University of Liverpool

ELEC450

MEng Group Project

Robot Object Search and Retrieval Team

Authors:
Jason Price
Gareth Walley
Peter Hill
Lyudmil Vladimirov
Tsvetan Zhivkov

Supervisors:
Prof. Simon Parsons
Dr. Elizabeth Sklar

May 18, 2015

Abstract

This document contains the report for the Robot Object Search and Retrieval Team. Included are sections outlining the aims and objectives, background theory, methods and design, results and conclusions. The project covers 5 sub-projects; mapping and navigation, object location, multi-robot communication, object retrieval using a robotic arm and iPad interface to display data. The robot team is able to map the arena, and retrieve objects from the arena using pose's uploaded through a server. The future work and conclusions section summarise the report and propose future work.

Contents

1 Introduction **1**
- 1.1 Aims and Objectives . . . 1
  - 1.1.1 Ground-Based Robot Team Objectives . . . 1
  - 1.1.2 Quad-copter Objectives . . . 1
  - 1.1.3 Robot Arm Object Manipulation Objectives . . . 2
  - 1.1.4 Multi-Robot Communication Objectives . . . 2
  - 1.1.5 iOS Application Objectives . . . 2
- 1.2 Industrial Relevance . . . 2

2 Theoretical Background **4**
- 2.1 Robot Localization . . . 4
- 2.2 Robot Mapping . . . 6
- 2.3 Robot Navigation . . . 7
- 2.4 Robot Arm Background . . . 8
- 2.5 Quad-copter Research . . . 11
  - 2.5.1 ARDrone2.0 . . . 11
  - 2.5.2 Nano Quad-copters . . . 11
- 2.6 Motivation to use Natural User Interface . . . 12
  - 2.6.1 Choice of Device . . . 13
  - 2.6.2 iOS Application Objectives: . . . 13

3 Materials and Equipment **14**
- 3.1 Hardware . . . 14
  - 3.1.1 uFactory uArm . . . 14
  - 3.1.2 Turtlebot 2 . . . 16
  - 3.1.3 ARDrone2.0 . . . 17
- 3.2 Software . . . 18
  - 3.2.1 ROS Packages used . . . 18
  - 3.2.2 Other Software . . . 19

4 Design and Methodology **20**
- 4.1 Ground Robot Team . . . 20
  - 4.1.1 Mapping Robot . . . 21
  - 4.1.2 Pickup/Arm Robot . . . 22
  - 4.1.3 Carrier/Bin Robot . . . 23
- 4.2 Robot Arm Methodology . . . 24
  - 4.2.1 Arm Assembly and Modifications . . . 24

4.2.2 Coloured Blob Detection 27
4.2.3 Controlling the Arm 28
4.3 Communications and Server 33
4.3.1 Server State Machine 33
4.3.2 Server NS Chart 34
4.3.3 ArmBot Client NS Chart 36
4.3.4 BinBot Client NS Chart 37
4.3.5 MapBot Client NS Chart 38
4.3.6 FlyBot Client NS Chart 39
4.4 Simulator Research 39
4.4.1 Install and Setup 39
4.4.2 Simulations 40
4.5 ARDrone 40
4.5.1 Connecting 40
4.5.2 Colour Detection 40
4.5.3 Navigation 41
4.5.4 Localisation 41
4.6 iPad Application Methods 42
4.6.1 Design view and properties: 43

5 Results 50
5.1 Ground Robot Team 50
5.1.1 Mapping Robot 50
5.1.2 Arm and Carrier Robots 54
5.2 Communications and Server 62
5.2.1 Design Files 62
5.2.2 Server Results 63
5.2.3 Client Results 66
5.2.4 ArmBot Client Results 68
5.2.5 BinBot Client Results 70
5.2.6 MapBot Client Results 70
5.2.7 FlyBot Client Results 71
5.2.8 ROS Data Transformations 71
5.3 Robot Arm 72
5.3.1 Flashing Code to the Arm 72
5.3.2 Accuracy and Repeatability 73
5.3.3 Final Design of State Machine for the Arm 75
5.4 Simulators 78
5.4.1 Install and setup 78
5.5 Quad-copter 79
5.5.1 Connecting 79
5.5.2 Colour Detection 80
5.5.3 Navigation 80
5.5.4 Localisation 82
5.6 iOS Application 83
5.6.1 Sending pose from iOS Application: 83

5.6.2 Receiving Pose, State and Images from Server 84
5.6.3 Warping Image and Unsuccessful Loading of Image in image-VC: 85

6 Discussion and Conclusions 88
6.1 Future Work 88
6.2 Conclusions 89
References 93

Chapter 1

Introduction

1.1 Aims and Objectives

The aim of the project is to build and program a robot team, using any resources within the laboratory that will be able to locate, pick-up and remove 'objects' from the robot arena. There are 5 main parts to the project, with each part being completed by one person. These are TurtleBot mapping and navigation, Quad-copter navigation and object location, iOS application development for data analysis, communication between robots and robot arm control and object recognition.

1.1.1 Ground-Based Robot Team Objectives

- Develop an approach for localisation, mapping and navigation of an unknown environment.
- Use these methods to develop a:
  - Robot to map the arena and share the map with the other robots.
  - Robot with an arm to move to and pick-up objects in the arena after receiving the locations of the objects from the quad-copter.
  - Robot to collect the object from the arm robot and return it safely to base.

1.1.2 Quad-copter Objectives

- Modify an existing quad-copter to run ROS.
- To implement a navigation stack on the quad-copter.
- The quad-copter must be able to locate objects of interest in the map, and pass their positions to the other robots.

1.1.3 Robot Arm Object Manipulation Objectives

- Integrate the arm with ROS so that it can send and receive data over serial connection.
- Use a camera mounted to the end of the arm to identify objects and the container to place them in.
- Pick-up objects and place them within the container on the carrier robot.

1.1.4 Multi-Robot Communication Objectives

- Develop a communication protocol for data transfer between robots.
- Co-ordinate robot actions using a server state machine.
- Interact with the iOS application to allow the user to control the system when necessary.

1.1.5 iOS Application Objectives

- Graphically display information from the system within an iOS application, such as the position of the robots on the map.
- Allow the user to control the robot team manually when required.

1.2 Industrial Relevance

With some of history's most significant technological advancements taking place over the past decade, one major example being the rapid increase in computer CPU performance, autonomous robotic systems become an increasingly important trend within the modern market environment. Due to this reason, a significant portion of today's research into state of the art robotics focuses on both entirely autonomous systems, as well as systems that promote human-robot interaction, which can provide solutions to a wide variety of emerging tasks.

Some of the most trending research topics, involve the development of efficient algorithms for performing essential robotic tasks, such as those of robot localization, navigation and mapping. Another subset of the undergone research focuses on multi-agent interaction, including inter-robot communication, as well as argumentation and auctioning mechanisms, which can be readily applied to a collective team of robots to ensure successful co-operation of its members. Commercial development makes use of the outcomes stemming from the above research fields, in order to avoid 're-inventing the wheel' and to identify the most suitable components, which shall provide the optimal solution to the application's specification and require the least amount of time and effort in order to be implemented.

The task of locating and retrieving target objects from an unknown environment, using a cooperative team of heterogeneous robots, can be thought of as having a wide range of profound applications. First and foremost, the scenario of emergency situations, where deployment of a human team to perform such a task can impose a great risk on the lives of involved members. One example of such situation is in the case of a terrorist bomb-threat where the detection, pickup and transportation of suspicious objects is required. Another example application could be the retrieval of objects from hostile/hazardous environments, such as a burning buildings or a nuclear plant following a catastrophic event (e.g. fire, nuclear reactor meltdown etc.). Looking at the situation from a different perspective, such a team of robots could be utilized in order to implement autonomous collection and disposal of trash from indoor environments, such as a home or an office.

As it becomes obvious, the different possible applications of the target system to be developed can range to a great degree and the system functionality can be easily extended depending on the exact end requirements for each application. To the best knowledge of the authors, at least at the time of formatting this paper, there do not exist any open-source implementations of a team with the same, or even closely related, characteristics. Thus, the resulting developed system, stemming from the work done in due course of this project, has a great potential of providing the base groundwork and motivation for future research and development.

Chapter 2

Theoretical Background

2.1 Robot Localization

By definition, robot (objective) localization is the problem of estimating a robot's pose within an objective frame of reference, while it moves and senses it's surroundings. With the pressuring need for hands-off autonomous systems, having knowledge of the robot's position inside it's environment becomes one of the most basic requirements. This in turn, creates the need for robots that are equipped with sophisticated sensors and algorithms, that shall allow them to build a sufficiently detailed perception of their environment and make efficient use of this perception to exhibit spatial behaviour.

Sensors are essentially information sources, which observe a certain process and report the outcome in terms of organised and usable data. Just like in all animals, including humans, the information sources available to a robot, can be categorised into two main classes; idiothetic and allothetic. Idiothetic sources (a.k.a. cues), produce data that emerge from internal sensors observing the robot's own behaviour. One major example of an idiothetic source is a servo motor, which provides odometry data computed by keeping track of the angle of rotation or the number of revolutions of a robot's wheels. Even though this could be sufficient to provide an absolute position of the robot within it's environment, it is prone to cumulative error which can grow rapidly (e.g. drifting wheels). On the other hand, allothetic cues provide data which is acquired through observation of the environment using sensors (such as a camera, radar or lidar system). Allothetic sources suffer mainly by the problem of 'perceptual aliasing', where two entirely different places could look, and thus be perceived as, the same. Examples of both idiothetic and allothetic information sources on a popular robot platform, which is also the one used extensively during the course of this project, are shown in Fig. 2.1. Most biologically inspired models, including robots, use a combination of the two sources to achieve spatial awareness. This can be achieved by the use of either Kalman [1] or Particle [2] filters which allow for fusion and correction of pose estimates from multiple sources. As a result, allothetic cues can be used to compensate for cumulative errors from idiothetic sources, while on the contrary

idiothetic information can be used to disambiguate between different locations with great morphological resemblance.

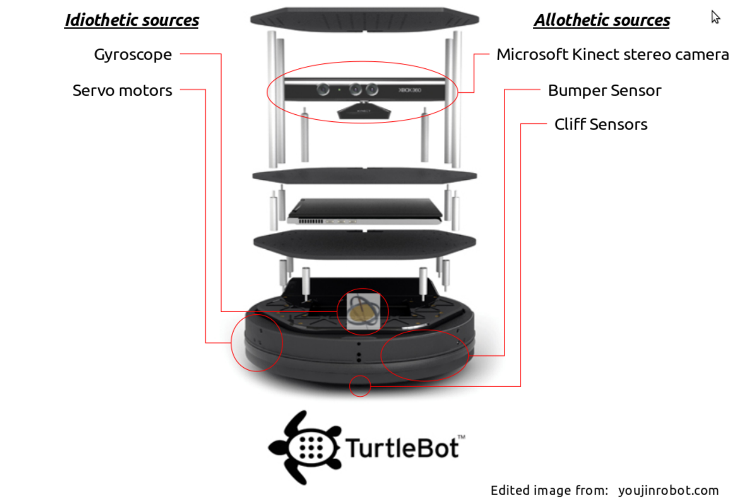


Figure 2.1: Examples of idiothetic and allothetic data sources illustrated on a commercially available development robot platorm.

The robot localization problem can be broken down further into two subproblems. First is the problem of Simultaneous Localization And Mapping (SLAM), in which case the robot has no previous knowledge about it's environment and is required to draw a map -or a floor plan- to represent it. This problem will be more thoroughly discussed in Section 2.2. The second case builds on the assumption that a priory -or previously learnt map- of the environment is already available and tackles the problem of localization within it. One of the most successful localization approaches, namely Monte Carlo Localization (MCL) [3], makes use of probability theory to build models of the uncertainty involved in all aspect of a robot, including the current pose estimate and the data acquired by the sensors. Classical kinematics provides the expected global trajectory of the robot, when certain controls are applied to it. However, this motion is uncertain, since kinematics involve deterministic calculations, which in turn follow certain assumptions that do not necessarily hold in practice. Similarly, there can exist many unforeseeable and uncotrollable factors that may affect the readings coming from sensors, which implies that sensor information is also uncertain. Probabilistic models of these uncertainties provide the basis for application of Bayesian inference, which constitutes the driving force behind localization. MCL, as described by F. Dellaert and D. Fox [3], makes use of the concepts described above and is implemented by recursive application of re-sampling and particle filtering processes to an initial set of randomly generated particles, in order to effectively reject samples with low weights/probability and promote once with high weights. The final result is a relatively accurate estimation of a robot's pose, including estimated covariance, relative to a predefined frame.

2.2 Robot Mapping

As research into autonomous robots -and robotic theory in general- advances, the need for acquiring and storing a more and more accurate perception of the surrounding environment arises. Within nature, evolutionary shaped blind action (such as triggered response) could be sufficient to allow some species to survive. However, in order for a robot -or an agent- to be able to plan ahead, making use of current perceptions and memorized events, while also foreseeing expected consequences, a more cognitive approach is required. This is where representation of the environment using maps comes in the spotlight.

In general, there are two different types of approaches for representing maps. In the first case, that of *topological* maps, the produced map is essentially a graph, in which different nodes are used to represent different places, while the edges between them show the respective paths. Distances between nodes are generally also stored, however this type of map is more concerned with the topological -hence the name- relations between different places rather than the actual composition of the environment. In contrast to the former type, *metric* maps provide a more human-like approach, where objects (i.e. obstacles) are placed with precise coordinates in a 2-dimensional (or even 3-dimensional) plane/grid. Since uncertainty is always an issue when working inside non-ideal environments, many mapping techniques, irrespective of the type of approach, use probabilistic theory to account for fluctuations in the received perceptions. The majority of the most successful mapping approaches involve a combination of both types of map representation in order to capture both the topological and geometrical characteristics of the environment.

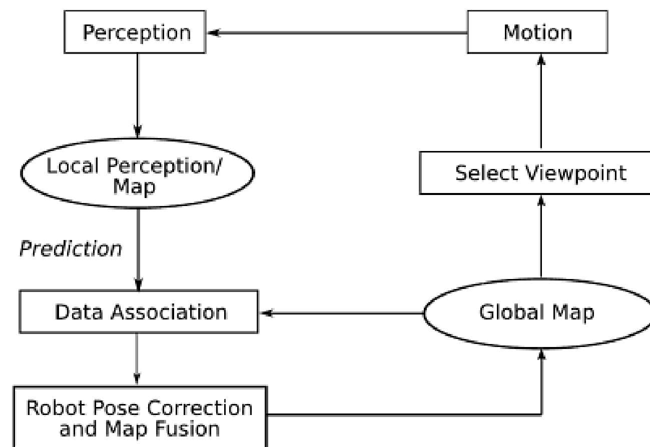


Figure 2.2: The general SLAM process. [4]

Having solved all the uncertainties involved around map representation, the map learning problem can be explained as follows. The robot moves around it's environment and gradually discovers new places within it. Newly discovered objects (i.e. obstacles), perceived from a certain location must therefore be associated and integrated with previously registered ones in a consistent manner. The out-

come of this integration is a map which represents the layout of the whole (or parts of) the surrounding environment. As it becomes obvious, the problem of map learning cannot be separated from the localization process, since information about the robot's position within it's environment, or the constructed map, is necessary in order to effectively perform the relation and integration of new areas. Thus, the problem that needs to be solved is that of Simultaneous Localization and Mapping (SLAM).

From it's definition in the world of computer science and robotics, SLAM is the computational problem of constructing and updating a map, while at the same time keeping track of an agent's location within it. While at a first glance, the problem seems quite similar to the widely known chicken and egg problem, intensive research over the past few decades has given birth to several different algorithms which manage to solve the problem, with high proximity and in tractable time. The techniques used are somewhat similar to the ones for solving the sole problem of robot localization, as described in 2.1, and again involve the use of particle or extended Kalman filters to provide estimates of the robot's location. S. Thrun provides an accurate summary of the different approaches to SLAM, including a rough comparison, in his article [5].The general SLAM process is depicted in Fig. 2.2, where the association and integration of the local perception to the global one becomes clear.

2.3 Robot Navigation

Whereas the term Navigation originally applied to the process of directing a ship to a certain location, over the past century the very same concept has been repetitively applied to all types of mobile devices. The general goal of navigation is, given a set of start-end points on a plane, to find the shortest possible path between them and create a plan in order to guide a mobile entity through it. For any autonomous platform, the ability to navigate it's environment, while maintaining spatial awareness and avoiding dangerous situations (i.e. collisions), is a very crucial, but also quite, complex task.

Robot navigation essentially builds on the two previously mentioned processes; robot localization (see 2.1) and robot mapping (see 2.2) and fuses the information provided by the two in order to achieve its goal. Representation of the surrounding environment using a map provides a robot with the ability to place itself within it and then make plans about it's actions. The navigation process could then be conceptually divided into two sub-processes; Global Path Planning and Local Path Following. Both sup-processes make use of the map and the robot's pose in order to compute their output. The first step involves the computation of the global plan that the robot needs to adhere to in order to achieve it's goal. This plan essentially includes the core trajectory that the robot should follow. Following, during the second step of the process the correct navigation instructions are provided to the motors of the robot by a reactive controller, in order to safely, navigate through the previously acquired path, while using sensory information in order to avoid

any obstacles met on the way. The two sub-processes generally interleave, since generation of an alternative global plan could be required, in the case that a previous one is deemed unachievable or too 'expensive' to achieve.

Even though change of robotic platform would imply that modifications in the navigation process are necessary, the general outline of all approaches follows the concept described in the previous paragraph. Path finding is generally implemented with the use of search algorithms such as several variations of D* [6], A* [7] or Dijkstra's [8] algorithms. Implementation of reactive Path following and obstacle avoidance include the use of concepts such as Vector Field Histograms (VFHs) [9] in order to utilise sensory information and generate a direction for the robot to head in, while velocity space approaches, such as the Dynamic Window [10] or the Lane Curvature [11], are used to perform a search of and generate the commands controlling the robot's movement, such as translational and rotational velocities.

2.4 Robot Arm Background

The four axes are controlled by:

- A motor in the base to rotate the arm 180° about the z-axis
- Two motors mounted to the side of the base to control the position of the end effector in 2-dimensional space (shown in figures 2.3 and 2.4).
- A motor within the end effector to rotate manipulated objects.

The servo motors manipulating the arm are located in the base, providing the required low centre of gravity. The end-effector is kept parallel to the floor by a set of beams connected to the base in parallel with the main beams. The arm has a much simplified work envelope to most robotic manipulators with higher degrees of freedom (see figures 2.5). As such, there is only one combination of joint angles possible to reach each location within the workspace. This makes the overall movement of the arm simple to model in comparison to more complex systems such as 6-axis arms where joints can be arranged in multiple ways to achieve the same positioning of the end effector, and transition from one position to another.

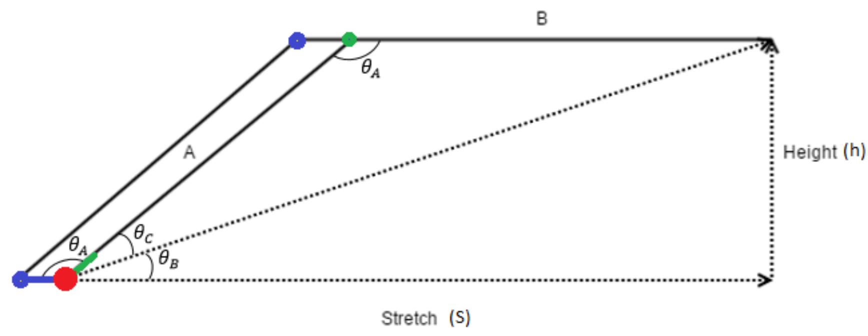


Figure 2.3: Geometry of the robotic arm.

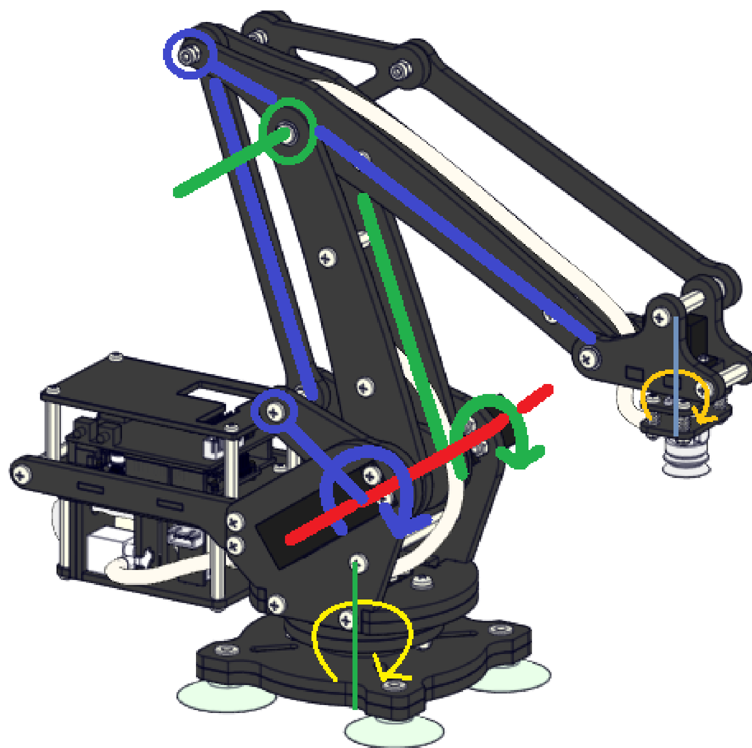


Figure 2.4: uArm with joints highlighted to show axes and joint rotation.

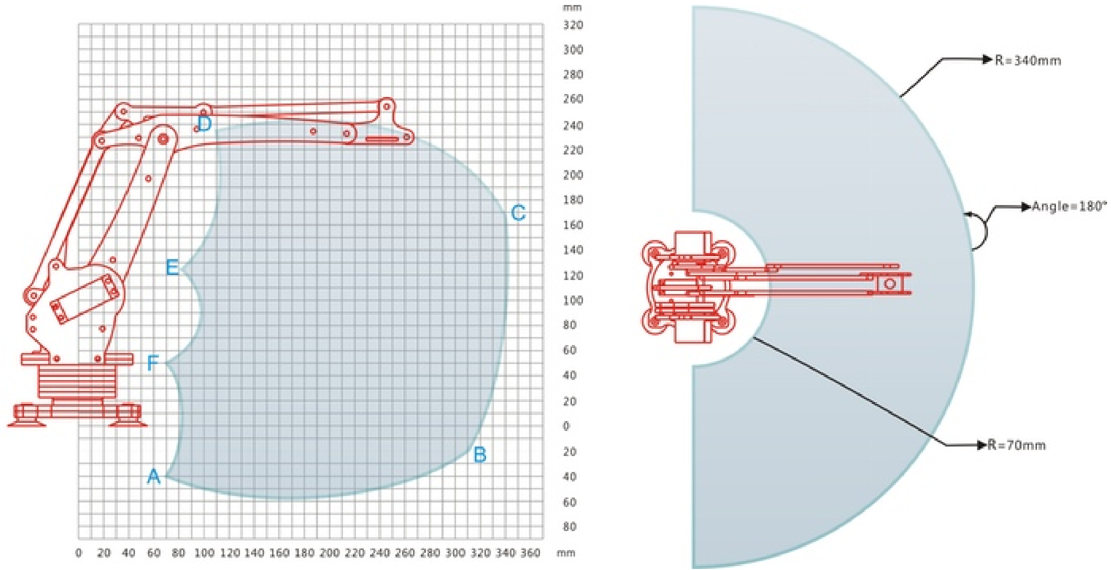


Figure 2.5: Work envelope enclosing the robotic arm.

With one servo controlling the pitch of arm section A (in figure 2.3, rotates about joint highlighted in red, connected to green bar) and another controlling the pitch of arm section B(in figure 2.3, rotates about joint highlighted in red, connected to blue bar) indirectly from the base. From figure 2.3, the required angles θ_A, θ_B, and θ_C for given values of height(h), and stretch(s) can be calculated by using the cosine rule:

$$A^2 = B^2 + C^2 - 2BC\cos(\theta_A) \tag{2.1}$$

Applying this to the model gives three equations which can be rearranged to determine relationship between s, h and each angle:

$$(s^2 + h^2) = A^2 + B^2 - 2ABcos\theta_A \tag{2.2}$$

$$\Rightarrow \theta_A = \arccos\frac{(A^2 + B^2) - (s^2 + h^2)}{2AB} \tag{2.3}$$

$$B^2 = A^2 + (s^2 + h^2) - 2A\sqrt{s^2 + h^2}\cos\theta_c \tag{2.4}$$

$$\Rightarrow \theta_c = \arccos\frac{A^2 + (s^2 + h^2) - B^2}{2A\sqrt{s^2 + h^2}} \tag{2.5}$$

The angle is given by:

$$\tan\theta_B = \frac{h}{s} \tag{2.6}$$

$$\Rightarrow \theta_B = \arctan \frac{h}{s} \tag{2.7}$$

The angle of the servo motor A, controlling section A of the arm can then be determined as:

$$\theta_{motorA} = \theta_B + \theta_C \tag{2.8}$$

To control the movement of section B of the arm, the servo motor B is reversed as it is located on the opposite side to motor A. The angle is therefore determined as:

$$\theta_{motorB} = 180 - \theta_A - \theta_B - \theta_C \tag{2.9}$$

This is implemented in the function `setPosition()`, within the `UF_uArm.h` library provided by uFactory on GitHub [12].

2.5 Quad-copter Research

2.5.1 ARDrone2.0

The ARDrone is the easiest and most popular pre-built quad-copter which has a ROS package to interface with it, this gives it a big advantage, as it is a robust, reliable and well developed platform, which can be easily linked to and used with ROS. The ARDrone has an onboard ARM Cortex A8 which allows linux 2.6.32 to run, this can allow some customisation of on-board parameters. With this size of quad-copter it would be simple to add a raspberry pi 2 that would allow on-board running of ROS, also would allow more complex code to be run on-board due to the added processing power.

2.5.2 Nano Quad-copters

Research into nano quad-copters has shown that they have very little on-board processing, very few nano quad-copters have automated flight, as-well as this they have a very small capacity for sensors, this often leads to nano quad-copters using external sensors to give them position. The crazy flie is a nano quad-copter that is designed to modified, this is advantageous as it has been built with extra flight power than it needs so it can carry extra weight, extra processing power to allow custom code to be run. Many people have used the crazy flie with ROS, however most use external sensors such as the kinect to provide positional data, this is demonstrated in this video [13]. There is a ROS package for the crazy flie, however it would have very limited uses as an automated quad-copter due to its processing power.

2.6 Motivation to use Natural User Interface

The need for a natural user interface in this project is an important stepping stone to improving the interaction and usability of robotic systems in a conventional environment. Human Computer Interaction has become very important with progression in technology and bespoke software, it is more common place to have visual interfaces for controlling home/work automated systems i.e. internet of things'. A GUI's main function is to hide unnecessary complexity from the user, allow for interaction with a computer system and display a simplified visual representation of data using WIMP style interface. The most important aspect of this is that a GUI interface can usually be interpreted by the user with a single glance. In 2007 with the release of the Apple iPhone [14], it very quickly and dominantly pushed forward the need for more powerful mobile devices. This popularised the use of the post-WIMP method of interacting with multi-touch sensitive devices. The post-WIMP method for interface design, also known as NUI goes beyond the simple WIMP method and makes use of other natural human interactions (i.e.touch, verbal commands, gestures etc.) to provide a natural experience [**?**]. The NUI interface reduces the need for full attention even further than that of GUI does for the user, this allows for multiple tasks to be done on the device even with fragmented observation. This kind of interface is becoming more useful to users as it enables, on-the-go updates and increased productivity, permitting attentiveness on surroundings while observing the interface and performing tasks. This new type of interfacing allows greater human integration in interacting with computer systems in many different respects which were not possible before. People with disabilities who did not have full control over such systems can now take advantage of them as an able-bodied person can. The change in NUI method is needed as mobile devices differ from traditional desktop computers [15] i.e.

Mobile devices are:

- Dynamically changing location
- Always connected
- Do not require continuous user attention
- Usually small screen and handheld

The need for improved human interaction with machines is due to:

- Higher training costs for specialised customer/staff (complex systems, steep learning curve)
- Constant customer/staff re-training when system is updated to stay current
- Higher usage and maintenance cost in the long run
- Increased erroneous data, from erroneous input in the system
- Lead to lower adoption, if too complex to use and understand

2.6.1 Choice of Device

Apple's mobile operating system, revealed in 2007 with the release of the iPhone, was quickly extended to support Apple's other mobile device range, including the iPad. The iPad is chosen as the mobile device for the NUI application program. This is the case due to the notable popularity of the device and the large amount of objective-C programming resources that can be found i.e. on iTunes store alone there are 725,000 native iPad applications written [16]. The apple mobile operating system, or also known as iOS, interface is based on *'direct manipulation, multi-touch gesture'* post-WIMP NUI. This style represents the rapid creation of objects and continuous action and feedback provided by the interface. The intention is that the user manipulates these objects, which closely correspond to physical objects.

2.6.2 iOS Application Objectives:

- The aim of writing an application on the iPad, is to visually display information from ROS to a remote device.
- The iPad will allow the user to control the robot team, graphically display sensor data and provide feedback to the robot team from user decisions.
- The iOS-app and the communication server are used to capture and display data, improve data analysis and integration of ROS in a variety of systems.
- The data will be used to display the positions of each robot in the map space and simulate the movement on a map view in real-time, using the converted map.png downloaded from the server.

Chapter 3

Materials and Equipment

3.1 Hardware

Table 3.1: Hardware Equipment used throughout the project

Equipment	Description
Turtlebot 2 platform	Robot Development Kit (see 3.1.2)
Asus Xtion Pro Live	Stereo Vision Camera sensor
Hokuyo URG-04LX-UG01	Laser Rangefinder (lidar)
Dell N5010 laptop	Intel Core i5 @ 2.5 GHz 6GB RAM
HP 250 G2	Intel Core i3 @ 2.4 GHz 6GB RAM
ASUS F201E	Intel Celeron @ 1.1 GHz 2GB RAM
Apple iPad (iOS version 8.0.2)	Tablet to display data

3.1.1 uFactory uArm

Most robotic manipulators are designed to allow for manipulation of objects within a large, hemi-spherical work envelope. This requires that joints can rotate more than 360 degrees. To achieve that, the control servos must be located at each joint, giving 6 degrees of freedom. This makes the arm itself heavier as the servos add significant weight. For the required purpose, at full extension, the arm could easily topple the mobile base it is mounted on. The servos rotating joints nearer the base of the arm would also have greater power requirements as they would also be required to expend more energy to support the weight of servos further up the arm.

Since the task specifically requires an arm to lift an object and place place it into a container (a task that industrial pallet packing robots carry out), an arm with a 4-axis parallel joint construction is ideal. It has the following advantages:

- Lower weight of the robot arm
- Increased payload capacity
- Reduced complexity

These have the effect of giving the robotic arm a low centre of gravity, reducing power requirements, which is desirable when mounting the arm on top of a mobile wheel base. It also allows quicker movements as reduced weight allows faster acceleration from standstill; faster reaction times. Therefore, the robot arm selected for this project is called the uFactory uArm [17], [18] (shown on right of figure 3.1), a small 4-axis arm modelled on the ABB IRB460 [19], an industrial high speed robotic palletizer (shown on left of figure 3.1). The uArm is constructed out of laser cut acrylic, powered by standard hobby servos, and controlled by an Arduino Uno. The physical arm itself has an action radius of between 70mm and 340mm, however the servos are configured to reduce this to 120mm-320mm to prevent the hardware from reaching or exceeding its limitations.

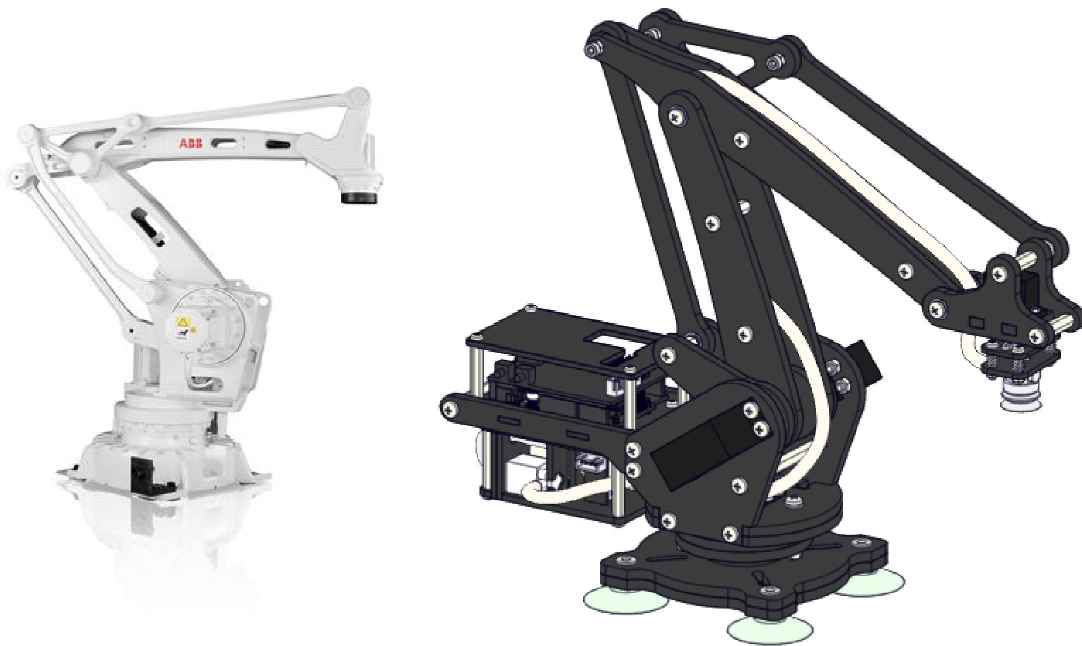


Figure 3.1: ABB IRB460(left)[20], Rendering of the uarm(right).

Not only does this arm have desirable mechanical traits, the software used to operate the arm is open-source and can therefore be modified as required. This would allow for commands to be sent to the arm over a serial connection through a custom serial interface. With this open design, future modifications could be made to the arm with the potential to upgrade servos, pump components, and

electronics.

The specific set of equipment used throughout the entire project is presented and described in Table (3.1). Continuing, the core robotic platforms, available during the design phase of the project, and their respective functionality shall be presented in the following sub-sections.

3.1.2 Turtlebot 2

Turtlebot 2 is a relatively low-cost, personal robot development kit which is widely supported by open-source software. One of it's main advantages is that, at the time of writing this paper, it is the cheapest readily available development platform that embodies the Robot Operating System (ROS) architecture. What is more, there exist a variety of packages within ROS specifically build to work with this platform, which effectively minimizes the time and effort required to set-up and start testing.

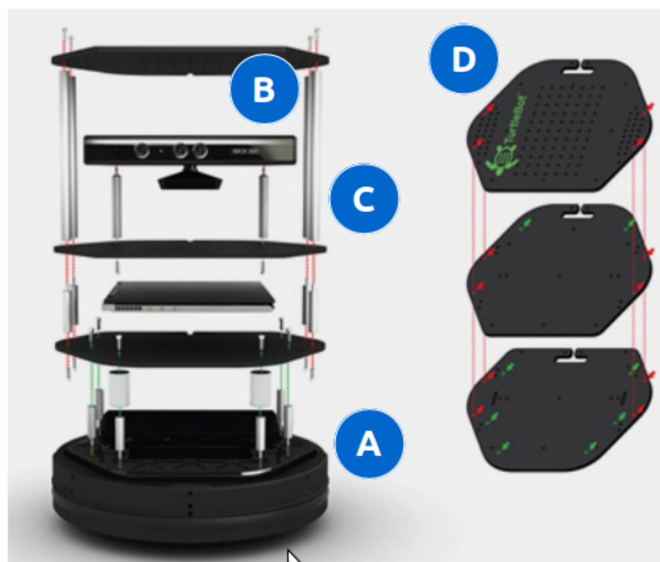


Figure 3.2: Structural layout of the Turtlebot 2 platform. (Edited image from: `http://www.robotnik.eu/mobile-robots/turtlebot-ros/`)

The included hardware can vary from manufacturer to manufacturer and from kit to kit, however a list of the main included modules, which also relates to Fig. 3.2, is presented below:

- Yujin Robot® Kobuki Differential kinematics mobile base (A);
- Microsoft® Kinect (inc. mounting hardware) (B);
- Turtlebot Structure Poles (C);
- Turtlebot Module Plates with 1-inch Hole Spacing Pattern (D);

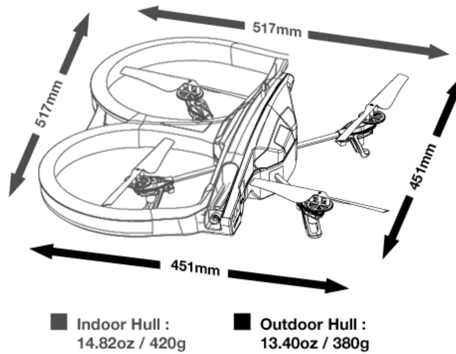


Figure 3.3: ARDrone2.0 chassis http://ardrone2.parrot.com/ardrone-2/specifications/

- Accessories, such as a 4S1P 2200 mAh battery, charger and interconnection wires.

The Kobuki base provides a range of in-built sensors and actuators, as well as power supplies to support expansion with external ones, including charging capabilities for an additional computer (laptop). The highly accurate odometry provided by the base, in conjunction with a factory calibrated gyro sensor, enable the platform to navigate precisely within an environment. Additionally, the Turtlebot structure is highly modular, which allows for a vast variety of possible structural compositions to be constructed. All the above mentioned capabilities, combined with the advantage of full compatibility with ROS and a relatively low cost, make this platform an ideal candidate for educational purposes and research into state of the art robotics algorithms.

3.1.3 ARDrone2.0

The ARDrone2.0 is a light versatile quad-copter designed to be driven via a mobile phone over wifi, also this quad-copter can be used with ROS using the 'ardrone_autonomy'[21] package. This quad-copter features HD video streaming, 1GHz ARM Cortex A8 running linux 2.6.32 allowing custom code to be run onboard on the linux OS, light weight durable chassis that allows for acrobatic flying and gives a long lifetime.[22] The onboard Wifi adapter supports Wifi b,g,n which allows for good quality of connections as-well as good speed, although this does not support the 5GHz bandwidth. The above figure 3.3 shows the removable indoor hull, this is made from polystyrene and plastic, this allows it to be flexible, so if it does collide with something the hull bend, not break and the quad-copter will bounce away.

3.2 Software

3.2.1 ROS Packages used

- **'cmvision' ROS package** [23] - This package provides a node for the Color Machine Vision Project[24], used for fast color blob detection. Additionally, it includes an interface that provides a means for graphically selecting desired colors for blobs.
- **'cmvision_3d' ROS package** [25] - Cmvision_3d uses the /blobs topic produced by cmvision, in addition to 3D data from a registered depth image, to publish both the position of each color blob relative to its camera frame and frames in the tf stack for each color.
- **'gmapping' ROS package** [26] - Provides a LIDAR based SLAM, which allows for the creation of a 2-D occupancy grid map from laser and pose data gathered form a mobile robot.
- **'hector_mapping' ROS package** [27] - An alternative SLAM approach which can be implemented without the use of odometry and produces a 2D occupancy grid map, while also providing 2D pose estimates at scan rate of sensors.
- **'hector_navigation' ROS stack** [28] - This stack provides packages related to the navigation of unmanned ground vehicles in USAR environments. The main packages are the following:
  - **'hector_exploration_planner' ROS package** - A planning library that generates goals as well as paths for the exploration of unknown environments.
  - **'hector_exploration_node' ROS package** - A package that provides a ROS node using the hector_exploration_planner.
- **'navigation' ROS stack** [29] - A 2D navigation stack that takes in information from odometry, sensor streams and a goal pose and outputs safe velocity commands that are sent to a mobile base. The main packages used are the following:
  - **'amcl' ROS package** - A probabilistic localization system for a robot moving in 2D. It implements the adaptive (or KLD-sampling) Monte Carlo localization approach (as described by Dieter Fox in [30]), which uses a particle filter to track the pose of a robot against a known map.
  - **'map_server' ROS package** - Provides the map_server ROS node, which offers map data as a ROS Service. It also provides the map_saver command-line utility, which allows dynamically generated maps to be saved to file. This package is included within the parent 'navigation' stack.

- **'move_base' ROS package** - The move_base package provides an implementation of an action (see the actionlib package) that, given a goal in the world, will attempt to reach it with a mobile base. The move_base node links together a global and local planner, while at the same time maintaining a costmap for each one of the planners, to accomplish its global navigation task.
- **'robot_pose_ekf' ROS package** - Used to estimate the 3D pose of a robot, based on (partial) pose measurements coming from different sources. It uses an extended Kalman filter with a 6D model (3D position and 3D orientation) to combine measurements from wheel odometry, IMU sensor and visual odometry.

- **'turtlebot' ROS stack**[31] - The turtlebot meta package provides all the basic drivers for running and using any version of the Turtlebot platform. The main packages used are the following:
  - **'turtlebot_bringup' ROS package** - This package contains the standard launch files for initializing the nodes required to start the Turtlebot.
  - **'turtlebot_description' ROS package** - This package provides a complete 3D model of the TurtleBot for simulation and visualization.
  - **'MORSE' Simulator** - [32] 'MORSE is an generic simulator for academic robotics. It focuses on realistic 3D simulation of small to large environments, indoor or outdoor, with one to tenths of autonomous robots.'
- **'Gazebo' ROS package** - [33] 'gazebo_ros_pkgs is a set of ROS packages that provide the necessary interfaces to simulate a robot in the Gazebo 3D rigid body simulator for robots. It integrates with ROS using ROS messages, services and dynamic reconfigure.'
- **'ardrone_autonomy' ROS package** - [21] 'ardrone_autonomy is a ROS driver for Parrot AR-Drone quadrocopter. This driver is based on official AR-Drone SDK version 2.0.1. The driver supports both AR-Drone 1.0 and 2.0. ardrone_autonomy is a fork of AR-Drone Brown driver. This package has been developed in Autonomy Lab of Simon Fraser University.'

3.2.2 Other Software

- Xcode 6.1 (Object-C),
- GCDAsyncSocket library,
- Python Socket and Threading libraries.

Chapter 4

Design and Methodology

4.1 Ground Robot Team

During the design phase of the project, it was decided that the ground-based team would be composed of a group of three heterogeneous robots; a) the Mapping Robot, b) the Pick-up/Arm Robot and c) the Carrier/Bin Robot. All three robots shall use the Turtlebot 2 platform as the base structure. However, each of the three robots should have a distinct architecture which shall allow it to complete a unique task. This shall involve correct rearrangement of the corresponding Turtlebot platforms, in order to allow for the robots to conform to specification.

One other design problem, which shall be common for all the robots, is the choice of a suitable sensor, which shall provide the 'eyes' of the system. Optimally, when building a 2-D map or localizing inside one, a sensor is required that can provide distance/depth information with relatively good accuracy and precision. Ultrasonic sensors are generally the easiest approach to this problem, however they are typically most suitable for short range sensing and the uncertainty involved in the measurements is quite immense. Thus, considering the dimensions of the Turtlebot 2 platform and the equipment readily available in the laboratory, two main alternative types of sensors where identified: a) Stereo Vision Camera (Microsoft Kinect/Asus Xtion) or b) Lidar (Hokuyo URG). The perceived depth-image of the 3-D cameras can be segmented to provide a horizontal 2-D line, in order to simulate a 2-dimensional beam, similar to that received by a lidar. With a quick glimpse through the technical specification of each component, the advantages of the lidar become relatively clear, given the significantly wider FOV (240° compared to 57°) and sensing distance range(2cm - 5.6m compared to 0.7m - 6m). However, in an effort to identify the optimal approach to combining the available physical resources to the open-source ROS compatible packages, the suitability and performance of both types of sensors will be evaluated for all the different robot types, and the obtained results shall be reported.

4.1.1 Mapping Robot

The Mapping Robot was chosen to be the first robot to explore and simultaneously create a 2-dimensional map of the unknown environment. Once the environment has been fully explored and a complete map has been produced, the map shall be forwarded to the rest of the team, through the server, to provide the main means of localization. Finally, upon confirmation that the map has been correctly received, the robot shall return to it's initial position. As it becomes clear, the robot shall be required to have the ability to navigate it's way safely through an unknown environment, while simultaneously mapping and localizing itself within it.

Within the ROS community, there exist a number of different approaches to 2-D SLAM, each one with it's own advantages. The two most successful, and well documented, packages that have been identified are: a) gmapping and b) hector_mapping. According to documentation, the two packages follow two slightly different approaches to SLAM, they are however, easily interchangeable in terms of interfacing. One key difference between the two packages, is the fact that hector_mapping does not make use of the odometry data provided by the motors, which can be advantageous in cases of robots that lack this kind of information, but could have a severe impact if odometry information is essential. Due to the fact that both packages have been used intensively over the past, by a wide range of people, this has resulted to a significant enrichment of the available documentation. This documentation will be very helpful in the case of troubleshooting or interfacing issues, during the development phase of the project. In order to reach a final decision, the performance of both packages shall be thoroughly tested and analysed, in combination with each on of the available sensors, and the results shall be documented.

The final decision to be made is that of navigation. Again, one simple approach would involve tele-operation of the platform by wireless means. However this solution, apart from striping away any sense of autonomy from the robot, it is also prone to fail in environments where performance of wireless communication with the platform is detrimented by external factors. The next step from simple tele-operation of the robot, would be to implement a point to point navigation system which shall allow a human to simply pass a goal pose for the platform to move. Again this scenario would involve wireless communication with the platform, however the amount of information communicated shall be significantly reduced and possible latencies would only be observed in the process of sending/receiving a pose, rather than the navigation process itself. The dominant ROS approach to point to point navigation within a (partially) known environment is provided by the move_base ROS package. In addition to being a finely documented package, move_base also comes with premade launch files, specifically tuned for the Turtlebot 2 platform, to allow for a quick start up.

Finally, a further improved approach would require the robot to navigate completely autonomously, in which case the only communication required would be the 'start-end' signal, followed by the inevitable map transmission. The ROS meta package 'hector_navigation' seems to be the only formal implementation of

this concept within the ROS framework, making use of a frontier based approach in order to autonomously identify unvisited regions on the map and, consequently, plan a path in order to navigate to them. Unlike all the packages considered so far, 'hector_navigation' does not come with great documentation and there is not much evidence of it being widely used. However, since this feature is a trending requirement for modern autonomous systems, efforts shall be made in order to incorporate it within the design.

Having completed all the above, a fully detailed state machine model for the desired behaviour of the Mapping Robot shall be developed and implemented within ROS. The overall behaviour of the robot shall be tested against the specifications and the results shall be analysed and presented.

4.1.2 Pickup/Arm Robot

The determined task for the Pickup/Arm Robot, involves the searching and picking up of a target object, from within a directed place on the map. Initially, the robot shall wait for the Mapping Robot to finish its task, after which it should use the acquired map to localize itself within it. Once the Copter Bot has finished scanning the arena for target objects and a target location has been specified, the Arm Robot should proceed to navigating to it. Continuing, once the object has been located, the Carrier Robot shall be invoked and upon arrival the robot shall pickup the object and place it within the container of the Carrier bot. Finally, upon disposal of an object the robot should remain in it's position until another target location has been received or all target objects have been removed from the arena.

The ROS 'navigation' meta package, provides a number of packages which shall allow the robot to utilize the received map from the Mapping Robot, in order to both localize and navigate within the environment. To begin with, the 'map_server' package, via the 'map_server' node, provides the functionality of making a previously created map accessible to multiple robots. This is implemented by converting the map - which is saved locally as a coupled set of an image and a configuration file - into a data type which is ROS compatible and subsequently publishing this information to a delegated ROS topic. Continuing, the 'amcl' package makes use of the published map and, given an initial pose estimation, considers odometry data together with the perceived image of the environment in order to continuously localize within it. Finally, having achieved correct localization of the robot, the previously mentioned 'move_base' package shall be used again, in order to provide the means of point-to-point navigation within the environment. The ROS stack 'hector_navigation' is not considered in this case, since the robot shall be required to travel from/to specific positions in the environment, rather than having to explore an unknown area. What is more, all internal packages of the stack have been strongly coupled, a fact which makes the 'hector_exploration_node' and 'hector_exploration_planner' packages very difficult to interface with individually.

As previously mentioned, color blob detection was chosen as the mechanism for identifying target objects within the environment. The 'cmvision' package provides

a ROS wrapper for the approach developed by the Colour Machine Vision Project, which allows for effective and efficient 2-dimensional tracking of colour blobs, using almost any conventional camera. Furthermore, since distance estimation through processing of a simple image is a highly complex and computationally expensive process, an alternative approach to tracking the 3-D position of objects needed to be identified. As an extension to the standard 'cmvision' package, 'cmvision_3d' uses the output of the former package and the input coming from a depth-image - such as the ones produced by the Microsoft Kinect and Asus Xtion devices - in order to effectively compute the corresponding 3-D position of a tracked blob, relative to the objective frame of the camera. Making use of this package, however, would imply the use of a Stereo Vision Camera as a compulsory requirement for the developed Pickup Robot Base structure. As a result, the object detection capabilities of both the Microsoft Kinect and the Asus Xtion devices shall be tested and the results shall be reported.

Upon having implemented and tested all the approaches mentioned above, a complete state machine model for the desired behaviour of the robot shall be created, based on which, a ROS compatible implementation shall be developed. Finally, the correct operation of the robot shall be evaluated and respective results from test runs shall be presented.

4.1.3 Carrier/Bin Robot

The Carrier/Bin Robot was included in the ground-based team composition to provide the means for transportation of the target objects. Just like in the previous case, the robot shall initially wait to receive the map produced by the Mapping Bot, upon which the localization process can be initialized. Once the Pickup/Arm Robot has located an object, the Carrier Robot shall be invoked and should begin navigating towards the Arm Robot's location. Upon arrival to the target location, the Carrier Robot should co-operate with the Arm Robot in order to collect the target object. Finally, once the object has been successfully collected, the Carrier Robot shall make its way back to its initial position, where it shall remain until invoked again.

As it is obvious, one first requirement for the respective robotic platform is to be equipped with a suitable container, where objects can be securely placed. Continuing, since the robot shall be required to cooperate with the Arm Robot, that implies the need for recognition between the two robots. The chosen solution approach to this problem was again colour blob detection. By placing unique colour identifiers on each different platform, the robots are given the ability to distinguish one another from the rest of the environment. Optimally, depth information can also allow the two robot's to accurately track the position of position of each other, when within FOV. As a direct outcome, both robots can reason and plan about their interactions with each other. Thus, once again, the use of a Stereo Vision Camera sensor becomes a necessary component of the overall robot structure.

By close observation of the Carrier Robot's basic behaviour and functionality, it can be identified that it resembles, to a high extent, that of the Arm Robot.

This implies that the core components that are used to provide the desired characteristics to the Arm Robot, thus all the ROS packages, mentioned in Section 4.1.2, can be re-utilized and combined in the same manner as before, in order to provide the underpinning software platform, based on which, the final (and fully developed) Carrier Robot behaviour shall be programmed. Having completed all the above, a state machine model of this behaviour can be produced and a ROS compatible node, implementing that model, shall be created. Finally, standard testing procedures on the behaviour and functionality of the developed robot shall be carried through, in order to evaluate the robots conformity to specification, and the obtained results shall be documented.

4.2 Robot Arm Methodology

4.2.1 Arm Assembly and Modifications

The arm was ordered as a kit to be assembled. To assemble the arm, detailed instructions are provided on the uFactory website [34]. It is vital that the servos are aligned properly during assembly, otherwise calibration will fail. The following steps should be carried out once the arm is assembled:

1. Download the latest Arduino software to program the Arduino board
2. Install FTDI Drivers (The board is a custom built Arduino that uses an FTDI FT232RL chip for serial communication)
3. Install uArm Arduino Libraries (can be downloaded at GitHub page[12])
4. Connect the Arduino Board via USB to PC
5. Connect power adapter to the motor shield (NOT TO ARDUINO) to power the arm
6. Program the calibration example sketch from uArm Library to Board

(These are listed in greater detail in the Getting Started Document[35]).
Calibration is required after assembly. The arm will make a beeping sound when it is turned on signalling that it must be calibrated. This can be done by following instructions in the getting started documentation[35]. The calibration data is stored in the EEProm memory so that even when the board is turned off and on, the data is kept. Once calibration is completed, the arm will no longer beep during initialisation. The arm is now ready to be programmed.

Like many small microcontrollers, the Arduino is coded using a set-up function and a super-loop. The set-up function is executed on start-up and the super-loop is then looped until the board is powered down or reset. In order to interface the arm with ROS, the package Rosserial was used. The package comes with an Arduino library that allows for the set-up of a ROS node on the Arduino. This node

then communicates over serial with a python node that is run within the ROS environment on the PC. Once set-up, this allows data to be sent and received by the arm using standardised message formats. An outline of the Arduino code is given in figure 4.1; a fully commented implementation can be obtained on GitHub [36].

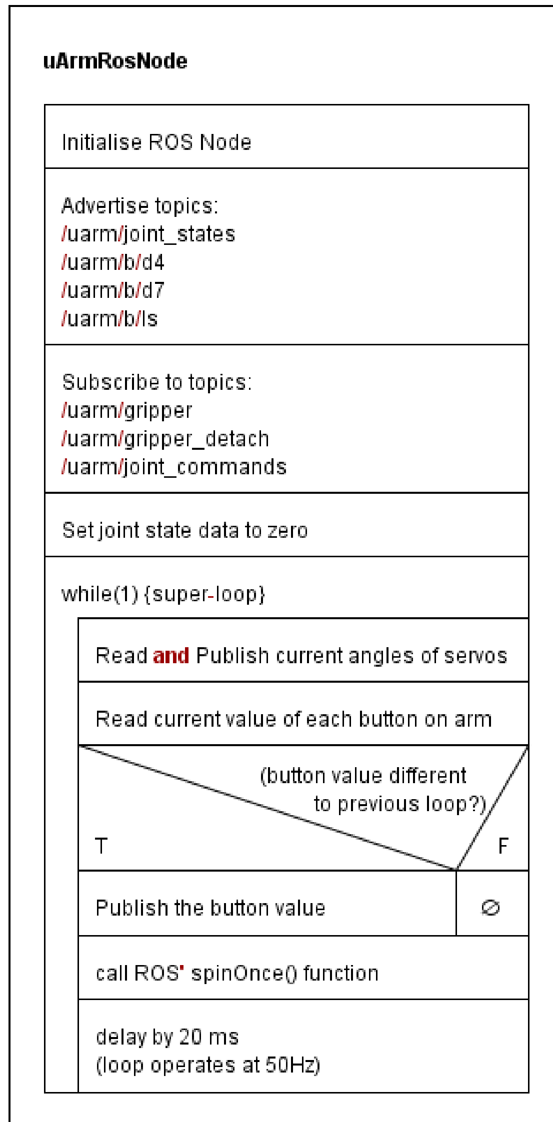


Figure 4.1: Nassi–Shneiderman diagram of Arduino Node.

Since one of the objectives for the arm was to have a visual feedback mechanism to help with alignment and object recognition, a camera can be affixed to the end of the arm. At first, the TechNet C016 USB camera[37] was selected as it was compatible with Ubuntu, a small form factor, and automatic white balance with built in LEDs which would help with colour recognition. The outer casing was removed and part of the monitor mount was attached to the arm as a mounting platform for the camera's bare circuit board. With further testing however, the camera did not operate as required with excessive tearing when the arm was moving quickly due to low frames per second. The LEDs also proved to cause problems when attempting to recognise coloured blobs in the image, washing out some of the colour and thus removing the required information from the image. Also, the camera could only conceivably be mounted to the end of the arm, which meant it could easily collide with the back of the Bin robot when attempting to place the block. With these problems in mind, a camera with better white balance, 30

fps, and better form factor was chosen; the Logitech C270 HD Webcam[38]. Once stripped of its outer casing, the camera's circuit board is long, with the camera itself located at one end. This allows the camera to be mounted on the side of the arm, providing the camera with a view of the container whilst the arm is holding an object, without increasing the length of the arm. This modification is shown in figure 4.2.

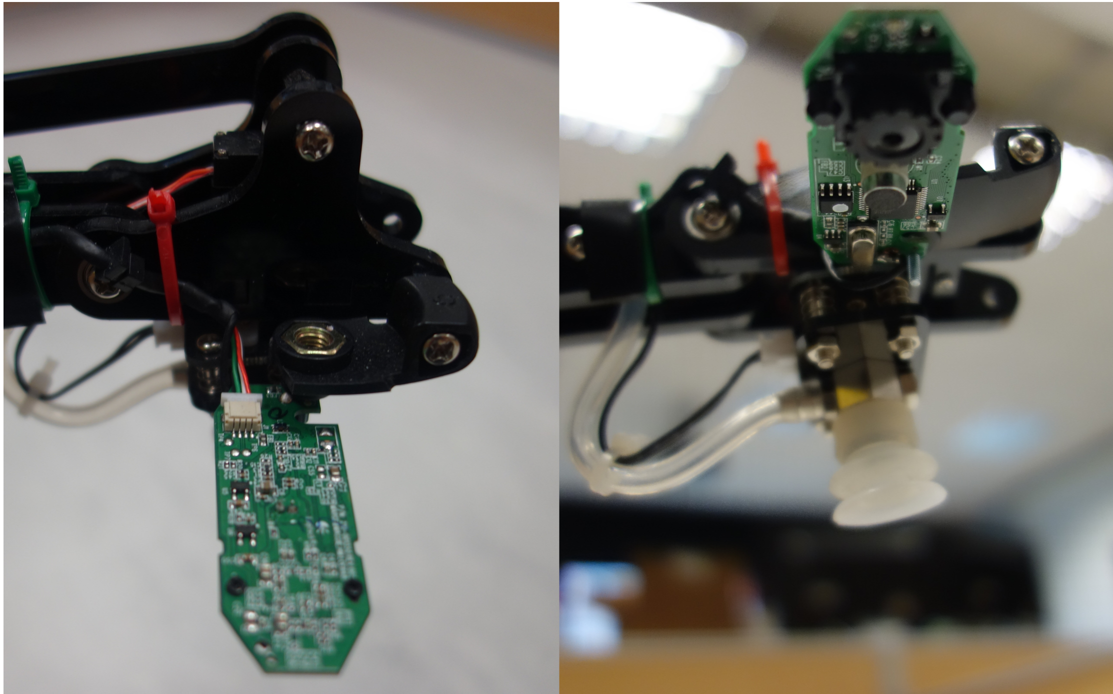


Figure 4.2: Camera attached to end of robot arm.

4.2.2 Coloured Blob Detection

As with other parts of this project, detection of coloured blobs in the image streaming from the camera on the arm is crucial to the positioning of the arm. The package CMvision is a ROS node that, given an input raw image feed from a camera, will locate coloured blobs in the image. It then publishes information about every blob of a given colour in the image. This information includes an array containing the x,y coordinates of the blobs and their areas (as well as other data). The area can be used to filter out blobs that either be too small or too large to be the object the arm is searching for. The message structure of the blobs message is as follows:

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
uint32 image_width
uint32 image_height
uint32 blob_count
cmvision/Blob[] blobs
  string name
  uint32 red
  uint32 green
  uint32 blue
  uint32 area
  uint32 x
  uint32 y
  uint32 left
  uint32 right
  uint32 top
  uint32 bottom
```

CMvision has a node called `colorgui` that can be run to determine the settings for colour detection. The object used in this project is a 70mmx70mmx70mm cube painted pink which is a colour that can be recognised relatively easily even in varying lighting conditions. To ensure that it does recognise it, the program allows you to calibrate for varying lighting conditions, and shows a box around any blobs it has recognised. The values it outputs can be put into the `colors.txt` file which is then used by CMvision as a list of colours to look for.

4.2.3 Controlling the Arm

The positions passed to the arm are in the form of an array containing stretch, height, base rotation, and hand rotation. The stretch and height values are converted by the method shown in section 2.4. The arm then reads the current position of the servos and publishes it. This data however does not match the data sent, and as such cannot be compared like for like. Also, it seemed from testing the arm, that the value returned from the servos was the target position it was moving to, not the current position of the motor. This means that the feedback values are useless for determining if the arm has reached its target position. It was decided that this would therefore be simulated by sending incremental movements to the arm, and keeping track of the current position in the controlling program. These values are therefore only an approximation of the current position of the arm, however for the purposes of this task, this is sufficient.

To control the arm, a python class was developed containing functions operate on the current estimated position. These tasks are:

- `Move`: calculates incremental changes in position to move the arm toward a given target at a given speed.
- `AtTarget`: Checks if current working position of controller is equal to target position.
- `UpdatePosition`: Given a set of incremental changes, adjusts current estimated position of the arm.

- `Error`: Given a colour of blob, and the required target position of that blob, calculates the error between blob's current position and target.
- `CenterOnBlob`: Given a colour of blob in image, and error between where it is and where it should be, calculates incremental changes to align the blob to target.

Outlines of these functions are shown as NS charts in figures 4.3, 4.4, 4.5, 4.6, and 4.7.

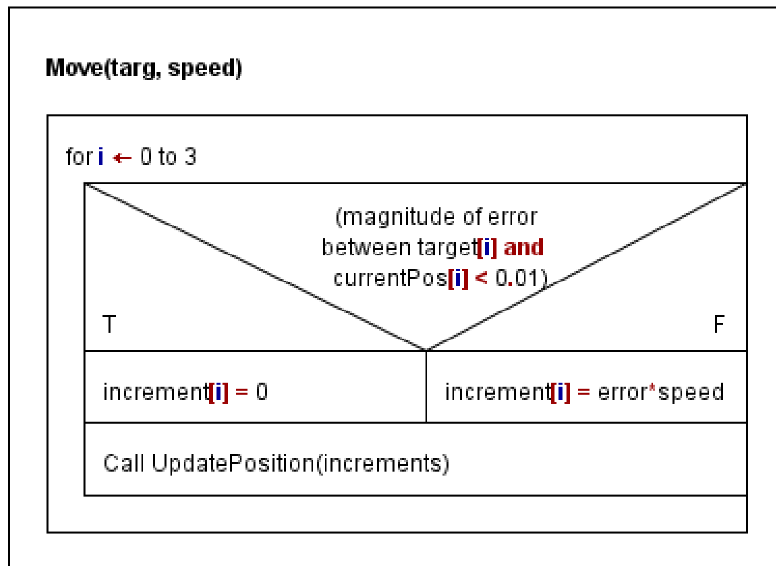


Figure 4.3: Nassi–Shneiderman diagram of `Move` function.

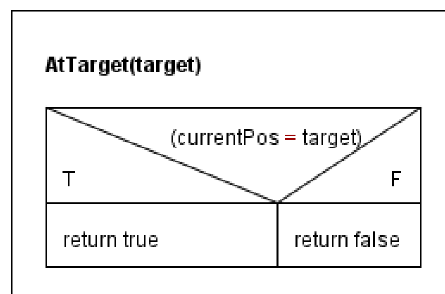


Figure 4.4: Nassi–Shneiderman diagram of `AtTarget` function.

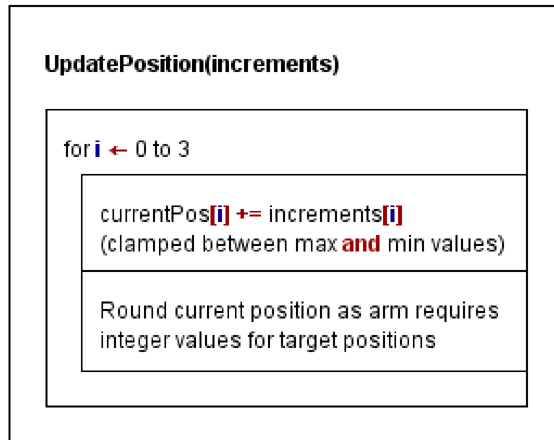


Figure 4.5: Nassi–Shneiderman diagram of `UpdatePosition` function.

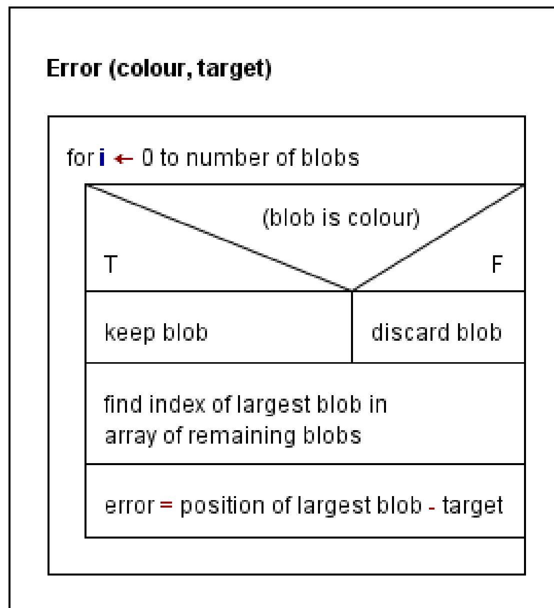


Figure 4.6: Nassi–Shneiderman diagram of `Error` function.

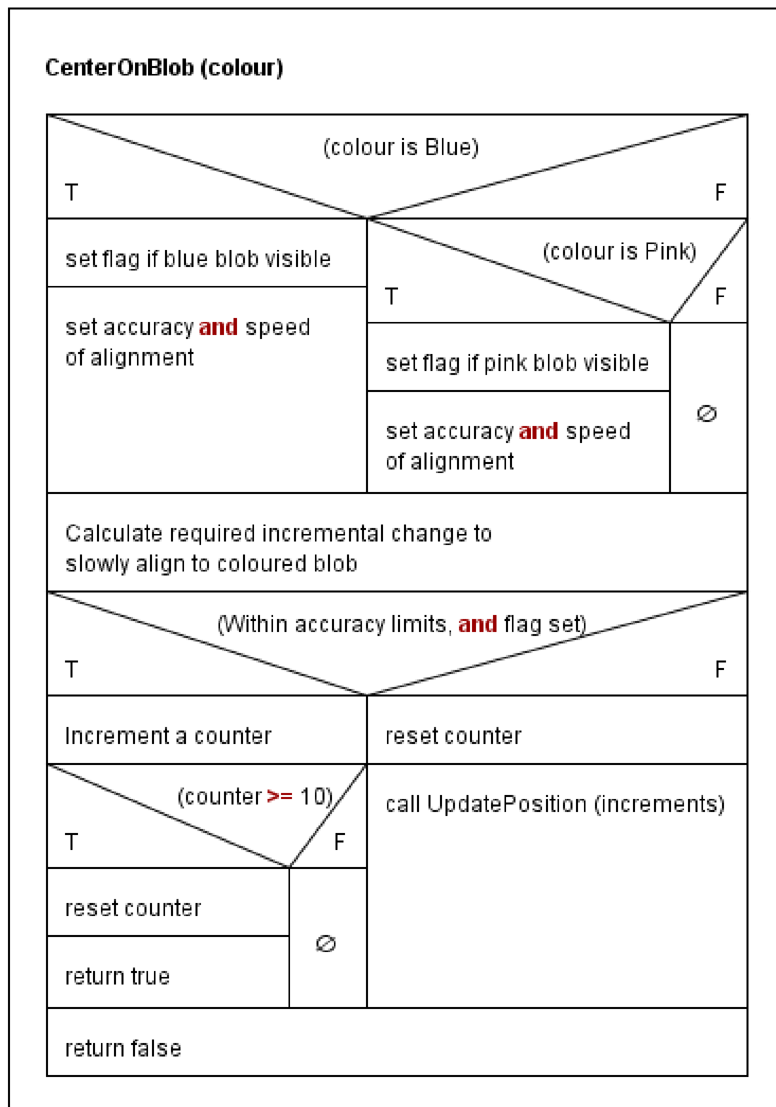


Figure 4.7: Nassi–Shneiderman diagram of `CenterOnBlob` function.

As well as these functions, various functions are required to publish to topics. Subscriptions are handled by callback functions that read the data from their topics and store it in class-wide variables. All of this functionality is implemented in python, in `ArmNode_1_5.py`.

4.3 Communications and Server

4.3.1 Server State Machine

The server requires a state machine to progress the object finding operation. The state machine outlines the progress of the whole process, demonstrating what conditions are necessary to progress. The server FSM is displayed in figure 4.8.

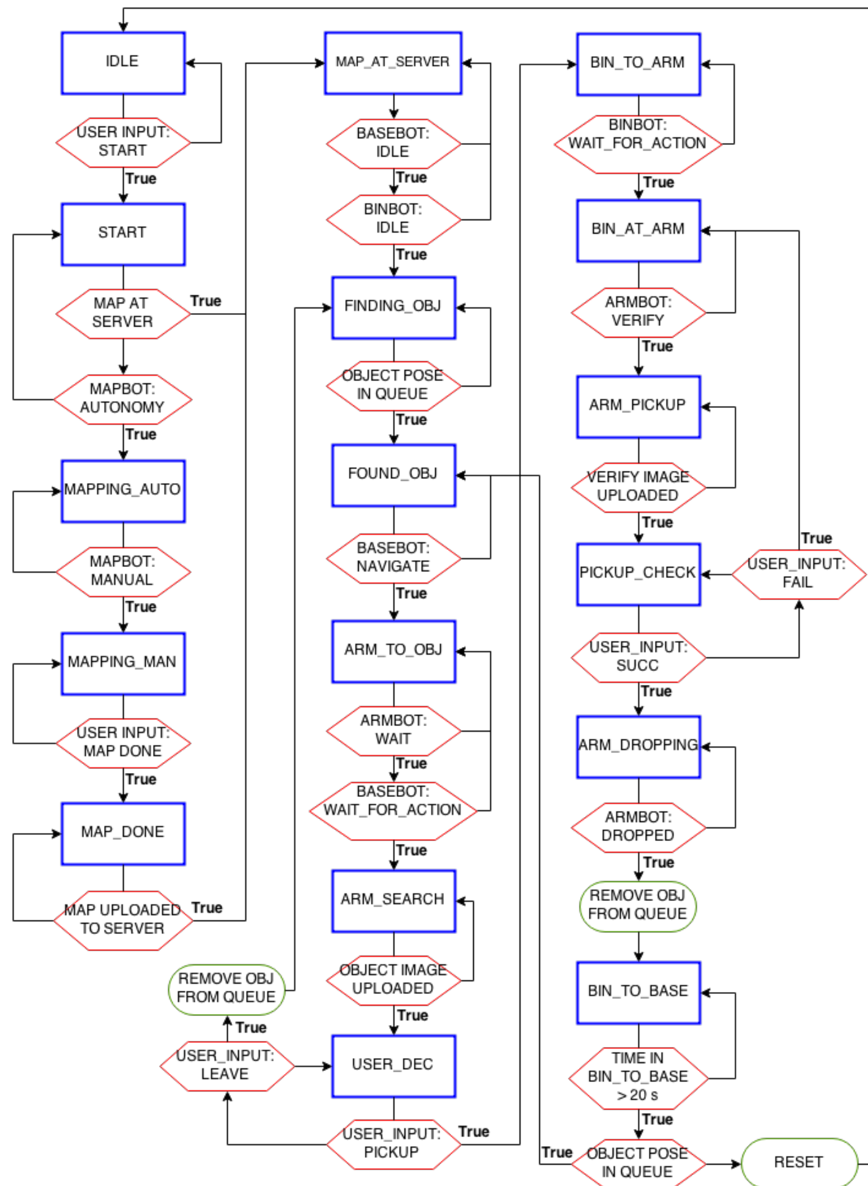


Figure 4.8: Finite state machine demonstrating the operation of the server and the conditions/transitions.

4.3.2 Server NS Chart

Aside from the FSM, the server needs the functionality to send and receive the appropriate messages. 3 main threads are needed for the server operation; a server thread to listen for client connections and assign sockets, a new thread for each incoming client connection and a thread to refresh and verify all server variables. An NS chart for the main server thread is displayed in figure 4.9. The client connection and data time-out thread NS charts are displayed in figure 4.10 and figure 4.11 respectively.

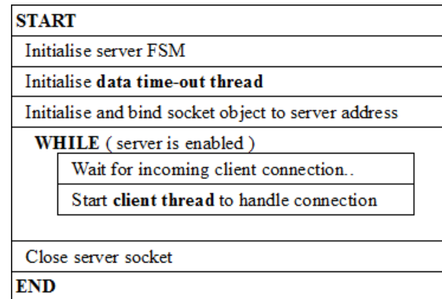


Figure 4.9: NS chart for the main server thread.

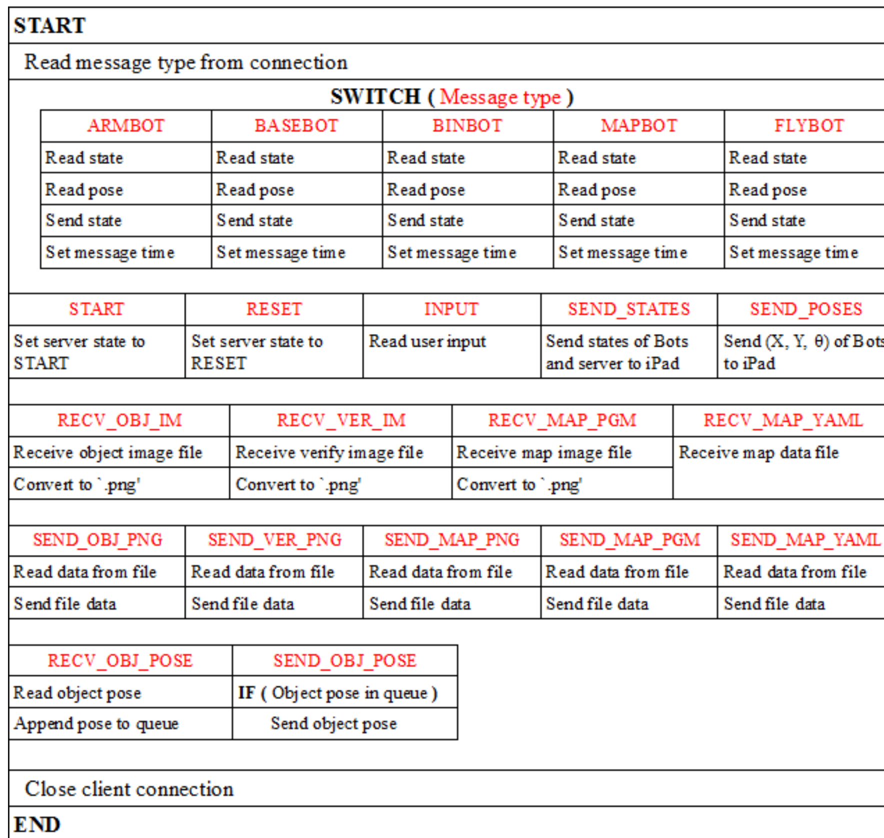


Figure 4.10: NS chart for the client connection message handling thread.

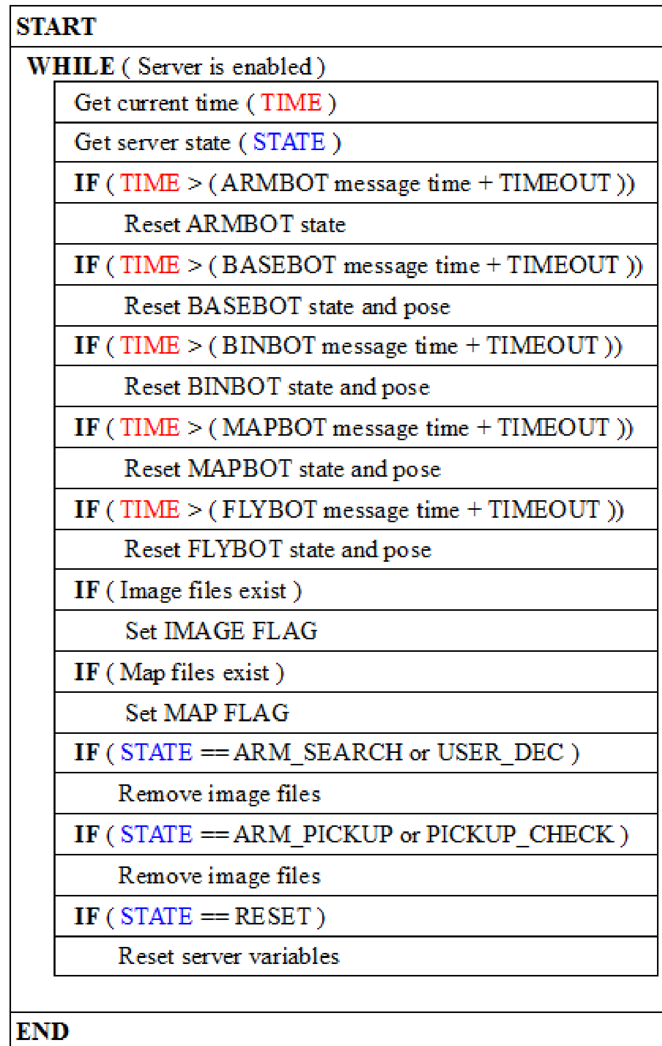


Figure 4.11: NS chart for the variable time-out code.

4.3.3 ArmBot Client NS Chart

The ArmBot client node will run on a ROS core, and utilise the ROS functionality to incorporate the required multi-process needs. The ArmBot client runs a number of tasks in parallel, and needs to:

- initialise and complete actions based on the server state;
  - move to object pose;
  - download map from server;
  - read server state;
- subscribe to ArmBot state topic;
- subscribe to BaseBot state topic;
- subscribe to BaseBot pose topic;
- subscribe to ArmBot camera image data topic;
- upload data to server.

The NS charts for ArmBot client are displayed in figure 4.12.

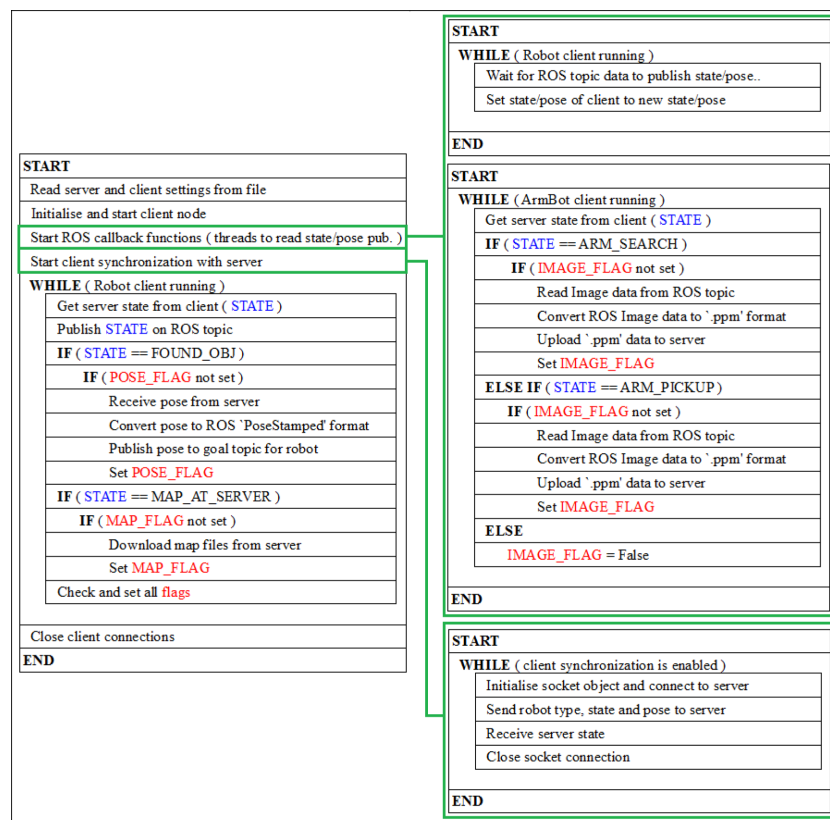


Figure 4.12: NS chart for the ArmBot client node.

4.3.4 BinBot Client NS Chart

The BinBot client node is identical to the ArmBot client node, with exception of the functionality to upload images from the robot. The BinBot client needs to:

- initialise and complete actions based on the server state;
  - move to object pose;
  - download map from server;
  - read server state;
- subscribe to BinBot state topic;
- subscribe to BinBot pose topic;
- upload data to server.

The NS charts for the BinBot client are displayed in figure 4.13.

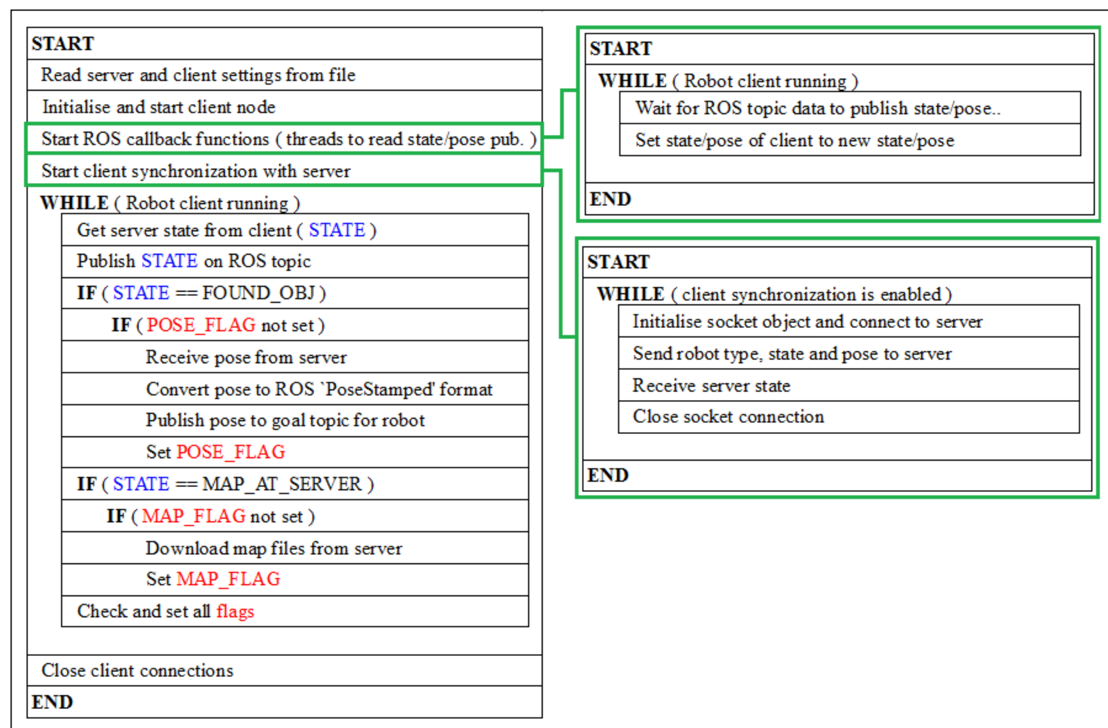


Figure 4.13: NS chart for the BinBot client node.

4.3.5 MapBot Client NS Chart

The MapBot client node is simpler than the Arm and Bin nodes, and needs to:

- initialise and complete actions based on the server state;
  - upload map files to server;
  - read server state;
- subscribe to MapBot state topic;
- subscribe to MapBot pose topic;
- upload data to server.

The NS charts for the MapBot client are displayed in figure 4.14.

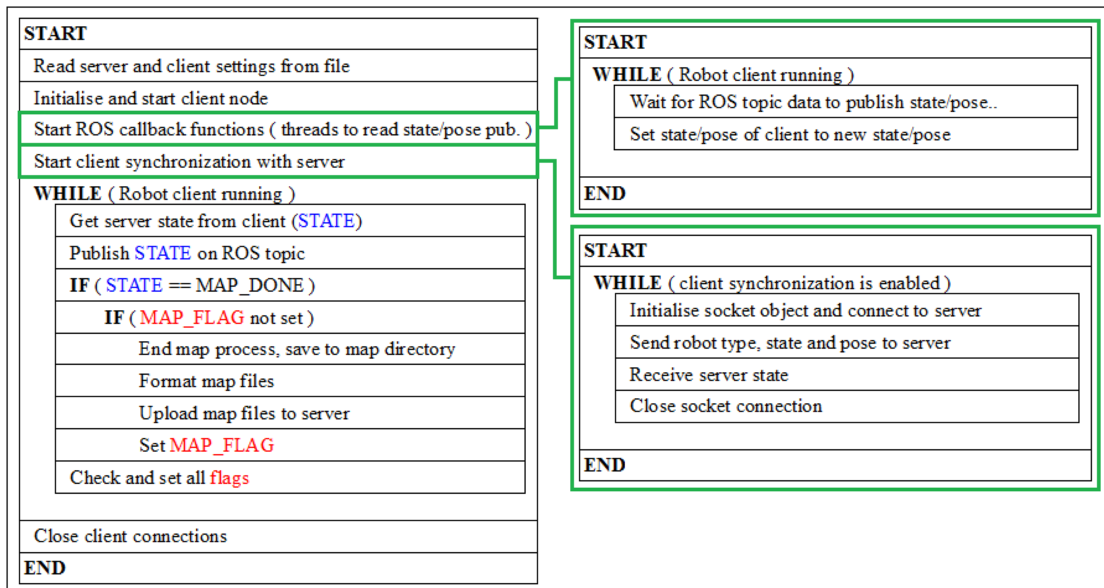


Figure 4.14: NS chart for the MapBot client node.

4.3.6 FlyBot Client NS Chart

The FlyBoy client node needs to:

- initialise and complete actions based on the server state;
  - download map from server;
  - read server state;
  - upload object pose to server;
- subscribe to FlyBot state topic;
- subscribe to FlyBot pose topic;
- upload data to server.

The NS charts for the FlyBot client are displayed in figure 4.15.

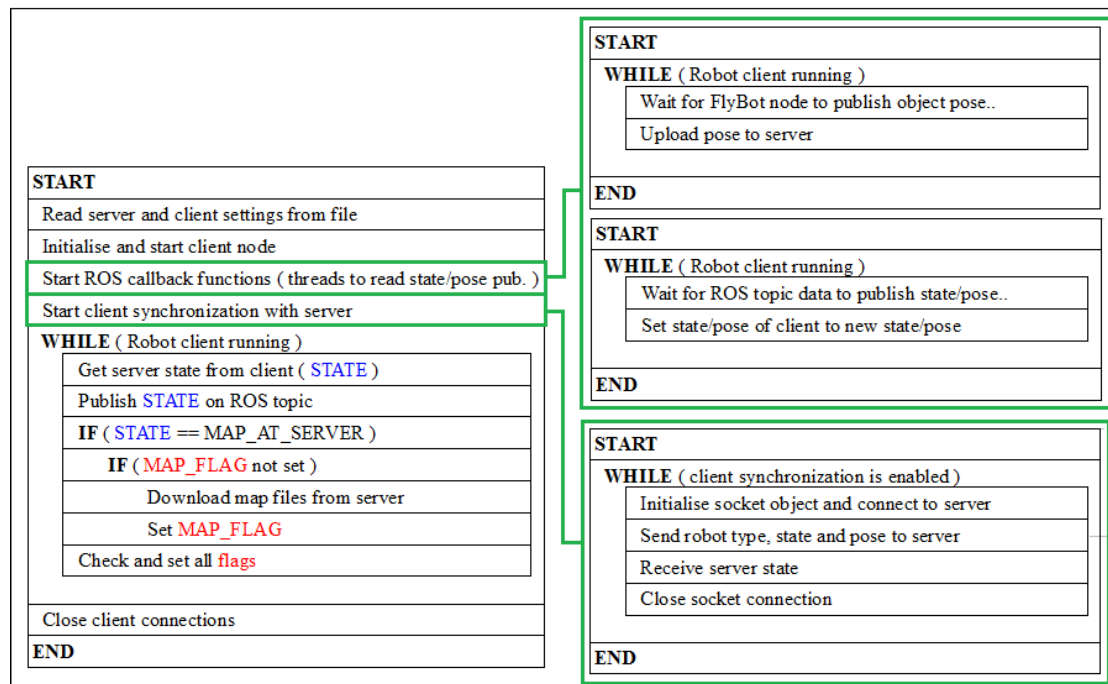


Figure 4.15: NS chart for the FlyBot client node.

4.4 Simulator Research

4.4.1 Install and Setup

A variety of simulators will be tested, the more sophisticated the simulator is the more realistic it will be. Two simulator packages will be investigated, the MORSE simulator and the gazebo simulator supported by ROS. These simulators will be

tested for ease of use, compatibility and predefined utility. Gazebo has a lot of support and predefined simulations as it is available through ROS and is widely used for simulations with ROS. Where as MORSE has less support due to it being newer and less developed. However MORSE utilises the blender game engine for 3d rendering and for 3d modelling, blender is well developed and a very stable platform. This could give MORSE higher accuracy and stability, using a more sophisticated 3d rendering and modelling.

4.4.2 Simulations

These simulators will both be tested by using the 'hector_mappping' and 'move_base' packages to navigate and map an area. The same parameters will be used for the packages on both simulators this will show if either perform differently, with regards to CPU usage, map quality, problems running with ROS.

4.5 ARDrone

For the ARDrone to function as part of ROS based team, it will need to be able to interface with ROS to allow it to use ROS messages to communicate with the rest of the team. Its task in the team is to locate the garbage and give a pose which it close to the garbage to the robot team, to do this the quad-copter will need to know the garbage's coordinates relative to the teams map.

For all testing on the ARDrone the tum_simulator [39] package will be used this contains some basic simulations and setup files for the ARDrone in gazebo, this simulation generates all the data as the ARDrone would.

4.5.1 Connecting

To connect the ARDrone to ROS the 'ardrone_autonomy' package will be used this allows a ROS device to connect to the quad-copter via wifi and broadcasts its data to ROS. This package will be tested on a variety of devices for compatibility, if this is package is compatible with ARM devices this will allow the quad-copter to have on-board intelligence.

4.5.2 Colour Detection

To detect the object in the arena colour detection will be used on the camera facing downwards, this will be done using the ROS package cmvision. An extra node will need to be developed that will receive the colour blob from 'cmvision' and convert it into a pose to send to the team.

4.5.3 Navigation

Initially a basic controller will be developed that converts an input from PlayStation controller using the ROS 'joy' package to target velocities for the quad-copter. This will be used in initial tests of the quad-copter as it allows for human correction.

For navigation the quad-copter will need some external feedback of its position, as any on-board sensor cannot track the quad-copters position over a long period of time. For this coloured markers will be used, which have known coordinates. The controller will be used to generate linear velocities for the quad-copter based on what state and room the controller is in. The controller has a set path/sequence of markers to follow by storing the current room the same colour can be used many times, so long as the next coloured marker is different to the current colour. This allows a path to be made around the arena only using 2 different colours. The quad-copter will move towards the next coloured marker in the sequence, when the marker is in sight of the down facing camera, the controller will align the quad-copter over the marker then update the quad-copter's 2D position to the markers position.

Another possible controller that will be tested is to use 2 simple PID controllers to generate x and y velocities based on the quad-copters current x and y position and a given target position. This controller allows poses to be given to the quad-copter which the quad-copter then moves to, this controller takes advantage of the fact the quad-copter can move at any direction in a 2D plane.

4.5.4 Localisation

Localisation will be needed to allow the ARDrone to communicate the objects position to the team. A node will be created to use the quad-copters linear velocity target to track the quad-copter's position, this should be accurate so long as there are only negligible other forces acting on the quad-copter. This node will publish the quad-copters position as a tf, this allows the quad-copter's position to be accessed by anything in ROS. This will be achieved by subscribing to the "cmd_vel" topic, and summing the linear velocities while scaling them with the update period of "cmd_vel". This node however also needs a mechanism to allow updating of position, this will be done by subscribing to the topic "/ardrone/poseUpdate", whenever a pose is given to this topic, localisation node will update its position to the poses x and y coordinates.

To test localisation the node will be run, the position will be monitored while the quad-copter is flown around. When it updates over the coloured markers that will show the error in the calculated position.

4.6 iPad Application Methods

Application Design

- First storyboard design of View controller (ViewController.h/ViewController.m)

  The first storyboard is used as a reference design, therefore it has poor design layout. It does not have well distinguished features or a good layout of the properties in the view and it has poor visual display of data. Initially a button-touch-event will open a separate popover view showing robot movement data and options for the server. Each image button corresponds to a different robot, when touched it will display the popover view tapping anywhere on the screen after or touching done will pop the view off. The limitations of this method are self evident, as it is noted that only a single robot's data can be viewed at a time and additionally shows options available for only that robot. Using this single popover-view method hides the options available for every other button which is not selected by the user.

- Final storyboard design of View Controller (ViewController.h/ViewController.m)

  The view controller is updated to display all the data from the server about the robots, all in the main view as can be seen in Figure 1. The option of a user inputting any data via a text-box on the application, is removed and instead requires only a user to touch a button or perform touch and drag gestures in response to an event/decision. As in the first storyboard design, the map view location is similar but the positions of any buttons and labels are now all at the top of the main view. This design decision improves a lot of what was wrong with the initial storyboard. It gives the user a full view of all the data, the properties have more importance as they are placed higher on the application screen and it is better visualisation for the user. For good design and future reference a higher priority placement of buttons eliminated any typing issues in a text-box where the virtual keyboard can cover the text-box.

4.6.1 Design view and properties:

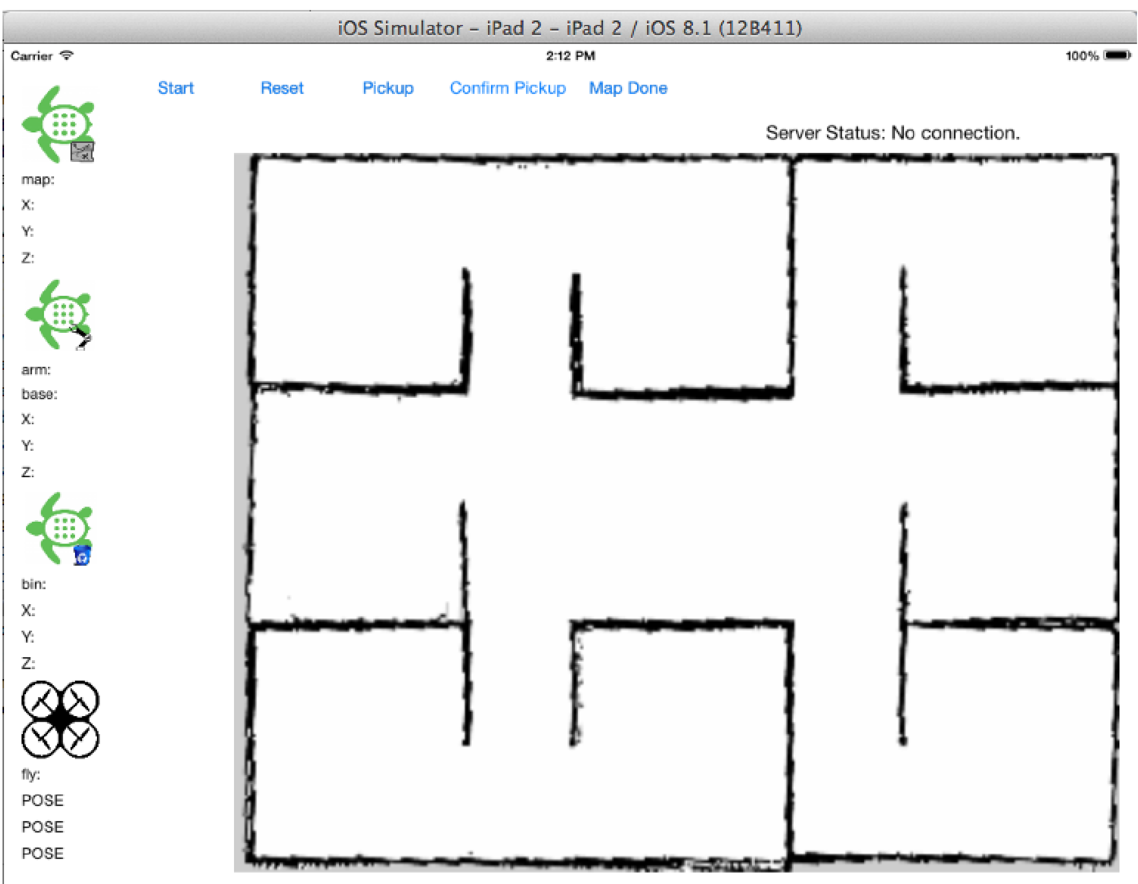


Figure 4.16: Final Interface view design and layout

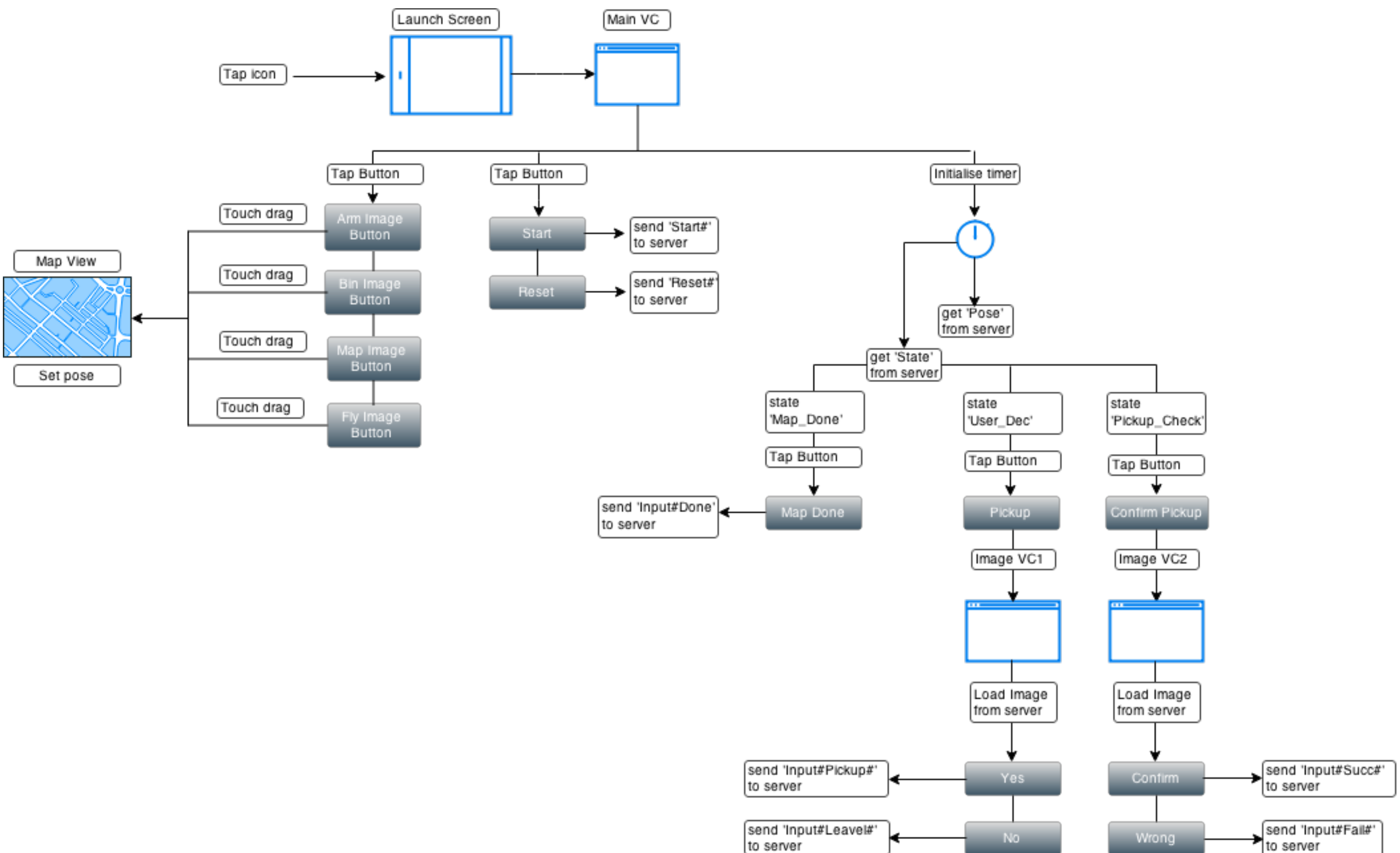


Figure 4.17: Flowchart diagram of iOS Application

Pseudo code - Library Methods

A custom library is used for image receiving, it is well commented in the repository please refer to [40].

A compulsory class or software library is required for communication with sockets for the application. Such a library is not included in Xcode and online research yielded a socket library that can be implemented for the application to server socket communication. The socket library chosen for the task is the *CocoaAsyncSocket Library/GCDAsyncSocket* [41], only a select few methods are needed from this library for message passing/receiving. In early testing phase a separate library called *square/SocketRocketLibrary* [42] which when used in conjunction with ROSbridge [43], uses industrial packages to enable non-ROS systems to connect to ROS. Since many modern web browsers can connect to ROSbridge using web sockets, this allows for communication between many of these systems. However the project goal is to use the server as a communication medium for sending/receiving messages and not necessarily directly linked to ROS. Since the server passes all messages in string format, this can be achieved in objective-C without the need for ROSbridge thus with no need for SR library. In turn this removes a layer of complexity between communications, therefore the aforementioned socket library (GCD) is used for the application. From the GCDAsyncSocket library only certain methods are needed from it for message passing/receiving. These methods are described below using pseudo code showing their functionality within the main view controller. More detailed analysis of the methods can be viewed in the GCDAsyncSocket library, look up the comments written in the *method called* sections within the library, obtained from below.

- **Check socket connected:**

  method called didConnectHost;
- **Check socket disconnected:**

  method called socketDidDisconnect;

Message sending pseudo code:
methods called connectToHost; writeData;

1. if socket is connectedToHost; andPort;
   1.1. string encoded “SEND_MESSAGE#”
   1.2. call function writeData:”SEND_MESSAGE#”
2. else if socket is not connectedToHost; andPort
   2.1. print “No Connection available”
3. Done

Message receiving pseudo code:
method called didReadData;

1. set data length to zero
2. call function readData:data
3. New Array is equal to data with components separated by "#"
4. if object at index zero of New Array is equal to "STATE"
    4.1. call function getStates: from New Array
5. else if object at index zero in New Array is equal to "POSE"
    5.1. call function getPose: from New Array
6. Done

Pseudo code - Map Conversion

The map view in the iOS application has been set to a certain size of width 800, height 650, pixels and ROS returns a certain range of x and y values respectively. An analysis in ROS pose coordinates is performed from the initial map, using stage in ROS to give an estimated pose at various locations around the map. The locations for pose estimates are recorded from top and bottom left hand corners, top and bottom right hand corners and a pose in the centre of the map. A conversion is then performed to get an estimate of where the robot images should be displayed on the simulated map space of the application. The pseudo code below is used to describe these conversion methods for vertical, horizontal and angular conversions.

ROS stage pose estimate results

- X-coordinate best estimate between ≈ -1.0 and $\approx< +8.0$
- Y-coordinate best estimate between $\approx +1.0$ and $\approx< -5.0$

Because the position of the robots vary slightly with every new map created by the map robot and the map can be displayed at different angles, it means this conversion method will need to be calibrated for every new map space created. As shown below in the pseudo code, only the 'offset' needs to be re-calibrated to fix any incorrect movement of the robots on the map view. It may seem as though the x or y coordinates contain a wasted if statement which is similar to the sequential else-if but this is not the case. It is required for improved calibration of offset values near the border of the map space, therefore should remain. Near the border the robot-images seem to float away from the map view and go over the border, this has to do with where the anchor position for the animation is. For the method of animation used, all the images are anchored at the top left hand corner therefore the offset is needed to account for that hence another critical reason for needing the extra if statements.

Vertical, Horizontal and Angular conversion pseudo code:
method called convertXXXVariables (where XXX is replaced by Arm || Bin || Map)

1. if x is less than 6.5
    1.1. x is equal to $(x * 100) + 50$
    1.2. print x

2. else if x is greater than or equal to 6.5 and less than 8.0
    2.1. x is equal to $(x * 100) + 25$
    2.2. print x
3. if y is greater than 0
    3.1. y is equal to $(y * -100) + 100$
    3.2. print y
4. else if y is less than or equal to 0 and y is greater than -4.5
    4.1. y is equal to $(y * -100) + 100$
    4.2. print y
5. else if y is less than or equal to -4.5
    5.1. y is equal to $(y * -100) + 25$
    5.2. print y
6. print z
7. convert degree to radian $= \frac{(z * \pi)}{180^\circ}$
8. Done

Table 4.1: A table showing all properties on the Main VC

Property type	Association	Quantity	Event	Function
UIButton	Robot-team Image Buttons	4	Touch up Inside	if UIButton highlighted and map view tapped, set pose of selected UIButton
UILabel	Under each robot-team image button	17	Display data	Display robot state and pose
UILabel	Server Status label	1	Display data	Display Server state
UILabel	Touch Events	1	Display data	Display Map tapped Co-Ordinates
UIButton	Start button	1	Touch up Inside	Send message to start server
UIButton	Reset Button	1	Touch up Inside	Send message to reset robot-team
UIButton	Pickup Button	1	Touch up Inside	Unwind-segue to image-VC1
UIButton	Confirm Pickup	1	Touch up Inside	Unwind-segue to image-VC2
UIButton	Map Done	1	Touch up Inside	Send message that Map is done to server
UIView	Map View	1	Touch Events	Track touches on map view
UIImageView	Map View Image	1	Load Image	Load Map Image over UIView
UIImageView	Robot-team Image Views	4	Display Image	Display Image when pose exists

Table 4.2: A table showing all properties on Image-VC1 and Image-VC2

Property type	**Association**	**Quantity**	**Event**	**Function**
UIButton	Yes/Confirm	1	Touch up Inside	Send message to server Pickup or Confirm
UIButton	No/Wrong	1	Touch up Inside	Send message to server No Pickup or Unsuccessful
UIImageView	Object View Image	1	Display Image	Display Image received from server

Chapter 5

Results

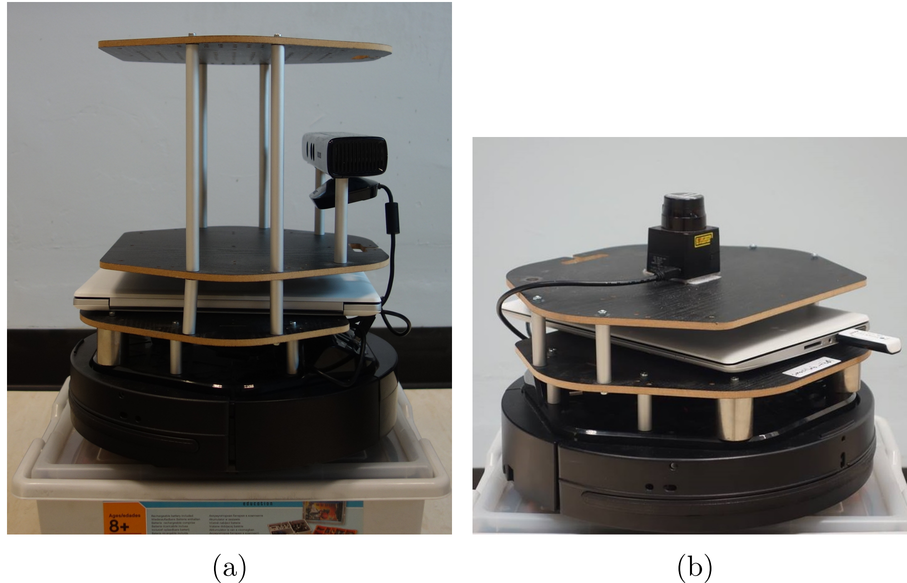
5.1 Ground Robot Team

As mentioned in the previous chapter, (see Section 4.1), it was decided that the ground-based robot team would be composed of three individual robots. The following subsections serve to present, with the lowest possible level of abstraction, the results received for each individual robot-member of the team, as well as to report any difficulties encountered during the tests. It should be noted that, due to the high similarity of the two approaches, the results subsections for the Arm and Carrier robots shall be merged together, while the main differences shall be highlighted. This is done in order to avoid repetitiveness of the same content, as well as to retain the paper's coherence and clarity.

5.1.1 Mapping Robot

Turtlebot 2 platform modifications

To begin with, the Turtlebot platform that would be used for the specific robot, had to be modified in order to provide the robot with it's desired functionalities. Figure 5.6 depicts the main differences between the original platform (on the left) and the modified one (on the right). Since the top plate was essentially not needed, it has been removed from the platform, along with any poles used to support it. This was done for two main reasons; 1) To minimize the dimensions of the platform, which imply significant limitations to the movement of the robot, and 2) To effectively lower the weight and mass center of the platform, which should result in a reduction of both, the overall power consumption and the danger of wheel drifting. This modification was possible, since the low computational cost of the ROS packages used for the mapping process, allowed for the default ASUS laptop 3.1 to be used, the dimensions of which enabled it to be slid inside the gap between the bottom and middle plates (see also Fig 3.2) Finally, the Microsoft Kinect sensor has been replaced by a Hokuyo URG-04 lidar, the reasons for which shall be described in the following sub-section.



(a) (b)

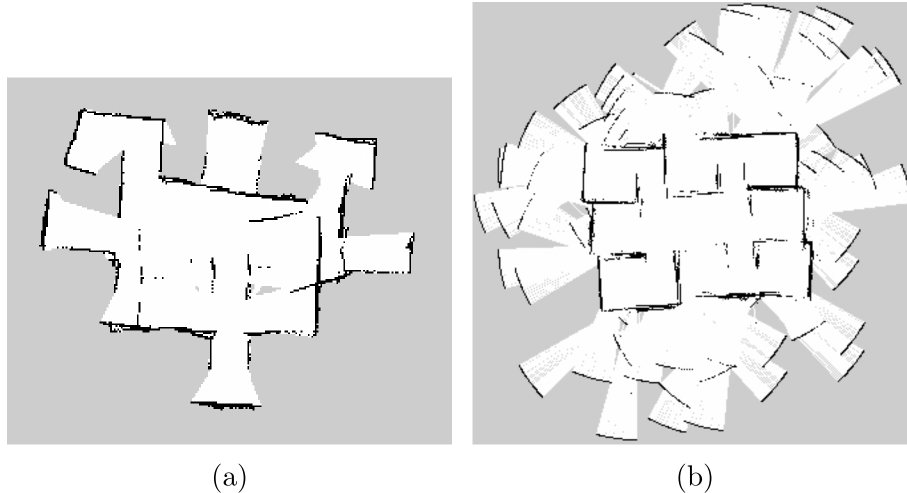
Figure 5.1: Photos of both the original Turtlebot platform (a) and the modified version (b), from which the main modifications can be observed.

Mapping approach

Continuing, the most effective combination of optical sensor and ROS package, to be used for the mapping process, needed to be identified. The first case to be examined was that of implementing 'gmapping' with the default Microsoft Kinect sensor. Since there already existed predefined open-source launch files for the use of this specific combination, it was expected that the results should be relatively accurate. However, as it can be seen in Fig. 5.2a the map does provide an consistent representation of the arena. Next, keeping 'gmapping' as the underlying mapping approach, the Kinect was swapped for a Hokuyo URG-04 laser scanner. As it can be observed in Fig. 5.2b, although the overall accuracy of the map has been improved, the map representation still does not seem quite satisfactory.

The main issues that were identified to cause the 'gmapping' package to fail in producing a sufficiently good representation of the arena were two; 1) The highly symmetric characteristics of the environment and 2) The relatively small scale of the arena. These issues, combined with the fact that 'gmapping' actively alters the transform between the global and local frames, resulted in the algorithm falsely altering the pose estimate of the robot, which in turn caused the robot to incorrectly place obstacles on the map. A fix to the above problems could be possibly implemented, by making changes to the parameters of the underlying SLAM algorithm. However, devoting all the time and effort in deeply understanding and improving the specific algorithm could prove to be wasteful, before first testing all the alternatives.

Since 'gmapping' was found to produce ambiguous map representations, 'hector_mapping' was identified as an alternative approach. This package does not



(a) (b)

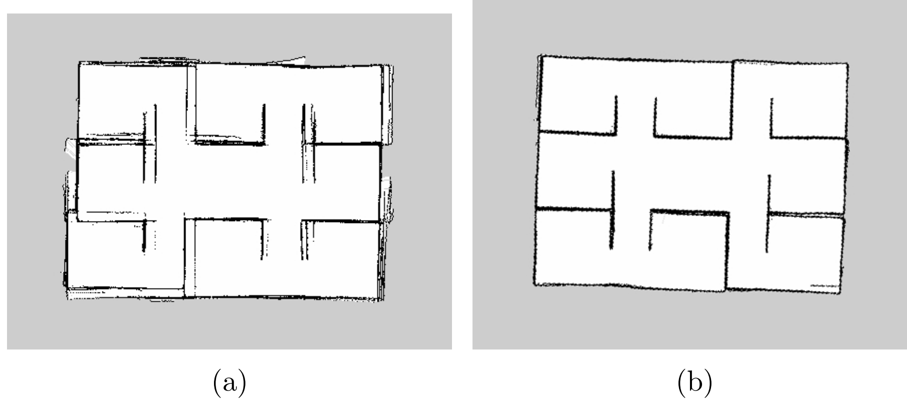
Figure 5.2: Map produced by implementing the 'gmapping' ROS package in combination with a Microsoft Kinect (a) and a Hokuyo URG-04 lidar (b). Manual teleoperation was used in both cases.

make use of odometry data in order to actively alter the global to local frame transform, but it implements a scan matcher module which publishes separately a current pose estimation of the robot based purely on the optical sensor data. As a result, this pose estimate can be fused with other odometry data externally using an algorithm (such as a Kalman filter) to provide an accurate pose estimate for the robot. The ROS package 'robot_pose_ekf' provide this exact solution, by making use of an extended Kalman Filter -hence the extension 'ekf'- in order to combine data from three different odometry sources -wheel odometry, IMU sensor data and visual odometry- and produce an accurate pose estimate of the robot.

Figures 5.3b and 5.3a show the resulting map from using 'hector_mapping' with and without implementing 'robot_pose_ekf', respectively. Due to the higher noise and much smaller FOV provided by the fake Kinect laser scans, 'hector_mapping' was producing internal errors, and thus only the Hokuyo lidar was used in this case. As it can be observed from the figures, use of 'robot_pose_ekf' showed a significant improvement of the produced map, which illustrates an almost perfect representation of the arena. Continuous testing of this combination, showed consistent map results which were not affected by the use of teleoperation or (semi)autonomous navigation. For this reason, the final approach which was chosen to be used and identified to produce optimal result was the combination of 'hector_mapping' and 'robot_pose_ekf' ROS packages, with the Hokuyo lidar.

Navigation Approach

The next decision to be made was what would be the chosen means of navigation for the robot. As described in Section 4.1 the main choices were three; 1) Fully manual teleoperation, 2) Semi-autonomous point-to-point navigation and 3) Fully autonomous exploration. Due to the nature of the project, the first option was automatically rejected and only used to test the mapping approach, where



(a) (b)

Figure 5.3: Map produced by implementing the 'hector_mapping' ROS package in combination with a Hokuyo URG-04 lidar with the use of the 'robot_pose_ekf' package(b) and without it (a).

navigation was not possible. One such example is the case of testing 'gmapping' where, due to the constant pose modifications of the robot and the misplacement of obstacles, the internally produced costmaps had an overwhelming effect on the navigation algorithm, causing a constant failure to identify possible paths.

In order to achieve the highest degree of autonomy, the last of the three options would be desirable. This approach would involve making use of the 'hector_exploration' package which makes use of a frontier-based exploration approach, where frontiers are identified as cells on the grid map, which have been marked as free of any obstacles and are adjacent to unknown (unexplored) cells. The closest reachable frontier is then identified and a path to this frontier is created. In order to create on obstacle free path to be followed, the path planner takes into consideration a costmap derived from the created map, produced by expanding the occupied cells of the map by a certain inflation distance. One major issue identified is the fact that the exploration path planner makes use only of this costmap in order to generate a path, which consists of a series of waypoints to the given location. The mapping package chosen to be used, 'hector_mapping', produces its map relative to the internally generated scan matcher frame, which is not necessarily the same as the actual pose estimation of the robot (robot frame), relative to the global frame. For this reason, if the robot happened to drift at any point, the robot frame would become different to the scan matcher frame, making the robot prone to collide with obstacles.

The 'move_base' ROS package, provides a solution to this problem by creating two separate sets of global and local costmaps and path planners. The global set essentially performs the same operation as the ones used in the case described above. The local costmap is produced solely depending on what the robot actually sees and the local path planner utilizes this information to by-pass the global planner and generate a path which is obstacle free but still adheres to the global plan. Thus, a possible approach was identified to use 'hector_exploration' in order to produce the initial path to the target frontier and then, extract the final pose of that path and send it to 'move_base' which would then handle the point-to-

point navigation. Unfortunately, even though a script was created to interface between the two packages, the overall concept did not eventually work, due to inability to configure the 'move_base' package to plan to unknown regions of the map. For this reason it was decided that the final approach would initially perform an autonomous exploration of the environment and in the case of failure, control could be taken over by a human operator using point-to-point navigation.

System Integration

The Finite State Machine model developed to describe the desired behaviour of the Mapping Robot is presented in Fig. 5.4. Before initialization, all required ROS packages, apart from the sub-packages for 'hector_exploration' are instantiated. When initialized the robot shall enter the 'FSM_WAIT' state, where it shall wait until a signal is received by the server to begin the mapping process. Upon receival of the signal, 'hector_exploration' shall be instantiated and the robot shall switch to 'FSM_AUTONOMY' state. In the case that a user sent navigation goal is received, the robot shall terminate all 'hector_exploration' sub-processes and proceed to 'FSM_MANUAL' state, where the robot uses the 'move_base' package to perform point-to-point navigation to user-defined goals. Once a signal is received by the server to stop the mapping process, the robot shall switch to state 'FSM_MOVE_TO_BASE', during which the robot sends itself the pose location of its base, and navigates to that location. Finally, upon arrival at the base location, the robot shall enter the 'FSM_IDLE' state in which it remains until shut down. To allow for a more detailed examination of the underlying programs, a Github repository has been created (see here [40]) and the internal README.txt file provides a general guideline to the packages, as well as files, included and the functionality provided by each one.

5.1.2 Arm and Carrier Robots

Turtlebot 2 platform modifications

A common characteristic of both the Arm and Carrier Robots is the fact that they both need to accommodate external equipment. In order to allow for such a functionality, the middle plate of the Turtlebot platform had to be moved back from its initial position. Fig. 5.5 depicts a profile image of each one of the three robots, from which main differences can be identified. In the case of the Arm Robot, this modification shall allow for successful mounting of the uArm platform and shall provide enough space for it to move freely. In the other case, that of the Carrier Robot, the same modification shall allow for a container to be mounted and provide sufficient clearance above the container so that the uArm can drop items without the danger of collision with any part of the platform.

As mentioned previously (see 4.1), since 3-D colour blob tracking is essential, both robots shall be equipped with a Stereo Camera Sensor. Instead of using the default Microsoft Kinect, the robots were mounted with an ASUS Xtion Pro

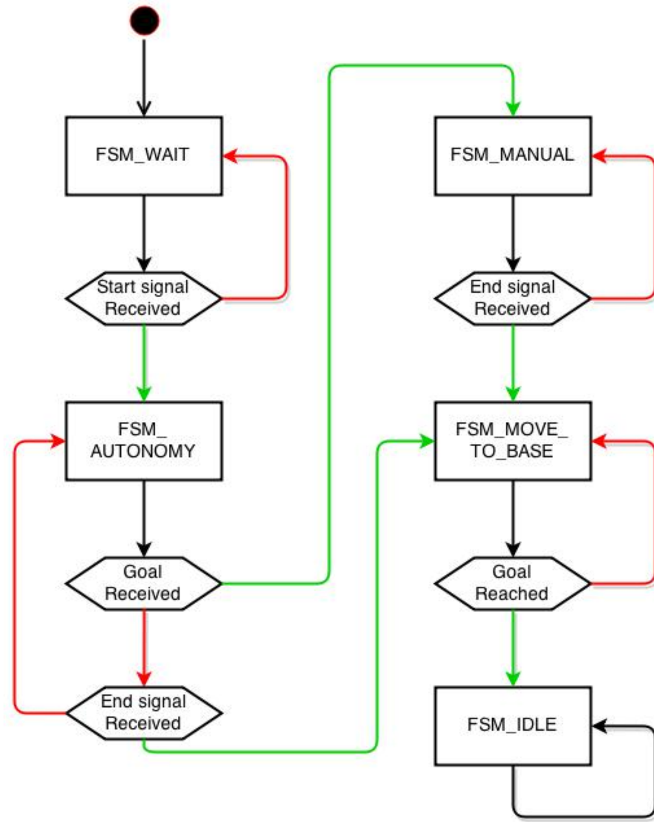


Figure 5.4: Finite State Machine (FSM) model developed to illustrate the basic behaviour of the Mapping Robot.

Live device, mainly due to size limitations of the platform. The target objects were chosen to be bright pink coloured cubes, which can be easily identified in the space and handled by the uArm. Additionally, the Arm Robot platform was mounted with a purple coloured stripe, to enable detection by the Bin Robot, while a blue square was attached to the container to act as an alignment landmark for the uArm. Finally, as identified in the process, the default ASUS laptops could not supply the required computational power in order to run all the required packages. For this reason, the top plate of the robots had to be utilized as a stand for the bigger laptops (see 3.1 and 3.1) which were too bulky to fit between the bottom and middle plates.

Localization and Navigation Approach

At an initial stage, it was essential for both robots to utilize a pre-existing map in order to localize within their environment. Once the Mapping Robot has finished mapping the environment, the 'map_saver' node of the ROS package 'map_server' is used in order to save the drawn map. The map is represented by an image (.pgm) file and a configuration (.yaml) file. Once saved, the server shall transfer these files onto the local machines for the two respective robots, and the 'map_server' node will be used in order to make use of these files and publish the respective map on

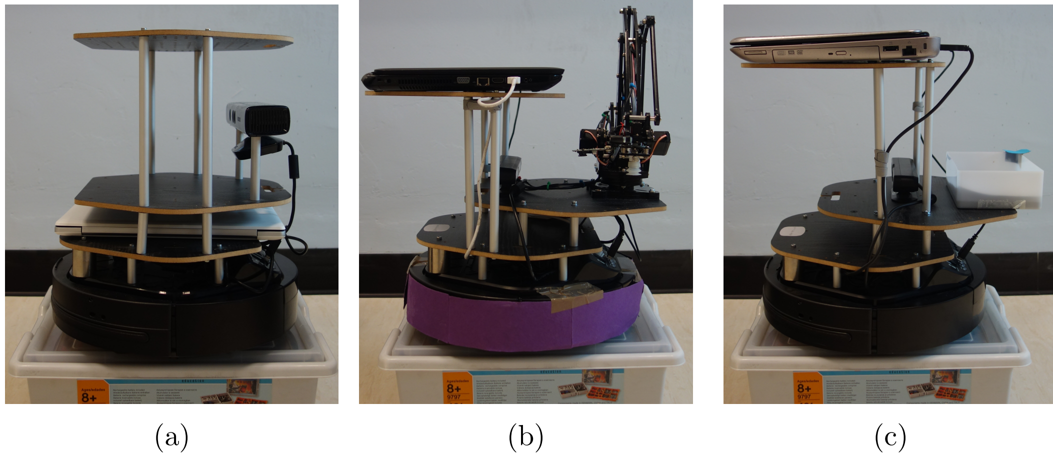


Figure 5.5: Photos of both the original Turtlebot platform (a) and the modified versions for the Arm Robot (b) and the Bin Robot (c), from which the main modifications can be observed.

a ROS topic. By making the map available onto the local machine of each robot, the localization approach can then be initialized.

The ROS package chosen for localization within a pre-existing map was 'amcl'. Just as the name implies this package provides a ROS wrapper implementation of the Adaptive Monte-Carlo Localization algorithm. Figures 5.6a to 5.6f show screenshots from the localization process of the Carrier Robot platform. Observation of the process is implemented using the 'RViz' ROS vizualization package, where the green arrows around the robot indicate all the sample pose estimates produced by the localization algorithm. As the robot spins, the set of pose estimates is continuously being resampled and a more accurate set of samples is drawn until eventually the robot has successfully localized. The resampling process is executed every time the robot travels a certain minimum distance or rotates by a minimum angle. Optimally these values should be tend to 0, however such settings would lead to the algorithm becoming too computationally intensive. This problem can be observed in the case of the Arm Robot, where due to the additional packages used for the implementation of the uArm platform, the pose estimate can be seen to diverge significantly from the robot platform. At this point, it is worth noting that a fix this problem was not identified and in its final form the Arm Robot still suffers from minor localization problems, which sometimes cause it to oscillate during navigation and rarely lead to collision with obstacles. Furthermore, due to the highly symmetric characteristics of the available arena (see Fig. 5.3b) at very rare occasions both robots were identified to suffer from 'perceptual alliasing', in which case the robot falsely believes it is in a different room than were it actually is, due to the perceived image in both rooms being nearly identical.

Coming to the navigation approach for the two robots, 'move_base' has already been proven to work successfully during the testing phase of the Mapping Robot. Thus, the same package has been used in order to handle point-to-point navigation, while the navigation goals are processed via the package's provided Simple Action Server/Client API (see the actionlib documentation [44]). However, the observed

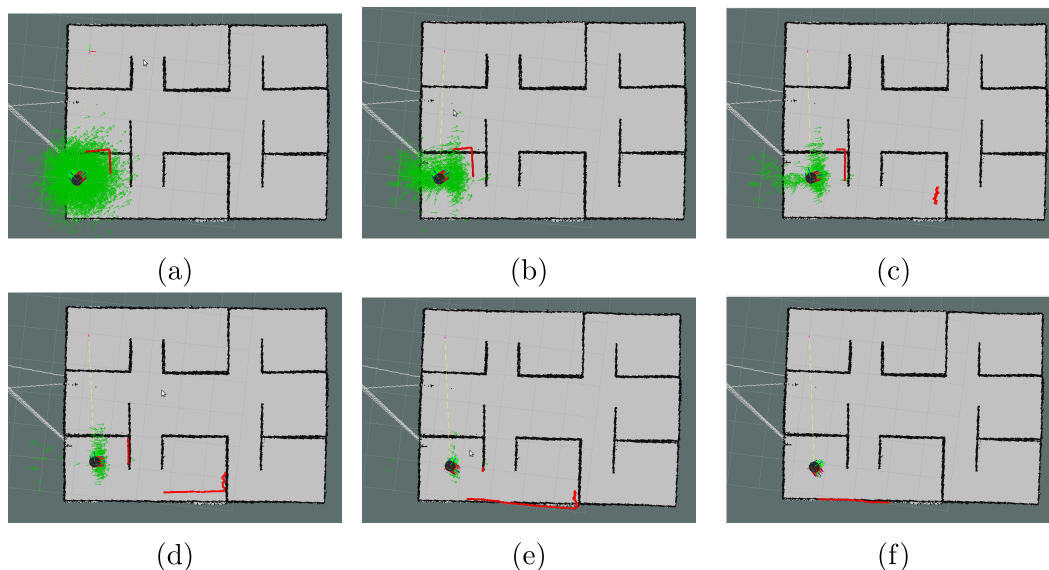


Figure 5.6: Screen shots taken during vizualization of the localization process of the Carrier Robot platform, using 'RViz'. The green arrows represent the pose estimate samples at each stage and sensor readings are shown with red. The process begins in sub-figure (a) with the initialization of the robot.

performance of the package, when used in combination with the ASUS Xtion devices, was sub-optimal compared to the one achieved using the Hokuyo lidar. From specification, the minimum sensing distance of the Xtion device is between 0.5 and 0.7 meters, compared to 2cm for the lidar. This fact, combined with the small scale of the available arena have a tremendous effect on the overall performance of the navigation process. One first example, is the case where the robot has to perform recovery behaviours when stuck between a narrow corridor.

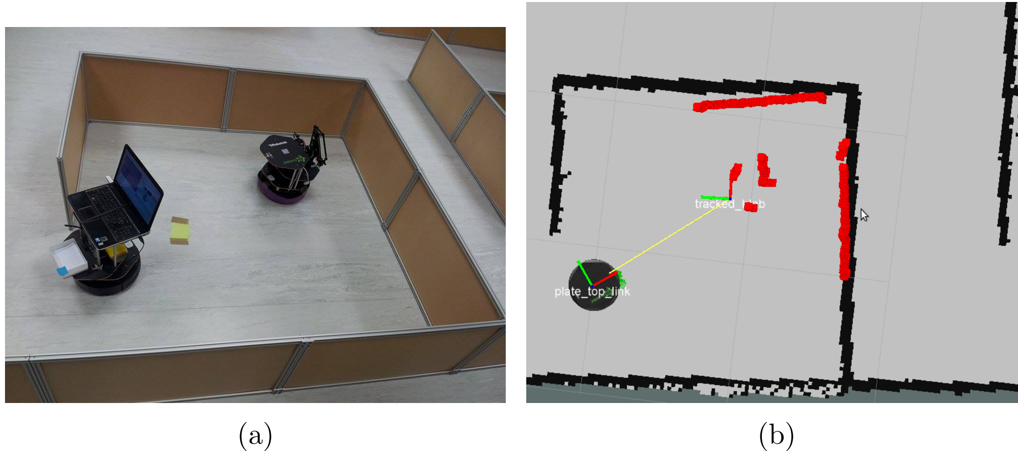
Recovery behaviours are implemented when the robot cannot identify a possible plan to a given location. One commonly used recovery behaviour is that of costmap clearing and rotate recovery, in which case all local costmaps are cleared and the robot spins in order to get a new perception of the environment. When such a recovery behaviour is executed inside a corridor of 1 meter width (such as the ones located inside the arena), given the minimum sensing distance of the Xtion device, it is very likely that at least one portion of the walls will not be detected. In such a case, if it so happens and an optimal plan is found that crosses such an undetected wall, then the robot shall collide. One possible solution to this problem would involve mounting a Hokuyo lidar in addition to the Xtion, the output of which shall be used as an input for the 'amcl' package, while the Asus Xtion shall be used solely for colour tracking. However, due to the performance limitations already being imposed on the underlying hardware, adding the necessary driver ROS packages for the lidar could potentially cause a system overload. Testing of this implementation was not executed, mainly due to time limitations of the project, however a potential continuation should consider this as a priority task.

Colour detection and tracking

In order to perform effective 3-D detection and tracking of colour blobs, a combination of two ROS packages was identified as the optimal solution. At a first stage, the 'cmvision' package uses the RGB image provided by the Xtion devices in order to produce a set of 2-dimensional coordinates, placing a detected color blob within the received image frame. Continuing, the 'cmvision_3d' package is used to fuse the depth image produced by the Xtion devices along with the data outsourced by the 'cmvision' package. The resulting output is an extended version of the output blob format provided by 'cmvision' with additional information about the 3-dimensional position of the center, as well as left end right edges, of each blob. At this point, it should be noted that an internal modification was applied to the underlying algorithm of 'cmvision_3d' (see the README.txt file in [40]), in order to allow for all different occurrences of the same colour to be published, rather than just the one with the largest area.

Due to color detection, as well as depth (infrared) registration, both being processes which are highly dependant upon external (ambient) light conditions, the outputed blobs from the above mentioned process were identified to have a great degree of error associated with them. To begin with, the depth information received by the Xtion devices was not always correct or consistent, which caused significant fluctuations in the registered distance information or led to blobs initially detected by the 'cmvision' package to be rejected during the dimensions validation phase of 'cmvision_3d'. The result of this error prone process, was a stream of color blobs which could not be guaranteed to be neither continuous or correct. Since the behaviour of both the Arm and the Carrier Robot is closely coupled to the correctness of this information, it was deemed necessary to apply an additional filtering/coordinate correction algorithm before the data would be considered by the robots.

The algorithm is described by the pseudo code presented inside Algorithm 1 in the Appendix Section. The time and resource complexity of the proposed algorithm can be analysed as O(n), where n is the number of input blobs. From repetitive examination of the output received by 'cmvision_3d', in several application specific test cases, it has been identified that the number of returned blobs is never higher than 20. Assuming that the above statement holds (without loss of generality) the algorithm complexity can be simplified further to O(1). The averaging approach for the x and z coordinates, has been implemented uniquely in each case in order to achieve different goals. The x coordinate is corrected solely depending upon the filtered blobs of each individual run, in order to smooth out any abrupt changes, which shall be advantageous when centering of the blob is essential. On the contrary, the z coordinate average is updated continuously upon determination of the largest sensible blob and has been configured to favour the smallest recorded distances. This shall optimally force saturation of the detected distance to a minimum, which shall enhance the stability of the overall blob distance detection. Finally, in case of a sudden loss of tracked blobs, the algorithm shall store the final tracked position of the blob for 10 individual runs before signaling that no blobs are being detected. This correction has been implemented



(a) (b)

Figure 5.7: Figure illustrating effective tracking and centering of the Arm Robot platform achieved by the Carrier Robot. Subfigure (a) shows the actual placement of the two robots in the arena. Subfigure (b) shows a vizualization of the process using 'RViz', where both the local transform frame of the robot (top plate link), as well as the published tracked blob transform frame, are depicted.

in order to avoid undesired behaviour due to constant target loss. Fig. 5.7 shows the visualized output of the algorithm during a test run, where the Carrier Robot has achieved to effectively track and center the color strip on the base of the Arm Robot.

System Integration

The Finite State Machine models used to describe the desired behaviours of the Arm and Carrier Robots, can be viewed in Figures 5.8 and 5.9, respectively. As it can be observed, the two behaviours are pretty much identical, apart from the very final few state transitions. Upon initialization, all essential ROS packages are launched and both robots enter the initial 'FSM_WAIT' state. Once the map files have been received on the local machines, the robots proceed to 'FSM_LOCALIZE' state, while at the same time launching the 'map_server', 'amcl' and 'cmvision'/'cmvision_3d' packages. 'FSM_LOCALIZE' shall be executed for 20 seconds (the approximate time required for a 360° spin of the platform), after the end of which the robots shall switch to the 'FSM_IDLE' state. Continuing, whilst in 'FSM_IDLE' both robots wait for a navigation goal to be received by the server, upon receival of which both state machines they advance to 'FSM_NAVIGATE'. If a some point during the navigation to the goal, a color blob is detected then the robots shall immediately switch to 'FSM_OPTIMIZE'. This specific state, makes use of a PD controller in order to systematically send velocity commands to the motors, such that the tracked blob is centered in the middle of the camera frame. In case of sudden loss of target, 'FSM_SEARCH' has been put in place in order to make the robots perform a searching spin in order to regain tracking control.

Once a tracked blob has been centered, the calculated distance shall be considered in order to determine whether or not final approach should be initialized.

Figure 5.8: Finite State Machine (FSM) model developed to illustrate the basic behaviour of the Arm Robot.

In the case that the distance is not sufficiently small, 'FSM_APPROACH_OBJ' shall be executed, in order to pass a waypoint to the robots which is 0.7 meters away from the tracked blob. Upon arrival to the set navigation goal, the robots reiterate through 'FSM_OPTIMIZE' upon completion of which, they shall move to 'FSM_STABLE', since they are now sufficiently close for final approach to be performs. During 'FSM_STABLE', the robots should perform a 2 second wait in the same position, which due to the structure of developed colour tracking enhancement algorithm (see Algorithm 1), shall let the calculated distance to saturate to a value relatively close to the minimum distance. This needs to be done before the robot goes any closer than 0.7 meters from the tracked object, since due to minimum sensing distance of the Xtion devices, any further approach shall risk losing track of the target. Once this measurement is complete, the robot calculates the amount of time that it needs to travel for in order to perform its final approach to the target object, before spin behaviours are initialized.

It is at this point that the two behaviours begin to diverge from one another. The Arm Robot, during its 'FSM_SPIN' state, spins until the Arm has detected a target object. Once, this has been done, 'FSM_WAIT_FOR_ACTION' shall be entered, during which the robot shall wait until the Arm has finished performing its actions. These two states should be looped through twice for each object that is being picked up. The first iteration is performed while the Arm is searching for and picking up the target object, and the second iteration is performed while the Arm is searching for and dropping an object inside the container. Upon successfully dropping an object, the Arm should be enter an Idle state, which shall force the

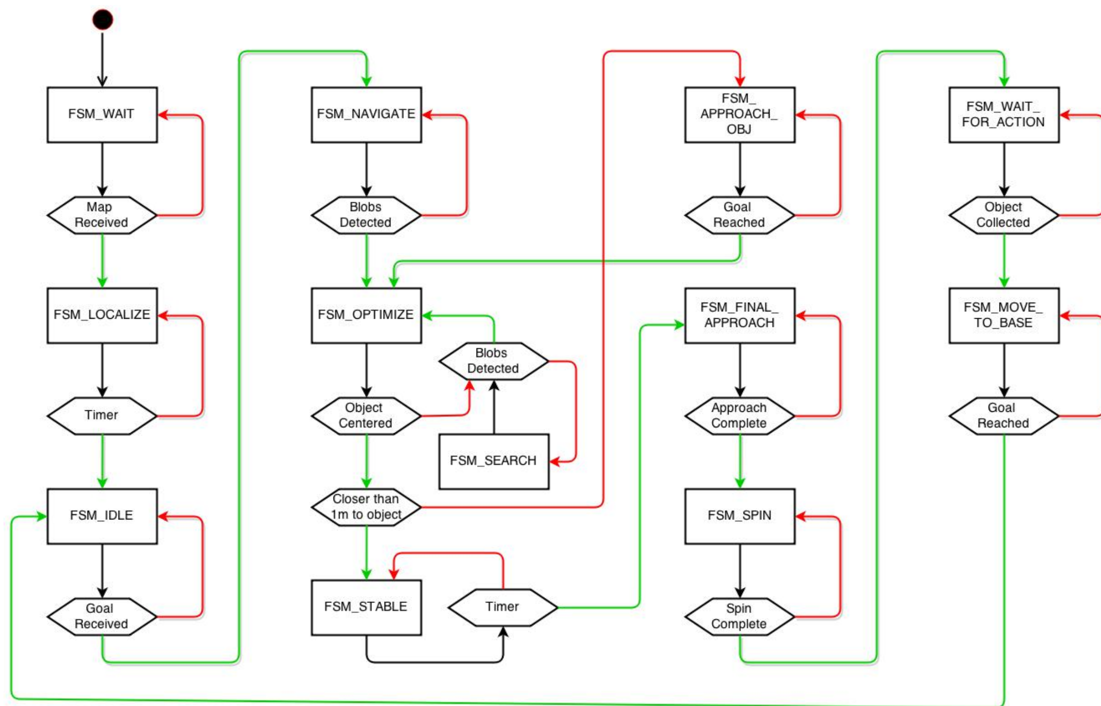


Figure 5.9: Finite State Machine (FSM) model developed to illustrate the basic behaviour of the Carrier Robot.

robot to loop back to 'FSM_IDLE', where it waits for a target navigation goal to be passed by the server. For the case of the Carrier Robot, 'FSM_SPIN' makes use of a PD controller which should force the robot to perform a 180° turn, once approach to the Arm Robot has been completed. Upon successfully performing the turn, the Carrier Robot advances to 'FSM_WAIT_FOR_ACTION', where it shall wait for the server to signify that the target object has been successfully dropped within the container. When the signal is received by the server, 'FSM_MOVE_TO_BASE' is invoked, during which the robot navigates back to its base location. Finally, once the base location is eventually reached the Carrier Robot shall loop back to 'FSM_IDLE', from which the entire process can be repeated. It should be noted that a reset state has been implemented for both robots, which is entered upon receival of a reset signal by the server and during which state both robots are instructed to fall back to their base locations. This state has been omitted by the FSM diagrams to avoid congestion.

5.2 Communications and Server

5.2.1 Design Files

A list and description of all the files in the robot communication module is given in table 5.2.1.

Table 5.2.1: Files contained within the robot communication package.

File Name	**Description**
server.py	Python server, run to enable server.
server_fsm.py	Server state machine.
clientclass.py	Client/server comm. class.
arm_client_node.py	ROS node to comm. with ArmBot
bin_client_node.py	ROS node to comm. with BinBot
map_client_node.py	ROS node to comm. with MapBot
fly_sim_node.py	ROS node to comm. with FlyBot
ipad_sim.py	Python script to simulate iPad input/user decisions.
pose_dict_tf.py	Functions to convert between ROS pose formats and python dictionaries.
ros_image_conv	Function to convert ROS sensor_msgs Image format to '.ppm'.
read_settings.py	Function to read 'setting.txt' to determine host address, file directories, etc..
settings.txt	Text file used to edit server address, server map/image directories, robot map directories.

The files for the robot communication package are hosted as a ROS package at [45].

5.2.2 Server Results

The server ('server.py') initialises using the settings specified in the 'settings.txt' file. When initialised, it pulls the host address, map file directory and image file directory from the settings and sets the server variables (listing 5.1). The robot data is stored in a dictionary object (line 13), and the object pose variable is a list (line 31), in order to act as a first-in first-out queue.

Listing 5.1: Initialisation function for the server, initialises server variables/FSM.

```
1     #####################################################################
2     # Initialised with HOST_ADDR as tuple consisiting of {IP, PORT},
3     # MAP_DIR a string pointing to directory for map files,
4     # IMAGE_DIR a string pointing to directory for image files.
5     def __init__(self, HOST_ADDR, MAP_DIR, IMAGE_DIR):
6         # Input variables
7         self.HOST_ADDR = HOST_ADDR
8         self.MAP_DIR = MAP_DIR
9         self.IMAGE_DIR = IMAGE_DIR
10        # Initialise server FSM.
11        self.InitServFSM()
12        # Set dictionary for bot data.
13        self.DATA = {'STATES': {'ARMBOT': {'NONE'},
14                                'BASEBOT': {'NONE'},
15                                'BINBOT': {'NONE'},
16                                'MAPEBOT': {'NONE'},
17                                'FLYBOT': {'NONE'}},
18                     'POSES': {'BASEBOT': 'NONE',
19                               'BINBOT': 'NONE',
20                               'MAPBOT': 'NONE',
21                               'FLYBOT': 'NONE'}}
22        # Initialise server variables.
23        self.START_PROCESS = False
24        self.SYNC = False
25        self.USER_INPUT = None
26        self.COUNT = 0
27        # Initialise MAP and IMAGE flags.
28        self.MAP_FLAG = None
29        self.IMAGE_FLAG = None
30        # Initialise object location list.
31        self.OBJ_POSE = []
32        # Initialise variables for bot data timeout.
33        self.ARM_TIME = 0
34        self.BASE_TIME = 0
35        self.BIN_TIME = 0
36        self.FLY_TIME = 0
37        self.MAP_TIME = 0
```

The server is started using the 'Run()' function, which begins threads for the 'Listen()' and 'DataTimeout()' functions. 'Run()' and 'Listen()' are shown in listing 5.2 and 'DataTimeout()' is shown in listing 5.3.

Listing 5.2: Server functions to start the server and listen for incoming clients.

```
1     #################################################################
2     # 'Run' enables the server to listen for incoming connections
3     # on one thread ('Listen') and spins 'DataTimeout' on another.
4     def Run(self):
5         self.SYNC = True
6         self.SERV_THREAD = threading.Thread(target=self.Listen)
7         self.SERV_THREAD.start()
8         self.DATA_THREAD = threading.Thread(target=self.DataTimeout)
9         self.DATA_THREAD.start()
10    #########################################################################
11    # Listens for incoming connections and assigns socket connections for
12    # client-server communication ('ClientConn' thread started for each
13    # client).
14    def Listen(self):
15        SOCK = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
16        SOCK.bind(self.HOST_ADDR)
17        SOCK.listen(15)
18        SOCK.settimeout(3)
19        while(self.SYNC):
20            try:
21                CONN, ADDR = SOCK.accept()
22                CONN.setblocking(1)
23                CLIENT_THREAD = threading.Thread(
24                    target=self.ClientConn, args=(CONN, ADDR))
25                CLIENT_THREAD.start()
26            except socket.timeout:
27                pass
28        SOCK.close()
29        print 'Socket closed.'
```

The 'Listen()' function listens for incoming client connections and assigns a temporary socket object to connected pairs. The socket connection that handles incoming connections has a time-out of 3 seconds (listing 5.2, line 19), which means the socket will re-establish every 3 seconds to check to see if user ended process.

Listing 5.3: Server function to refresh/reset server variables based on time-outs/server state.

```
1  ###############################################################
2  #  Reset robot data variables after TIMEOUT seconds of no new
3  #  data.  Check and set IMAGE and MAP flags, allow user input
4  #  during allowed server states (USER_INPUT).
5  def DataTimeout(self):
6      while(self.SYNC):
7          #  Check if data is less than TIMEOUT seconds old,
8          #  reseting to 'NONE' if so.
9          CUR_TIME = time()
10         TIMEOUT = 5
11         CUR_STATE = self.FSM.curState.StateName()
12         if(CUR_TIME > (self.ARM_TIME + TIMEOUT)):
13             self.DATA['STATES']['ARMBOT'] = 'NONE'
14         if(CUR_TIME > (self.BASE_TIME + TIMEOUT)):
15             self.DATA['STATES']['BASEBOT'] = 'NONE'
16             self.DATA['POSES']['BASEBOT'] = 'NONE'
17         if(CUR_TIME > (self.FLY_TIME + TIMEOUT)):
18             self.DATA['STATES']['FLYBOT'] = 'NONE'
19             self.DATA['POSES']['FLYBOT'] = 'NONE'
20         if(CUR_TIME > (self.BIN_TIME + TIMEOUT)):
21             self.DATA['STATES']['BINBOT'] = 'NONE'
22             self.DATA['POSES']['BINBOT'] = 'NONE'
23         if(CUR_TIME > (self.MAP_TIME + TIMEOUT)):
24             self.DATA['STATES']['MAPBOT'] = 'NONE'
25             self.DATA['POSES']['MAPBOT'] = 'NONE'
26         #  Set map flag (check for .pgm and .yaml).
27         CON_1 = os.path.isfile(self.MAP_DIR + '/map.pgm')
28         CON_2 = os.path.isfile(self.MAP_DIR + '/map.yaml')
29         self.MAP_FLAG = CON_1 and CON_2
30         #  Set image flag (check for png's in image dir).
31         CON_1 = os.path.isfile(self.IMAGE_DIR + '/object_image.png')
32         CON_2 = os.path.isfile(self.IMAGE_DIR + '/verify_image.png')
33         self.IMAGE_FLAG = CON_1 or CON_2
34         #  Remove image files/reset user input if not in decision state.
35         if((CUR_STATE != 'ARM_SEARCH') and (CUR_STATE != 'USER_DEC')):
36             if(os.path.isfile(self.IMAGE_DIR + '/object_image.png')):
37                 os.remove(self.IMAGE_DIR + '/object_image.ppm')
38                 os.remove(self.IMAGE_DIR + '/object_image.png')
39         if((CUR_STATE != 'ARM_PICKUP') and (CUR_STATE != 'PICKUP_CHECK')):
40             if(os.path.isfile(self.IMAGE_DIR + '/verify_image.png')):
41                 os.remove(self.IMAGE_DIR + '/verify_image.ppm')
42                 os.remove(self.IMAGE_DIR + '/verify_image.png')
43         if((CUR_STATE != 'USER_DEC') and (CUR_STATE != 'PICKUP_CHECK') and
44                 (CUR_STATE != 'MAPPING_MAN')):
45             self.USER_INPUT = None
46         #  Reset pose list and start variable in RESET.
47         if(CUR_STATE == 'RESET'):
48             self.START_PROCESS = False
49             self.OBJ_POSE = []
50         sleep(1)
```

The 'DataTimeout()' function ensures that the robot data is only valid for a given period after receiving the data. The time-out is set to 5 seconds (listing 5.3, line 5), which is sufficient time to show a dropped connection. Flags are set based on whether map or image files exist in the server directories (listing 5.3, lines 27-33) and unwanted files are removed (listing 5.3, lines 35-42).

The 'Send()' function is used to send data on a socket connection, shown in listing 5.4.

Listing 5.4: Server function to send variable packet-size data over socket connection.

```python
################################################
#  Send input DATA to socket connection CONN,
#  using packet size PACK_SIZE.
def Send(self, CONN, PACK_SIZE, DATA):
    SIZE = len(DATA)
    PACK = 0
    INDEX = 0
    SENT = 0
    while(True):
        INDEX = PACK * 1
        CHUNK = DATA[INDEX:INDEX + PACK_SIZE]
        CONN.send(CHUNK)
        SENT = SENT + len(CHUNK)
        PACK = PACK + PACK_SIZE
        if(SENT == SIZE):
            break
```

The packet size can be specified so the function can be used to send all data types. There are 3 functions to receive data; 'Recv()', 'RecvLine()' and 'RecvFile()', shown in listing 5.5.

Listing 5.5: Server functions to receive different data types over a socket connection.

```python
################################################
#  Receive data from socket connection CONN,
#  using packet size PACK_SIZE.
def Recv(self, CONN, PACK_SIZE):
    DATA = ''
    while(True):
        CHUNK = CONN.recv(PACK_SIZE)
        if(CHUNK):
            DATA = DATA + CHUNK
        else:
            break
    return DATA

################################################
#  Receive data from socket connection CONN,
# until end-of-line character '#'.
def RecvLine(self, CONN):
    DATA = ''
    while(True):
        CHAR = CONN.recv(1)
        if(len(CHAR) == 1):
            if(CHAR == '#'):
                break
            else:
                DATA = DATA + CHAR
        else:
            break
    return DATA

################################################
#  Receive data from socket connection CONN,
#  using packet size PACK_SIZE and save to
#  file specified by PATH.
def RecvFile(self, CONN, PACK_SIZE, PATH):
    FILE = open(PATH, 'wb')
    while(True):
        CHUNK = CONN.recv(PACK_SIZE)
        if(CHUNK):
            FILE.write(CHUNK)
        else:
            break
    FILE.close()
```

'Recv()' allows for a variable packet size string to be received from the client, 'RecvLine()' reads 1 character at a time and stops at an end-line character (i.e. read until '#') and 'RecvFile()' reads data from the client and writes it directly to a file.

5.2.3 Client Results

The client class ('clientclass.py') is used in each of the robot client nodes to communicate with the server. When initialised with a robot type, the client-server synchronization can be started with 'Start()'. 'Start()' begins a thread, 'Sync()', which connects to the server and transmits data at 5 Hz while enabled. After initialising and starting the client, the user only needs to update the client state/pose to upload the robot data to the server. The 'Start()' and 'Sync()' functions are shown in listing 5.6.

Listing 5.6: Client functions to start and synchronise the client to the server.

```
1   #############################################################
2   #  Starts threads for 'Sync' and 'DataTimeout' functions
3   def Start(self):
4       self.SERV_SYNC = True
5       self.CLIENT_CONN = threading.Thread(target=self.Sync)
6       self.CLIENT_CONN.start()
7       self.TIMEOUT_THREAD = threading.Thread(target=self.DataTimeout)
8       self.TIMEOUT_THREAD.start()
9
10  #############################################################
11  #  While sync'd update robot pose/state to server
12  #  and request server state.
13  def Sync(self):
14      logging.info('Synchronization started.')
15      while(self.SERV_SYNC):
16          SOCK = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
17          SOCK.setblocking(1)
18          SOCK.settimeout(5)
19          try:
20              SOCK.connect(self.HOST_ADDR)
21              DATA = self.BOT_TYPE + '#' + \
22                  self.STATE + '#' + self.POSE + '#'
23              self.Send(SOCK, 1, DATA)
24              self.SERV_STATE = self.Recv(SOCK, 32)
25              sleep(0.2)
26              SOCK.close()
27          except socket.timeout:
28              logging.info('Cannot connect, re-establishing \
29                  connection to server...')
30              sleep(0.2)
31      logging.info('Synchronization stopped.')
```

The 'Sync()' functions sends a string to the server containing the current robot type, state and pose, before receiving the current server state (listing 5.6, lines 21-22).

Functions to read/write the client state/pose and read the server state are shown in listing 5.7. When the state/pose is updated the time is recorded in order to reset the variables after a period of no connection (listing 5.7, lines 6 and 10).

Listing 5.7: Client functions for accessing/setting state and pose variables.

```
1   ###############################################
2   #  Set client state and pose, read client
3   #  and server states.
4   def SetState(self, STATE):
5       self.STATE = STATE
6       self.STATE_TIME = time()
7
8   def SetPose(self, POSE):
9       self.POSE = POSE
10      self.POSE_TIME = time()
11
12  def ServState(self):
13      return self.SERV_STATE
```

The ‘Send()’, ‘Recv()’ and ‘RecvFile()’ functions are identical to the server versions (listings 5.4 and 5.5). The ‘RecvMap()’ and ‘RecvPose()’ functions retrieve the map files and object pose from the server respectively, shown in listing 5.8.

Listing 5.8: Client functions for receiving poses and files from the server.

```
1      ######################################################
2      #  Depending on TARGET, requests and retrieves
3      #  pose string for ARMBOT or OBJECT.
4      def RecvPose(self, TARGET):
5          SOCK = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
6          SOCK.connect(self.HOST_ADDR)
7          if(TARGET == 'OBJ'):
8              SOCK.send('SEND_OBJ_POSE#')
9          elif(TARGET == 'ARM'):
10             SOCK.send('SEND_ARM_POSE#')
11         POSE = self.Recv(SOCK, 32)
12         SOCK.close()
13         if(POSE):
14             return POSE
15         else:
16             return None
17
18     ########################################################
19     #  Receive 2 map files, 'map.pgm' and 'map.yaml',
20     #  saving in directory specified by DEST.  Using
21     #  threads to avoid holding up program.
22     def RecvMap(self, DEST):
23         SOCK_0 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
24         SOCK_0.connect(self.HOST_ADDR)
25         SOCK_0.send('SEND_MAP_PGM#')
26         THREAD_0 = threading.Thread(target=self.RecvFile,
27                                     args=(SOCK_0, 32, DEST + '/map.pgm'))
28         THREAD_0.start()
29         SOCK_1 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
30         SOCK_1.connect(self.HOST_ADDR)
31         SOCK_1.send('SEND_MAP_YAML#')
32         THREAD_1 = threading.Thread(target=self.RecvFile,
33                                     args=(SOCK_1, 32, DEST + '/map.yaml'))
34         THREAD_1.start()
```

‘RecvMap()’ starts 2 threads of ‘RecvFile()’, one for each map file downloading from the server (i.e. ‘map.pgm’ and ‘map.yaml’). This is to ensure that the client node isn’t halted while it waits for files to download. The ‘SendFile()’ functions enables the upload of map and image files to the server, as seen in listing 5.9.

Listing 5.9: Client function to send files to server via thread.

```
1      ####################################
2      #  Send input DATA of type TYPE.
3      def SendFile(self, DATA, TYPE):
4          if(TYPE == 'MAP_YAML'):
5              MSG = 'RECV_MAP_YAML#'
6          elif(TYPE == 'MAP_PGM'):
7              MSG = 'RECV_MAP_PGM#'
8          elif(TYPE == 'OBJ'):
9              MSG = 'RECV_OBJ_IM#'
10         elif(TYPE == 'VER'):
11             MSG = 'RECV_VER_IM#'
12         THREAD = threading.Thread(target=self.Send,
13                                   args=(None, 32, MSG + DATA))
14         THREAD.start()
```

5.2.4 ArmBot Client Results

The ArmBot client node is initialised from ROS, and reads the ‘settings.txt’ file to determine the server address and file directories. The ArmBot client node initialisation is displayed in listing 5.10.

Listing 5.10: Initialisation function for the ArmBot client node.

```
    ##########################################################
    # Initialise client with server address HOST_ADDR and
    # map file directory MAP_DIR. A client is initialised
    # for the ARM and the BASE, and any exisiting map files
    # are deleted on startup.
    def __init__(self, HOST_ADDR, MAP_DIR):
        self.HOST_ADDR = HOST_ADDR
        self.MAP_DIR = MAP_DIR
        self.ARM_CLIENT = BotClient(HOST_ADDR, 'ARMBOT')
        self.ARM_BASE_CLIENT = BotClient(HOST_ADDR, 'BASEBOT')
        self.IMAGE_FLAG = None
        self.PRINT_FLAG = None
        self.POSE_FLAG = None
        if(os.path.isfile(self.MAP_DIR + '/map.pgm')):
            os.remove(self.MAP_DIR + '/map.pgm')
        if(os.path.isfile(self.MAP_DIR + '/map.yaml')):
            os.remove(self.MAP_DIR + '/map.yaml')
        self.MAP_FLAG = None
        # Subscriptions for ARM state, BASE state,
        # BASE pose and ARM image data.
        rospy.Subscriber('/uarm/state',
                         String,
                         self.ArmStateCallback)
        rospy.Subscriber('/arm_bot_base/state',
                         String,
                         self.ArmBaseStateCallback)
        rospy.Subscriber('/amcl_pose',
                         PoseWithCovarianceStamped,
                         self.ArmBasePoseCallback)
        rospy.Subscriber('/usb_cam/image_raw',
                         Image,
                         self.ImageCallback)
        # Publications for server state and object pose goal.
        self.POSE_TOPIC = rospy.Publisher('/arm_bot_base/goal', PoseStamped)
        self.SERV_STATE = rospy.Publisher('/client_node/serv_state', String)
```

The ROS subscriptions for the ArmBot state, BaseBot state/pose and ArmBot image feed are at lines 21-32 and the ROS publications for move goal and server state are at lines 34-35.

The callback functions for the states and poses are shown in listing 5.11.

Listing 5.11: Callback functions to update client/upload to server.

```
    ##########################################
    # Updates ARM state in client.
    def ArmStateCallback(self, DATA):
        self.ARM_CLIENT.SetState(DATA.data)

    ##########################################
    # Updates BASE state in client.
    def ArmBaseStateCallback(self, DATA):
        self.ARM_BASE_CLIENT.SetState(DATA.data)

    ##################################################
    # Updates ARM pose in client. Converts pose
    # to a dictionary, converts the data to euler
    # co-ords from quaternion and send to server.
    def ArmBasePoseCallback(self, DATA):
        POSE_DICT = PoseCovarianceToDict(DATA)
        POSE_DICT_STRING = str(QuaternionToEulerDict(POSE_DICT))
        self.ARM_BASE_CLIENT.SetPose(POSE_DICT_STRING)
```

The ArmBot and BaseBot states are uploaded to the server, and the BaseBot pose is converted to a dictionary format (line 16), transformed to Euler co-ordiates (line 17) before being uploaded to the server (line 18).

The callback function for the ArmBot camera feed is displayed in listing 5.12.

Listing 5.12: Callback function for the ArmBot camera upload thread.

```
1   ############################################################
2   #  When the server is waiting for an image file to be
3   #  uploaded (for user decision), ROS geometry_msgs
4   #  Image converted to PPM and sent to server.
5   def ImageCallback(self, DATA):
6       CUR_SERV_STATE = self.ARM_CLIENT.ServState()
7       if(CUR_SERV_STATE == 'ARM_SEARCH'):
8           if(not self.IMAGE_FLAG):
9               PPM_DATA = RosImageToPPMString(DATA)
10              self.ARM_CLIENT.SendFile(PPM_DATA, 'OBJ')
11              self.IMAGE_FLAG = True
12      elif(CUR_SERV_STATE == 'ARM_PICKUP'):
13          if(not self.IMAGE_FLAG):
14              PPM_DATA = RosImageToPPMString(DATA)
15              self.ARM_CLIENT.SendFile(PPM_DATA, 'VER')
16              self.IMAGE_FLAG = True
17      else:
18          self.IMAGE_FLAG = None
```

An image is uploaded to the server for 2 user decisions; 1 to confirm a valid object has been found and another to confirm successful pickup of the object. For each upload, the Image data is read and converted to a '.ppm' format (lines 9 and 14) and uploaded to the server (lines 10 and 15).

The main callback function for ArmBot client node is displayed in listing 5.13.

Listing 5.13: Main function for the ArmBot client node.

```
1   ##########################################################
2   #  Work callback, executed at RATE (defined at top).
3   def WorkCallback(self):
4       #  Publish current server state.
5       CUR_SERV_STATE = self.ARM_CLIENT.ServState()
6       self.SERV_STATE.publish(CUR_SERV_STATE)
7       #  Display current server state (if updated).
8       if(self.PRINT_FLAG != CUR_SERV_STATE):
9           self.PRINT_FLAG = CUR_SERV_STATE
10          rospy.loginfo(self.PRINT_FLAG)
11      #  Reset IMAGE_FLAG when in un-related state.
12      if((CUR_SERV_STATE != 'ARM_SEARCH') or
13              (CUR_SERV_STATE != 'ARM_PICKUP')):
14          self.IMAGE_FLAG = None
15      #  When object is found (server state FOUND_OBJ),
16      #  an object pose is downloaded, converted to PoseStamped()
17      #  message type and published to necessary ROS topic.
18      if(CUR_SERV_STATE == 'FOUND_OBJ'):
19          if(not self.POSE_FLAG):
20              POSE_STRING = self.ARM_CLIENT.RecvPose('OBJ')
21              rospy.loginfo('POSE: ' + POSE_STRING)
22              if(isinstance(POSE_STRING, str)):
23                  POSE_DICT = ast.literal_eval(POSE_STRING)
24                  POSE_STAMPED = DictToPoseStamped(POSE_DICT)
25                  self.POSE_TOPIC.publish(POSE_STAMPED)
26                  self.POSE_FLAG = True
27      else:
28          self.POSE_FLAG = None
29      #  If no map present, check server state for MAP_AT_SERVER
30      #  and download to MAP_DIR if it's available from server.
31      if(CUR_SERV_STATE == 'MAP_AT_SERVER'):
32          if(not self.MAP_FLAG):
33              self.ARM_CLIENT.RecvMap(self.MAP_DIR)
34              self.MAP_FLAG = True
35      else:
36          self.MAP_FLAG = os.path.isfile(self.MAP_DIR + '/map.pgm') and \
37              os.path.isfile(self.MAP_DIR + '/map.yaml')
```

5.2.5 BinBot Client Results

The BinBot client node is identical to the ArmBot client node with some reduced functionality. Firstly, the BinBot client node only uploads 1 state to the server, where-as the ArmBot client uploads state for the ArmBot and BaseBot. Secondly, the BinBot client does not have the functionality to upload images, as only the ArmBot requires this user-decision check.

5.2.6 MapBot Client Results

The MapBot client is similar to the BinBot client, except for a different work callback because of the different tasks for the robot. The main work thread ('WorkCallback()') for the MapBot client node is shown in listing 5.14.

Listing 5.14: Main function for the MapBot client node.

```
1     ######################################################
2     #  Work callback, executed at RATE (defined at top).
3     def WorkCallback(self):
4         #  Publish current server state.
5         CUR_SERV_STATE = self.MAP_CLIENT.ServState()
6         self.SERV_STATE.publish(CUR_SERV_STATE)
7         #  Display current server state (if updated).
8         if(self.PRINT_FLAG != CUR_SERV_STATE):
9             self.PRINT_FLAG = CUR_SERV_STATE
10            rospy.loginfo(self.PRINT_FLAG)
11        #  When server state is MAP_DONE, the mapping is stopped,
12        #  the map is saved, the '.yaml' file is amended and both
13        #  the '.pgm' and modified '.yaml' are sent to the server.
14        if(CUR_SERV_STATE == 'MAP_DONE'):
15            if(not self.MAP_FLAG):
16                subprocess.call('rosrun map_server \
17                    #map_saver -f map',  shell=True)
18                YAML_DATA = self.ReadFile(self.MAP_DIR + '/map.yaml')
19                YAML_DATA = self.FixYAML(YAML_DATA)
20                self.MAP_CLIENT.SendFile(YAML_DATA, 'MAP_YAML')
21                PGM_DATA = self.ReadFile(self.MAP_DIR + '/map.pgm')
22                self.MAP_CLIENT.SendFile(PGM_DATA, 'MAP_PGM')
23                self.MAP_FLAG = True
24        else:
25            #self.MAP_FLAG = os.path.isfile(self.MAP_DIR + '/map.pgm') and \
26                #os.path.isfile(self.MAP_DIR + '/map.yaml')
27            self.MAP_FLAG = None
```

The MapBot begins mapping when the server stays in 'START', indicating that there is no available map for the system. The MapBot will begin to automatically map until the map is complete. If the map is unfinished, the user can finish the map by giving the necessary move goals using RViz. When the user decides that the map is finished, the server is put into the 'MAP_DONE' state. When in 'MAP_DONE', the MapBot will stop recording a map and save the current data (line 16), modify/fix the '.yaml' to ensure that the map works on all robots (i.e. change the name of the 'map.yaml' name region, lines 18 and 19) and upload the '.pgm' and '.yaml' files to the server (lines 20 and 22).

5.2.7 FlyBot Client Results

The integration of the FlyBot into the object retrieval process is still under development, as issues with the network quality and wind-drafts made it very difficult for the drone to properly localise. Because of this, the FlyBot client node was developed to simulate this functionality by allowing the user to upload the object pose through the server using RViz and the arena map. The 'WorkCallback()' function for the FlyBot client node is displayed in listing 5.15.

Listing 5.15: Main function for the FlyBot client node.

```
1   ####################################################
2   # Work callback, executed at RATE (defined at top).
3   def WorkCallback(self):
4       CUR_SERV_STATE = self.FLY_CLIENT.ServState()
5       # Display current server state (if updated)
6       if(self.PRINT_FLAG != CUR_SERV_STATE):
7           self.PRINT_FLAG = CUR_SERV_STATE
8           rospy.loginfo(self.PRINT_FLAG)
9       # If no map present, check server state for MAP_AT_SERVER
10      # and download to MAP_DIR if it's available from server.
11      if(CUR_SERV_STATE == 'MAP_AT_SERVER'):
12          self.MAP_FLAG = os.path.isfile(self.MAP_DIR + '/map.pgm') and \
13              os.path.isfile(self.MAP_DIR + '/map.yaml')
14          if(not self.MAP_FLAG):
15              self.FLY_CLIENT.RecvMap(self.MAP_DIR)
16              self.MAP_FLAG = True
17      # Reset map flag variable, removing existing
18      # files on reset.
19      elif(CUR_SERV_STATE == 'RESET'):
20          if(os.path.isfile(self.MAP_DIR + '/map.pgm')):
21              os.remove(self.MAP_DIR + '/map.pgm')
22          if(os.path.isfile(self.MAP_DIR + '/map.yaml')):
23              os.remove(self.MAP_DIR + '/map.yaml')
24          self.MAP_FLAG = None
```

5.2.8 ROS Data Transformations

To transmit ROS data formats using socket connections, it must be transmitted as a string. To allow for easier reconstruction after upload, the ROS data formats are converted to more manageable formats. The poses that are being published for the Arm, Base, Bin and MapBots consist of ROS 'PoseStampedWithCovariance' data types. The 'PoseStamped' and 'PoseStampedWithCovariance' messages can be mapped to a dictionary object using python, and functions to convert back and forth between the ROS pose formats and dictionary's have been created. The file 'pose_dict_tf.py' contains functions 'PoseStampedToDict()', 'PoseCovarianceToDict()' and 'DictToPoseStamped()', which convert between ROS pose formats and dictionaries.

Another transformation is used to convert the robot pose dictionaries from Quaternion (x, y, z, w) co-ordinates to Euler (pitch, roll, yaw) co-ordinates. This is done using the 'tf.transformations' functions 'euler_from_quaternion()' and 'quaternion_from_euler()'.

The 'QuaternionToEulerDict()' and 'EulerToQuaternionDict()' allow for conversion between both formats and are included in the 'pose_dict_tf.py' file. The 'QuaternionToEulerDict()' function is displayed in listing 5.16.

Listing 5.16: Function to convert Quaternion co-ord. dict. to Euler co-ord. dict.

```
##########################################################
#  Converts quaternion dictionary to euler dictionary.
def QuaternionToEulerDict(DICT_IN):
    QUAT = [DICT_IN['pose']['orientation']['x'],
            DICT_IN['pose']['orientation']['y'],
            DICT_IN['pose']['orientation']['z'],
            DICT_IN['pose']['orientation']['w']]
    EUL = euler_from_quaternion(QUAT)
    TF_DICT = {}
    TF_DICT['pose'] = {}
    TF_DICT['pose']['orientation'] = {}
    TF_DICT['pose']['orientation']['roll'] = EUL[0]
    TF_DICT['pose']['orientation']['pitch'] = EUL[1]
    TF_DICT['pose']['orientation']['yaw'] = EUL[2]
    DICT_IN['pose']['orientation'] = TF_DICT['pose']['orientation']
    return DICT_IN
```

The ArmBot camera records image data in the ROS 'sensor_msgs/Image' format, which is incompatible with most image viewing software and so needs to be converted to a suitable format. The 'ros_image_conv.py' file contains the 'RosImageToPPMString' function, which takes an input 'sensor_msgs/Image' data string and converts it to a '.ppm' string. The '.ppm' format is a common format which can be converted to a '.png' server-side, to allow for compatibility with the iPad. The 'RosImageToPPMString()' function is shown in listing 5.17.

Listing 5.17: Function to convert 'sensor_msgs/Image' to '.ppm'.

```
def RosImageToPPMString(SENSOR_IMAGE):
    IMAGE = SENSOR_IMAGE
    IM_CONV = ''
    IM_CONV = IM_CONV + 'P6\n' + \
        str(IMAGE.width) + ' ' + str(IMAGE.height) + '\n'
    IM_CONV = IM_CONV + '255\n'
    print 'sensor_msgs/Image to .ppm:\n [resolution: ' + \
        str(IMAGE.width) + 'x' + str(IMAGE.height) + ']'
    for y in xrange(IMAGE.height):
        for x in xrange(IMAGE.width):
            RED_IDX = int(y * int(IMAGE.step) + 3 * x)
            GREEN_IDX = RED_IDX + 1
            BLUE_IDX = RED_IDX + 2
            IM_CONV = IM_CONV + str(IMAGE.data[RED_IDX])
            IM_CONV = IM_CONV + str(IMAGE.data[GREEN_IDX])
            IM_CONV = IM_CONV + str(IMAGE.data[BLUE_IDX])
    print 'Conversion complete.'
    return IM_CONV
```

5.3 Robot Arm

5.3.1 Flashing Code to the Arm

When the code was uploaded for testing to the Arduino Uno supplied with the arm, the program became unresponsive. When it tried to execute the program, it simply failed to finish initialising. This appeared to be due to the serial buffer size of the Uno being too small at only 64 bytes. It could not handle the large volume of serial data being sent and received by the program, whilst simultaneously executing the code within the uArm control library. Since there didn't seem to be a way to fix the issue, the Uno was replaced with an Arduino Mega 2560. This board is an upgrade to the Uno with increased RAM size, a faster processor, as

well as a larger serial buffer.

5.3.2 Accuracy and Repeatability

With the ability to send and receive data to and from the arm, it is now possible to control it as desired. From within the ROS environment, the controller has access to any topic that is published to the roscore. In order to test this functionality and to run some accuracy and repeatability tests on the arm, a simple script was created to repeatedly move the arm between two points. A pen was mounted to the end of the arm to leave a mark on a piece of paper at each location, shown in figure 5.10.

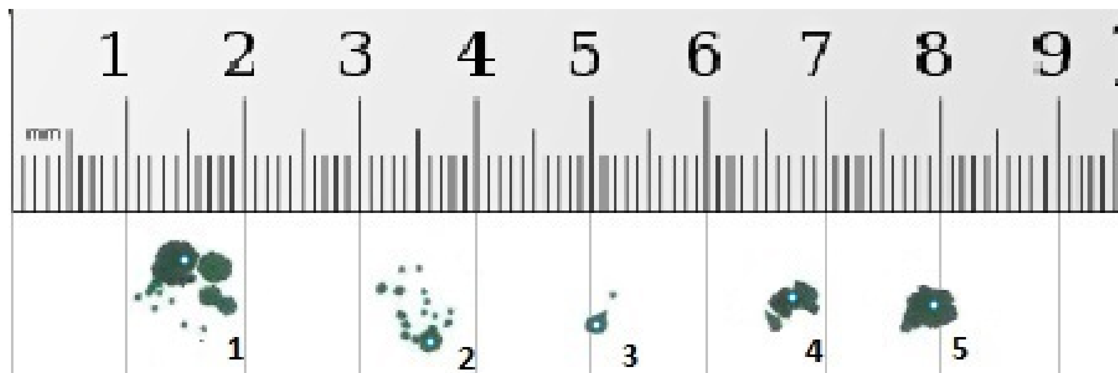


Figure 5.10: Accuracy and repeatability test results.

The arm was told to move to 5 different positions on a plane in the work envelope. The marks made by the pen are green, and the white spots were added to show the position on the paper the arm was aiming for. The ruler along the top of the image is for scale and the results are analysed below:

1. This position was 250mm from the base of the arm. Some points to note:
    - When the arm moves to the position at full speed, the end effector can forced down into the paper before reaching its target. This is due to the design of the arm, and poor resolution of the servos. This is shown by the little spots located down and to the left of the larger spots. Where the pen was held for a longer duration, the spots are larger.
    - As the test was repeated, it appeared to hit the target many times, but occasionally missed it.
2. After the previous test, it was noted that if the pen was held in position touching the paper for too long, a larger spot was drawn. The pen was therefore moved out of position straight away for each subsequent test. The target position was 200mm from the base of the arm. Some points to note:
    - Similar outcome to the previous test.

- The arm appears to have low accuracy near its range limit.

3. This position was only 100mm from the base. Some points to note:
   - The arm only missed the target by a significant margin once.
   - When the target is closer, the accuracy increases

4. Both position 4 and 5 were 150mm from the base. Some points to note:
   - These points were located roughly halfway between the inner and outer limits of the arm's reach.
   - Position 4 was to the right of the arm's work envelope, and 5 was nearer the middle.
   - The arm managed to reach the target position with decent accuracy each time, but it was not able to do it as accurately as it did for position 3.

uFactory states that the worst-case accuracy of the arm is 6mm-10mm. This appears to match the results obtained from these tests. It is important to note that the difference in accuracy between positions close and further away to the base arise due to the accuracy in the servos, and the geometry of the work envelope. Since the motors are only basic servos, they only have a resolution of 180°. The arm can only reach positions that can be defined with integer values. The servo that determines the rotation of the arm will also not be able to position itself to align the arm along the radial lines emanating from the base. This effect is shown in figure 5.11. This has the effect of reducing the accuracy of the arm when trying to position the end effector further away as the arm could be slightly to the left or right of its target position, with the servo thinking it has reached its target.

Since the objects to be manipulated are 70mmx70mm, even at full extension, accuracy of 6-10mm is acceptable.

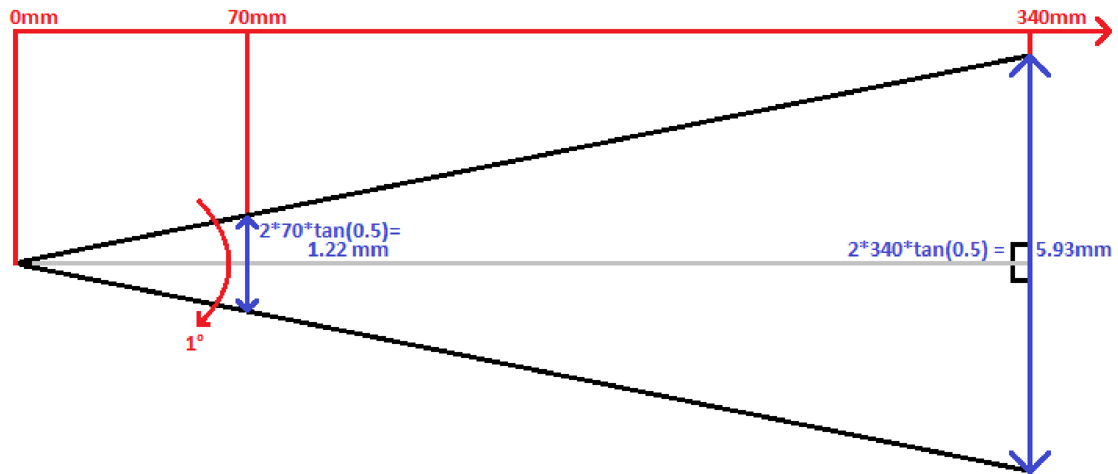


Figure 5.11: Effect of targets distance from base on accuracy. Black lines are possible positions the servos can take. Angle between them 1°.

5.3.3 Final Design of State Machine for the Arm

To keep track of what the arm is meant to be doing at any given time, a finite state machine was developed. The state machine transitions between states when it has finished tasks, or is prompted to by either the robot base, or the server. The full design of the state machine can be seen in figure 5.12.

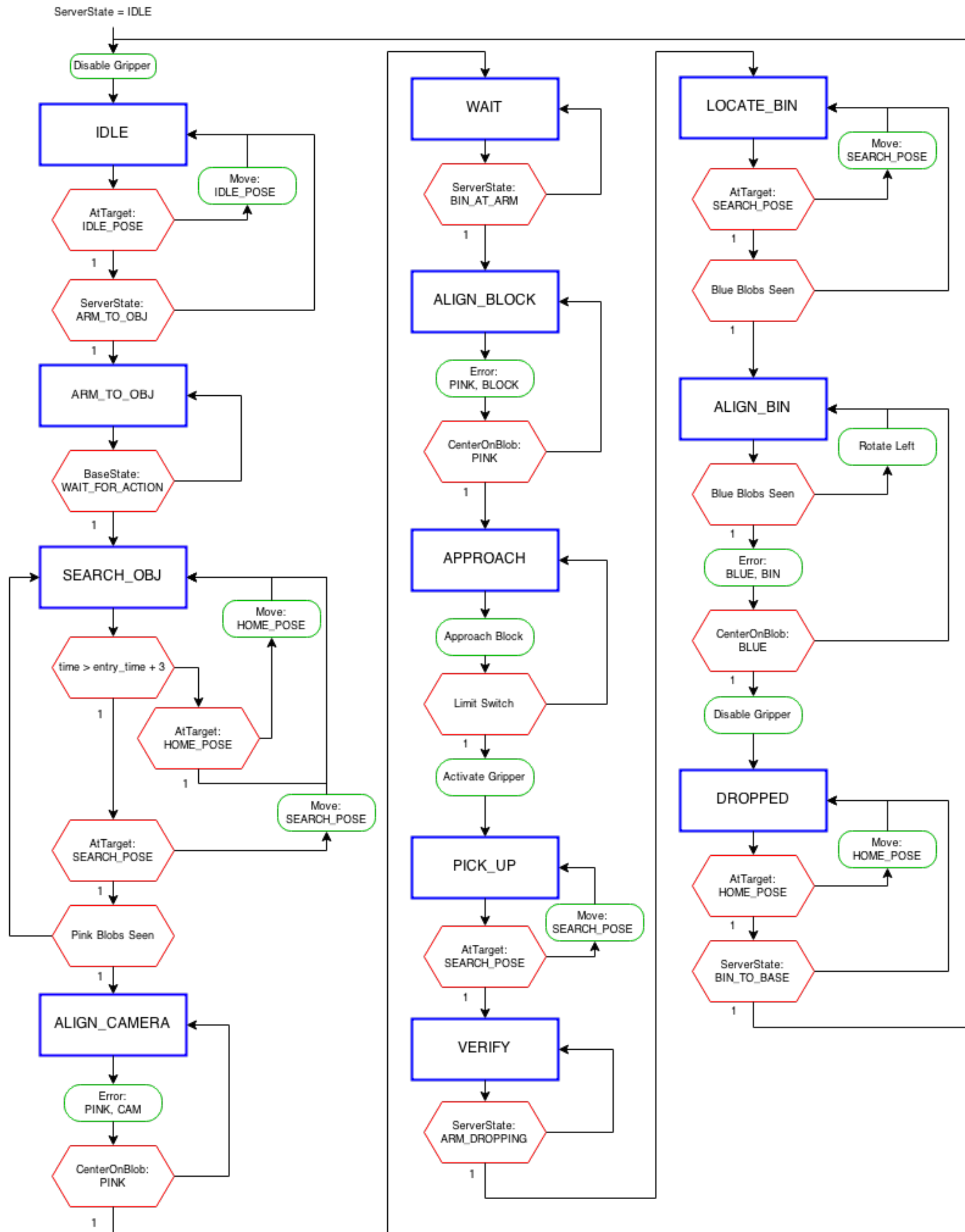


Figure 5.12: Full Finite State Machine for the Robot Arm.

Each state was implemented as a class (each defined as a state object), to allow for clearly defined entry and exit functionality for each state. When the state is not in transition, the execute function is called repeatedly until the state is transitioned. The basic outline of the state object is shown below:

```
#=================================================================
# Definition of Base State Class
#=================================================================
class State(object):

    def __init__(self, FSM, Arm):
        self.FSM = FSM
        self.Arm = Arm

    def Enter(self):
        pass

    def Execute(self):
        pass

    def Exit(self):
        pass

    def ReturnName(self):
        pass
```

To transition between states, a transition class object was defined to keep track of current transition, like so:

```
#=================================================================
# Transition class. Called when ToTransition set
#=================================================================
class Transition(object):

    def __init__(self, toState):
        self.toState = toState

    def Execute(self):
        print("Transitioning...")
```

The last part of the FSM definition is the FSM class itself. This class allows for the creation of states and transitions from the main of the program.

```
#=================================================================
# Top-level of the Finite State Machine
#=================================================================
class FSM():
    # Initialise finite state machine
    def __init__(self):
        self.states = {}
        self.transitions = {}
        self.curState = None
        self.prevState = None
        self.trans = None

    # Each transition should be added by calling this function
    def AddTransition(self, transName, transition):
        self.transitions[transName] = transition

    # Adding a state to the FSM is done using this function
    def AddState(self, stateName, state):
        self.states[stateName] = state
        print 'Adding State: ' + stateName

    # Used to set state manually
    def SetState(self, stateName):
        self.prevState = self.curState
        self.curState = self.states[stateName]

    # When transitioning between states, use this function
    def ToTransition(self, toTrans):
        self.trans = self.transitions[toTrans]

    # Call repeatedly from a loop. When a transition has been added,
    #   the state machine will call the exit function from current
    #   state, set the new state and call its entry function, and
    #   reset the transition.
    # The execute function of each state is called on every function
    #   call.
    def Execute(self):
        if(self.trans):
            self.curState.Exit()
            self.trans.Execute()
            self.SetState(self.trans.toState)
```

```python
            self.curState.Enter()
            self.trans = None
        self.curState.Execute()
```

All the transitions and states are stored in dictionaries, with strings as identifiers. As such, they are initialised like so:

```python
# State initialisation
fsm.AddState("IDLE",          IDLE(fsm, rArm))
# Transition Initialisation
fsm.AddTransition("to_IDLE",          Transition("IDLE"))
```

The main function then calls the execute function of the FSM on every loop, like so: `fsm.Execute()`.
Here is an example of a fully implemented state:

```python
    class SEARCH_OBJ(State):

    def __init__(self, FSM, Arm):
        super(SEARCH_OBJ, self).__init__(FSM, Arm)
        self.EntryTime = None

    def Enter(self):
        rospy.loginfo("Entering SEARCH_OBJ State")
        self.EntryTime = rospy.get_time()

    def Execute(self):
        if rospy.get_time() < self.EntryTime + 3.3:
            if self.Arm.AtTarget(HOME_POSE) is False:
                self.Arm.Move(HOME_POSE, 0.1)
        else:
            if self.Arm.AtTarget(SEARCH_POSE) is False:
                self.Arm.Move(SEARCH_POSE, 0.1)
            elif self.Arm.PinkBlobsSeen() is True:
                self.FSM.ToTransition("to_ALIGN_CAMERA")

    def Exit(self):
        rospy.loginfo("Exiting SEARCH_OBJ State")

    def ReturnName(self):
        return "SEARCH_OBJ"
```

5.4 Simulators

5.4.1 Install and setup

Both simulators will need to be installed for Gazebo it was simply installed via the ROS Ubuntu repository, this includes some basic launch files although there are many more ROS packages which provide simulations to run in gazebo.

MORSE is available in the Ubuntu repository however only for Ubuntu 13.04 onwards, in these tests Ubuntu 12.04 was used in conjunction with ROS groovy, and MORSE 1.0. The git page was used to download MORSE, following the instructions on there it was compiled then installed. Compiling and installing MORSE is quite simple following the instructions, however the most important thing is to match the python versions of MORSE, ROS and Blender, without this MORSE will compile and work but it will not be able to interface with ROS/Blender. This can be quite a difficult task in older versions of ROS such as groovy. However newer versions of Ubuntu and ROS have an easier setup due to matching python version by default and Ubuntu 15.04 has MORSE with automatic middle-ware installation in its repository allowing as simpler installation as Gazebo.

5.5 Quad-copter

The 'tum_simulator' package was used to test all code before testing on the quad-copter, this package provides simulation files for gazebo, this allows gazebo to generate all the ROS topics the ARDrone would and recreate the data it would produce in the simulated environment. This was useful to test states transitions function correctly as-well as to test how to control the quad-copter initially without damaging the ARDrone.

5.5.1 Connecting

Connecting to the ARDrone was extremely easy this has been tested on many devices, desktop, laptop and Raspberry pi 2, all that is required is a wifi connection and the 'ardrone_autonomy' package. The only time when connection to the quad-copter was not possible was at bench inspection when there where many new temporary wifi networks that caused too much interference. As the ARDrone runs linux this allows many parameters for the wifi connection to be changed, such as the channel which did help avoid the majority of interference.

Table 5.1: Sample of Pings to the ARDrone

PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_req=1 ttl=64 time=2.65 ms
64 bytes from 192.168.1.1: icmp_req=2 ttl=64 time=1.32 ms
64 bytes from 192.168.1.1: icmp_req=3 ttl=64 time=1.04 ms
64 bytes from 192.168.1.1: icmp_req=4 ttl=64 time=0.874 ms
64 bytes from 192.168.1.1: icmp_req=5 ttl=64 time=0.721 ms
64 bytes from 192.168.1.1: icmp_req=6 ttl=64 time=1.56 ms
64 bytes from 192.168.1.1: icmp_req=7 ttl=64 time=0.838 ms
64 bytes from 192.168.1.1: icmp_req=8 ttl=64 time=0.840 ms
64 bytes from 192.168.1.1: icmp_req=9 ttl=64 time=0.905 ms
64 bytes from 192.168.1.1: icmp_req=10 ttl=64 time=3.39 ms
64 bytes from 192.168.1.1: icmp_req=11 ttl=64 time=3.52 ms
64 bytes from 192.168.1.1: icmp_req=12 ttl=64 time=0.824 ms
64 bytes from 192.168.1.1: icmp_req=13 ttl=64 time=4.43 ms
64 bytes from 192.168.1.1: icmp_req=14 ttl=64 time=0.894 ms
64 bytes from 192.168.1.1: icmp_req=15 ttl=64 time=0.944 ms
64 bytes from 192.168.1.1: icmp_req=16 ttl=64 time=1.14 ms
64 bytes from 192.168.1.1: icmp_req=17 ttl=64 time=0.778 ms

The pings show that for the majority of the time there is a good connection, but there is a periodic lag spike where pings increase and data being received from the quad-copter slows down or freezes.

5.5.2 Colour Detection

To implement colour detection a node was created that, will subscribe to all blobs produced by 'cmvision' this node requests the quad-copters position, when it receives a blob then sends the position plus a small offset depending on where abouts in the image the blob was detected, if there has not already been a blob sent within 1 metre of that location. The offset of the position sent based on the blobs location within the image was calibrated by knowing the quad-copter will be flying at a set height then testing the camera to find scaling constants. This node also sends colour marker blobs to a separate topic for the controller node except that those blobs are repeated whenever they are seen.

5.5.3 Navigation

The Controller node will produce 'cmd_vel' based on the current state and room, in state 0 the quad-copter is landed so not moving, in state 1 the quad-copter moves in a preset direction depending on what room it is in. State 2 is used to align the quad-copter over the coloured marker, then state 3 is used to update the coordinates to the coordinates of the room. The node resets in state 4 where the quad-copter lands or whenever the the quad-copter lands.

This nodes states and transitions are shows in figure 5.13 this shows that the node starts in the landed state 0, the node subscribes to the 'navdata' topic which provides the node with the quad-copters state. When the quad-copters state is no longer landed the node will move to state 1. State 1 is used to move the quad-copter towards the next colour marker in the sequence, the direction depends on what rooms its in. State 2 is used to align the quad-copter over the marker so it can get an accurate update of its coordinates. State 3 publishes the update pose to the localisation node, this pose is set within the controller, each colour marker needs to have a pose given to it. State 4 is used to land the quad-copter when the node has reached room 8.

This node worked very well in simulation where sensor error was minimal, 5.14 this figure shows the simulation used to test the controller the quad-copter follows a serpentine path of alternating coloured markers. In simulation different markers are simulated using red balls and wood boxes, these were used as they are two contrasting colours that already exist on Gazebo. However in practice the quad-copter struggled to move smoothly around the arena. It was later determined when the quad-copter was directly over a wall it would hover almost perfectly, however in the middle of rooms or positions where air currents could be redirected back towards the quad-copter it would be blown off course. This presented problems to determine how far to move in each direction, as each part of the arena required different amounts of thrust to travel the same speed. To correct for this more sensor data will be needed.

The quad-copter could also struggle to maintain its orientation while moving around the arena, as this was unanticipated, there was no data that could provide

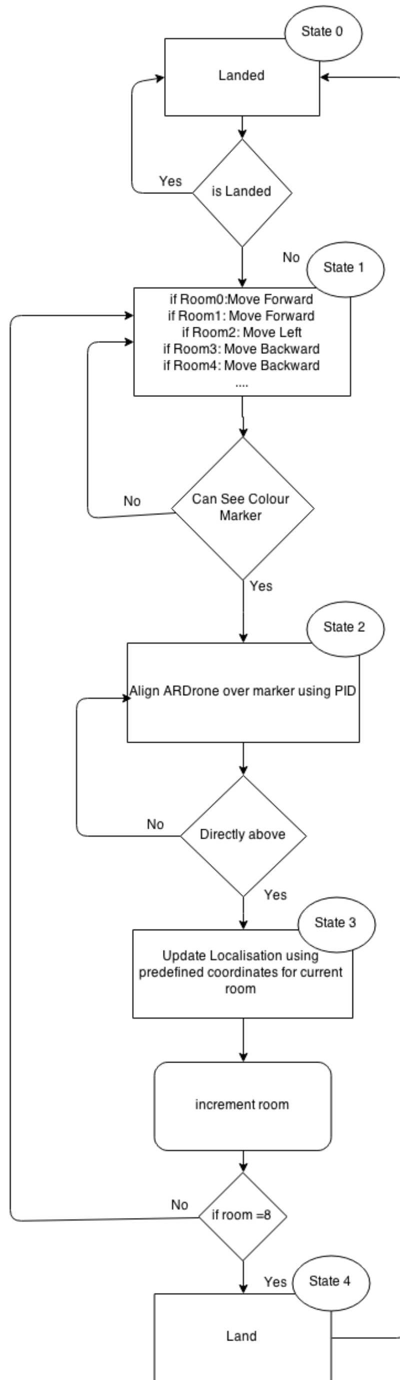


Figure 5.13: ASM of the controller node showing states and state transitions.

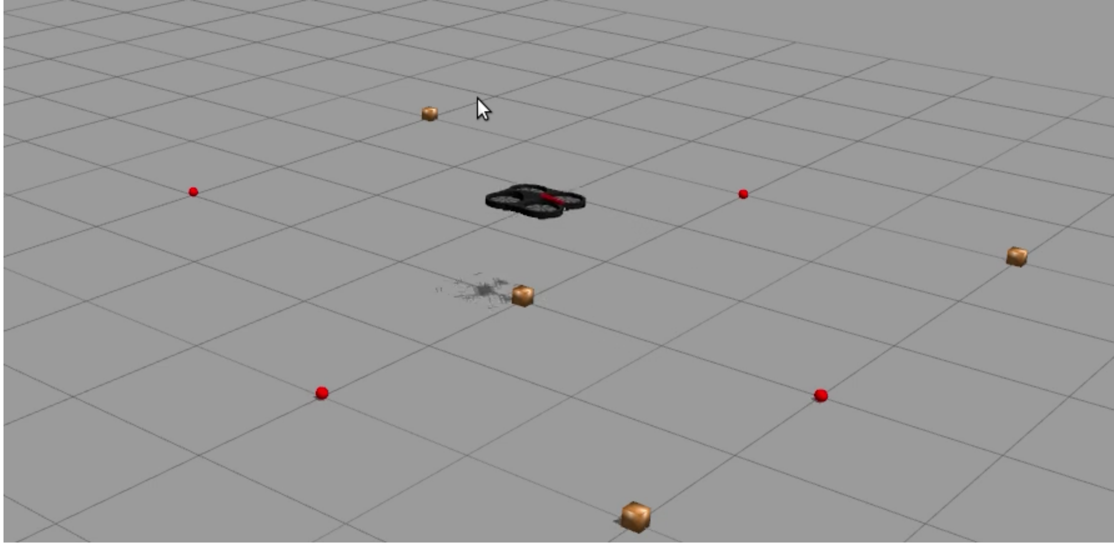


Figure 5.14: Controller node Simulation

reliable feedback of orientation.

The wireless network reliability made centering the quad-copter over the colour markers very difficult, as it was common for the connection to the quad-copter to drop out for up to 3 seconds. By this time the quad-copter had drifted off course and needed manual intervention to get back to a known location. However this behaviour is quite predictable, as the ARDrone has on board processing it simply continues doing whateve it was doing when the connection dropped, this could allow the ARDrones position to be re-estimated. An attempt to implement a recovery behaviour after a connection drop out was to reverse the quad-copters direction so it would go back to where it was when the connection dropped.

5.5.4 Localisation

In simulation gazebo generated a pose for the quad-copter automatically, this had a small amount of error introduced into it, however using this it was easy to develop a node that would publish the objects position. Therefore it was needed that a node be created that would generate an estimated pose, as there was no single sensor that could provide this data, the ARDrones input velocities would be logged if the quad-copter was ideal this would be sufficient. The ARDrone is not ideal so the quad-copters position will be updated from the controller, when it is above a colour marker.

In practice in certain parts of the map, usually in the middle of rooms, the quad-copter would drift away from its estimated position, this was because of the air turbulence generated from the arena, that was pushing the quad-copter out of position. Only when the quad-copter moved over a marker would the position correct its self, this allowed the error generated here to be measured.

Table 5.2: Sample of Data Showing Quad-copter Drift from Estimated Pose

Estimated	Actual	Difference
x:3.32, y:-4.41	x:1.97, y:-3.64 -68	x:1.35, y:-0.77
x:2.85, y:-3.93	x:3.04, y:-3.42	x:-0.19, y:-4.12
x:3.9, y:-5.91	x:1.92, y:-3.89	x:1.98, y:-7.89

5.6 iOS Application

5.6.1 Sending pose from iOS Application:

The functionality to send user generated poses is implemented within the application at the final stage of completion. However due to the reason that it is implemented very late in the project it is decided by the group that it is not practical to test it as many other changes will need to be catered for this functionality to work. A demonstration however is shown in the image below of the functioning user sent poses by tapping on the map and sending poses to the robot-image labels and robot-image view.

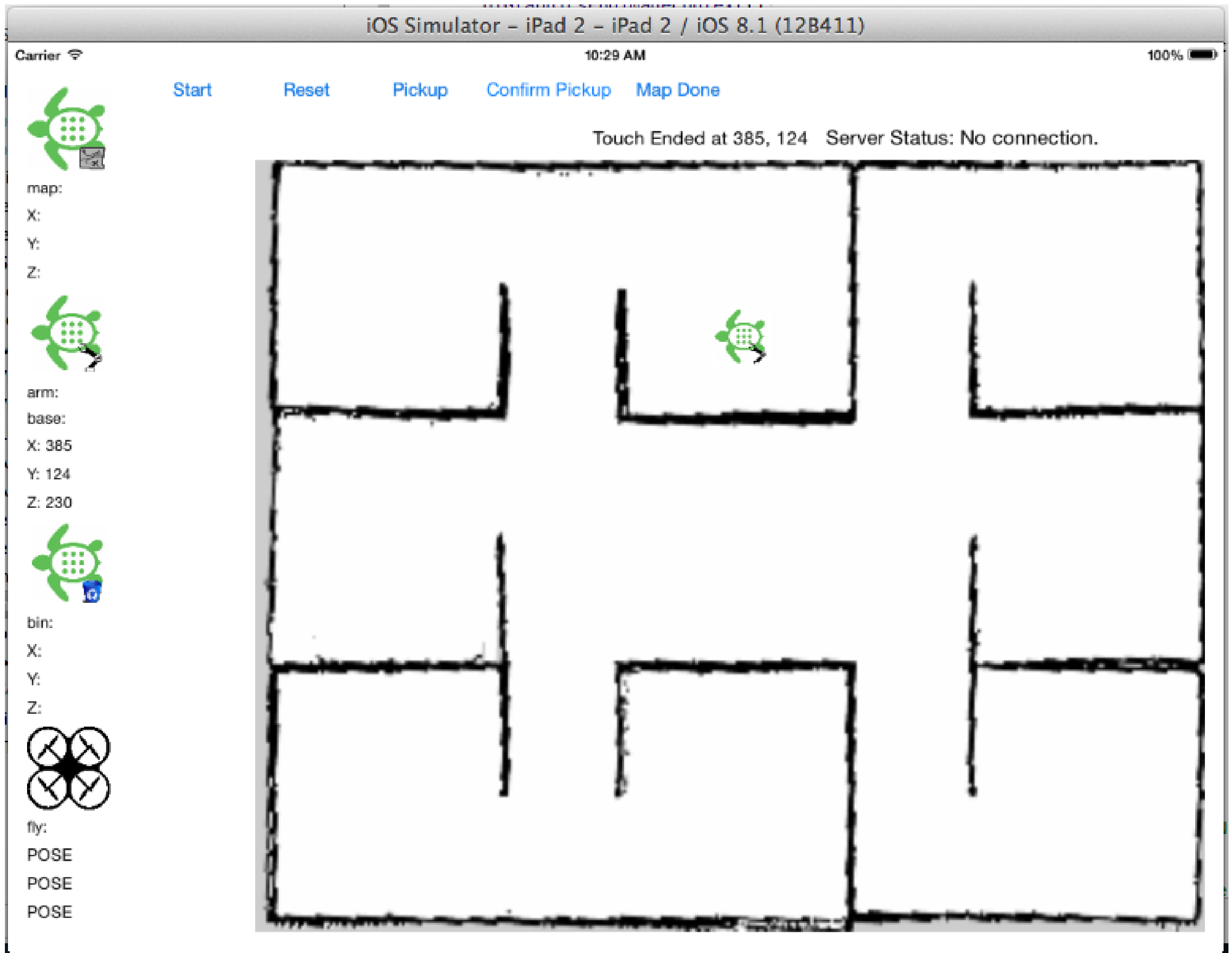


Figure 5.15: Demonstrating user sent poses by tapping on map view area

5.6.2 Receiving Pose, State and Images from Server

- It can be seen from the figure below that receiving poses for the currently active robots is fully functioning and being refreshed at a period of every 0.5 seconds. This is considered enough to display smooth motion, included with the timed animation, at the same instance it does not overload the server or connection with unnecessary traffic.
- The states are refreshed at a lower rate of every 1.0 seconds as they do not change as frequently or quickly as the poses do.
- When user decision is required the popover view controller is loaded when a specific task is to be carried out and the image is loaded successfully. In the figure below it is demonstrated by pulling the map.png file and loading it in the image view of the popover view controller.
- The loaded image also shows the position and angular tilt of the most recent map to be used for the robot-team, the map is updated and then kept

consistent for further testing to yield the most regular results.

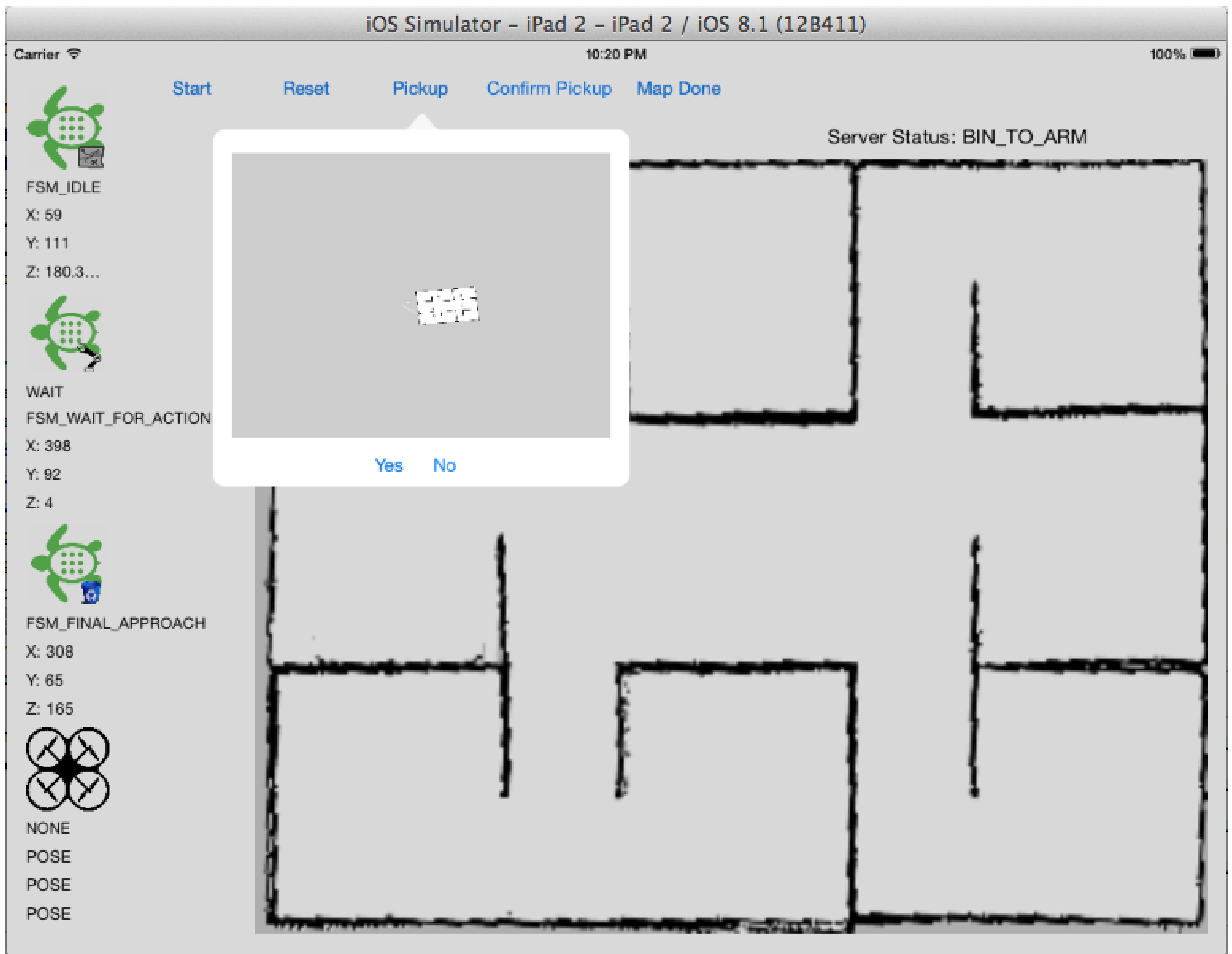


Figure 5.16: Displaying data and demonstrating an Image loaded from server

5.6.3 Warping Image and Unsuccessful Loading of Image in image-VC:

- The custom SimpleSocket.h/SimpleSocket.m library is used for downloading images from the server with a socket time-out of zero and no packet recovery if packets are lost during transmission. The library is very primitive for this project but the GCDAsyncSocket library method of receiving the NSData format is not interpreted in time to make full use of it. However the GCDAsyncSocket library does have a method to append data read from a socket and put it in an NSData format, but the image file created thereafter is corrupted and varying in size with each attempt at downloading an image from the server.

- The method for motion animation works in the vertical and horizontal plane but the angular plane is not functioning as expected. The figure below depicts an instance when the image disappears, warp or is squished away using *CGAffineTransformMakeRotation* function. This function is used for rotating *CGContextRef* [46] which is a graphics context containing drawing parameters to render an image or paint on a destination i.e. window in an application for a bitmap image for instance. Performing a static test rotating a UIImageView of a robot on the map with fixed vertical, horizontal and angular variables is successful. The irregularity which causes a robot-image to 'warp' is due to the fact that the image-view is not translated in a CGContextRef as a graphics context. Please refer to the repository [40] for the commented code.

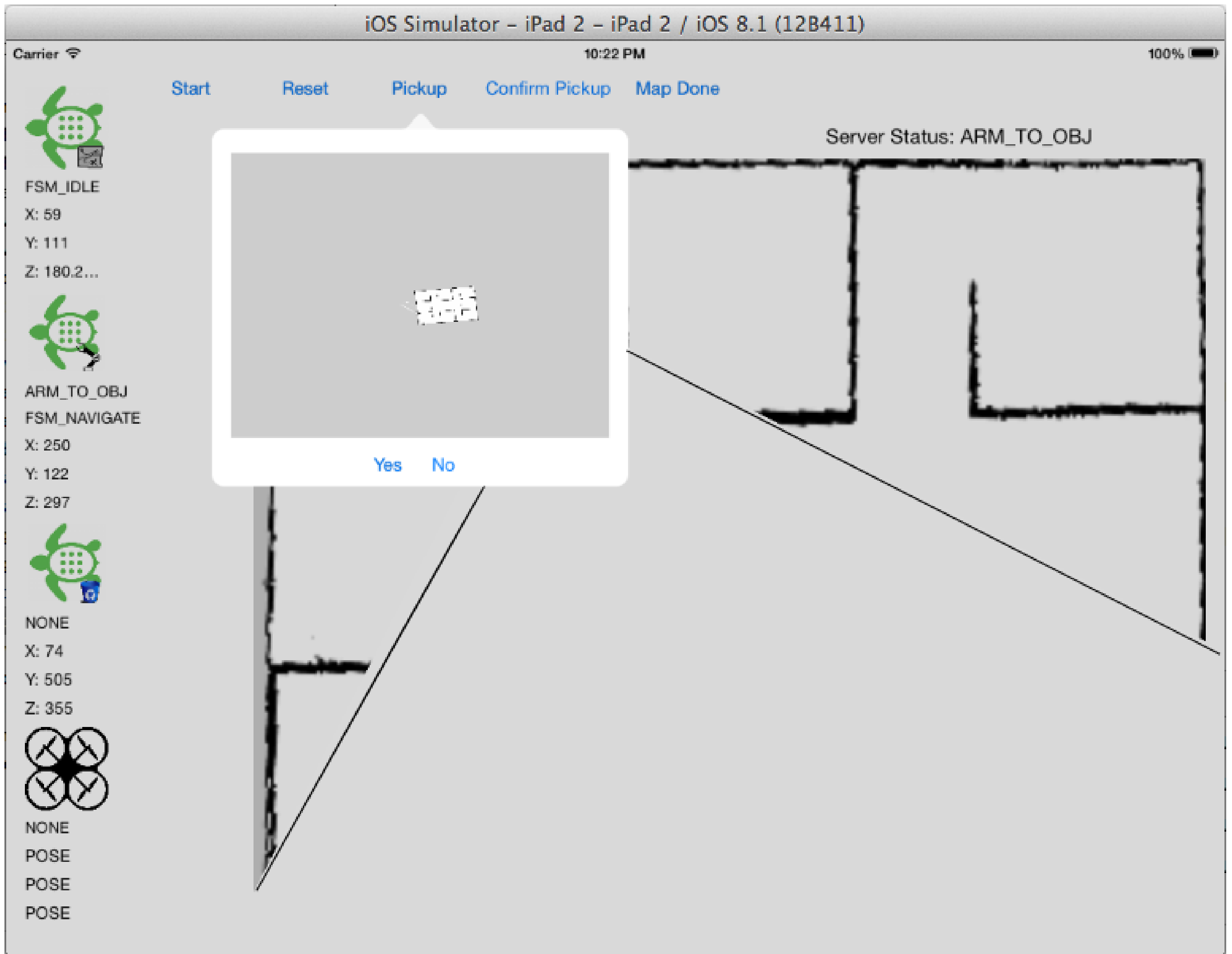


Figure 5.17: Showing Warping effect with black lines to highlight fault

Chapter 6

Discussion and Conclusions

6.1 Future Work

For the communications server, the future work lies in adapting the network configuration so that a distributed server system is achieved. To do this, each robot will need to host a server and client and continuously transmit and recieve data with all other robots in the team. A script to compare the current data for each robot would allow a decision to be made on which robot has the most up-to-date data.

To further improve the arm's capabilities, the following improvements can be made. Modelling the current position of the end effector in relation to the base of the robot using ROS tfs. This would allow the arm to know where the object and container are prior to moving, instead of relying soley on the camera. This is especially useful for placing the object in the container, as the carrier robot can tell the arm robot its current position, and adjust the reach of the arm so that it will be able to see the container every time. This would also require conversion from cartesian coordinate system to cyclindrical co-ordinates in order to move the arm to those target positions. Replacing the arm's servos so that they provide more feedback would allow for more accurate control, so that the controlling program can keep track of the arm's physical location. Adding a sensor to the pump so that it can tell when an object has been picked-up, which can be done by using a flow sensor as when an object is not picked-up, air will flow freely through the tube.

To further improve the application, the layout can be made to look more professional as well as including an application starting icon and launch screen. Another possibility is to allow the user to change the host IP and port at any time, as well as map offsets if a new map is loaded. When a new map is loaded, it could also be possible to allow the user to re-size and rotate the image from within the image window instead of using fixed values.

There is little left to test for Gazebo as it is so well supported, there are many simulations already available for it, there are many on-line guides, forums and documentation. Which make it the easiest simulator to use on ROS. Where as

MORSE is still developing and adding more features such as sensors and robot models, for this reason MORSE should be tested again in the future, on a newer version of Ubuntu and ROS, to keep track of the projects progress. However to run MORSE's more sophisticated graphics, the lab computers may need upgrading.

Navigation on the quad-copter is limited by its processing power, therefore it is a sensible idea to keep the navigation simple such as close waypoints with no object avoidance. This limits further progress on this although improvement of localisation or accuracy of sensor readings would give improvements for navigation.

To improve localisation further more data will be needed to track the quad-copter more accurately, this could be achieved with more sophisticated markers to update the quad-copters position. This would help as using colour detection, limits the amount of unique markers based on the cameras quality. Along with this the colour markers give no data on the quad-copters orientation.

Future work for the Ground-based Robot Team can be subdivided into the individual potential extensions that can be applied to each one of the robot members, in case of a potential continuation of the project. Since the only issue which was identified with the developed Mapping Robot, was its inability to perform autonomous exploration without the risk of collision, future approaches could focus on improving on this issue. Continuing, both the Arm and Carrier (Bin) Robots were limited by the measurement performance of the optical sensor used (Asus Xtion). In order to provide a solution to this limitation, future attempts can potentially add a Hokuyo lidar to each one of the platforms, which should enable both robots to detect objects which are closer than the minimum sensing distance of the currently employed optical sensor. Finally, a major limiting factor which was identified to have a tremendous effect on the performance of all robots, was the set of computer hardware used to execute all the required programs on each individual robot. The relatively poor, in terms of performance, default Asus laptops could not even closely supply the required computational power, while even the alternative laptops which were eventually used, demonstrated inability to appropriately run computationally expensive packages (such as 'amcl'). Thus, future work on this part of the project should optimally make use of significantly more powerful hardware.

6.2 Conclusions

By the end of the project the robot team can autonomously map the unknown environment, locate objects within the mapped arena, move to and pick-up those objects and handover the objects to another bot to safely return them to base, while allowing human interaction to verify robot actions. To achieve this a mapping robot was implemented that would autonomously map the unknown area using the Hokuyo laser scanner, locating the object within the unknown environment was achieved using the ARDrone being flown manually due to on board sensor performance and wireless network limitations. To pick up the object an Arm Bot was designed that would autonomously pickup the object once it was within line of

sight. To transport the object back to base a bot was modified to hold a bin, this bot would go to the Arm Bot, wait for the object to be placed in it's bin and then return to base with the object in its bin. However due to modifications which made them larger and sensor limitations the Arm/Bin Bots had minor difficulty passing through the arenas narrow corridors. To allow all the robots to communicate a centralised server was created that could run on a raspberry pi, this allowed each robot's state and position to be shared with the team.

Bibliography

[1] R. E. Kalman, "A New Approach to Linear Filtering and Prediction Problems," Research Institute for Advanced Study, Baltimore Md., 1960.

[2] M.S. Arulampalam, S. Maskell, N. Gordon, T. Clapp, "A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking," Defence Sci. & Technol. Organ., Adelaide, SA, Australia, 2002.

[3] F. Dellaert, D. Fox, W. Burgard, S. Thrun, "Monte carlo localization for mobile robots," Computer Science Department, Carnegie Mellon University, Pittsburgh PA 15213, 1999.

[4] R. Chatila, "Robot Mapping: An Introduction," *Springer Tracts in Advanced Robotics - Robotics and Cognitive Approaches to Spatial Mapping*, vol. 38, 2008.

[5] S. Thrun, "Simultaneous Localization and Mapping," *Springer Tracts in Advanced Robotics - Robotics and Cognitive Approaches to Spatial Mapping*, vol. 38, 2008.

[6] A. Stentz, "Optimal and Efficient Path Planning for Partially-Known Environments," The Robotics Institute; Carnegie Mellon University; Pittsburgh, PA 15213, 1994.

[7] P. E. Hart, N. J. Nilsson, B. Rafael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," AI Group of the Applied Physics Laboratory, Stanford Research Institute, Menlo Park, California, 1967.

[8] E. W. Dijkstra, "A Note on Two Problems in Connection with Graphs," Mattreoatisch Centlum 2e Boerhaavestraat 49 Amsterdam-O, 1959.

[9] J. Borenstein, Y. Koren, "The vector field histogram-fast obstacle avoidance for mobile robots," Adv. Technol. Lab., Michigan Univ., Ann Arbor, MI, USA, 1991.

[10] D. Fox, W. Burgard, S. Thrun, "The dynamic window approach to collision avoidance," *Robotics & Automation Magazine, IEEE*, vol. 4, no. 1, 1997.

[11] N. Y. Ko, R. Simmons, K. Reid, G. Simmons, "The Lane-Curvature Method for Local Obstacle Avoidance," 1998.

[12] uFactory, “Github - ufactory,” https://github.com/ufactory, June 2014.

[13] Youtube, “’Crazy Flie Automated Flight’,” https://www.youtube.com/watch?v=UzFwg2Fpv4E.

[14] Various, “ios,” May.

[15] Dr.T.Payne, “User interface design,” ”http://cgi.csc.liv.ac.uk/~trp/COMP327_files/LS8%20UIDesign%2014.pdf”, 2015.

[16] Various, “App store (ios),” May.

[17] uFactory, “Uarm - put a miniture industrial robot arm on your desk,” ”https://www.kickstarter.com/projects/ufactory/uarm-put-a-miniature-industrial-robot-arm-on-your”, January 2014.

[18] ——, “Acrylic replaceable kit,” hhttp://store.ufactory.cc/acrylic-replaceable-kit/”, January 2014.

[19] ABB, “Irb 460 - high speed robotic palletizer,” http://new.abb.com/products/robotics/industrial-robots/irb-460, May 2015.

[20] ——, “Image of irb 460 - high speed robotic palletizer,” http://abbib.cloudapp.net/public/default/product/9AAC171535/presentation, May 2015.

[21] AutonomyLab, “Ardrone_Autonomy Git Page,” https://github.com/AutonomyLab/ardrone_autonomy/tree/groovy-devel.

[22] Parrot, “ARDrone2.0 Specification Page,” http://ardrone2.parrot.com/ardrone-2/specifications/.

[23] N. Koenig, “cmvision - ros wiki,” ”http://wiki.ros.org/cmvision”, May 2015.

[24] “The coral group’s color machine vision project,” ”http://www.cs.cmu.edu/~jbruce/cmvision/”, May 2015.

[25] A. Hubers, “cmvision_3d - ros wiki,” ”http://wiki.ros.org/cmvision_3d”, May 2015.

[26] B. Gerkey, “gmapping - ros wiki,” ”http://wiki.ros.org/gmapping”, May 2015.

[27] S. Kohlbrecher, “hector_mapping - ros wiki,” ”http://wiki.ros.org/hector_mapping”, May 2015.

[28] S. Kohlbrecher, T. Graber, “hector_navigation - ros wiki,” ”http://wiki.ros.org/hector_navigation”, May 2015.

[29] E. Marder-Eppstein, “navigation - ros wiki,” ”http://wiki.ros.org/navigation”, May 2015.

[30] D. Fox, "Adapting the Sample Size in Particle Filters Through KLD-Sampling," Department of Computer Science & Engineering, University of Washington, Seattle, WA 98195, 2003.

[31] T. Foote, M. Ferguson, M. Wise, "turtlebot - ros wiki," "http://wiki.ros.org/turtlebot", May 2015.

[32] MORSE Project, "'What is MORSE ?" http://www.openrobots.org/morse/doc/stable/what_is_morse.html.

[33] Gazebo, "'gazebo_ros_pkgs'," http://wiki.ros.org/gazebo_ros_pkgs.

[34] uFactory, "uarm assembly instructions," "http://www.ufactory.cc/downloads/documents/uArm_Assembly_Instructions_v1.2.3.pdf", May 2014.

[35] ——, "Getting started with uarm v1.1," "http://www.ufactory.cc/downloads/documents/Getting_Started_with_uArm_v1.1.pdf", March 2014.

[36] G. Walley, "uarm rosart code on github," http://github.com/gWalley/uarm_scripts.git, May 2015.

[37] TeckNet, "Tecknet c016 hd webcam," http://www.tecknet.co.uk/c016.html, 2014.

[38] Logitech, "Logitech hd webcam c270," http://www.logitech.com/en-gb/product/hd-webcam-c270, 2015.

[39] Tum Computer Vision Group, "'tum_simulator' ROS wiki page," http://wiki.ros.org/tum_simulator.

[40] ROSART, "Robot Object Search and Retreival Team GitHub repository," https://github.com/rosart/Robot-Object-Search-and-Retreival-Team.git, May 2015.

[41] R.Hanson, "Cocoaasyncsocket," August.

[42] dfed, "square/socketrocket," May.

[43] rctoris, "rosbridge_suite," May.

[44] "actionlib ROS Wiki," http://wiki.ros.org/actionlib, May 2015.

[45] J. Price, "robot_comm," "https://github.com/jkprice07/robot_comm", 2015.

[46] A. Developer, "Cgcontext reference," "https://developer.apple.com/library/ios/documentation/GraphicsImaging/Reference/CGContext/index.html", 2015.

Appendices

```
Data: blobs_3d - Output list of 3-D blobs from 'cmvision_3d'
      filtered_blobs - Internal list of filtered blobs
      max_blob_area - Variable to store the larget recorded blob area
      blob_count - Variable to store the number of filtered blobs
      sum_x - Variable to store sum of x coordinates
      miss_counter - Variable to count number of callbacks without detecting blobs
Result: tracked_blob - Single optimized blob
        blobs_detected - Variable to indicate blob detection

Callback initialization;
for each blob in blobs_3d do
    if blob closer than 1.2 meters and below horizontal level then
        append blob to filtered_blobs_3d list;
        increment blob_count;
    end
end
if filter_blobs_3d contains blobs then
    set blobs_detected to true;
    reset miss_counter;
    for each blob in filtered_blobs_3d list do
        sum_x += x coordinate of blob;
        if blob area > max_blob_area then
            max_blob_area = blob area;
            tracked_blob = blob;
        end
        ;
    end
    avg_x = sum_x / blob_count;
    if no previously tracked blob then
        avg_z = z coordinate of tracked_blob;
    else
        if z coordinate of tracked_blob < avg_z then
            avg_z = (avg_z + 10(z coordinate of tracked_blob))/11;
        else
            avg_z = (10avg_z + (z coordinate of tracked_blob))/11;
        end
    end
    set x and z coordinates of tracked_blob to avg_x and avg_z;
    publish transform frame of tracked_blob;
else
    increment miss_counter;
    if miss_counter>=10 then
        reset all storage variables;
        set blobs_detected to false;
    end
end
```

Algorithm 1: Pseudo code describing the additional blob processing algorithm developed to optimize blob detection for the purposes of the application.