Technische Universiteit **Eindhoven** University of Technology

UC San Diego

Department of Mechanical Engineering

Implementation of a distributed algorithm for coverage control

Internship report

R.A. Ottervanger BSc

Supervisors:
prof. J. Cortes
prof.dr.ir. M. Steinbuch

CST 2014.NUMBER

San Diego, California
June 2014

Abstract

This report describes the progress made on the robotics project at UCSD by Rokus Ottervanger in spring 2014. The work is focused on the deployment of a scalable number of robots using given theoretical algorithms for coverage control. The report describes the successful design and development of a functional implementation of one of such algorithms aiming for other similar algorithms to follow. To do this, the architecture is modular. It is set up so that parts of the implementation can easily be replaced by other programs, making future development easier. Finally, a number of recommendations with respect to the hardware and software setup is given, as well as suggestions for future development.

Contents

Chapter 1

Introduction

In this project, an implementation is delivered of algorithms for the autonomous distributed control of mobile sensor networks. This chapter gives the motivation for this, as well as the formal problem statement of this work. Finally, an outline is given of this report.

1.1 Motivation and background

In recent years, interest has grown in numerous applications of networks of autonomous cooperative mobile robots or sensors. Numerous examples of these can be found in literature, including sensor networks that may be applied to monitor environmental matters such as in oceanographic research [5]. Another example is Google's project Loon [1], which aims to provide an Internet connection in remote areas using a network of stratospheric balloons much like weather balloons. In the recent event of the Malaysia Airlines aircraft that was lost over the Indian ocean, an autonomous naval sensor network could have been deployed to search for the lost airplane's fading distress signal and black box. In this report, the individual members of such networks are referred to as agents, sensors or robots.

Each of these networks benefits from the minimization of noise occurring in the transfer of information, either sensing or communicating, between any point in the area of deployment and the nearest agent. To minimize this noise, such a network may distribute itself over the predefined area using the algorithms proposed in [4], which are specifically designed to do this. The important property of these algorithms is that they are distributed, so they are designed to work without the use of centralized computation in the network.

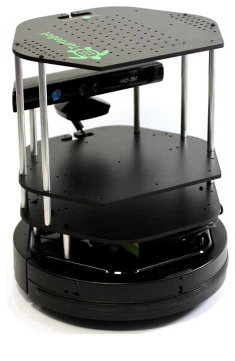


Figure 1.1: *Turtlebot 2*

1.2 Problem statement and objective

In the lab of professors Cortes and Martínez ten second generation Turtlebots, as shown in figure 1.1, are available for testing and demonstrations of these algorithms. However, before the algorithms can be deployed, they need to be implemented first. Software to determine a ground truth as to the localization of the robots exists and is in development to become more reliable. To perform demonstrations with the Turtlebots in deployment for coverage control, software needs to be designed for the position control of the individual sensors. To be more specific, algorithms provided by [4] are to be implemented. Among these are algorithms for communication, determination of the locally important peers, goal position determination and the position control of the individual robots. The goal of this work is to deliver this implementation, aiming to stay as close as possible to a real world implementation such as the ones stated in section 1.1 to be able to give realistic demonstrations. This means that the software should run distributedly, with as little as possible, if any, central computation and/or coordination.

1.3 Outline

This report starts by explaining relevant theoretical background on the subject, after which the setup of this work is laid out. The actual implementation is discussed in the next chapter and after that, the test cases for this implementation and their results are given. In the final chapter, conclusions are drawn with respect to the problem statement and objective of this work and recommendations for further work are given.

Chapter 2

Theory

The theoretical foundation of the implementations done in this work were laid by J. Cortes et al. In this chapter, relevant basics of the algorithms proposed in that work are described. For more into depth information about the theoretical coverage control algorithms, refer to [3] and [4]. Furthermore, the implementation given in this work uses an image processing algorithm called flood filling, which is also explained in this chapter. Without loss of generality, in this chapter a network of sensors is observed. Identical reasoning would apply for a network providing internet access or other similar networks.

2.1 Coverage control

Let n be the number of agents deployed within a convex polygon, including its interior, Q in $\mathbb{R}^N$ and $P = (p_1, \ldots, p_n)$ the location of the agents, moving in Q. Because of noise and loss of resolution, the agents' sensing performance degrade with the distance between the sensed event and the agent. So if agent i senses some event at point $q \in Q$, the sensing noise behaves as a non-decreasing differentiable function $f(\|q - p_i\|)$. We also consider the density function $\phi(\cdot)$ that can be seen as the probability of an event occurring, or the importance of recording an event at a every point q. A *partition* of Q can be defined as a collection of n polygons $\mathcal{W} = W_1, \ldots, W_n$ of which the union is Q. If we consider the task of minimizing the total sensing error, the following locational optimization function comes to mind:

$$H = \sum_{i=1}^{n} \int_{W_i} f(\|q - p_i\|)\phi(q)dq. \tag{2.1}$$

It can easily be seen that the optimal partition is the Voronoi partition $\mathcal{V}(P) = V_1, \ldots, V_n$ with P as generators:

$$V_i = q \in Q | \|q - p_i\| \leq \|q - p_j\|, \forall j \neq i. \tag{2.2}$$

It is shown in [4] that for $f(\|q - p_i\|) = \|q - p_i\|^2$, the local minima of H are at the centroids of the Voronoi cells, resulting in a centroidal Voronoi partition V_C. In general, multiple solutions are possible for an arbitrary Q and ϕ.

Another sensible choice for H would be the one to minimize the worst case sensing error, i.e. to minimize the maximum distance from any point to the nearest sensor. This would be

$$H(P) = \max_{q \in S} \min_{i \in \{1, \ldots, n\}} \|q - p_i\|. \tag{2.3}$$

[4] also shows that the local optima for this can be found where P is the circumcenter locations of the Voronoi partition generated by P.

Even more possibilities can be found when looking at non-convex environments, such as in [2]. In that case, the Euclidean distance does not generally equal the minimum travel distance between any q and p_i. This current work however, focuses on implementation of the first distributed optimization problem, where a centroidal Voronoi partition is an optimum.

2.2 Lloyd's algorithm

A way to find centroidal Voronoi partitions is Lloyd's algorithm. The discrete case of this algorithm calculates the Voronoi diagram and the centroids of its Voronoi cells. Moving the Voronoi generator points to the centroids of their Voronoi cells each step makes the partition converge asymptotically to a centroidal Voronoi partition. An example of this is shown in figure 2.1.

According to [4] Lloyd descent also converges to a centroidal Voronoi partition. In Lloyd descent, it is enough for the generators to move *towards* the centroids of their last computed Voronoi cells instead of actually reaching them in that step. This property is very useful when the generators have their own dynamics, limiting their movement in some way. This idea is used in the implementation following from this work.

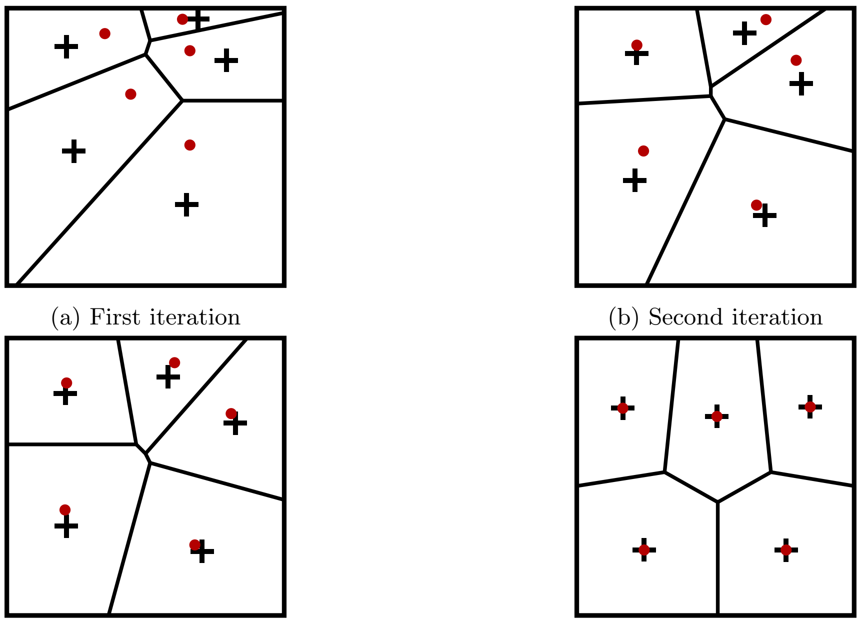

(a) First iteration (b) Second iteration (c) Third iteration (d) Fifteenth iteration

Figure 2.1: *Demonstration of Lloyd's algorithm to find centroidal Voronoi partitions [9].*

2.3 Flood fill algorithm

For an agent in the deployed system to find its own Voronoi cell, it draws its model of the multi-agent system and the environment in an image and draws the Voronoi diagram over it. For more information on this, see section 4.2.2. To find the cell that corresponds to the robot in this image, the flood fill algorithm is used.

Flood filling is a way to find enclosed areas in an image. The algorithm takes a seed point, a substitute value and an image. It starts at the seed point and replaces the pixel with the substitute color. It then moves on to the (either 4-connected or 8-connected) neighbors and replaces their values with the substitute one by one if the difference with the value of the seed pixel is below a certain threshold. It keeps looking for neighbors of the pixels that were replaced in the previous cycle until there are no more pixels to replace.

This algorithm is very intuitive when observing a single channel (gray value) image as a landscape that is high at high valued pixels and low at low valued pixels. It fills the 'valley', pointed to by the seed point, with the substitute value, up to a certain height.

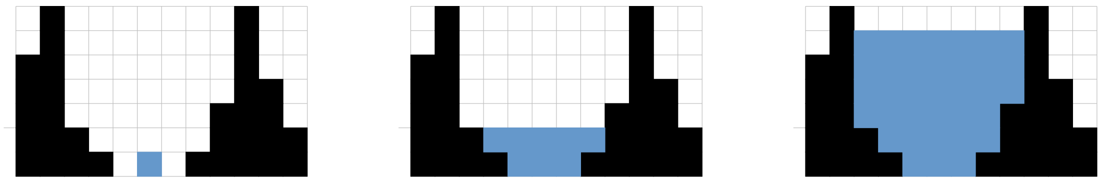

(a) Seed point (b) Filled up to a difference of 2 (c) Filled up to a difference of 6

Figure 2.2: *One dimensional cross sections of a two dimensional 8 bits gray scale image on which flood filling is performed.*

2.4 Reference frames

In the implementation, several reference frames are used. To explain their function and the relations between them, they are explained below. All reference frames except the image frame are right-handed, so positive angles turn counter clockwise.

2.4.1 Robot fixed frame and odometry frame

One reference frame is fixed to each of the robots. These frames are oriented such that the x-axis points forward, the y-axis to the left and the z-axis completes the right-handed frame by pointing up through the rotational center of the robot. Generally, this frame is kept track of by ROS in the transform tree as `base_footprint` with respect to the initial position and orientation of this frame, called the odometry frame. Note that as each of the robots is initiated in a different position and orientation, each has its own odometry frame.

2.4.2 Map frame

The robots deploy themselves in the map frame. This frame has its base in the bottom left corner of the map of the deployment area. Its x-axis points along the lengthwise direction of the deployment area and its y-axis in the widthwise direction. The z-axis points upward. To find the positions and orientations of the robots, the relationship between the respective odometry frames and the (global) map frame is kept track of by ROS.

2.4.3 Camera frame

The software for the top-view camera records the positions and orientations of the robots in the camera frame. In this way, the robots are localized in the camera frame. In this work, the camera frame is assumed to be perfectly aligned to the map frame, so that the localization data can be seen as a direct overlay to the map, without translation or rotation. In reality, this means that the camera should be mounted such that the edges of the image are aligned to the edges of the deployment area. The localization software also scales this frame such that it matches the map frame.

2.4.4 Image frame

The image frame is the reference frame that is used to draw the map and Voronoi diagram data in. Because in image processing, convention is to handle an image as a matrix, the x-axis runs along the top of the image (the column indicator) and the positive y-axis points down along the left of the image (the row indicator). When drawing the map and localization data, scaling is needed because of the (adjustable) resolution of the image.

Chapter 3

Setup

For the programming in this work, a foundation of hardware and software is chosen to build upon. This foundation can be seen as the test setup on which the designed software is tested and executed. Both the hardware and the software aspects of this setup are discussed in this chapter.

3.1 Hardware setup

For demonstrations of various deployment algorithms, a simple and flexible platform is needed. Ten Turtlebots (see figure 1.1) are available to fulfill that task. Turtlebots are readily available wheeled robots equipped with a Kinect for vision and depth perception, encoders on the wheels and an IMU to provide odometry data, and a netbook with WiFi capabilities for computation and communication with peers and/or a central computer.

3.1.1 Wifi setup: Infrastructure

Two different wireless network setups are used. Firstly, as wireless network infrastructure is available in the form of the UCSD-PROTECTED network, this is the most convenient option to use. The robots all have an Internet connection through this network, which makes it easy to update their software; automatic updates through the Ubuntu package manager as well as custom software from the SVN. However, to simulate real deployment, it may be interesting to have the robots communicate directly with each other without infrastructure. Additionally, while testing, the UCSD-PROTECTED network infrastructure showed bad performance (bad signal strength, occasional dropped packets).

3.1.2 Wifi setup: Ad hoc

Therefore, the second network setup is an ad hoc structure. In this configuration, no infrastructure is needed and robots communicate directly with each other. It is also possible to include a computer with Internet access into such a network to provide the network with a web connection. However, this does not seem to be possible through the UCSD-PROTECTED network environment because of security measures. For details on this network setup, refer to Appendix A.

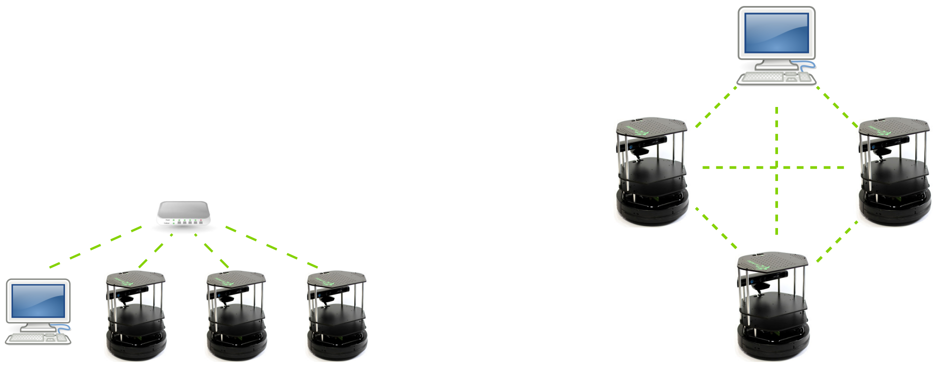

(a) Infrastructure (b) Ad hoc

Figure 3.1: *Network configurations.*

3.2 Software setup

The software developed in this work is built in a pre-existing software environment. This section describes the environment from the operating system to custom programs that were written before and used in this work.

3.2.1 Operating system, framework and language

The Turtlebots come preinstalled with Ubuntu 12.04 LTS and also the lab computers are equipped with this operating system. Additionally, ROS (the Robot Operating System) is installed on each of these computers [10]. This offers libraries for C++ and Python to use in robotics applications, as well as tools for debugging, simulation and visualization. The currently used version of ROS is Hydro Medusa. Furthermore, the Turtlebot software, a series of ROS packages specifically designed for the Turtlebot, is installed on each of the computers. In this project, programming is done in C++ using the IDE Qt Creator. Finally, the image processing toolbox OpenCV for C++ is used in the deployment implementation. All of this software, from the operating system to the image processing toolbox, is open source.

3.2.2 Distributed control

As ROS needs a single core to have its nodes communicate to one another and localization is done using a lab PC, the most obvious choice for a computer to run this core is the lab PC. Research was done into options to create a more realistic distributed environment by using software such as LCM [6], RoCon [12] or ROS-rt-wmp [13]. However, although the end goal is to deploy the Turtlebots completely distributedly, the most practical solution now is to simulate this behavior by distributing the running programs as much as possible within the framework of ROS, as this works out of the box. The fact that a single core is running on a centralized computer does not diminish the fact that the implemented algorithms are distributed and they would work without alterations in a completely distributed ROS setup.

3.2.3 Localization node

A way to localize the robots was implemented before and improved during this project. This localization is done using a ROS node receiving data from a top-down view camera mounted above the deployment area. The node is programmed such that it localizes one robot using a (robot specific) color tag on top of the robot. The tags consist of two slightly differently colored areas to be able to determine the heading of the robot. The ROS node only identifies one robot, but it can be tuned to different colors. This way the system is scalable by running multiple instances of this node, albeit with different color settings, in parallel.

3.2.4 Navigation

Navigation is readily implemented in the ROS navigation stack [8]. However, this software is complicated and can lead to problems unrelated to the software currently in development while debugging. Therefore, this navigation stack is not used yet. A simpler navigation node is developed for testing purposes, which is described further in chapter 4.

Chapter 4

Software Design

This chapter aims to explain the general structure of the software design as well as the inner workings of the separate programs within this structure. Furthermore, decisions were made in the process of designing this structure and writing its components. These decisions are also explained here, in order to clarify why the programs are designed in this way.

4.1 Architecture

A ROS robot control structure is based on nodes communicating with one another as in a publisher-subscriber model or using services and requests. This makes it possible to structure the software into nodes in order to separate tasks. Figure 4.1 shows the control structure in a multi-robot system. The hardware receives velocity commands and moves accordingly. The control loop is closed because the localization measures the new position of the robot. The localization node exists in the form of the top-down camera localization node, and the other software components are developed in this work.

The localization node publishes location data for a robot over a location topic specific to that robot. For the deployment however, Voronoi cells need to be calculated, so the locations of possible other robots should also be known. To do this, the communication node is positioned between the localization and the deployment nodes. This node makes the position of its associated robot available to the other robots in the system by publishing it on a global topic. The deployment node can then tap from that information to create a model of where the respective robot and its peers are located in the deployment space as well as what their orientations are.

The deployment node is designed so that its output is a new goal position (namely the centroid of the corresponding Voronoi cell. The hardware however, can only handle velocity commands. To translate goal positions to velocity commands, the navigation node is written.

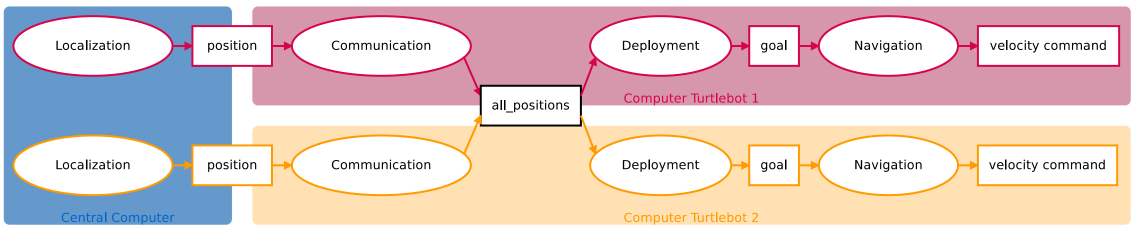


Figure 4.1: *Multiple robot control structure.*

4.2 Implementation

In this section, the inner workings of each of the nodes is explained, the order following the stream of information, starting with the communication node. Diagrams are used to show the interface of each node. In these diagrams, the topics that the node subscribes to and/or publishes on are shown, as well as the parameters that need to be set by the user. Each of the nodes returns an error when parameters are not set or not valid. When a topic name is prefixed by '/', that means the name is fully qualified, and the name is resolved to that exact topic name. Conversely, if a topic name is not prefixed by '/', that means that the topic name will be resolved to '/some_namespace/topic_name', if the node runs within a group namespace.

4.2.1 Communication

As can be seen in figure 4.2, the original position messages published by the localization node are published on a topic within the specific robot's namespace. Also, the messages themselves do not contain information about which robot they refer to, as that information is given by the namespace of the topic they are published on. On the other hand, a node cannot subscribe to a variable number of topics. So to make the system scalable, the deployment software preferably listens to one topic on which information on all agents is published. However, the receiving deployment software needs to know which position is its own to know which Voronoi cell is its own, so the agent name of each message is relevant. To avoid hard coding the robot names in any program, a new message type is created that contains the pose data that is published by the localization node as well as a name tag of the robot it refers to. As soon as the communication node receives a pose from the localization node, it repackages it into that new message type, adding the name tag, and publishes it to the global position topic. This makes the system scalable as a communication node runs for each agent and names can be assigned by the user (but should be coordinated to match the robot names, for instance in a launch file).

The attentive reader may notice that the functionality of the communication node could easily be integrated in the localization node. To do this, the localization node may be provided with the robot's name and it should include the new message. Then it could publish the pose information on a combined topic right away, without intervention of a communication node. However, in a later stage, the current localization node may be replaced by other methods, such as a GPS node, or the existing Adaptive Monte-Carlo Localization (AMCL) node, as these are distributed ways for the robots to localize themselves in a deployment space. The latter publishes on the amcl_pose topic within the robot namespace just like the current localization node. New localization nodes would also best be written to publish a single robot's pose on a topic similar to amcl_pose within the robot's namespace. This way, they can also be used for localization of an isolated robot. To ensure compatibility with these other ways of localization, the communication functionality is isolated from the localization node.

4.2.2 Deployment

At the highest level, the deployment node uses the positions of all robots in the system to determine a Voronoi diagram. It uses the Voronoi cell of the agent it works for and calculates its center

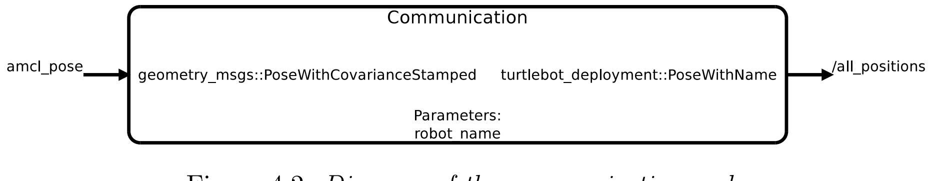

Communication
amcl_pose
geometry_msgs::PoseWithCovarianceStamped
turtlebot_deployment::PoseWithName
/all_positions
Parameters:
robot_name

Figure 4.2: *Diagram of the communication node*

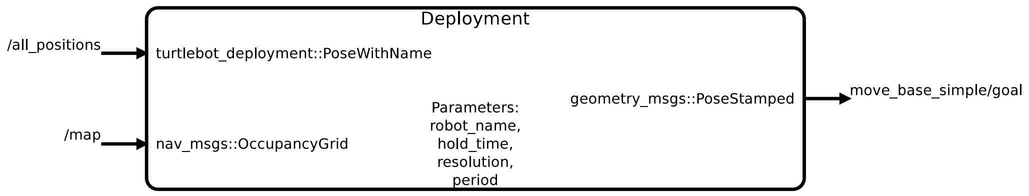


Figure 4.3: *Diagram of the deployment node*

of mass. This point is sent on as the robot's goal position.

To this end, each robot maintains its own database of robots in the system (including itself). This database is programmed as a vector of objects of the 'agent' class. For more information about this class, see subsection 4.2.3. Upon reception of information about a robot, first the database is checked for availability of that robot. If the robot is already available, its pose in the database is updated; if the robot is not yet available, it is added to the database, also if this information is about the agent the node works for. Finally, the database is sorted by distance, from short to long. Furthermore, upon reception of a map, the map is stored in the node's map member variable.

A timer is used to trigger the generation and publishing of a new goal position. This is the main part of the deployment node. In the `publish` method, which is called by the timer, availability of the map and the associated robot's position is checked and database entries of which the age exceeds the hold time are removed from the database. This new database now contains all necessary agents for the Voronoi diagram.

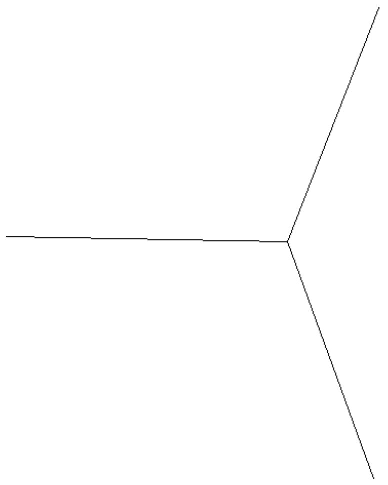
To generate this diagram, a third party open source Voronoi diagram generation implementation is used [11]. The Voronoi diagram generation algorithm is implemented as a class that contains methods to generate and extract the Voronoi diagram. The generation method takes arrays of x-values and y-values of the generator points, the lengths of these arrays and the minimum and maximum x- and y-values in between which it will generate the diagram. The algorithm returns using the `getNext` method, which overwrites the values of the input arguments with the endpoints of the next line and returns true while there are more line segments in the diagram.

After conversion to image coordinates, which is done as late as possible to avoid rounding errors, the extracted lines are drawn in an image with a user defined resolution. Next, the map of the environment, which was received from the map server, is drawn in a separate image and using a bitwise OR operation, added to the Voronoi image. Using a flood fill algorithm, which is readily available in OpenCV, and the position of the robot itself as seed point, the right Voronoi cell is extracted and drawn onto a new image. These steps are shown in figure 4.4. The center of mass of this last image is (approximately, because of a limited resolution) the center of mass of the robot's Voronoi cell. This point is transformed from coordinates in the image to coordinates in the map frame and published as goal.

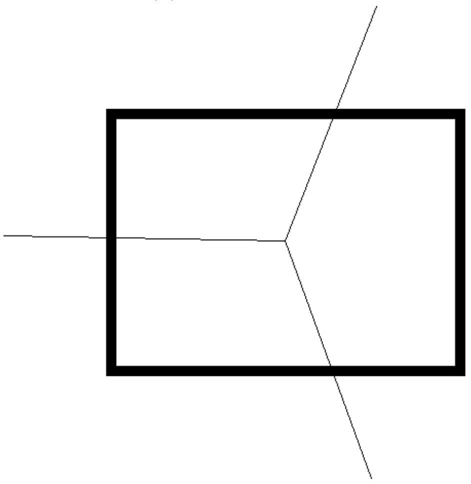
4.2.3 Agent

The robot database that is maintained by the deployment node is a vector of elements of the 'agent' class. This class contains an identification number, a name, a pose, a time stamp and a distance. The identification number is zero if the agent is another, and one if it is the one the node works for, but this information is currently not used. The time stamp contains the time of the last position update for that agent. The distance is the distance between the stored agent and the agent the node works for. Furthermore, the agent class has methods for retrieving and setting all of this information except the time stamp, which can only be set to the current time by updating the pose. There is also a method to get the age of the latest pose update. There are

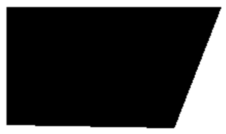
(a) Agent representations



(b) Voronoi diagram



(c) Voronoi diagram with map



(d) Resulting Voronoi cell

Figure 4.4: *Intermediate steps to determination of the Voronoi cell's centroid. Program output is color-inverted for printing purposes.*

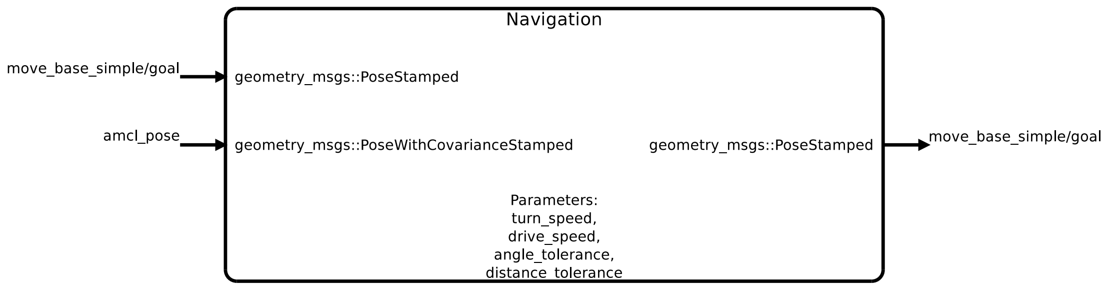


Figure 4.5: *Diagram of the navigation node*

several different constructors available to initialize an object of the agent class. Details on this can be found in the header file of the Agent class.

4.2.4 Navigation

The navigation node is programmed to be as simple as possible in producing velocity commands for the robot given a position and a goal. It stores goal pose data about its associated agent at arrival of such messages and calculates velocity commands at arrival of pose messages (from the localization node). As long as the robot's heading is not within a certain tolerance around the direction of the goal position, the robot will turn in its place towards the goal position at a fixed angular velocity that can be defined by the user. As soon as the heading is within that tolerance, the robot starts moving forward at a fixed speed which can also be user defined. It will stop driving forward as soon as the heading direction is again outside of the tolerance area, at which point it will start turning again, or when it reaches the linear proximity tolerance.

At this point, the navigation node publishes raw velocity commands to the robot's base. These should however be filtered to obtain a smoother signal. Also the user of this node must be aware that this is only suitable for deployment in convex spaces as it will try to move directly towards the goal position. Additionally, although the deployment algorithm should always spread the robots out in space and never let them cross paths, turned off or malfunctioning robots may be in the way of deployed robots. In such a case, this node does not avoid obstacles. To obtain obstacle avoidance and behavior suitable for deployment in non-convex spaces, the ROS navigation stack may be used. This is however more complicated and so far not used because of complications in applying that in a multi-robot environment. Using the navigation stack would also not be enough to deploy robots in non-convex spaces as the currently implemented deployment algorithm is not suited for this.

Chapter 5

Results

A vital part of software design is verification of correctness. Although this is generally a difficult problem, using appropriate test cases, many bugs can be found and fixed. This chapter explains what test cases are set up for the verification of the ROS nodes implemented in this work and it presents the results of these test cases.

5.1 Simulation

In simulation, each part of the software is subjected to one or more test cases. This section explains the setup of the test cases and shows the results.

5.1.1 Communication

Firstly the communication node is tested. The function of this node is simply to output its input with a name stamp, so its working can be verified by simply examining the output after giving manual input. This experiment is done with success. After verification of correctness, the communication node is included in the test scheme of the deployment node.

5.1.2 Deployment

The deployment node is tested in three parts: Voronoi diagram generation, goal position generation and the overall result of the node's application.
To test the diagram generation, a node is written to generate dummy data of randomly placed robots at a fixed frequency of 0.5 Hz. Using this data as input for the Voronoi diagram generation, about 30 generated diagrams are visually checked for plausibility, verifying that the generated line segments are indeed bisectors of the generator points. Results of this can be seen in figures 5.1

After verifying that the generated diagrams are viable, the generation of goal positions can be tested. In figure 5.2, the extracted Voronoi cells are shown in black with a white dot indicating the location of the centroid (or goal position) in several randomly generated configurations. Again, the generation of the centroids is checked visually on about 30 images, with positive results.

If goal positions are generated and published correctly, a basic simulation of deployment can be performed. To this end, a dummy node is written to simulate the behavior of an agent in the sense that it receives a goal pose and publishes its new pose. A diagram of the simulation software structure can be seen in figure 5.3. To keep the dummy node simple and avoid the opportunity for bugs in that node, the robot is assumed to attain the goal position within one time step. As a result, the dummy node can just republish the goal pose as its most recently attained pose. This results in an implementation of the true Lloyd's algorithm, as opposed to a Lloyd descent behavior.

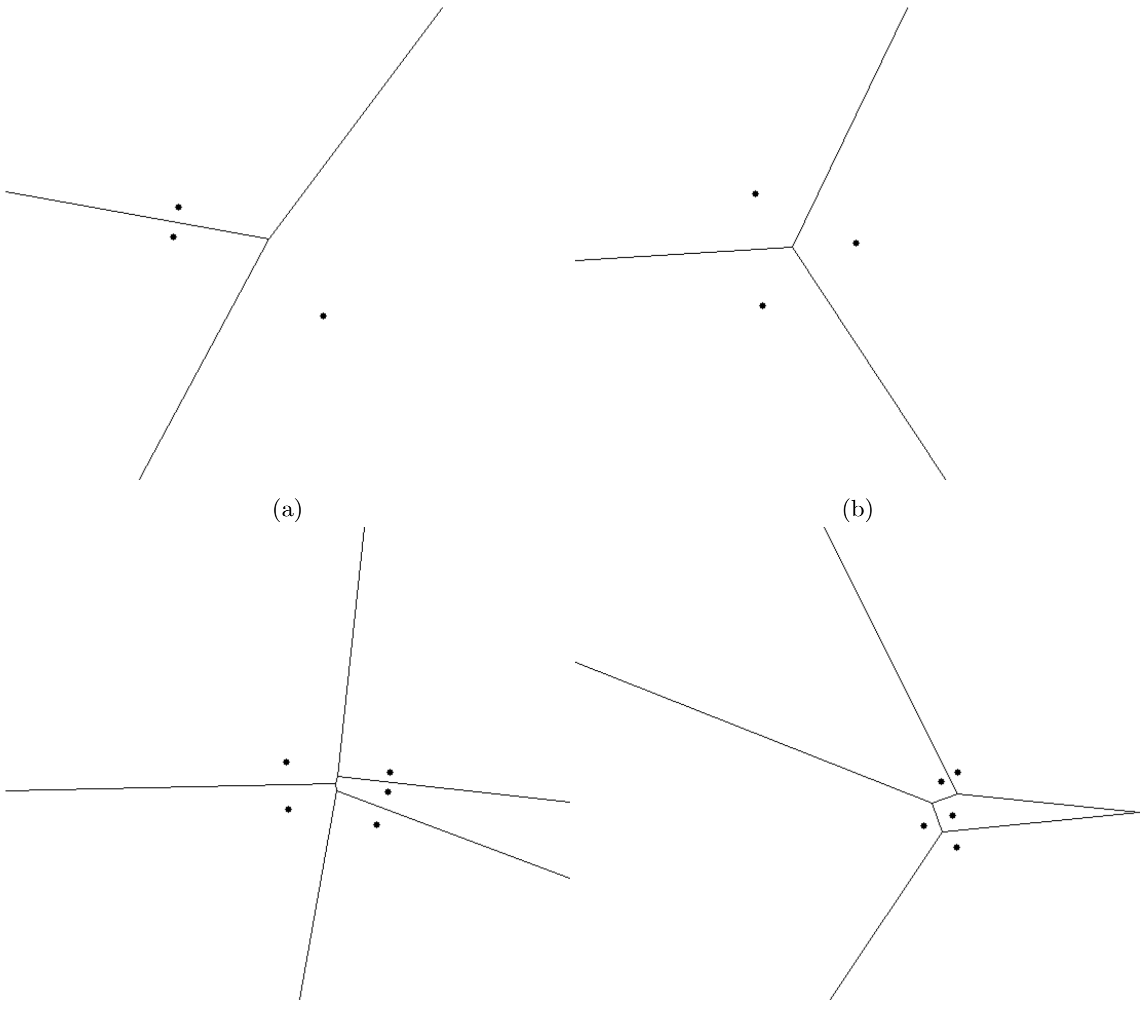
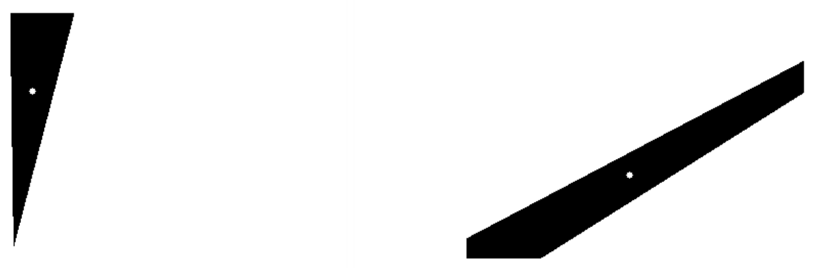
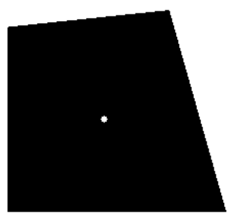


Figure 5.1: *Voronoi diagram generation results. Program output is color-inverted for printing purposes.*



(a) (b)



(c)

Figure 5.2: *Results of calculating centroids of Voronoi cells. Program output is color-inverted for printing purposes.*

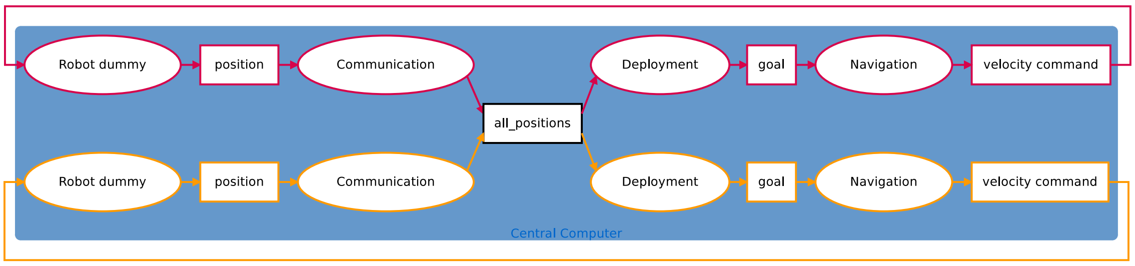
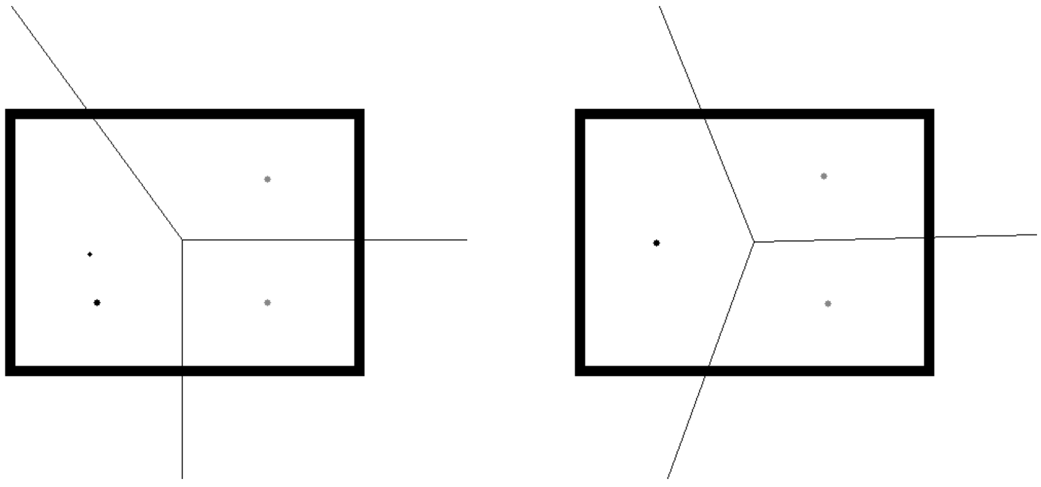


Figure 5.3: *Diagram of the deployment simulation setup for two robots*



(a) Just after removal (b) Near equilibrium

Figure 5.4: *Deployment simulation results after removal of fourth agent. The agent this node works for is indicated in black, its peers in gray and its goal position with a smaller black dot. Program output is color-inverted for printing purposes.*

In this simulation, the scalability of the system can be tested. In practice, it should be possible to add an agent to the system or remove one from it by simply running a launch file or by shutting one down. This can also be done in simulation to test the robustness against addition and deletion. Several experiments are performed to verify that the system has this robustness. An example result of such a simulation can be seen in figures 5.4. In this case, from a four-agent system (which has already successfully undergone addition of agents), one agent is removed by shutting it down. The network adapts to the absence of the fourth robot by redistributing its remaining three robots over the deployment space, including the space the fourth robot left behind.

5.2 Hardware deployment

After testing in simulation, the software can be tested on the hardware. For the communication node there is no difference between simulation and hardware deployment, so there is no need to additionally test this part of the software. It can therefore directly be used in the test schemes of the deployment node and the navigation. This section discusses the hardware tests and their results.

5.2.1 Localization

Localization is an issue in testing on the hardware. The existing node relies on identifying colored tags on top of the robots using a top-view web cam. However, due to changes in lighting conditions, the quality of localization is highly dependent on the time of day. The lighting conditions are also not constant in the deployment area, which made the quality of localization and especially consistently finding the orientation of the deployed robots dependent on their location in the lab. These issues may be solved by switching to shapes instead of colors for finding orientation. Localization and identification may then still be done using colors, but using shapes for the orientation relaxes constraints on the width of the color bands used in thresholding for localization. Of course, localization may also be done using shapes.

Another problem in the localization is that there is no filtering. Additionally, the published position and orientation are zero when the localization node is not able to find any object large enough, within its threshold values. This, together with the changing lighting conditions makes the published data unreliable. The reliability may be increased by only implementing sample-and-hold. This would result in the robots at least receiving the last known position of their peers and themselves. On the other hand, this could lead to unwanted behavior. A robot that is not found in the camera image any longer, will continue to move, because its perceived position does not change. The robot may uncontrollably move outside of the deployment area. A more reliable solution would be to use data from the IMU when not receiving location information from the camera or additionally use a Kalman filter.
Finally, the color bands the localization node looks for are now hard coded into the source. The right values are selected in the program by using the robot name as input and comparing it to hard coded robot names. This means that it can only work with the preprogrammed robot names and their corresponding preprogrammed colors. This is not only very confusing when switching robots and their color tags, but it also makes it impossible to scale the system to more than three robots other than by adding robot names and threshold values to the program source code.

5.2.2 Wireless networking

Testing on the hardware revealed issues with the wireless network connection as more robots were deployed. A single robot does not show any signs of issues. As soon as a second robot is deployed, the robots seem to occasionally lose connection, resulting in removal of deployed peers from their database; even with a relatively long hold time of 8 seconds, and the localization node running at a rate of 20 Hz. With more robots, the connection seems to drop more often. The problem is not with the computational power of the Turtlebots or the lab computer as CPU levels don't reach 100% while running the software. On the ad-hoc network, performance seems to be better, but this is not extensively tested, and in any way, the problem is not completely solved.

A possibility might be that the network cannot handle the amount of transferred data. This means that buffers of the infrastructure fill up and packets of data are dropped. Some suggestions of locating and/or solving the problem are to reduce the publishing frequency of (at least) the localization node to reduce network traffic.

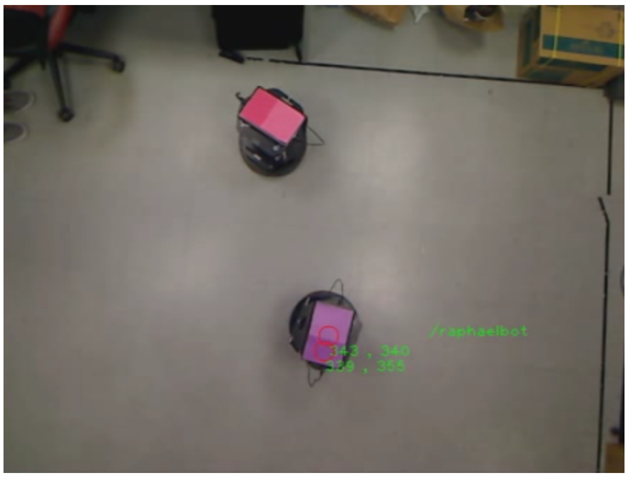
A more sustainable way could be to communicate more efficiently by using services instead of publishing and subscribing. ROS offers the possibility for one node to ask for the service of another node. This could be a request for information, upon which the service provider answers with the information asked for. The localization now runs fairly quickly (20 Hz) to offer the most recent information possible. All of this information is broadcast over the network. As the deployment node only runs at a fixed, relatively slow rate (0.5 Hz, it would be more efficient to only ask for location information when it needs that information. The localization can then respond by offering the latest information.

5.2.3 Navigation

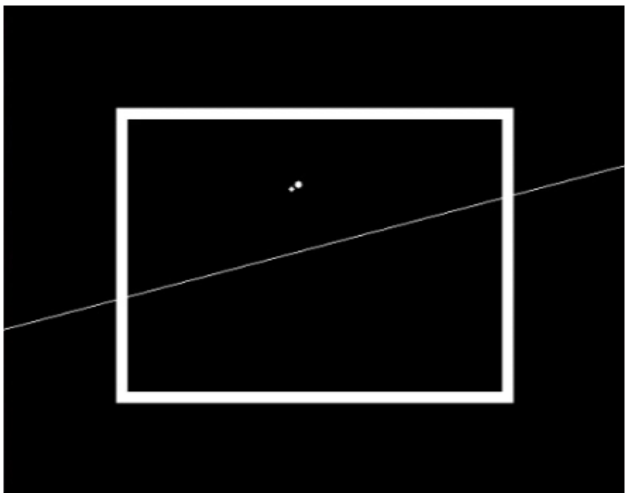
The navigation software is sufficient for testing, but lacks elegance. The raw velocities sent out by this node are jerky as recalculation of the velocities is triggered by changes in position information, which are received at 20 Hz and velocity commands can go from zero to the maximum velocity in one time step (0.05 seconds). A velocity smoother node such as the one readily implemented in ROS may suffice for this purpose.

5.2.4 Deployment

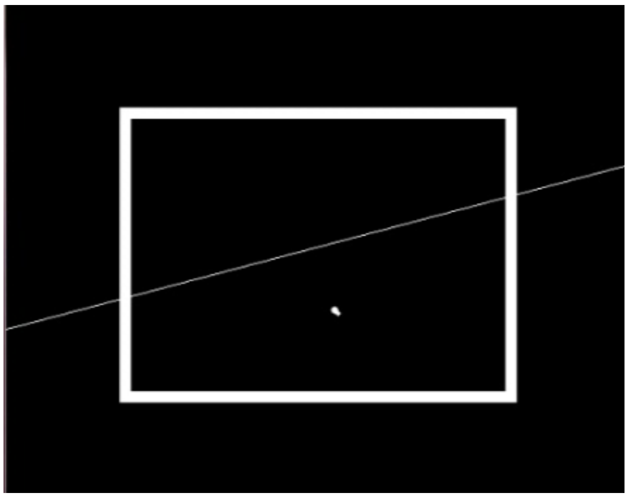
Deployment of two or three robots is shown to be possible. As long as robots receive up to date information, they are capable of navigating to the desired goal position resulting in centroidal Voronoi partitions up to the limits defined by the resolution of the Voronoi image. Results of a deployment with two robots are shown in figures 5.5 to 5.7.



(a) Top view camera image

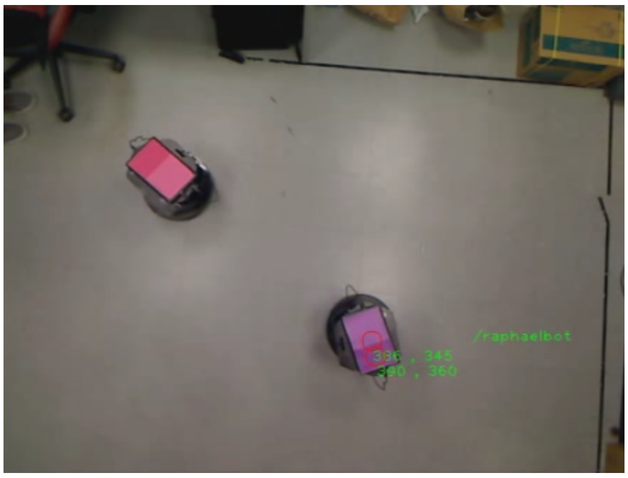


(b) Pink robot's world model

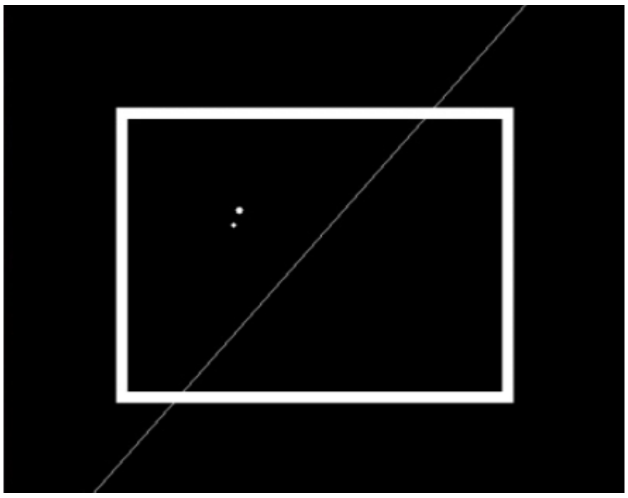


(c) Purple robot's world model

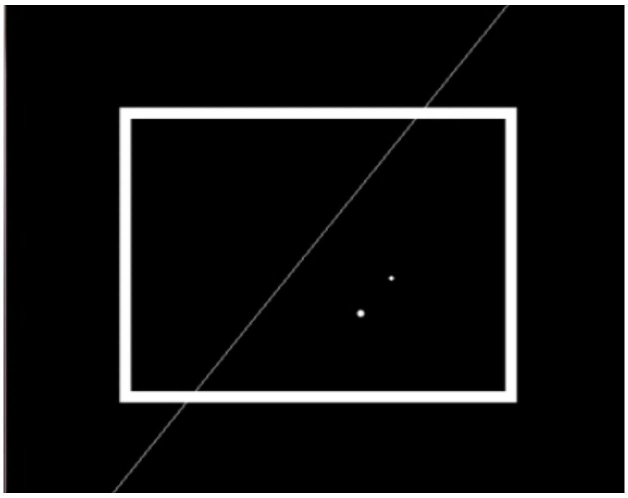
Figure 5.5: *Screenshots of two robot deployment at* $t = 0$



(a) Top view camera image

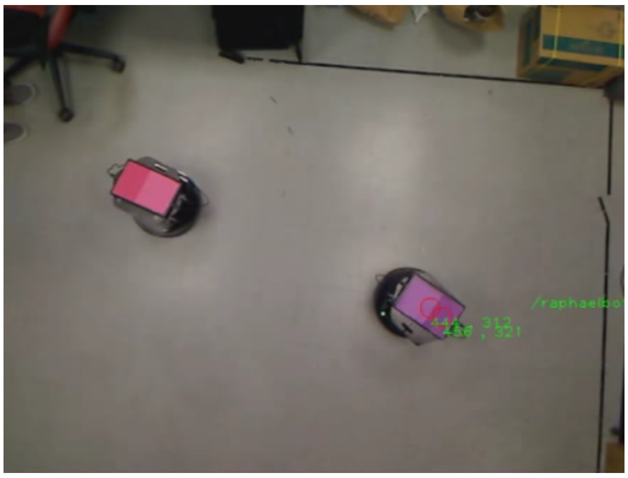


(b) Pink robot's world model

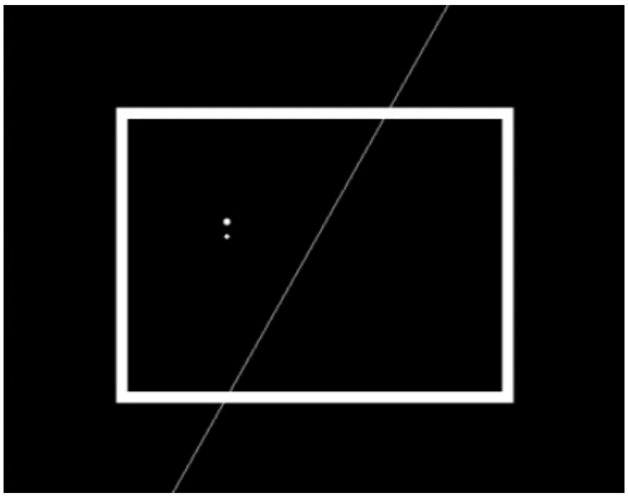


(c) Purple robot's world model

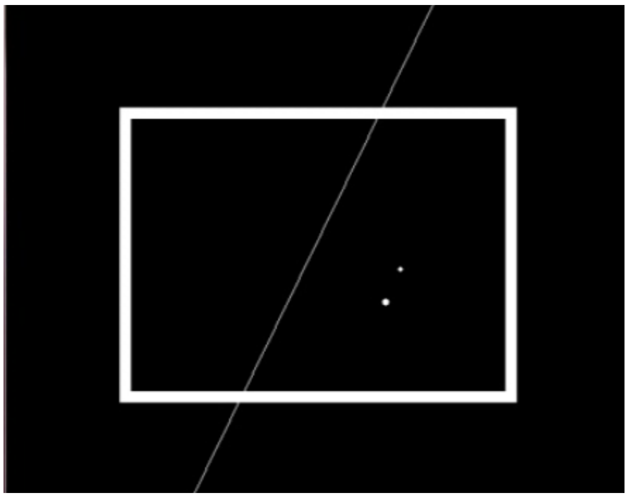
Figure 5.6: *Screenshots of two robot deployment at* $t = 9$



(a) Top view camera image

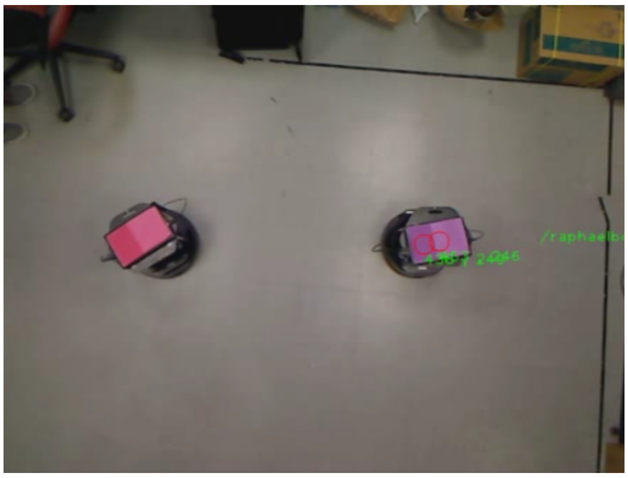


(b) Pink robot's world model

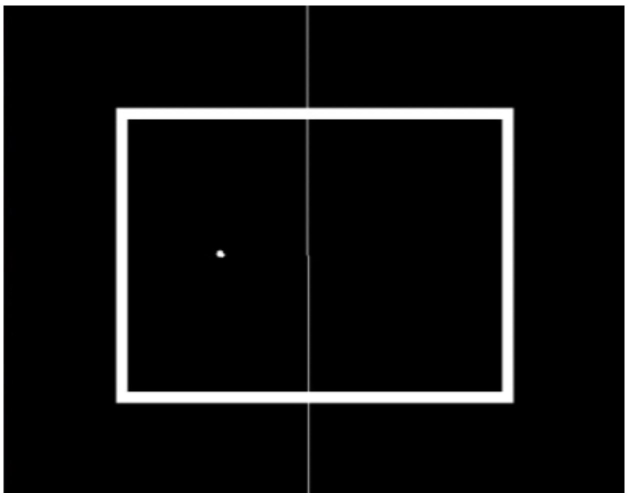


(c) Purple robot's world model

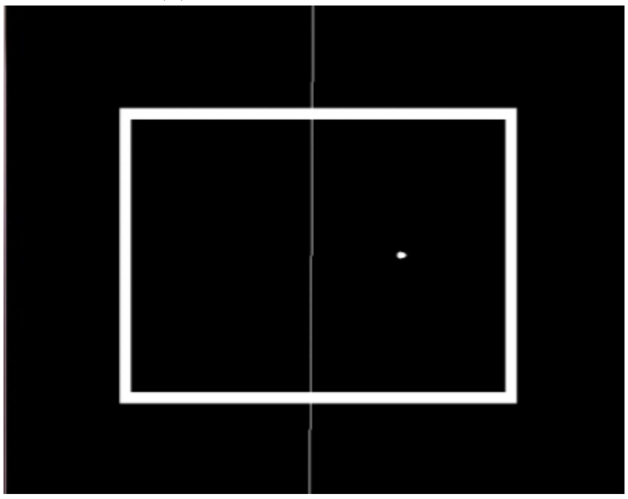
Figure 5.7: *Screenshots of two robot deployment at* $t = 13$



(a) Top view camera image



(b) Pink robot's world model



(c) Purple robot's world model

Figure 5.8: *Screenshots of two robot deployment at* $t = 23$

Chapter 6

Conclusions and recommendations

The goal of this project is to deliver the implementation of at least one coverage control algorithm to deploy a set of Turtlebots, staying as close as possible to a real world implementation on a sensor network. At the start of this work, a primitive program to localize the Turtlebots in their deployment environment was available as well as theoretical algorithms for coverage control of that environment. The algorithm resulting in a centroidal Voronoi partition of the deployment area is chosen for implementation, leading the way for similar algorithms and their implementations to follow.

6.1 Results

In the process deploying Turtlebots, a software architecture was designed, ROS nodes were written in C++ implementations for communication, goal location calculation and navigation, simulations were performed, and finally the implementation was tested on a group of Turtlebots in two different network configurations.

The results of these experiments were successful with respect to the software designed in this work. Problems in communication arise when deploying numbers of robots larger than, say, 3. The cause of this was not yet found, but it may have to do with a high demand on the network. Other issues are found with the localization node. This node works, but is not scalable to larger systems, position data is not very consistent and its performance depends strongly on the lighting conditions.

6.2 Recommendations

These problems directly yield ideas for further work. Localization may be improved by identifying shapes instead of colors, the node should be programmed such that it is scalable to more than the three hard coded color tags it can handle now and a filter should be implemented to make sure that there is always useful position data available. See subsection 5.2.1 for details. Also the navigation software should be improved with a filter to create a smoother velocity command. Refer to subsection 5.2.3 for more on that.

The demand on the wireless network may be reduced to examine if this solves the issues with communication between the the lab PC and the netbooks. A possible way to do that is to use services instead of simple publishers and subscribers. See subsection 5.2.2 for more information.

This implementation is meant to lead similar algorithms to find their way to the hardware. Variations on the distributed optimization problem implemented in this work are therefore also a recommendation for future work. In the current implementation, density functions are not used, but could be implemented by simply multiplying the binary cell image with a density map. The current implementation results in the centroidal Voronoi partition, but as stated in section 2.1, the Voronoi partition where the generators are the circumcenters is also a useful one. Furthermore, the implemented algorithm does not work for non-convex spaces, which may be interesting when optimizing visibility in a building that needs surveillance. A more complicated problem arises when the visibility of a 2.5 dimensional terrain is observed such as in [7]. These are all suggestions for further implementations very similar to the one resulting from this work.

Aside from the recommendations that can be done for further research, some advice may be given regarding practical matters. Keeping track of software developments made by a team of researchers can become increasingly complicated when time passes and more software is developed. A good way to document the current state of software that is under development, as well as thoughts on further improvements is by maintaining a Wiki. Using a Wiki, every person can document his/her own work such that it is always up to date. New contributors such as undergrads or foreign exchange students can easily find where others left off so they can get started more quickly. Information on such a page is not necessarily strictly formal, so it can also be more practical. Manuals such as the one in appendix A and B are an example of such practical, slightly less formal information.

Bibliography

[1] Google Loon. http://www.google.com/loon/. Accessed: September 26, 2014. 1

[2] Andreas Breitenmoser, Mac Schwager, Jean-Claude Metzger, Roland Siegwart, and Daniela Rus. Voronoi Coverage of Non-Convex Environments with a Group of Networked Robots. In *IEEE International Conference on Robotics and Automation*, pages 4982–4989. Institute of Electrical and Electronics Engineers (IEEE), 2010. 4

[3] Francesco Bullo, Jorge Cortés, and Sonia Martínez. *Distributed Control of Robotic Networks.* 2009. 3

[4] Jorge Cortés, Sonia Martínez, Timur Karatas, and Francesco Bullo. Coverage Control for Mobile Sensing Networks. *IEEE Transactions on Robotics and Automation*, 20(2):243–255, 2004. 1, 2, 3, 4

[5] Thomas Curtin, James Bellingham, Josko Catipovic, and Doug Webb. Autonomous Oceanographic Sampling Networks, 1993. 1

[6] Albert S. Huang, Edwin Olson, and David C. Moore. LCM: Lightweight Communications and Marshalling. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4057–4062, Taipei, 2010. IEEE. 8

[7] Ferran Hurtado, Maarten Loffler, Ines Matos, Vera Sacristan, Maria Saumell, Rodrigo I. Silveira, and Frank Staals. Terrain Visibility with Multiple Viewpoints. In *International Symposium on Algorithms and Computation*, pages 317–327, 2013. 28

[8] Eitan Marder-Eppstein. ROS Navigation. http://wiki.ros.org/navigation. Accessed: July 19, 2014. 9

[9] Dominik Moritz. Lloyd's algorithm. http://en.wikipedia.org/wiki/Lloyd's_algorithm, 2013. Accessed: September 30, 2014. 4

[10] Open Source Robotics Foundation. ROS.org. http://www.ros.org/. Accessed: June 30, 2014. 8

[11] Shane O'Sullivan. Masters - Map Building with Mobile Robots. http://www.skynet.ie/ sos/-masters.php, 2003. Accessed: July 19, 2014. 13

[12] Daniel Stonier. RoCon. http://wiki.ros.org/rocon. Accessed: July 19, 2014. 8

[13] Danilo Tardioli. ROS-rt-wmp. http://wiki.ros.org/ros-rt-wmp. Accessed: May 7, 2014. 8

Appendix A

Deployment Manual

Starting a deployment may seem complicated at first, but as soon as you know what's happening, it's actually quite easy. The general procedure can be divided in the following steps:

0. Preliminary: Updating the software
1. Network setup;
2. Environment setup;
3. Starting and calibrating the camera node;
4. Starting the map server;
5. Deploying the robots.

This manual explains how to perform these steps and tries to explain what they do and why they are necessary. But you are reading this because you want to deploy those robots, so let's get started.

A.0 Updating the software

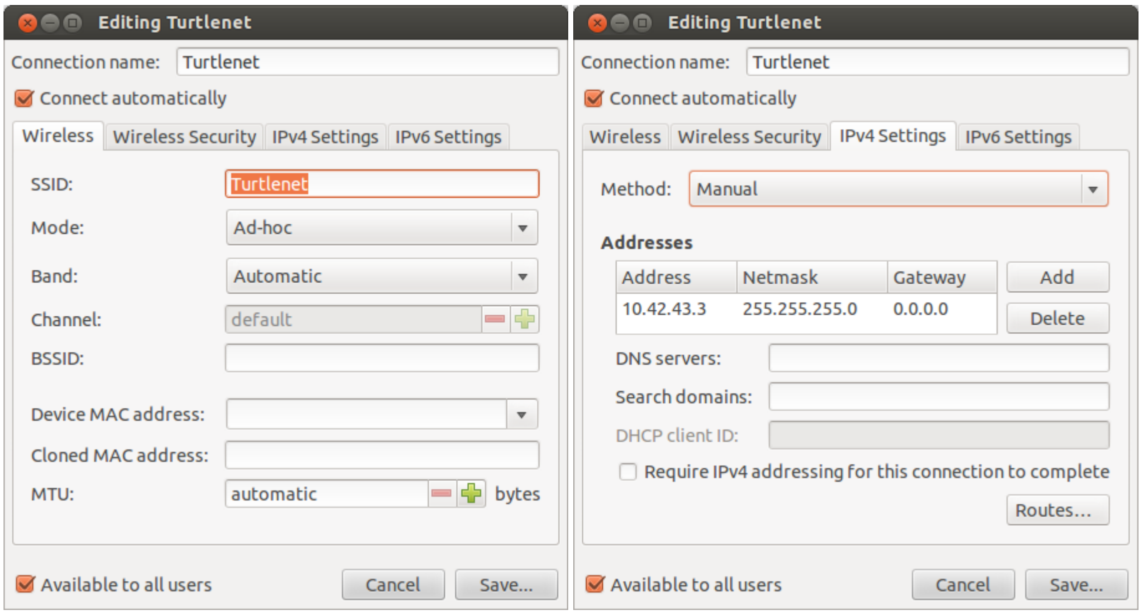
This is a preliminary step if all netbooks are connected to the Internet. If they are not, first go to step 1 to connect to the Internet, then return here and, if need be, go on to step one.

The Ubuntu software is under constant development. To keep up to date, Ubuntu uses a package manager that lets you update the software. The lab software is also under constant development. To keep also that software up to date, the lab uses SVN (more on this in B). Both of these systems need an Internet connection to update. To update the Ubuntu software directly on the netbook, just follow the steps in the graphical user interface (GUI). It is also possible to update through the command line, which is less work if there is more than one Turtlebot to update and more convenient if you are connected to the Turtlebot through ssh. To do this, perform these steps (after ssh into the Turtlebot's netbook):

```
$ sudo apt-get update
$ sudo apt-get dist-upgrade
```

and if, after updating, the system asks for reboot, do:

```
$ sudo reboot
```



Editing Turtlenet

Connection name: Turtlenet

☑ Connect automatically

Wireless | Wireless Security | IPv4 Settings | IPv6 Settings

SSID: Turtlenet

Mode: Ad-hoc

Band: Automatic

Channel: default

BSSID:

Device MAC address:

Cloned MAC address:

MTU: automatic bytes

☑ Available to all users

Cancel | Save...

Editing Turtlenet

Connection name: Turtlenet

☑ Connect automatically

Wireless | Wireless Security | IPv4 Settings | IPv6 Settings

Method: Manual

Addresses

Address	Netmask	Gateway
10.42.43.3	255.255.255.0	0.0.0.0

Add | Delete

DNS servers:

Search domains:

DHCP client ID:

☐ Require IPv4 addressing for this connection to complete

Routes...

☑ Available to all users

Cancel | Save...

(a) Wireless tab (b) IPv4 settings tab

Figure A.1: *Settings for ad hoc network*

A.1 Network setup

At the time of writing, two network configurations have been tried with good results: infrastructure and ad hoc. The first option uses the university's network infrastructure in the form of the UCSD-PROTECTED network. The second option is a peer-to-peer network among the Turtlebots and (optionally, but so far always) the lab PC, without any infrastructure. It is important to note that the network setup for the ad hoc network structure does not offer an Internet connection so if you need to update software, connect to UCSD-PROTECTED. An Internet connection should be possible for an ad hoc network with the right setup, in the right environment, but this has not worked so far (which might have to do with protection on the UCSD-PROTECTED network).

To use the infrastructure setup, just connect every netbook to the UCSD-PROTECTED network.

To use the ad-hoc network setup, it must be configured correctly in the network manager. To configure the Turtlenet network, follow these steps:

- Click the network manager icon in the top-right corner of the screen;
- Select "Edit Connections..."
- In the Wireless tab, look for a network called Turtlenet. If this is already available, you can probably connect to it right away, but to be sure, you might want to check the settings first.
- If Turtlenet is not yet available, click Add; if it is available, select it and click Edit. In the window that appears, make sure that the settings are as shown in figures A.1 and set the Wireless Security to None.
- Click Save and close the Network Manager window.

You are now ready to connect to the ad-hoc network. To do that, follow these steps:

- Click the network manager icon in the top-right corner of the screen;

- Select “Connect to Hidden Wireless Network...” (not directly “Turtlenet”, as this may be a network with outdated settings);
- Select the appropriate network (Turtlenet);
- Click “Connect”.

The computer should now connect to (or set up, if it is the first on the network) the Turtlenet network.

A.2 Environment setup

ROS needs some environment variables to run properly. Especially for network communication, it is important that these variables are set correctly, otherwise the software will not run, or communication fails.

To set up the environment variables, we use the .bashrc file in your Home folder. This file is *sourced* every time you log in or open a new terminal window. This means that the definitions stated in this file are read and used. To open this file, run in a terminal:

```
$ gedit ~/.bashrc
```

or just

```
$ gedit .bashrc
```

if you are in your home folder.

In the bottom of this file, there are several lines that are interesting to us, namely the ones that define the ROS_MASTER_URI and ROS_HOSTNAME. These must be defined differently depending on whether the ad hoc or infrastructure network structure is used.

If you are using the ad hoc network, make sure the following lines are active (not commented with a #)

```
export ROS_HOSTNAME=localhost
export ROS_MASTER_URI=http://[IP of master]:11311
```

where [IP of master] should be replaced with the IP address of the master computer that it has on the ad hoc network. Normally, when deploying with the Calipso lab PC as the master, this should be `10.42.43.1`. Other lines exporting these environment variables should be commented out. It has also been shown that it sometimes works to use `http://[hostname of master].local:11311`, but only under certain circumstances. The IP method is more reliable.

If you are using the infrastructure, use

```
export ROS_HOSTNAME=localhost
export ROS_MASTER_URI=http://[hostname of master].dynamic.ucsd.edu:11311
```

where [hostname of master] is `calipso` for the Calipso lab PC. These changes should be done on every netbook or PC on the network when changing network configurations. After applying changes to the .bashrc file, it needs to be sourced again. This is automatically done in a new terminal window, but open terminal windows each need to source this file before the changes will be used. This can be done by executing

```
export source ~/.bashrc
```

A.3 Starting and calibrating the camera node

Starting the camera node can be done using convenience launch files. To check if all robots are tracked well, run the following command on the lab PC

```
$ roslaunch blueBot localization_demo_multiple.launch
```

Be sure to check the names defined in this launch file before applying it. Also, at the time of this writing, names of robots as well as their thresholding values are hard coded into the localization node. This works, for now, but is very ugly, not scalable and may need to be changed in the future (so feel free to fix it).

If not all robots are found robustly, edit the calibration launch file (localization_calibration.launch) to run the node for the relevant robot and `roslaunch` it. The threshold values can then be changed to work under the current lighting conditions using the sliders. To store these values, they need to be copied to the source code (ugly), which can be found in the src folder of the blueBot package. This package has been Catkinized and put on the SVN (where the package name has changed to Turtlebot_camera_localization to follow Catkin naming conventions). However, at the time of writing, there are SVN conflicts with this package when updating that need to be resolved. The source code in the Catkinized package is outdated, but this can easily be solved by copying the C++ code from blueBot to the Catkin package.

A.4 Starting the map server

The map server is an integral part of ROS, so it can be used on any PC, with or without the lab's custom Turtlebot software. However, a launch file was written to load the right parameters and a map file of the deployment area. This file can be found in the Turtlebot_deployment/launch folder and is conveniently called map_server.launch. The map file it loads can be found in the 'maps' folder of the same package and can be opened and changed if that is necessary. The map metadata must be edited the first time it is used on a computer, because it contains the full path of the map image. The existing map file defines a rectangular deployment area, which should correspond to the taped-off region in the lab. To launch the map server, call

```
$ roslaunch Turtlebot_deployment map_server.launch
```

A.5 Deploying the robots

The robots can now be deployed by running

```
$ roslaunch Turtlebot_deployment deploy_robot.launch robot:=[robot name]
```

on the robot, where [robot name] needs to be replaced with the respective robot name. These names need to correspond to the names used in the localization node, otherwise topic names will not match.

Appendix B

SVN

Apache Subversion, abbreviated SVN, is a version control system much like the better known Dropbox or Google Drive, but has the advantage that files are stored on a private server, which makes it more secure. The SVN for this lab is used for documents such as reports and papers, but also for the software. When all the necessary software for the Turtlebots and the lab PC on the SVN are up to date, one terminal command is enough to update a PC or netbook with the latest software.

An SVN account can be obtained from professor Jorge Cortes. This will let you download the lab folder, make changes, add other files and commit those changes and new files to the server. To download a copy of the lab SVN, open a terminal, navigate to the folder where you want to download this (tip: put it in /catkin_ws, like on the lab computers and the Turtlebots), and execute the following command:

```
$ svn co http://carmenere.ucsd.edu/svn/repos/lab
```

Now, before you go any further and commit changes to the SVN, read the manual on this web page: http://cstwiki.wtb.tue.nl/index.php?title=SVN_instruction. Of course, the checkouts they do in the tutorial are not relevant, but the information given in the manual is very useful! Information on this page may be copied to the project Wiki page.