Waiting to be connected....'
        while not self.connected:
            time.sleep(2)


         # initialize the visualizating module
        self.outQ = Queue.Queue()       # queue for messages to the
                                        # visualization thread
        self.display = Visualize(self.outQ, self.store)
        self.inQ = self.display.outQ    # queue for messages from the
                                        # visualization thread


        # start visualization thread
        self.viso_thread = threading.Thread(target=self.display.runVisualize)
        self.viso_thread.setDaemon(True)
        self.viso_thread.start()

        # start handling mouse events from user
        mouse_thread = threading.Thread(target=self.handleMouseEvents)
        mouse_thread.setDaemon(True)
        mouse_thread.start()

        self.startpos = 10

        # ready to take events from controller
        self.ready = True

        while True:
            time.sleep(5)
```



```
    def initServer(self):
        '''Initializes server and registers server functions.
    '''
        addr = '129.242.19.46'     # addr to computer where map
                                    # service has been tested
        port = 8091

        self.my_server = SimpleXMLRPCServer( (addr, port) )

        self.my_server.register_function(self.makeMap)
        self.my_server.register_function(self.getMapPiece)
        self.my_server.register_function(self.moveRobot)
        self.my_server.register_function(self.markNewObject)
        self.my_server.register_function(self.noContactRobot)


    def serve(self):
        '''Starts server
    '''
        self.my_server.serve_forever()


#-------------------------------------------------------------------------------
# Main functions
#-------------------------------------------------------------------------------
    def handleMouseEvents(self):
        '''Handles mouse events from visualization thread. Calls
    controller to perform actions.
    '''
        path = False
        while True:
            data = self.inQ.get()
            # mouseclick occured
            if self.robots.has_key(data):
                # user clicked a robot
                tag = data
                try:
                    # check if double click or task giving
                    data = self.inQ.get(True, 1.5)
                    if data == tag:
                        # get monitor data of robot
                        if self.robots[tag].status == 'Connected':
                            self.showMonitorData(tag)
                    else:
                        # make path
                        path = []
                        while True:
                            path.append(data)
                            data = self.inQ.get(True, 1)
                except:
                    # no new mous event, task giving finished
                    if path:
                        if len(path) == 1:
                            # only one position given => goTo-task
                            task = ('goTo', path[0])
                        else:
                            # folloPath-task
                            task = ('followPath', path)

                        if self.robots[tag].status == 'Connected':
                            # give task
                            self.controller.newRobotTask(tag, task)
                        path = False

            elif self.list_of_objects.has_key(data):
                # user clicked a discovered object
                if self.list_of_objects[data].movie:
                    # show information of object
```

```
                    print self.list_of_objects[data].movie
                    #cmd = 'DISPLAY=:0 screen -d -m  mplayer ' + \
                    #      self.list_of_objects[tag].video
                    #os.system(cmd)


    def showMonitorData(self, tag):
        '''Stores received monitor data in a .txt file and show it
    in an oen window.
    '''
        data = self.controller.getMonitorData(tag)
        filename = 'monitor'+str(tag)+'.txt'
        f=open(filename,'w')
        f.write(data)
        f.close
        os.system('xmessage -file '+filename+'&')


#-------------------------------------------------------------------------------
# Server functions
#-------------------------------------------------------------------------------
    def makeMap(self, contr_addr, contr_port, data):
        '''Initiates a client to the controller and makes the map
    based on the parameters
    '''
        self.controller = ServerProxy('http://' + contr_addr + \
                                      ':' + str(contr_port))
        self.store = StoreHouse(data)
        self.list_of_objects = self.store.list_of_objects
        self.connected = True
        return True


    def getMapPiece(self, tag):
        '''Finds map information to a robot and puts robot in the
    queue to be drawn.
    '''
        # wait until ready
        while not self.ready:
            time.sleep(1)

        # get position from a non existing position system
        pos = (self.startpos, 10)
        self.startpos = self.startpos + 50

        # aquire store house's lock to edit data stored
        while self.store.lock:
            time.sleep(0.5)
        self.store.lock = True

        # make map piece
        map_piece, map_size, rob_pos, data = self.store.makeMapPiece(tag, pos)

        # release store house's lock
        self.store.lock = False

        if not self.robots.has_key(tag):
            # make new drawing object if robot is new
            self.robots[tag] = Drawing(tag, rob_pos, map_piece, map_size)
        else:
            self.robots[tag].status = 'Connected'

        # put the drawing object on queue to be drawn
        msg = 'newObj', self.robots[tag]
        self.outQ.put(msg)

        return map_piece, map_size, rob_pos, data
```



```
    def moveRobot(self, tag, pos, direction):
        '''Finds the right robot to be moved and returns it
    with the new position.
    '''
        # wait until ready
        while not self.ready:
            time.sleep(1)

        robot = self.robots[tag]

        #robot.pos = self.positioning.getPosition(tag)
        robot.pos = pos
        robot.direction = direction
        robot.getImage()

        # draw changes
        msg = 'move', robot
        self.outQ.put(msg)

        return robot.pos


    def markNewObject(self, pos, tag, movie):
        '''Appends new unidentifyed object to list_of_objects.
    Puts object in the queue to be drawn.
    '''
        # wait until ready
        while not self.ready:
            time.sleep(1)

        # aquire store house's lock to edit data stored
        while self.store.lock:
            time.sleep(0.5)
        self.store.lock = True

        #store new object
        mark = self.store.newObject(pos, tag, movie)

        # release lock
        self.store.lock = False

        # put object on queue to be drawn
        msg = 'newObj', self.list_of_objects[mark]
        self.outQ.put(msg)

        return True


    def noContactRobot(self, tag):
        '''Lost conntact with a robot. Uppdates the status.
    '''
        # wait until ready
        while not self.ready:
            time.sleep(1)

        self.robots[tag].status = 'lost'

        # show changes in graphical output
        msg = 'lost', self.robots[tag]
        self.outQ.put(msg)

        return True



if __name__ == '__main__':
    exSrv = MapService()
    exSrv.main()
```

```python
##############################################################################
#
# The store house.
# Makes and maintains the map.
#
# Written by Marte Karidatter Skadsem, 2005/2006
#
##############################################################################

from objects import UnidfObj

class StoreHouse:
    def __init__(self, data):
        x, y, factor, map_piece_size, known_obj = data

        # the mapping factor
        self.display_factor = factor

        # size of robot maps
        self.map_piece_size = map_piece_size

        # list of all known objects
        self.known_obj = known_obj

        # size of the display
        self.disp_size_x, self.disp_size_y = self.findDisplayPositions((x,y))

        # make the map
        self.map = self.initMap(x, y)
        for o in self.known_obj:
            o_x , o_y = o
            self.map[o_y][o_x] = 'w'

        # some global variables
        self.mark = 0              # unique number for new objects
        self.list_of_objects = {}  # all found object key=id, value=unidf obj
        self.pieces = {}           # key=robot id, value=map_piece
        self.lock = False          # to prevent several threads from
                                   # accessing the same datastructure


    def initMap(self, x, y):
        '''Makes the map
    '''
        new_map = []
        for i in range(y):
            new_map.append([])
            for j in range(x):
                new_map[i].append('')

        return new_map


    def findDisplayPositions(self, real_pos):
        '''Take a position from the real world and find
    the corresponding display position.
    The map is not allways drawn 1:1. The relationship
    is stored in self.display_factor.
    '''
        real_x, real_y = real_pos
        disp_x = real_x * self.display_factor
        disp_y = real_y * self.display_factor
        return (disp_x, disp_y)


    def findRealPositions(self, disp_pos):
        '''Take a display position and find the corresponding
    real world position.
    The disp is not allways drawn 1:1. The relationship
```



```python
    is stored in self.display_factor.
    '''
        disp_x, disp_y = disp_pos
        real_x = disp_x / self.display_factor
        real_y = disp_y / self.display_factor
        return (real_x, real_y)


    def newObject(self, pos, tag, movie):
        self.mark += 1
        self.list_of_objects[self.mark] = UnidfObj(self.mark,
                                                   pos,
                                                   tag,
                                                   movie)
        return self.mark


    def makeMapPiece(self, tag, pos):
        '''Finds the map mpiece to be given to a robot. The maximum
    size of the piece is determined by self.map_piece_size. The
    piece is made so that the robot is placed in the midle.
    The only information given back to the robot is how big the
    map is, where it is placed, known objects inside the map
    piece and if it shares som aera of the map piece with another
    robot.
    '''
        if self.map_piece_size == len(self.map):
            # size of map piece equals map size
            map_piece = (0, 0)
            map_size = (self.map_piece_size, self.map_piece_size)
            rob_pos = pos
            data = self.known_obj, []

        else:
            robx, roby = pos

            # find the start and end x-values for the piece
            startx, endx = self.findvalue( len(self.map[0]),
                                  robx - (self.map_piece_size/2) )
            # map's length
            lenx = endx - startx

            # find the start and end y-values for the piece
            starty, endy = self.findvalue( len(self.map),
                                  roby - (self.map_piece_size/2) )
            # map's height
            leny = endy - starty

            # find known objects in map piece
            obj = []
            for o in self.known_obj:
                ox, oy = o
                if ox >= startx and ox <= endx:
                    if oy >= starty and oy <= endy:
                        local_ox = ox - startx
                        local_oy = oy -  starty
                        obj.append(local_ox, local_oy)

            # check if mp piece overlaps with ither pieces
            overlap = self.checkMapPiece(tag, startx, starty)
            self.pieces[tag] = (startx, starty)

            map_piece = (startx, starty)
            map_size = (lenx, leny)
            rob_pos = (robx, roby)
            data =  obj, overlap

        return map_piece, map_size, rob_pos, data
```

```
    def findvalue(self, length, start):
        '''Find a valid value for start- and end-
coordinates.
'''
        end = start + self.map_piece_size - 1
        if start < 0: start = 0
        if end >= length: end = length - 1
        return (start, end)


    def checkMapPiece(self, id, startx, starty):
        '''Check if there are other robots sharing some
parts of the map piece.
'''
        overlap = []
        endx = startx + self.map_piece_size - 1
        endy = starty + self.map_piece_size - 1

        for tag in self.pieces:
            if not tag == id:
                robstartx, robstarty = self.pieces[tag]
                robendx = robstartx + self.map_piece_size - 1
                robendy = robstarty + self.map_piece_size - 1

                if startx >= robstartx and startx <= robendx:
                    # the left corner is inside another robot's map piece
                    if starty >= robstarty and starty <= robendy:
                        # up-left corner
                        overlap.append(self.findIntersection('ul',
                                                             startx,
                                                             starty,
                                                             robstartx,
                                                             robstarty)
                                       )
                    elif endy >= robstarty and endy <= robendy:
                        # down-left corner
                        overlap.append(self.findIntersection('dl',
                                                             startx,
                                                             starty,
                                                             robstartx,
                                                             robstarty)
                                       )
                elif endx >= robstartx and endx <= robendx:
                    # the right corner is inside another robot's map piece
                    if starty >= robstarty and starty <= robendy:
                        # up-right corner
                        overlap.append(self.findIntersection('ur',
                                                             startx,
                                                             starty,
                                                             robstartx,
                                                             robstarty)
                                       )
                    elif endy >= robstarty and endy <= robendy:
                        # down-right corner
                        overlap.append(self.findIntersection('dr',
                                                             startx,
                                                             starty,
                                                             robstartx,
                                                             robstarty)
                                       )
        return overlap


    def findIntersection(self, corner, startx, starty, robstartx, robstarty):
        '''Find the exact coordinates of shared aeras in the map piece
'''
        endx = startx + self.map_piece_size - 1
        endy = starty + self.map_piece_size - 1
```



```
        robendx = robstartx + self.map_piece_size - 1
        robendy = robstarty + self.map_piece_size - 1

        overlap = []
        if corner == 'ul':
            for i in range(startx, robendx+1):
                for j in range(starty, robendy+1):
                    self.map[j][i] = 's'
                    localx = i - startx
                    localy = j - starty
                    overlap.append((localx, localy))
        elif corner == 'dl':
            for i in range(startx, robendx+1):
                for j in range(robstarty, endy+1):
                    self.map[j][i] = 's'
                    localx = i - startx
                    localy = j - starty
                    overlap.append((localx, localy))
        elif corner == 'ur':
            for i in range(robstartx, endx+1):
                for j in range(starty, robendy+1):
                    self.map[j][i] = 's'
                    localx = i - startx
                    localy = j - starty
                    overlap.append((localx, localy))
        else:
            #corner == 'dr':
            for i in range(robstartx, endx+1):
                for j in range(robstarty, endy+1):
                    self.map[j][i] = 's'
                    localx = i - startx
                    localy = j - starty
                    overlap.append((localx, localy))
        return overlap
```

```
###############################################################################
#
# Th evisualizatioon module
# This class takes care of all the pygame-stuff and produces
# a graphical output.
#
# Written by Marte K Skadsem, autum 2005
#
###############################################################################

import pygame
from pygame.locals import *
import threading, Queue, time


class Visualize:
    def __init__(self, inQ, storeHouse):
        self.inQ = inQ              # queue to get uppdates from map server
        self.outQ = Queue.Queue()  # queue to put messages to mapserver

        self.store = storeHouse    # where the map is stored

        # aquire store house's lock to edit stored data
        while self.store.lock:
            time.sleep(0.3)
        self.store.lock = True

        # get size odf displayed map
        x = self.store.disp_size_x
        y = self.store.disp_size_y

        self.stop = False          # decides when thread stops

        # initialize PyGame
        pygame.init()
        pygame.display.set_caption('Where is the robot?')

        # adds the size of the robot image and the size of the
        # display so that robots will not be drawn outside display
        self.robot_image = pygame.image.load("../pygameImg/N.png")
        rob_img_width, rob_img_height = self.robot_image.get_size()
        x = x + rob_img_width
        y = y + rob_img_height

        # create the display surface
        self.screen = pygame.display.set_mode( (x,y) )
        self.screen.fill((255,255,255))

        # makes a white background, blits on the screen, and shows the updates
        self.background = pygame.Surface((x,y))
        self.background.fill((255,255,255))

        #fill in walls
        for x,y in self.store.list_of_objects:
            for i in range(self.store.display_factor):
                for j in range(self.store.display_factor):
                    background.set_at((x+i,y+j), (0,0,0))

        # release store lock
        self.store.lock = False

        self.screen.blit(self.background, (0,0))
        pygame.display.update()

        # displayed objects key=tag/id value=rect
        self.map_objects = {}

        # size of mouse
        self.mouse = pygame.Rect(0, 0, 5, 5)
```



```
    def runVisualize(self):
        '''The main function.
Starts a thread for checking whether to kill
the display window, and gets events from the in-queue.
'''
        mouse_thread = threading.Thread(target=self.captureEvents)
        mouse_thread.setDaemon(True)
        mouse_thread.start()

        while not self.stop:
            task, obj = self.inQ.get()

            if task == 'newObj':
                self.newObject(obj)
            elif task == 'move':
                self.moveObject(obj)
            elif task == 'lost':
                self.lostContact(obj)
            else:
                print 'Unkknown task'

        print 'QUIT!'


    def newObject(self, obj):
        '''Puts a new object in map_objects
type(obj) = Drawing object
'''
        self.map_objects[obj.id] = obj.rect

        self.moveObject(obj)


    def moveObject(self, obj):
        '''Moves the given robot to the new position.
'''
        self.screen.blit(self.background, obj.rect, obj.rect)

        for tag, drawn_rect in self.map_objects.iteritems():
            if not tag == obj.id:
                if drawn_rect.topleft == obj.rect.topleft:
                    image = self.store.list_of_objects[tag].image
                    self.screen.blit(image, drawn_rect)

        if obj.status == 'lost':
            obj.status = 'ok'

        obj.rect = obj.image.get_rect()

        while self.store.lock:
            time.sleep(0.3)
        self.store.lock = True
        obj.rect.topleft = self.store.findDisplayPositions(obj.pos)
        self.store.lock = False

        self.drawObject(obj)
        self.map_objects[obj.id] = obj.rect


    def drawObject(self, obj):
        '''Draws the given robot on the display screen.
'''
        self.screen.blit(obj.image, obj.rect)
        pygame.display.update()


    def lostContact(self, obj):
```

```
        ''' Lost conntact with a robot. Uppdates display
    accordingly.
    '''
        rect = obj.rect
        self.screen.blit(self.background, rect, rect)

        noContact_image = pygame.image.load("../pygameImg/NoConntact.png")
        new_rect = noContact_image.get_rect()
        new_rect.topleft = rect.topleft
        self.screen.blit(noContact_image, new_rect)

        pygame.display.update()

        obj.rect = new_rect
        obj.status = 'lost'
        self.map_objects[obj.id] = obj.rect


    def captureEvents(self):
        '''The quit-thread. If a certant event occur, it
    kills the display window and set a global stop-
    variable so that the whole visualization thread
    stops. It also registers mouse events.
    '''
        while not self.stop:
            for event in pygame.event.get():
                if event.type == QUIT:
                    self.stop = True
                elif event.type == KEYDOWN and event.key == K_ESCAPE:
                    self.stop = True
                elif event.type == MOUSEBUTTONDOWN:
                    self.selectRect()
        pygame.quit()


    def selectRect(self):
        '''If the mouse clicked on a robot or an object,
    send the tag, else send teh position of the mouse.
    '''
        mouse_pos = pygame.mouse.get_pos()
        self.mouse.topleft = (mouse_pos)

        for tag, rect in self.map_objects.iteritems():
            if self.mouse.colliderect(rect):
                self.outQ.put(tag)
                return

        while self.store.lock:
            time.sleep(0.3)
        self.store.lock = True
        real_pos = self.store.findRealPositions(mouse_pos)
        self.store.lock = False

        self.outQ.put(real_pos)
```

```
################################################################################
#
# Classes for storing data relevant for the graphical output.
#
# Written by Marte Karidatter Skadsem, 2005/2006
#
################################################################################

import pygame
from pygame.locals import *

class UnidfObj:
    '''Class for storing drawing inforamtion about a discovered object.
  '''
    def __init__(self, mark, pos, tag, movie):
        self.id = mark
        self.pos = pos
        self.finder = tag
        self.movie = movie
        self.image = pygame.image.load("../pygameImg/uidfObj.png")
        self.rect = self.image.get_rect().move(pos)
        self.video = False
        self.status = 'new'



class Drawing:
    '''Class for storing drawing information about a robot.
  '''
    def __init__(self, tag, pos, map_piece, map_size):
        self.id = tag
        self.pos = pos
        self.status = 'Connected'
        self.map_piece = map_piece
        self.map_size = map_size
        self.direction = 'N'

        self.image = pygame.image.load("../pygameImg/N.png")

        #surface.get_rect() returns a rect covering the entire surface
        #rect.move() returns a new rect that is moved by the given offset
        # => type(dis_pos)=rect
        self.rect = self.image.get_rect().move(pos)


    def getImage(self):
        '''As the robot moves it changes directions. The image of the
    robot changes accordingly to reflect this changes.
    '''
        if self.direction == 'N':
            self.image = pygame.image.load("../pygameImg/N.png")
        elif self.direction == 'E':
            self.image = pygame.image.load("../pygameImg/E.png")
        elif self.direction == 'S':
            self.image = pygame.image.load("../pygameImg/S.png")
        else:
            # 'W'
            self.image = pygame.image.load("../pygameImg/W.png")
```

```
################################################################################
#
# The robot.
#
# Written by Marte K Skadsem, 2005/2006
#
#
################################################################################

# python modules
import sys, os, threading, time
from xmlrpclib import ServerProxy

# own written modules
import common
from robotERintf import RobotERIntf
from robotMapping import RobotMapping
from navigation import Navigation

class Robot:
    def __init__(self, connected):
        # connect to robot control center (RCC)
        if connected:
            self.er1 = RobotERIntf('localhost', 9000)
        else: self.er1 = False

        self.addr = common.ROBOT_ADDR  # robot's ipaddress
        self.tag = 0                   # robot's tag/id
        self.my_pos = (0,0)            # robot's pos

        # start navigation module
        self.driving_control = Navigation(self.er1)

        # some global variables used to stop running threads
        self.interrupt = False
        self.stopped = True

        # list of modules imported during runtime
        self.new_modules = {}

        #infrastructure interface
        self.infra_intf = False


    def main(self):
        '''Downloads interface from infrastructure, registers at
    the infrastructure, gets map information and makes a map,
    starts the server, registers pre-defined tasks, and ends
    in a while loop polling the infrastructure for new tasks.
    '''
        # 1) init phase
        self.initphase()

        # 2) call welcome to get map and stuff
        reply = self.infra_intf.callWelcome(self.addr)
        tag, rest = reply[0], reply[1]
        map_piece = rest[0][0], rest[0][1]
        map_size = rest[1][0], rest[1][1]
        rob_pos = rest[2][0], rest[2][1]
        data= rest[3][0], rest[3][1]
        self.tag = tag
        self.my_pos = rob_pos

        # 3) make map
        self.map = RobotMapping(map_piece, map_size, data)

        # 4) set up server thread
        self.server_thread = threading.Thread(target=self.infra_intf.serve)
        self.server_thread.setDaemon(True)
```



```
        self.server_thread.start()

        # 5) send a list of all possible tasks that tm can call
        self.sendTaskOptions()

        # 6) main loop
        print 'Going in while loop'
        while True:
            if self.stopped:
                if not self.interrupt:
                    task = self.infra_intf.requestNewTask(self.tag)
                    if task:
                        self.task = task
                        print 'Received task from request'
                        self.doTask()
            time.sleep(5)


    def initphase(self):
        '''Connects to the infrastructure and downloads a file
    containing the interface to use in communication with it.
    The file is a python module. This is dynamically imported,
    and the interface is initiated.
    '''
        infrastru_addr = common.INFRASTRU_ADDR
        infrastru_port = common.INFRASTRU_PORT

        infrastructure = ServerProxy('http://' + infrastru_addr + \
                                     ':' + str(infrastru_port))

        filename, obj = infrastructure.initphase()

        new_file = open(filename, 'w')
        new_file.write(obj.data)
        new_file.close()

        module_name = filename.split('.')[0]
        infra_intf = __import__(module_name)

        self.infra_intf = infra_intf.RobotInfrastructureIntf(self,
                                                              infrastru_addr,
                                                              infrastru_port,
                                                              common.SERVE_PORT)


    def sendTaskOptions(self):
        '''Sends a descriotion of all predefined tasks to
    infrastructure.
    '''
        possible_tasks = "goTo( (x,y) )       - go to point (x,y)\n" +\
                         "followPath( path )  - path is a list of\n" +\
                         "                      at least one point of\n" +\
                         "                      the type (x,y)\n" +\
                         "examineArea( (x,y) ) - go to (x,y) and take\n" +\
                         "                      a video of the area\n"

        self.infra_intf.taskOptions(self.tag, possible_tasks)


#-----------------------------------------------
# SERVER FUNCTIONS (called by infrastructure)
#-----------------------------------------------
    def executeTask(self, task):
        '''Starts a task thread. When this is called, we know that
    the robot is idle because the infrastructure stoped it first.
    '''
        print 'Got a task from user.'
        self.task = task
```

```
        self.interrupt = False
        self.driving_control.interrupt = False

        task_thread = threading.Thread(target=self.doTask)
        task_thread.setDaemon(True)
        task_thread.start()

        return True


    def runCode(self, module_name):
        '''Imports and starts execution of given code in a thread.
    The file is stored by the interface. Returns when a
    "ready"-flag is raised.
    '''
        print 'Preapare code execution'

        module_name = module_name.split('.')[0]

        # import module
        self.importModule(module_name)

        module = self.new_modules[module_name]

        ready = []      # the ready-flag
        code_thread = threading.Thread(target=module.init,args=(self,ready))
        code_thread.setDaemon(True)
        code_thread.start()

        # wait until flag is raised
        while len(ready) == 0:
            time.sleep(0.5)

        return True


    def importModule(self, module_name):
        '''Imports a new module. It is stored in a dictionary
    where key = module name and value = module, so that it
    can be accessed afterwards.
    '''
        print "import module: " + module_name

        mod = __import__(module_name)
        self.new_modules[module_name] = mod

        return True


    def getMonitorData(self):
        '''Returns information that can be shown in monitoring of
    the robot.
    '''
        data = "The goal is that this would be a stream of sensor data," +\
               " i.e. video stream. Have not thought on how to do it."
        return data


    def uppdateMap(self, list_of_objects):
        '''Gets a list of new objects and obstacles discovered in
    the work space. Uppdates map accordingly.
    '''
        self.map.uppdateMap(list_of_objects)
        return True


    def stop(self):
        '''Stops the movement of the robot and interrupts task
    execution.
```



```
    '''
        print "Got STOP order"

        if self.er1: self.er1.stop()
        self.stopped = True

        self.interrupt = True
        self.driving_control.interrupt = True

        return True


#-------------------------------
# TASK EXECUTION FUNCTIONS
#-------------------------------
    def doTask(self):
        '''Executes the new task and starts the task-kill thread.
    Cleans up after execution.
    '''
        print 'Execute task ' + str(self.task)

        self.stopped = False

        # start taskkill thread
        taskkill_thread = threading.Thread(target=self.taskKill)
        taskkill_thread.setDaemon(True)
        taskkill_thread.start()

        # check what task to do
        if self.task[0] == "goTo":
            msg = self.goTo( (self.task[1][0],self.task[1][1]) )
        elif self.task[0] == "followPath":
            msg = self.followPath(self.task[1])
        elif self.task[0] == "examineArea":
            msg = self.examineArea( (self.task[1][0],self.task[1][1]) )
        else:
            msg = "do not support this task"

        # report that task is done
        if not self.interrupt:
            self.infra_intf.done(self.tag, self.task, msg)
            self.interrupt = True   # set in order to stop taskkill thread

        # wait until thread is stopped
        taskkill_thread.join()

        # make ready for new task
        self.interrupt = False
        self.driving_control.interrupt = False
        self.stopped = True


    def taskKill(self):
        '''The task-kill thread. Checks for interrupt signal. When
    received, stops task execution at the driving control.
    '''
        while not self.interrupt:
            time.sleep(0.3)

        self.driving_control.interrupt = True


# -------------------------------------
    def goTo(self, point):
        '''The goTo-task. Goes to the point given.
    '''
        print 'Going to: ' + str(point)

        # report position
```

```
        direction = self.driving_control.direction
        self.my_pos = self.infra_intf.reportPosition(self.tag,
                                                     self.my_pos,
                                                     direction)
        self.my_pos = self.my_pos[0], self.my_pos[1]

        # find a path between current position and goal
        path = self.map.findPath(self.my_pos, point)

        if not path:
            return 'done'
        if path == 'unwalkable':
            return 'Cannot go there. Path is unwalkable.'
        elif path == 'nopath':
            return 'No path exists between ' + str(self.my_pos) +\
                   ' and ' + str(point) + '. Cannot go there.'

        # follow path
        msg = self.driving_control.followPath(path)

        direction = self.driving_control.direction
        if msg == 'done':
            # all went well, report position, which is at goal point
            self.my_pos = self.infra_intf.reportPosition(self.tag,
                                                         point,
                                                         direction)
            self.my_pos = self.my_pos[0], self.my_pos[1]
        elif type(msg) == tuple :
            # got interrupted, report current point
            self.my_pos = self.infra_intf.reportPosition(self.tag,
                                                         msg,
                                                         direction)
            self.my_pos = self.my_pos[0], self.my_pos[1]

        print 'Return message: ' + str(msg)
        return msg


    def followPath(self, _path):
        '''The followPath-task. Follows a user given path.
'''
        print 'Path to follow: ' + str(_path)

        # report position
        direction = self.driving_control.direction
        self.my_pos = self.infra_intf.reportPosition(self.tag,
                                                     self.my_pos,
                                                     direction)
        self.my_pos = self.my_pos[0], self.my_pos[1]

        # translates path into a list of tuples
        _path = self.makePath(_path)

        path = [self.my_pos]
        path.extend(_path)
        for i in range( len(path) - 1 ):
            # for each step in path
            if self.interrupt: return

            # finds path between current step and next step in path
            part = self.map.findPath(path[i], path[i+1])

            if part:
                if part == 'unwalkable':
                    return 'Cannot go there. Path is unwalkable.'
                elif part == 'nopath':
                    return 'No path exists between ' + str(path[i]) +\
                           ' and ' + str(path[i+1]) + '. Cannot go there.'
```



```
                # follow path
                msg = self.driving_control.followPath(part)

                direction = self.driving_control.direction
                if msg == 'done':
                    # all went well, report new position
                    self.my_pos = self.infra_intf.reportPosition(self.tag,
                                                                 path[i+1],
                                                                 direction)
                    self.my_pos = self.my_pos[0], self.my_pos[1]
                elif type(msg) == tuple :
                    # got interrupted, report current point
                    self.my_pos = self.infra_intf.reportPosition(self.tag,
                                                                 msg,
                                                                 direction)
                    self.my_pos = self.my_pos[0], self.my_pos[1]
                    return msg
                elif not msg == 'done':
                    return msg

        # finished. report position
        direction = self.driving_control.direction
        self.my_pos = self.infra_intf.reportPosition(self.tag,
                                                     path[len(path)-1],
                                                     direction)
        self.my_pos = self.my_pos[0], self.my_pos[1]

        return msg


    def makePath(self, _path):
        '''The xmlrpclib treats makes lists of tuple arguments. Need
to "translate" the list of lists, that the _path is, to a
list of tuples.
'''
        for i in range( len(_path) ):
            _path[i] = (_path[i][0], _path[i][1])
        return _path


    def examineArea(self, point):
        '''The examineArea-task. Goes to one step before goal point.
Starts a process that takes pictures and makes a  movie.
Concurrently with this process, moves the last part of the
path and turns 360 degrees. Sends the movie to the
infrastructure.
'''
        print 'Examine area: ' + str(point)

        # report position
        direction = self.driving_control.direction
        self.my_pos = self.infra_intf.reportPosition(self.tag,
                                                     self.my_pos,
                                                     direction)
        self.my_pos = self.my_pos[0], self.my_pos[1]

        # find path
        path = self.map.findPath(self.my_pos, point)

        if path == 'unwalkable':
            return 'Cannot go there. Path is unwalkable.'
        elif path == 'nopath':
            return 'No path exists between ' + str(self.my_pos) +\
                   ' and ' + str(point) + '. Cannot go there.'

        last_part = [path[len(path)-2],path.pop()]

        # follow path to one step before goal point
        msg = self.driving_control.followPath(path)
```

```
        direction = self.driving_control.direction
        if msg == 'done':
            # all went well, report new position
            self.my_pos = self.infra_intf.reportPosition(self.tag,
                                                         path[len(path)-1],
                                                         direction)
            self.my_pos = self.my_pos[0], self.my_pos[1]
        elif type(msg) == tuple :
            # got interrupted, report current point
            self.my_pos = self.infra_intf.reportPosition(self.tag,
                                                         msg,
                                                         direction)
            self.my_pos = self.my_pos[0], self.my_pos[1]
            return msg
        else:
            return msg

        if self.interrupt: return

        # thread in which the robot drives the last part of the path
        video_thread = threading.Thread(target=self.driving_control.driveNturn,
                                        args=(last_part,))
        video_thread.setDaemon(True)
        video_thread.start()

        # start taking pictures
        print 'Start picture taking process'
        imageid = str(point[0]) + '_' + str(point[1]) + '_'

        os.system("python "+common.PATH+"video.py "+imageid)

        if self.interrupt: return

        # join the two threads
        video_thread.join()

        # report position
        self.my_pos = last_part[1]
        direction = self.driving_control.direction
        self.my_pos = self.infra_intf.reportPosition(self.tag,
                                                     self.my_pos,
                                                     direction)
        self.my_pos = self.my_pos[0], self.my_pos[1]

        # make movie and delete pictures
        print 'Make movie'
        filename = self.camera.makeMovie(common.FFMPEG, common.INFILES_DIR,
                                         imageid, common.OUTFILE_DIR)
        os.system('del ' + common.INFILES_DIR + '*.jpg ')

        # stream movie file to task manager

        self.infra_intf.streamFile(self.tag, common.INFILES_DIR, imageid)

        os.system('del ' + common.INFILES_DIR + '*.jpg')

        return imageid


if __name__ == '__main__':
    exSrv = Robot(True)
    exSrv.main()
```

```python
######################################################################
#
# The communication with the ER1 (RCC's command line inteface).
# Sends commands to  the robot using telnet.
#
# Written by Marte K Skadsem, spring 2005
# Modifyed by Marte K Skadsem, autumn 2005
#
######################################################################

import telnetlib
import sys
import time

class RobotERIntf:
    def __init__(self, host, port):
        '''Establish the telnet connection
   '''
        self.host = host
        self.port = port
        print 'connecting to RCM'
        self.connection = telnetlib.Telnet(self.host, self.port)

        #check if connection is okay
        self.connection.write('\n')
        self.recAck(1,2)


    #------- move commands ----------------------------#
    def move(self, cmd):
        '''Moves the robot in specified direction
   '''
        self.connection.write('move ' + cmd + '\n')
        ack = 1
        self.recAck(ack,2)


    def rotateToward(self, what, args):
        '''The robot rotates toward what is specifyed (can
   be object or color)
   '''
        self.connection.write('move rotate toward ' + \
                              what + ' ' + args + '\n')
        self.recAck(1,2)


    def driveToward(self, what, args):
        '''The robot drives toward what is specifyed (can
   be object or color)
   '''
        self.connection.write('move drive toward ' + \
                              what + ' ' + args + '\n')
        self.recAck(1,2)


    #------- play command -----------------------------#
    def playPhrase(self, phrase):
        '''Robots says the specified phrase
   '''
        self.connection.write('play phrase "' + phrase + '"\n')
        self.recAck(1,2)


    #------- stop command -----------------------------#
    def stop(self):
        '''Stops any robot motion or soounds which are in progress
   '''
        self.connection.write('stop\n')
        self.recAck(1,2)
```



```python
    #------- sense command ----------------------------#
    def senseOn(self, sensor):
        '''Turns on the specified sensror
   '''
        self.connection.write('sense ' + sensor + '\n')
        self.recAck(1,2)


    def senseOff(self, sensor):
        '''Turns off the specified sensror
   '''
        self.connection.write('sense ' + sensor + ' off\n')
        self.recAck(1,2)


    #------- clear command ----------------------------#
    def clear(self):
        '''Throws away all events which have not yet been sent to user
   '''
        self.connection.write('clear\n')
        self.recAck(1,2)


    #------- events command ---------------------------#
    def eventsOn(self):
        '''Turns on the events-command
   '''
        self.connection.write('events\n')
        self.recAck(1,2)


    def eventsOff(self):
        '''Turns off the events-command
   '''
        self.connection.write('\n')
        self.recAck(1,2)


    #------- set command ------------------------------#
    def set(self, cmd):
        self.connection.write('set ' + cmd + '\n')
        self.recAck(1,2)


    #---- read commands  (NOT INCLUDED IN THE API) ----#
    def waitFor(self, cmd, timeout):
        '''Reads the connection until cmd or timeout appears
   '''
        t_taken1 = time.time()
        reply = self.connection.read_until(cmd, timeout)
        t_taken2 = time.time()

        if t_taken2 - t_taken1 >= timeout:
            return 'timeout'

        return reply


    def recAck(self, acks, time):
        '''Receive specifyed number of acks
   '''
        while not acks == 0:
            acks -= 1
            ok = self.connection.read_until('OK\r\n', time)
            if ok == '':
                #did nor receive an OK, try again
                ok = self.connection.read_until('OK\r\n', time)
```

```
                if ok == '':
                    print 'robot: Did not receive an OK'
                    self.connection.close()
                    sys.exit(0)
            elif ok.__contains__('Error'):
                #some error occured
                print 'Error message: ' + str(ok.split('\r\n'))
                self.connection.close()
                sys.exit(0)


    def readConnection(self):
        '''Reads the connection.
Returns whatever was read.
'''
        reply = self.connection.read_until('\r\n', 5)
        # reply is a string
        return reply
```

```
################################################################################
#
# The map module. Makes and maintains the robot's map and finds paths in it.
# Gets information about the map at start up. Uses the a searching
# algorithm to find paths. In this case, the A* algorithm.
#
# Written by Marte K Skadsem, 2005/2006
#
################################################################################

import time

from astar import Astar

class RobotMapping:
    def __init__(self, map_piece, map_size, data):
        '''Initializes the map. The parameters are:
    map_piece = the top left coordinate in the map (inicates where
    map piece starts)
    map_size = size of map_piece
    data = two lists. One with knownj object and one with overlaping
    areas.
    '''
        self.map_piece = map_piece
        self.lenx, self.leny = map_size
        self.known_obj, self.overlap = data

        self.map = self.initMap(self.lenx, self.leny)

        if len(self.map[0]) > 100:
            self.high_level_map = self.splitMap()
        else: self.high_level_map = False

        self.search = Astar()

        self.free = True    # a lock to prevent more than one
                            # thread access the map


    def initMap(self, x, y):
        '''Makes the map and fills it with information.
    '''
        new_map = self.makeMap(x,y)

        map_p_x, map_p_y = self.map_piece
        for o in self.known_obj:
            o_x , o_y = o
            new_map[o_y - map_p_y][o_x - map_p_x] = 'w'

        for o in self.overlap:
            o_x , o_y = o
            new_map[o_y - map_p_y][o_x - map_p_x] = 's'

        return new_map


    def makeMap(self, x, y):
        '''Makes a map.
    '''
        new_map = []
        for i in range(y):
            new_map.append([])
            for j in range(x):
                new_map[i].append('')
        return new_map


    def splitMap(self):
        '''Makes a high level map out of the originally map.
```



```
    Done to make searches over big areas faster.
    '''
        high_level_map = self.makeMap(10, 10)
        self.map_factor = len(self.map[0]) / 10

        for o in self.known_obj:
            o_x , o_y = o
            high_level_map[o_y/self.map_factor][o_x/self.map_factor] = 'w'

        return high_level_map


    def findPath(self, start, stop):
        '''Finds a path between start and stop. If a high level
    map exists and the distance bewteen start and stop is big,
    searches the hig level map first.
    '''
        if start == stop:
            return  False

        while not self.free:
            time.sleep(0.5)

        self.free = False

        path = []

        if self.high_level_map and self.distance(start, stop) > 100:
            # searchng high level

            # empty earlier searching data from map
            self.search.emptyLists(self.map)

            hl_start = start[0]/self.map_factor, start[1]/self.map_factor
            hl_stop = stop[0]/self.map_factor, stop[1]/self.map_factor

            path_hl = self.search.aStar(hl_start, hl_stop,
                                        self.high_level_map, 1)

            if path_hl == 'nopath':
                path_hl = self.search.findClosest(hl_start,
                                                  self.high_level_map)

            # translate into normal level coordinates
            path_hl = self.resolvePath(path_hl)

            # empty earlier searching data from map
            self.search.emptyLists(self.high_level_map)

            if not path_hl[0] == start:
                path = self.search.aStar(start, path_hl[1], self.map, 1)

                path.pop()
                path_hl.remove(path_hl[0])
                path.extend(path_hl)
            else: path = path_hl
            start = path[len(path)-1]

        # empty earlier searching data from map
        self.search.emptyLists(self.map)

        # search normal level
        part = self.search.aStar(start, stop, self.map, 1)

        self.free = True

        if not path == []:
            path.pop()
            path.extend(part)
```

```
        else: path = part

        return path


    def distance(self, start, stop):
        '''Roughly estimated distance bewteen start and stop
'''
        if stop[0] > start[0]: dist = stop[0] - start[0]
        else: dist = start[0] - stop[0]

        if stop[1] > start[1]: dist2 = stop[1] - start[1]
        else: dist2 = start[1] - stop[1]

        return max(dist, dist2)


    def resolvePath(self, path):
        '''Translates a path with high level coordinates to
a path with normal level coordinates.
'''
        _path = []
        for step in path:
            _path.append( (step[0]*self.map_factor, step[1]*self.map_factor) )

        return _path


    def uppdateMap(self, list_of_objects):
        '''Uppdates the map with new information.
'''
        while not self.free:
            time.sleep(0.5)

        self.free = False

        for o in self.known_obj:
            o_x , o_y = o
            new_map[map_p_y - o_y][map_p_x - o_x] = ''

        self.known_obj = list_of_objects
        for o in self.known_obj:
            o_x , o_y = o
            new_map[map_p_y - o_y][map_p_x - o_x] = 'w'

        self.free = True
        return True
```

```
################################################################################
#
# The A* algorithm.
# The opened-list is implemented with a binary heap.
#
# Written by Marte K Skadsem, autum 2005
#
################################################################################

import heapq

#-------------------------------------
# Objects of the Node class is used
# in the heap.
#-------------------------------------
class Node:
    def __init__(self, name, parent, g, h, t):
        self.name = name
        self.parent = parent
        self.g = g      #movement cost to get to this node from startpoint
        self.h = h      #estimated movement cost from this node to the endpoint
        self.f = g + h  #score of the node
        self.time = t   #used to deside which node is newest
        self.list = 'open'  #the list this node belongs to

    def __cmp__(self, y):
        '''This function will override compare() when
    two nodes are compared. We want the node that was
    added last in the heap (the open list) to be on
    top of the others with the same f-value
    '''
        if self.f < y.f:
            return -1
        if self.f == y.f and self.time > y.time:
            return -1
        return 1


    def printInfo(self):
        '''Used to print all the information of a node.
    '''
        print self.name, self.parent, self.f, self.g, self. h


#-------------------------------------
# The class tha implements the A*
# algorithm.
#-------------------------------------
class Astar:
    def __init__(self):
        self.open_list = []
        self.closed_list = []
        self.neighbours = [(1,0),(0,1),(-1,0),(0,-1)]


    def aStar(self, start, stop, robo_map, map_factor):
        '''The main function.
    start - startpoint
    end - endpoint
    robo_map - memory reference to the map to be used
    map_factor - denotes if we are woriking on low lwvwl map or
    high level map
    Returns path if found else nopath or unwalkable.
    '''
        if map_factor == 1:
            if not self.checkWakable(start, stop, robo_map):
                return 'unwalkable'
        g = 10
        time = 0
```



```
        current = Node(start,0,0,0,0)

        while not current.name == stop:
            time += 1
            current_x, current_y = current.name

            self.switchToClosedList(current,robo_map)

            for n in self.neighbours:
                #for all neighbours of current
                n_x = n[0] + current_x
                n_y = n[1] + current_y

                if self.neighbourInsideMap(n_x, n_y, robo_map):
                    #if neighbour is inside map:

                    if robo_map[n_y][n_x].__class__ == Node:
                        if robo_map[n_y][n_x].list == 'open':
                            #check if shorter path
                            tmp = robo_map[n_y][n_x]
                            if tmp.g > current.g + g:
                                #change parent to the neighbour
                                tmp.parent = current.name
                                #recalculate g anf f
                                tmp.g = current.g + g
                                tmp.f = tmp.g + tmp.h

                    elif robo_map[n_y][n_x] == '':
                        #add neighbour to open_list
                        h = self.findH( (n_x,n_y), stop ) * g
                        new = Node((n_x,n_y),
                                   current.name,
                                   current.g+g,
                                   h,
                                   time)
                        heapq.heappush(self.open_list, new)
                        robo_map[n_y][n_x] = new

            if len(self.open_list) == 0:
                return 'nopath'

            current = heapq.heappop(self.open_list)

        self.switchToClosedList(current,robo_map)
        return self.savePath(start, current, robo_map)


    def emptyLists(self, robo_map):
        '''Need to clean the map for old
    data before used again.
    '''
        for i in self.open_list:
            if not type(robo_map[i.name[1]][i.name[0]]) == str or not type(robo_
map[i.name[1]][i.name[0]]) == str:
                robo_map[i.name[1]][i.name[0]] = ''

        for i in self.closed_list:
            if not type(robo_map[i.name[1]][i.name[0]]) == str or not str(robo_m
ap[i.name[1]][i.name[0]]) == str:
                robo_map[i.name[1]][i.name[0]] = ''

        self.open_list = []
        self.closed_list  = []


    def checkWakable(self, start, stop, robo_map):
        '''Cheks if there is an object at the start
    position or end position.
    '''
```

```
        if robo_map[start[1]][start[0]] == 'w':
            return False
        if robo_map[stop[1]][stop[0]] == 'w':
            return False
        if robo_map[stop[1]][stop[0]] == 's':
            return False
        return True


    def switchToClosedList(self, current, robo_map):
        '''Switches the current node from the opened list
to the closed list.
'''
        self.closed_list.append(current)

        if type(robo_map[current.name[1]][current.name[0]]) == str:
            robo_map[current.name[1]][current.name[0]] = current

        robo_map[current.name[1]][current.name[0]].list = 'closed'


    def neighbourInsideMap(self, n_x, n_y, robo_map):
        '''Checks that n_x and n_y are inside the map.
'''
        if n_x >= 0 and n_y >= 0 :
            if n_x < len(robo_map[0]) and n_y < len(robo_map):
                return True

        return False


    def findH(self, here, stop):
        '''Finds the h-value for the here-node.
'''
        x,y = here
        endx, endy = stop
        n = 0

        if endy >= y: n = endy - y
        else: n = y - endy

        if endx >= x: n = n + (endx - x)
        else: n = n + (x - endx)

        return n


    def savePath(self, start, this, robo_map, txt=None):
        '''Returns the path from the this-node to the sart
position.
'''
        path = []
        while not this.name == start:
            path.append(this.name)
            parent = this.parent
            this = robo_map[parent[1]][parent[0]]

        path.append(this.name)
        path.reverse()

        if txt: path.append(txt)

        return path


    def findClosest(self, start, robo_map):
        '''Returns the path from the start position to
the node with the lowest h-value <=> the node assumingly
closest to the end position.
```



```
'''
        h = self.closed_list[1].h
        pos = 0
        for i in range(1, len(self.closed_list)-1):
            if self.closed_list[i].h <= h:
                h = self.closed_list[i].h
                pos = i
        return self.savePath(start, self.closed_list[pos], robo_map)


    def printMap(self, robo_map):
        '''Prints the map
'''
        print '__________________'
        for p in robo_map:
            cmd = '|'
            for q in p:
                if type(q) == str:
                    if q == '':
                        cmd += '-|'
                    else:
                        cmd = cmd + q + '|'
                else:
                    cmd = cmd + q.list + '|'
            print cmd
            print '---------------------'


    def printList(self, which):
        '''Prints the given list.
'''
        if which == 'open':
            print 'Opened List:'
            for o in self.open_list:
                o.printInfo()
        else:
            print 'Closed List:'
            for o in self.closed_list:
                o.printInfo()
```

```
###############################################################################
#
# The navigation module.
# Contains code for following a given path
#
# Written by Marte K Skadsem, 2005/2006
#
###############################################################################
import time, sys

import common

class Navigation:
    def __init__(self, er1):
        # the er1 is the robotER1Interface module
        self.er1 = er1

        # some driving relevant information
        self.fw = common.FW                     # denotes if robot's motors are
                                                # turned back-forward
        self.direction = common.DIRECTION       # robot's heading
        self.nittideg = common.NITTIDEG         # denotes how many degrees makes
                                                # robot turn 90
        self.map = None
        self.interrupt = False                  # set if robot is interrupted


    def followPath(self, path):
        '''Follows a given path.
    The parameter path is a list of points starting with the current
    position.
    '''
        idx = 0
        while idx < len(path)-1:
            if self.interrupt: return path[idx]

            if not self.validate(path[idx], path[idx+1]):
                msg = 'Error! tries to go from ' + str(path[idx]) + \
                      ' to ' + str(path[idx+1])
                return msg

            # turn
            self.turn( self.findHeading(path[idx], path[idx+1]) )

            if self.interrupt: return path[idx]

            # find distance to move
            dist, idx = self.findDist(path, idx)

            # move robot
            self.move(dist)

        if self.interrupt: return path[idx]

        # all went well
        msg = 'done'
        return msg


    def validate(self, this, next):
        '''Validates that start and end points in the
    path are not the same
    '''
        if this == next:
            return True
        elif this[0] == next[0] or this[1] == next[1]:
            return True
        else:
            return False
```



```
    def findHeading(self, this, next):
        '''Returns which heading to take next
    '''
        myx, myy = this
        nextx, nexty = next
        if nextx > myx and nexty == myy:
            return 'E'
        elif nextx < myx and nexty == myy:
            return 'W'
        elif nexty > myy and nextx == myx:
            return 'S'
        elif nexty < myy and nextx == myx:
            return 'N'
        else:
            return self.direction


    def turn(self, heading):
        '''Truns the robot in right direction.
    '''
        # find how much to turn
        dirs = ['N', 'E', 'S', 'W']
        deg = dirs.index(heading) - dirs.index(self.direction)

        if deg == 1 or deg == -3:
            cmd = '-' + str(self.nittideg)
        elif deg == -1 or deg == 3:
            cmd = str(self.nittideg)
        elif deg == 2 or deg == -2:
            cmd = str(self.nittideg * 2)
        else:
            return
        self.direction = heading

        cmd = cmd + ' degrees'

        if self.interrupt: return

        if self.er1:
            # turn
            self.er1.move(cmd)
            self.er1.eventsOn()
            while True:
                reply = self.er1.waitFor('move done\r\n', 3)
                if reply.__contains__('move done'):
                    break


    def findDist(self, path, idx):
        '''Finds distance to move in same direction
    '''
        dist = 0
        x, y = path[idx]
        idx += 1
        nextx, nexty = path[idx]
        try:
            if nextx == x:
                # moving North-South
                while nextx == x:
                    if self.direction == 'N':
                        dist = dist + (y-nexty)
                    else:
                        dist = dist + (nexty-y)
                    x, y = path[idx]
                    idx += 1
                    nextx, nexty = path[idx]
            else:
```

```
            # moving East-West
            while nexty == y:
                if self.direction == 'W':
                    dist = dist + (x-nextx)
                else:
                    dist = dist + (nextx-x)
                x, y = path[idx]
                idx += 1
                nextx, nexty = path[idx]
        except IndexError:
            idx -= 1
            return (dist, idx)

        idx -= 1
        return (dist, idx)


    def move(self, dist):
        '''Moves the robot the spesifyed distance forward.
Returns when move is done
'''
        if self.interrupt: return

        if self.er1:
            self.er1.move(self.fw + str(dist) + ' cm')
            self.er1.eventsOn()
            while True:
                reply = self.er1.waitFor('move done\r\n', 3)
                if reply.__contains__('move done'):
                    break


    def driveNturn(self, last_part):
        '''Used in the examina area task. Dirves the last part
of path forward and makes a 360 degree turn.
'''
        self.followPath(last_part)

        if self.interrupt: return

        if self.er1:
            self.er1.move('360 d')
            self.er1.eventsOn()
            while True:
                reply = self.er1.waitFor('move done\r\n', 3)
                if reply.__contains__('move done'):
                    break
```

```
################################################################################
#
# The picture taking and video making process.
#
# OBS! Uses the VideoCapture module. Make sure it is installed.
# Can only run on Windows.
#
# Written by Marte K Skadsem, 2005/2006
#
################################################################################

import time, string, os, sys
from VideoCapture import Device

import common

class VideoMaking:
    def __init__(self):
        '''If you get horizontal stripes or other errors in the captured
    picture (especially at high resolutions), try setting
    showVideoWindow=0.
   '''
        self.cam = Device(devnum=0, showVideoWindow=1)
        #self.cam = Device(devnum=1, showVideoWindow=1)


    def main(self, imageid):
        '''The parameter imageid is the name to identify
    the pictures taken.
   '''
        self.takePictures(common.PIC_IN_SEC,
                          common.DURATION,
                          common.IMAGEDIR,
                          imageid)


    def takePictures(self, pic_in_sec, duration, savedir, imgid):
        '''Takes pictures. The pictures are saved as jpeg files.
    The parameters means:
    pic_in_sec = how many pictures to take in a second
    duration = how long time to take pictures (in sec)
    savedir = where to save pictures
    imgid = identifyer for the pictures taken
   '''
        # Specify the amount of seconds to wait between individual captures.
        sec_betw_cap = 1.0 / pic_in_sec

        num_pic = 0

        starttime = time.time()
        elapsedtime = 0
        while elapsedtime < duration:
            # take a picture and store it
            self.cam.saveSnapshot(savedir + imgid + \
                                  string.zfill(str(num_pic), 4) + '.jpg',
                                  timestamp=3, boldfont=1)
            num_pic += 1
            time.sleep(sec_betw_cap)
            elapsedtime = time.time() - starttime


    def makeMovie(self, ffmpeg, infiles_dir, imageid, outfile_dir):
        '''Uses ffmpeg to merge jpeg files to a mp4 file.
   '''
        os.system(ffmpeg + "-r 5 -i " + \
                  infiles_dir + imageid + "%04d.jpg " + \
                  outfile_dir + imageid + "movie.mp4")
        return outfile_dir + imageid + "movie.mp4"
```



```
if __name__ == '__main__':
    exSrv = VideoMaking()
    exSrv.main(sys.argv[1])
```

```
################################################################################
#
# This file contains variables used by robot.py, video.py and
# navigation.py
#
# Written by Marte Karidatter Skadsem 2005/2006
################################################################################

# robot's ipaddr
ROBOT_ADDR = '129.242.18.166'

# address and port to infrastructure
INFRASTRU_ADDR = '129.242.19.46'
INFRASTRU_PORT = 8081

# port to robot's simpleXMLRPCServer
SERVE_PORT = 8090

# paths for storing images and videos
PATH = "~\\therobot\\"
IMAGEDIR = PATH+"images\\"
INFILES_DIR = PATH+"images\\"
OUTFILE_DIR = PATH+"images\\"

FFMPEG = "~\\FFmpeg\\ffmpeg.exe "

# data for picture taking
PIC_IN_SEC = 24
DURATION = 20

# some driving relevant information
FW = '-'
DIRECTION = 'N'
NITTIDEG = 90
```

```
#################################################################################
#
# This is the robot code that uses the ulstrasound sensors.
# The robot stays in a distance bewteen 30 and 60 cm from the wall on the
# right hand side.
#
# We stopped using the fourth sensor, the one pointing forward and downward,
# because of many error readings.
#
# Written by Marte Karidatter Skadsem, spring 2006
#
#################################################################################
import threading, time, Queue, sys

from ultradist import Ultradist
from robotERintf import RobotERIntf

class UltraRobot:
    def __init__(self, connected):
        # connect to robot control center
        self.connected_robot = connected
        if self.connected_robot:
            self.er1 = RobotERIntf('127.0.0.1', 9000)
        else: self.er1 = False

        # some driving relevant information
        self.fw = False             # robot's motors are turned back-forward
        self.direction = 'N'        # robot's heading

        # init ultrasound sensors
        self.sensorQ = Queue.Queue()
        self.flagQ = Queue.Queue()
        self.ultraSense = Ultradist(self.sensorQ, self.flagQ)
        self.falseAlarm = False


    def main(self):
        # set the robot's speed
        self.er1.set('v 10')

        # how many sensors are used
        ports = self.ultraSense.ports

        senslist = [(0,0)]
        dist3 = 0

        # start ultradist thread (starts the sensors)
        self.udthread = threading.Thread(target=self.ultraSense.main)
        self.udthread.setDaemon(True)
        self.udthread.start()

        # read first readings from all sensors
        # this reading is almost allways wrong, so we ignore them
        for i in range(1,ports+1):
            self.flagQ.put(i)
            print self.sensorQ.get()
            senslist.append(time.time())

        # start driving forward
        self.er1.move('-1000 cm')
        self.moving = True
        self.stairs = False

        # check sensors
        s = 1
        while True:
            t = time.time()
#            if (t - senslist[4]) > 1.0:
#                s = 4
```



```
#                senslist[4] = t
            if (t - senslist[1]) > 6.0:
                s = 1
                senslist[1] = t

            self.flagQ.put(s)

            # read sensor
            sensor, dist = self.sensorQ.get()
            print sensor, dist

#            if sensor == 4:
#                self.checkS4( (sensor,dist) )

            if sensor == 1:
                self.checkS1( (sensor, dist) )

            elif sensor == 3:
                if dist <= 30:
                    print 'turn right'
                    self.moving = False
                    self.er1.move('-20 d')
                    self.er1.waitFor('move done\r\n', 60)

            else:
                # sensor == 2
#                if self.stairs:
#                    if dist <= 61:
#                        self.stairs = False
                if dist <= 30:
                    print 'turn left'
                    self.moving = False
                    self.er1.move('20 d')
                    self.er1.waitFor('move done\r\n', 60)
                elif dist >= 62:
                    if dist > 100:
                        # first reading - cannot be a corner
                        if not dist3 == 0:
                            if max(dist,dist3) - min(dist,dist3) > 100:
                                self.turning(sensor,dist3)
                    else:
                        print 'adjust alignment'
                        self.moving = False
                        self.er1.move('-20 d')
                        self.er1.waitFor('move done\r\n', 60)
                dist3 = dist

            if not self.moving:
                self.er1.move('-1000 cm')
                self.moving = True

            s += 1
#            if s == 5: s = 1
            if s == 4: s = 1

    def turning(self, sensor, dist):
        '''Turns to the right to turn a corner
    '''
        print 'turn corner'
        self.er1.eventsOff()
        self.er1.move('-30 cm')
        t1 = time.time()
        while True:
#            self.flagQ.put(4)
#            if not self.checkS4(self.sensorQ.get()):
                # stairs ahead
#                return
            self.flagQ.put(1)
            if not self.checkS1(self.sensorQ.get()):
```

```
                # wall ahead
                return
            t2 = time.time()
            if t2 − t1 >= 3.0:
                break

        self.er1.move('−90 d')
        self.er1.waitFor('move done\r\n', 30)

        length = dist + 30
        delay = length / 10
        t1 = time.time()
        self.er1.move('−' + str(length) + ' cm')
        while True:
#            self.flagQ.put(4)
#            if not self.checkS4(self.sensorQ.get()):
                # stairs ahead
#                return
            self.flagQ.put(1)
            if not self.checkS1(self.sensorQ.get()):
                # wall ahead
                return
            t2 = time.time()
            if t2 − t1 >= delay:
                break
        self.moving = False
        return


    def checkS1(self, sensorData):
        '''Check if there is something in front of the robot.
    If so, backs and turn left to follow the wall in front.
    '''
        dist = sensorData[1]
        if dist < 51:
            print 'Crash!!'
            length = str(50 − dist)
            self.er1.move(length + ' cm')
            self.er1.waitFor('move done\r\n', 60)
            self.er1.move('90 d')
            self.er1.waitFor('move done\r\n', 60)

            self.moving = False
            return False

        return True


    def checkS4(self, sensorData):
        '''Check if robot is close to stairs
    '''
        dist = sensorData[1]
        if dist > 90:
            if self.falseAlarm:
                print 'stairs!'
                self.er1.move('35 cm')
                self.er1.waitFor('move done\r\n', 60)
                self.er1.move('90 d')
                self.er1.waitFor('move done\r\n', 60)

                self.stairs = True
                print 'self.stairs'
                self.moving = False
                return False
            else:
                print 'possible false alarm'
                self.flaseAlarm = True
                self.flagQ.put(4)
                self.checkS4(self.sensorQ.get())
```



```
        elif dist > 35:
            print 'stairs!'
            self.er1.move('35 cm')
            self.er1.waitFor('move done\r\n', 60)
            self.er1.move('90 d')
            self.er1.waitFor('move done\r\n', 60)

            self.stairs = True
            print 'self.stairs'
            self.moving = False
            return False

        return True



if __name__ == '__main__':
    exSrv = UltraRobot(True)
    exSrv.main()
```

```
################################################################################
#
# This program measures distance by using the Velleman K8055 PIC USB
# test kit attached to BASIC stamp 2 Devantech SRF04 ultrasonic Range
# Finder #28015
#
# KAJ feb 2006
# Ultrasound distance usb Velleman K8055 hack
# Thanks to
# Bob Dempsey
# bdempsey_64@msn.com
# for supplying me with the k8055dll library for unix
#
# This python code is tranlsated from a originale c code.
#
# Translated to Python by Marte Karidatter Skadsem, feb/march 2006
#
################################################################################

from ctypes import *
import time, Queue

class Ultradist:
    def __init__(self, sensorQ, flagQ):
        '''Adapt the frequency to the 555 timer to reflect distance
    Current 555 oscillating frequency setting is: 1.700 Khz
    Gives 1 sample every 20 cm speed of sound
    This gives distance: 10cm/pulse (back and forth)
    Multiply this number with the number og count in the counter
    '''
        # Define speed of sound
        self.sndspd = 331.46

        # Room temperature higher temperature gives higher speed of sound
        self.roomtemp = 25.0  # In degrees Celcius

        # Set current 555 timer oscillation frequency in HZ
        self.frequency = 1700.0

        # Number of attached sensors, maximum 8
        self.ports = 3

        # Minimum time (in sec) from end of last trig pulse to next
        self.min_delay = 0.28

        # Verbose flag - for debug purpose
        self.verbose = False

        # Number of loops, 0 means infinite
        self.loops = 0

        self.dev = 0x00
        self.str_len = 256

        # Queues used in communication with ultradistRobot.py
        self.sensorQ = sensorQ
        self.flagQ = flagQ


    def my_sleep(self, min_delay):
        '''sleep min_delay sec
    '''
        i = delay = 0
        t1 = time.time()
        while delay < min_delay:
            while not i == 1000:
                i+=1
            delay = time.time() - t1
```



```
    def main(self):
        usec_start = []
        line = ""
        loops = self.loops

        k8055d = windll.k8055D
        fd = k8055d.OpenDevice(c_long(self.dev))

        if not fd == 0:
            print "Couldn't open DEV=" + str(fd) + \
                  " OpenDevice returned " + str(self.dev)

        # Set counter debounce time to 0 ms, max sample rate 2000s/sec
        k8055d.SetCounterDebounceTime(c_int(1), c_long(0))
        k8055d.SetCounterDebounceTime(c_int(2), c_long(0))

        # Init all timers
        for p in range(self.ports):
            usec_start.append(time.time())

        while loops >= 0:
            if not self.loops == 0:  # Don't count down if LOOPS == 0
                loops -= 1

            # Trig all sensors 1-4, 1-front, 2-right, 3-left, 4-down
            #for s in range(1,self.ports+1):
            s = self.flagQ.get()

            # Check that it was at least 10 ms since last trig
            delay = time.time() - usec_start[s-1]
            if self.verbose:
                line = line + "delay port " + str(s) + \
                       " = " + str(delay) + " usec"

            if delay < self.min_delay:
                if self.verbose:
                    line = line + "Sleeping additional " + \
                           str(self.min_delay - delay) + " microseconds"
                self.my_sleep(self.min_delay - delay)

            # Set ultrasound control pin high for next trig pulse

            # Reset counters
            # I do a read of counter registers after ResetCounter()
            # Without they may not reset properly - a k8055dll BUG here?
            k8055d.ResetCounter(c_long(1))
            cnt1 = k8055d.ReadCounter(c_long(1))
            k8055d.ResetCounter(c_long(2))
            cnt2 = k8055d.ReadCounter(c_long(2))

            k8055d.ClearDigitalChannel(c_long(s))

            line = line + "Distance sensor " + str(s) + " = "

            # Wait for response pulse to go high  attached to dig. inp1
            cnt1 = 0
            tmout = 0

            while cnt1 == 0:
                cnt1 = k8055d.ReadCounter(c_long(1))
                self.my_sleep(self.min_delay)
                if self.verbose: line = line + "cnt1=" + str(cnt1) + ","

                # Report timeout, and break loop
                # - may indicate failure in sensor
                tmout += 1
                if tmout > 5:
                    line = line + "Timeout sensor " + str(s)
```

```
                    self.sensorQ.put( (s, 'tmout') )
                    break

            # Here we have either tmout or just reached end of sensor pulse
            # Start timer since we have to wait at least 10 ms for next trig
            usec_start[s-1] = time.time()

            # Read digital counter on port 2
            cnt2 = k8055d.ReadCounter(c_long(2))
            if self.verbose: line = line + "cnt2=" + str(cnt2) + " "

            # On the new version on the PING sensor from Parallax we need to
            # hold the signal output low during response sampling
            k8055d.SetDigitalChannel(c_long(s))

            # Report distance in cm, unless timeout
            if tmout <= 5:
                dist = float(cnt2) * 100 * (self.sndspd+0.6*self.roomtemp)/self.
frequency/2
                line = line + str(dist) + " cm (" + str(cnt2) + " ticks) "
                self.sensorQ.put( (s, dist) )

            # Print a visual bar
            for i in range(cnt2):
                line = line + "#"
            if self.verbose: print line + "\n"
            line = ""

        # Next sensor
        # Next loop

        k8055d.CloseDevice()


if __name__ == '__main__':
    exSrv = Ultradist()
    exSrv.main()
```

```
################################################################################
#
# The user application.
#
# Written by Marte K Skadsem, 2005/2006
#
################################################################################

import sys, threading, time

from xmlrpclib import ServerProxy, Binary

import longstrings

class User:
    def __init__(self):
        self.connected = False   # set when connected and running
        self.userhistory = []    # stores the last user action

    def main(self, infra_addr, infra_port):
        '''Connects to the infrastructure and staarts approproate
    actions based on the last element in user history.
    '''
        try:
            self.infrastructure = ServerProxy('http://' + infra_addr + \
                                  ':' + str(infra_port) )

            self.infrastructure.testConnection()
            self.connected = True
        except Exception, msg:
            print msg
            sys.exit(0)

        self.userhistory.append('welcome')
        print longstrings.USER_WELCOME

        # thread to report when a task is finished
        checkDone_thread = threading.Thread(target=self.checkDone)
        checkDone_thread.setDaemon(True)
        checkDone_thread.start()

        # main loop
        while self.connected:
            if self.userhistory[len(self.userhistory)-1] == 'welcome':
                self.mainChoice()

            elif self.userhistory[len(self.userhistory)-1] == 'monitor':
                self.monitor()

            elif self.userhistory[len(self.userhistory)-1] == 'task':
                self.giveTask()

            elif self.userhistory[len(self.userhistory)-1] == 'code':
                self.sendCode()


    def checkDone(self):
        '''Polls the infrastructure for tasks that are executed.
    Tells user when a task is done.
    '''
        while True:
            result = self.infrastructure.checkDone()
            if result:
                print '\n*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*'
                print 'A robot is done with its task:'
                print result
                print '*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*\n'
            time.sleep(5)
```



```
    def mainChoice(self):
        '''Makes appropriate actions based on user choices
    from main menu.
    '''
        choice = raw_input()
        if not choice.isdigit():
            print longstrings.WRONG_TYPE
            return

        if choice == '1':
            if self.getRobotList():
                print longstrings.MENU
            return

        elif choice == '2':
            self.userhistory.append('monitor')
            self.monitormenu()
            return

        elif choice == '3':
            self.userhistory.append('task')
            self.taskmenu()
            return

        elif choice == '4':
            self.userhistory.append('code')
            self.codemenu()
            return

        elif choice == '5':
            self.stopRobot()
            return

        if choice == '6':
            print 'OK. Quitting...'
            self.userhistory.append('quit')
            self.connected = False
            return
        else:
            #smth wrong
            print longstrings.WRONG_INPUT


    def getRobotList(self):
        '''Prints a list of all robot tags from the infrastructure.
    '''
        data = self.infrastructure.getRobotList()

        if not data:
            print 'No robots registered.\nReturning to main menu.\n'
            print longstrings.MENU
            return False

        print '\nList of all robot tags:'
        for tag in range(len(data) -1):
            print str(data[tag])
        print data[len(data)-1]
        print '----------------------'

        return True


    def monitormenu(self):
        '''Prints the monitor menu
    '''
        if not self.getRobotList():
            # robot list is empty
            self.userhistory.remove('monitor')
```

```
        else:
            print longstrings.MONITOR


    def monitor(self):
        '''Gets the monitor data for the specifyed robot from
    infrastructure and prints it.
    '''
        choice = raw_input()
        if not choice.isdigit():
            print longstrings.WRONG_TYPE
            return

        data = self.infrastructure.monitorRobot(int(choice))
        if not data:
            # if wrong tag is typed three times, return to main menu
            print longstrings.WRONG_TAG
            self.userhistory.append('monitor')
            if self.userhistory.count('monitor') == 4:
                print longstrings.RETURN_MENU
                for i in range(self.userhistory.count('monitor')):
                    self.userhistory.remove('monitor')
            return

        else:
            print data
            print longstrings.MENU

        for i in range(self.userhistory.count('monitor')):
            self.userhistory.remove('monitor')


    def taskmenu(self):
        '''Prints the task menu
    '''
        if not self.getRobotList():
            # robot list is empty
            self.userhistory.remove('task')
        else:
            print longstrings.GIVE_TASK_TAG


    def giveTask(self):
        '''Sends a user given task to the infrastructure together
    with the tag of the robot to perform it. Makes the task from
    serveral inputs from user.
    '''
        choice = raw_input()
        if not choice.isdigit():
            print longstrings.WRONG_TYPE
            return

        # get a list of possible tasks for this robot
        data = self.infrastructure.getPossibleTasks(int(choice))
        if not data:
            # if wrong tag is typed three times, return to main menu
            print longstrings.WRONG_TAG
            self.userhistory.append('task')
            if self.userhistory.count('task') == 4:
                print longstrings.RETURN_MENU
                for i in range(self.userhistory.count('task')):
                    self.userhistory.remove('task')
            return
        print data

        tag = int(choice)

        # register task name
        print longstrings.GIVE_TASK
```



```
        task_name = raw_input()

        # register arguments
        print longstrings.GIVE_ARGUMENTS
        args = raw_input()
        if args == '':
            # no arguments given
            task = task_name
        else:
            # translate from string type to list of tuples
            args = self.makeArgs(args)
            task = (task_name, args)

        # interrupt the robot?
        print longstrings.WAIT
        _wait = raw_input()

        # send task
        if _wait == 'yes':
            self.infrastructure.giveTask( tag, task )
        else:
            self.infrastructure.giveTask( tag, task, True )

        for i in range(self.userhistory.count('task')):
            self.userhistory.remove('task')

        # return to main menu
        print longstrings.MENU


    def makeArgs(self, args):
        '''Translates a string of arguments to a list of tuples.
    '''
        i=0
        x = ''
        y = ''
        path = []
        while not i >= len(args):
            #print i
            if args[i] == '(':
                i+=1
                while not args[i] == ',':
                    x = x+args[i]
                    i+=1
                i+=1
                while not args[i] == ')':
                    y = y+args[i]
                    i+=1
                i+=1
                path.append( (int(x), int(y) ) )
                x = ''
                y=''
            i+=1

        if len(path) == 1:
            return path[0]

        return path


    def codemenu(self):
        '''Prints code menu.
    '''
        if not self.getRobotList():
            # robot list is empty
            self.userhistory.remove('code')
        else:
            print longstrings.CODE_TAG
```

```
    def sendCode(self):
        '''Sends a user given file containing a python module to the
    infrastructure together with the tag of the robot to
    import it. The file is wrapped and sent in an instance of
    the Binary class because of xml rpc rules.
    '''
        tag = raw_input()
        if not tag.isdigit():
            print longstrings.WRONG_TYPE
            return

        tag = int(tag)

        # name of file containing code
        print longstrings.CODE_NAME
        filename = raw_input()

        infile = open(filename, "r")
        code = infile.read()
        infile.close()
        binary_obj = Binary(code)

        print longstrings.CODE_OR_MODULE
        choice = raw_input()
        if choice == 'code':
            # sends code to execute
            data = self.infrastructure.sendCode(tag, filename,
                                                binary_obj, True)
        else:
            # send module to import
            data =  self.infrastructure.sendCode(tag, filename,
                                                 binary_obj)

        if not type(data) == str:
            print 'All went well\n'
        else:
            print data

        # return to main menu
        self.userhistory.remove('code')
        print longstrings.MENU


    def stopRobot(self):
        self.getRobotList()
        print 'tag to stop: '
        tag = raw_input()
        if not tag.isdigit():
            print longstrings.WRONG_TYPE
            return

        tag = int(tag)
        t1 = time.time()
        if self.infrastructure.stopRobot(tag):
            t2 = time.time()
            print 'Robot stoped'
        else:
            t2 = time.time()
            print 'Goodbye robot!'
        print t2 -t1


if __name__ == '__main__':
    exSrv = User()
    exSrv.main('129.242.19.46', 8082)
```

```
################################################################################
#
# All the menus and long string printed to the user.
#
# Written by Marte K Skadsem, 2005/2006
#
################################################################################

MENU = '_____ MAIN MENU ______________________\n' + \
       'What do you want to do? (Make a choice)\n' + \
       '1. Get List of robots\n' + \
       '2. Monitor robot\n' + \
       '3. Give task to a robot\n' + \
       '4. Give new code or module to a robot\n' + \
       '5. STOP ROBOT\n' + \
       '6. Quit\n' +\
       '_____________________________________'

USER_WELCOME = '**** WELCOME! ****\n' + MENU

MONITOR = 'Print the tag of the robot you wish to monitor\n' + \
          'tag: '

GIVE_TASK_TAG = 'Print the tag of the robot you wish to give a task\n' + \
                'tag: '

GIVE_TASK = 'Print the task you want to give\n' +\
            'task: '

GIVE_ARGUMENTS = 'Print the arguments to the task\n' +\
                 'arguments: '

WAIT = 'Do you want to interrupt the robot in its current task?\n' +\
       'yes or no: '

CODE_TAG = 'Print the tag of the robot you wish to give the code\n' + \
           'tag: '

CODE_NAME = 'Print the name of the file containing the code\n' +\
            'filename: '

CODE_OR_MODULE = 'If the file is a module, write module. If it is code to execute, write code.'

#--------------------------------------
# Error messages
#--------------------------------------
WRONG_INPUT = 'Wrong input. Please choose among the numbers given.\n\n' +\
              MENU

WRONG_TAG = 'There is no robot with this tag. Try again!\n' +\
            'tag: '

RETURN_MENU = 'Wrong again. No more tries for you.\n' + \
              'Returning to main menu\n\n' + \
              MENU

WRONG_TYPE = 'Wrong input type. Please enter the number of choise\n'
```

```
################################################################################
#
# A module to be exported to robots.
#
# Written by Marte K Skadsem, 2005/2006
#
################################################################################

def square(parent):
    '''Drives in a square with size 50X50
  '''
    size = 50

    # report position
    direction = parent.driving_control.direction
    parent.my_pos = parent.infra_intf.reportPosition(parent.tag,
                                                     parent.my_pos,
                                                     direction)
    parent.my_pos = parent.my_pos[0], parent.my_pos[1]

    for i in range(4):
        if parent.interrupt:
            return
        if i == 0:
            x = parent.my_pos[0] + size
            y = parent.my_pos[1]
        elif i == 1:
            x = parent.my_pos[0]
            y = parent.my_pos[1] + size
        elif i == 2:
            x = parent.my_pos[0] - size
            y = parent.my_pos[1]
        else:
            x = parent.my_pos[0]
            y = parent.my_pos[1] - size

        next_corner = ( x,y )

        # find path
        path = parent.map.findPath(parent.my_pos, next_corner)

        # follow path
        msg = parent.driving_control.followPath(path)

        parent.my_pos = next_corner

        if msg == 'done':
            # report position
            direction = parent.driving_control.direction
            parent.my_pos = parent.infra_intf.reportPosition(parent.tag,
                                                             next_corner,
                                                             direction)
            parent.my_pos = parent.my_pos[0], parent.my_pos[1]

    return msg
```

```
################################################################################
#
# A module to be exported to and executed by robots.
# It replaces a metod in the robot class and adds a task.
#
# Written by Marte K Skadsem, 2005/2006
#
################################################################################
import threading, time

def new_doTask(self):
    '''Replaces the originally doTask().
  Executes the new task and starts the task-kill thread.
  Cleans up after execution.
  '''
    print 'Execute task ' + str(self.task)

    self.stopped = False

    # start taskkill thread
    taskkill_thread = threading.Thread(target=self.taskKill)
    taskkill_thread.setDaemon(True)
    taskkill_thread.start()

    # check what task to do (only predefined tasks)
    if self.task[0] == "goTo":
        msg = self.goTo( (self.task[1][0],self.task[1][1]) )
    elif self.task[0] == "followPath":
        msg = self.followPath(self.task[1])
    elif self.task[0] == "examineArea":
        msg = self.examineArea( (self.task[1][0],self.task[1][1]) )
    elif self.task == "square":
        msg = self.new_modules["squareCode"].square(self)

    # report that task is done
    if not self.interrupt:
        self.infra_intf.done(self.tag, self.task, msg)
        self.interrupt = True   # set in order to stop taskkill thread

    # wait until thread is stopped
    taskkill_thread.join()

    # make ready for new task
    self.interrupt = False
    self.driving_control.interrupt = False
    self.stopped = True


def init(parent, flag):
    '''The method that all imported modules to execute must have.
  Control is given to the code through this method. Important
  responsibility: Must raise the flag parameter when controll
  can be given back to main thread.
  '''
    parent.__class__.doTask = new_doTask

    parent.interrupt = False
    parent.driving_control.interrupt = False

    # finish, raise flag
    flag.append(1)
```