VISVESVARAYA TECHNOLOGICAL UNIVERSITY
BELAGAVI-590 018

Internship / Professional Practice Report

on

"Autonomous Surveillance Robot"

Submitted in partial fulfillment of the requirements for the final year degree of
Bachelor of Engineering in Computer Science and Engineering
of Visvesvaraya Technological University, Belagavi

by

Rahul Devajji	**1RN16CS078**
Sagnik Das	**1RN16CS086**
Yash Vora	**1RN16CS123**

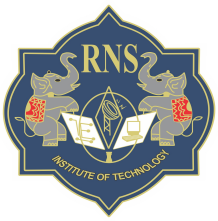
Carried out at

RoboThoughts

Under the guidance of:

Internal Guide:
Prof. Devaraju B M
Assistant Professor
Dept. of CSE

External Guide (Company):
Mr. Anbu Kumar
Founder and CEO
RoboThoughts



Department of Computer Science and Engineering
(NBA Accredited for academic years 2018-19, 2019-20, 2020-21)
R N S Institute of Technology
Channasandra, Dr. Vishnuvardhan Road, Bengaluru-560 098

2019-2020

R N S Institute of Technology
Channasandra, Dr. Vishnuvaradana Road,
Bengaluru-560 098

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
(NBA Accredited for academic years 2018-19, 2019-20, 2020-21)

INTERNSHIP CERTIFICATE

Certified that the Internship/ Professional Practice work entitled ***"Autonomous Delivery Robot"*** has been successfully carried out at **RoboThoughts**, by **Rahul Devajji** bearing USN **1RN16CS078**, **Sagnik Das** bearing USN **1RN16CS086** and **Yash Vora** bearing USN **1RN16CS123**, bonafide students of **RNS Institute of Technology** in partial fulfillment of the requirements for the final year degree in **Bachelor of Engineering Computer Science and Engineering** of **Visvesvaraya Technological University, Belagavi** during the academic year 2019-2020. It is certified that all corrections/suggestions indicated for Internal Assessment have been incorporated in the report deposited in the departmental library. The internship report has been approved as it satisfies academic requirements in respect of Internship work for the said degree.

Prof. Devaraju B. M	**Dr. G T Raju**	**Dr. M K Venkatesha**
Assistant Professor	**Vice-Principal**	**Principal**
	Professor, HoD-CSE	

External Viva:

Name of the Examiners	**Signature with date**
1.	
2.	

ABSTRACT

Surveillance is major thing when we are going to secure any thing as it is tedious job peoples are getting boarded because of that it will might risky to observing all this things we are going to make a robot which is continuously monitor thing. This robot continuously watch and sending a live streaming of it to a authorized person. Because of that monitoring the work will be some what easy and it will be make accurate because of technology.

The implementation of this project to resolve the problem of replacing human to surveillance robot, because of this we reduce harm of human resource. Robot are usually miniature in size so they are enough capable to enter in tunnels, mines and small holes in building and also have capability to survive in harsh and difficult climatic conditions for life long time without causing any harm. Military robots were designed from last few decades.

Nowadays, most of the system uses a mobile robot with a camera for surveillance. The camera mounted on the robot can move to different locations. These types of robots are more flexible than the fixed cameras. In it is given that mostly used surveillance robots are wheel robot. The wheel based robots are more suitable for flat platform. With the development in wireless communication and internet, the videos captured by wheel robot can be seen remotely on computer or laptop.

During the course of this internship, we built a simple ground vehicle platform which has the ability to move along people or any other obstacles. By adding camera, sensors and wireless connectivity the robot was made fully autonomous and also remotely controllable for difficult to pass through areas.

ACKNOWLEDGEMENTS

At the very onset, we would like to place on record our gratefulness to all those people who have helped us in making this internship work a reality. Coming up with this Internship work to be a success was not easy. Our Institution has played a paramount role in guiding us in the right direction.

We would like to profoundly thank the **Management of RNS Institute of Technology** for providing such a healthy environment for the successful completion of this Internship / Professional Practice work.

We would like to express our sincere thanks to our respected Director, **Dr. H N Shivashankar** for his constant encouragement that motivated me for the successful completion of this work.

We would also like to thank our beloved Principal, **Dr. M K Venkatesha**, for providing the necessary facilities to carry out this work.

We are extremely grateful to our beloved Vice–Principal and HoD-CSE, **Dr. G T Raju**, for having accepted to patronize me in the right direction with all his wisdom.

We place our heartfelt thanks to all the Coordinators- Internship / Professional Practice. We would like to thank the internal guide **Prof. Devaraju B. M** , Asst. Professor, for his continuous guidance and constructive suggestions for this work.

It's our pleasure to thank **RoboThoughts** for providing us the best platform to complete the internship work and glad to thank the extremal guide **Mr. Anbu Kumar**, for corporate exposure, training and guidance. Last but not the least, we are thankful to all the staff members of Computer Science and Engineering Department for their encouragement and support throughout this work.

Rahul Devajji 1RN16CS078
Sagnik Das 1RN16CS086
Yash Vora 1RN16CS123

Contents

List of Figures

Chapter 1

Introduction

1.1 Robotics

Robotics is a scientific and engineering discipline that is focused on the understanding and use of artificial, embodied capabilities. The people who work in this field (roboticists) come from mechanical engineering, electronic engineering, information engineering, computer science, and other fields. On the engineering side, roboticists deal with the design, construction, operation, and use of robots, especially through computer systems for their control, sensory feedback, and information processing. On the scientific side, roboticists study how a robot's environment and design affect how well it does its job.

1.2 Problem Statement

Develop a robot that will be able to navigate and map any foreign environment autonomously. The robot will stream the camera footage continuously to the mobile application managed by the handler. Robot should also have the remote -control feature to navigate in the event of failure of autonomous operation.

1.3 Proposed Solution

Develop an electric-powered robot equipped with a sensor suite that includes LIDAR, camera, GPS and inertial measurement unit. This robot will be guided by the algorithms running on the on-board system which takes input from all the sensors, processes them and then gives the input to move the wheels in the right direction. The handler will also be able to navigate the robot using the mobile application and the camera feed streamed to the app. We will employ the Robot Operating System (ROS) which will orchestrate the entire architecture by enabling the interfacing and communication between all the moving parts of the robot.

Chapter 2

Requirement Analysis

2.1 Hardware Requirements

- Raspberry Pi 4 Model B
- NVIDIA Jetson
- Sparkfun 9dof razor imu m0
- Rplidar A2
- USB Camera
- Internet Dongle
- Four nos of high torque motors with inbuilt encoders
- Four rugged wheels with Metal chassis
- Lipo Battery with charger

2.2 Software Requirements

- Ubuntu (18.04) installed with ROS (Melodic-Full Installation)
- Python 3.x
- C++
- Android app for remote control

2.3 Tools, Technologies and Platform

2.3.1 Robot Operating System

Robot Operating System (ROS) is robotics middleware (i.e. collection of software frameworks for robot software development). Although ROS is not an operating system, it provides services designed for a heterogeneous computer cluster such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management. Running sets of ROS-based processes are represented in a graph architecture where processing takes place in nodes that may receive, post and multiplex sensor data, control, state, planning, actuator, and other messages. Despite the importance of reactivity and low latency in robot control, ROS itself is not a Real-Time OS (RTOS). It is possible, however, to integrate ROS with real-time code. Software in the ROS Ecosystem can be separated into three groups:

- Language-and platform-independent tools used for building and distributing ROS-based software.
- ROS client library implementations such as roscpp, rospy and roslisp.
- Packages containing application-related code which uses one or more ROS client libraries.

The main ROS client libraries are geared toward a Unix-like system, primarily because of their dependence on large collections of open-source software dependencies. For these client libraries, Ubuntu Linux is listed as ”Supported” while other variants such as Fedora Linux, macOS, and Microsoft Windows are designated ”experimental” and are supported by the community.

2.3.2 Python

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python’s design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.

2.3.3 Ubuntu

Ubuntu is a free and open-source Linux distribution based on Debian. Ubuntu is officially released in three editions: Desktop, Server, and Core (for internet of things devices and robots). All the editions can run on the computer alone, or in a virtual machine. Ubuntu is a popular operating system for cloud computing, with support for OpenStack.

Ubuntu is released every six months, with long-term support (LTS) releases every two years. The latest release is 19.10 (”Eoan Ermine”), and the most recent long-term support release is 18.04 LTS (”Bionic Beaver”), which is supported until 2023 under public support.

2.3.4 Android

Android is a mobile operating system based on a modified version of the Linux kernel and other open source software, designed primarily for touchscreen mobile devices such as smartphones and tablets. Android is developed by a consortium of developers known as the Open Handset Alliance, with the main contributor and commercial marketer being Google.

2.3.5 Android App

An Android app is a software application running on the Android platform. Because the Android platform is built for mobile devices, a typical Android app is designed for a smartphone or a tablet PC running on the Android OS. Although an Android app can be made available by developers through their websites, most Android apps are uploaded and published on the Android Market, an online store dedicated to these applications. The Android Market features both free and priced apps.

Android apps are written in the Java programming language and use Java core libraries. They are first compiled to Dalvik executables to run on the Dalvik virtual machine, which is a virtual machine specially designed for mobile devices. Developers may download the Android Software Development Kit (SDK) from the Android website. The SDK includes tools, sample code and relevant documents for creating Android apps.

2.4 Devices

2.4.1 LIDAR

Lidar is a surveying method that measures distance to a target by illuminating the target with laser light and measuring the reflected light with a sensor. Differences in laser return times and wavelengths can then be used to make digital 3-D representations of the target.

RPLIDAR is a low cost LIDAR sensor suitable for indoor robotic SLAM application. It provides 360° scan field, 5.5Hz/10Hz rotating frequency with guaranteed 8 meter ranger distance, current more than 16m for A2 and 25m for A3 . By means of the high speed image processing engine designed by RoboPeak, the whole cost are reduced greatly, RPLIDAR is the ideal sensor in cost sensitive areas like robots consumer and hardware hobbyist.

2.4.2 Camera

This USB 2.0 camera module is sold by Ailipu Technology Inc. (Shenzhen) under the model name ELP-USBFHD01M. It comes with a resolution of 1280×720 @ 60 fps (MJPEG). It's ideal for many applications like security systems, portable video system, video phones, industrial machine monitoring and toys. It use high quality image sensors made by OmniVision, one of the world leaders in this field of electronics.

2.4.3 Inertial Measurement Unit

An Inertial Measurement Unit (IMU) is an electronic device that measures and reports a body's specific force, angular rate, and sometimes the orientation of the body, using a combination of accelerometers, gyroscopes, and sometimes magnetometers. IMUs are typically used to maneuver aircraft (an attitude and heading reference system), including Unmanned Aerial Vehicles (UAVs), among many others, and spacecraft, including satellites and landers.

An inertial measurement unit works by detecting linear acceleration using one or more accelerometers and rotational rate using one or more gyroscopes. Some also include a magnetometer which is commonly used as a heading reference. Typical configurations contain one accelerometer, gyro, and magnetometer per axis for each of the three vehicle axes: pitch, roll and yaw.

2.4.4 Nvidia Jetson

Nvidia Jetson is a series of embedded computing boards from Nvidia. The Jetson TK1, TX1 and TX2 models all carry a Tegra processor (or SoC) from Nvidia that integrates an ARM architecture Central Processing Unit (CPU). Jetson is a low-power system and is designed for accelerating machine learning applications. JetPack is a Software Development Kit (SDK) from Nvidia for their Jetson board series.

2.4.5 Raspberry Pi 4

The Raspberry Pi 4 Model B is the latest version of the low-cost Raspberry Pi computer, boasting an updated 64-bit quad core processor running at 1.5GHz with built-in metal heatsink, USB 3 ports, dual-band 2.4GHz and 5GHz wireless LAN, faster Gigabit Ethernet, and PoE capability via a separate PoE HAT. The Raspberry Pi 4 can do a surprising amount. Amateur tech enthusiasts use Pi boards as media centers, file servers, retro games consoles, routers, and network-level ad-blockers. One of the most popular OSs used for the Raspberry Pi is the Raspbian Operating system. The Raspbian OS is based on the Debian OS, optimized for the Raspberry Pi hardware. The Raspbian OS boots off a micro-SD card and the entire operating system runs off the card. A typical Class 4 8GB micro-SD card is sufficient for most purposes, but you have the option to connect it to an external hard disk or flash drive for more storage.

2.4.6 Internet Dongle

A dongle — also called an internet dongle — is a small USB device that allows you to access the internet. Dongles are popular because they offer greater flexibility than fixed line connections and can be used on the go. It run on the battery of the computer, therefore do not need to be charged.

2.4.7 LIPO battery

A lithium polymer battery, or more correctly lithium-ion polymer battery, is a rechargeable battery of lithium-ion technology using a polymer electrolyte instead of a liquid electrolyte. High conductivity semisolid (gel) polymers form this electrolyte. These batteries provide higher

specific energy than other lithium battery types and are used in applications where weight is a critical feature, like mobile devices and radio-controlled aircraft.

Chapter 3

DESIGN AND IMPLEMENTATION

3.1 Flowcharts, Diagrams

3.1.1 Robot Framework

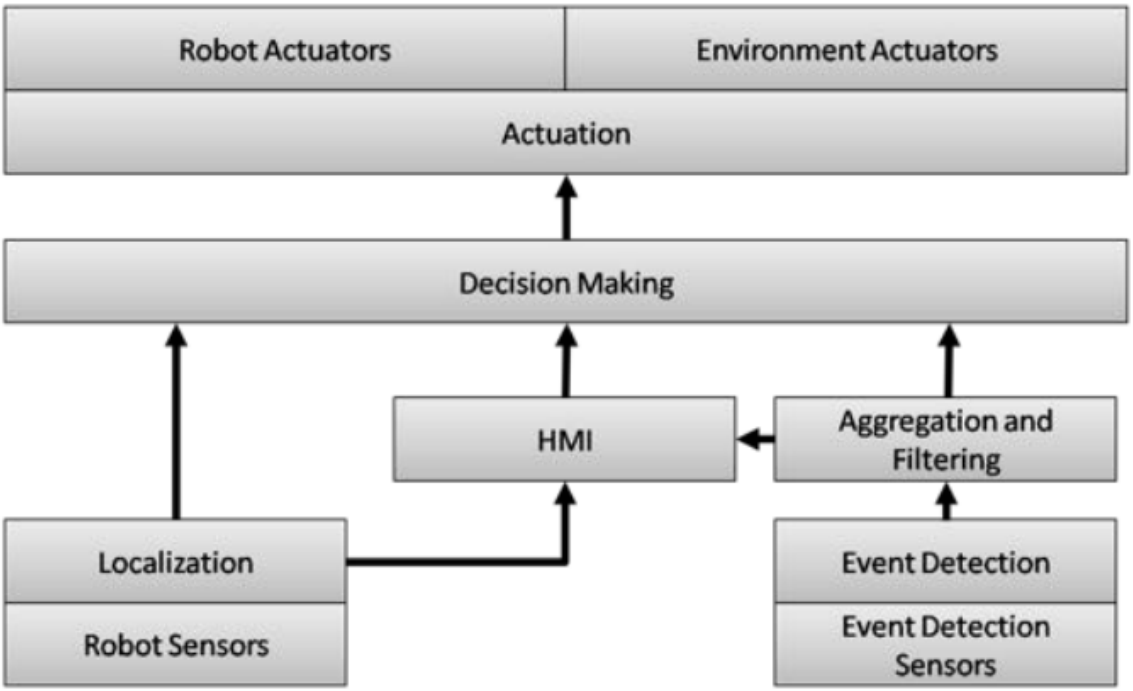


Figure 3.1: Modular design of our surveillance framework

In general, a mobile surveillance robot will experience a sequence of decisions about where to go and what to do, as long as it is operating in the environment and events are being detected by the network. In order to increase the autonomy of the networked robotic system, planning methodologies should consider several relevant aspects within the decision-making problem.

Apart from the heterogeneous sensor/actuator modules, a Human-Machine Interaction (HMI) module is included to display information (e.g. detected events) to the operator, to receive remote commands (e.g. sending a robot to a desired position).

3.1.2 Python Driver

A device driver is a computer program that operates or controls a particular type of device that is attached to a computer. A driver provides a software interface to hardware devices, enabling operating systems and other computer programs to access hardware functions without needing to know precise details about the hardware being used.

The main purpose of device drivers is to provide abstraction by acting as a translator between a hardware device and the applications or operating systems that use it. Programmers can write higher-level application code independently of whatever specific hardware the end-user is using.

Pseudocode:

```
class MotorDriver:
    def __init__(self, max_speed=10):
        """
        Init communication, set default settings, ...
        """
        self.max_speed = max_speed
        self.current_speed = 0
        self.voltage = 12
        self.temperature = 47
    def set_speed(self, speed):
        """
        Give a speed that the motor will try to reach.
        """
        if speed <= self.max_speed:
            self.current_speed = speed
        else:
            self.current_speed = self.max_speed
    def stop(self):
        """
        Set speed to 0 and thus stop the motor
        """
        self.current_speed = 0
    def get_speed(self):
        """
        Return current speed
        """
        return self.current_speed
    def get_status(self):
        """
        Get hardware information from the motor
        """
        return {
            'temperature': self.temperature,
            'voltage': self.voltage
        }
```

3.1.3 ROS Wrapper

When we talk about a wrapper class for a library (for any language, not specific to ROS), it means that we create a class around another class (or a function around another function, a module around another module, etc.), in order to add new functionalities or a different interface.

If you have a class A wrapping a class B, then the client developer will only use the interface provided by class A. Then, class A will be responsible for making the right calls on class B.

A ROS wrapper is simply a node that you create on top of a piece of (non-ROS) code, in order to create ROS interfaces for this code.

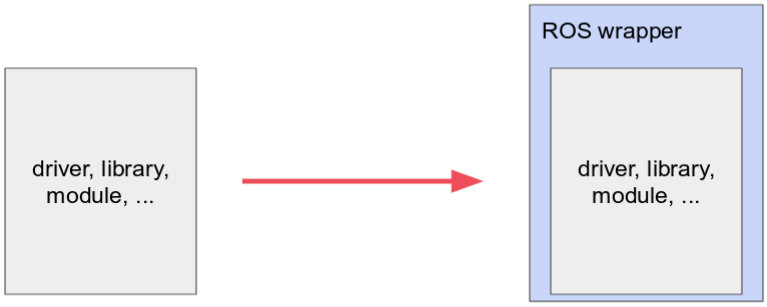


Figure 3.2: ROS Wrapper

Why use a ROS wrapper?

Creating a ROS wrapper has a few big advantages:

- You can easily integrate any driver/library into your ROS stack.
- Anyone knowing how to use ROS will know how to use your driver, without having to learn the driver's interface.
- If you decide to change some code in the driver later, you can do it without changing the ROS interface, and thus it won't be the source of breaking changes in other people's code.
- With the same ROS wrapper interface, you can easily switch between various hardware modules underneath.

In the motor driver class you have those main functions

- Constructor with max_speed as a parameter.
- set_speed to give a new speed for the motor to reach.
- stop to set motor speed to zero and thus stop the motor.
- get_speed to get current motor speed.
- get_status to get current temperature and voltage from the motor.

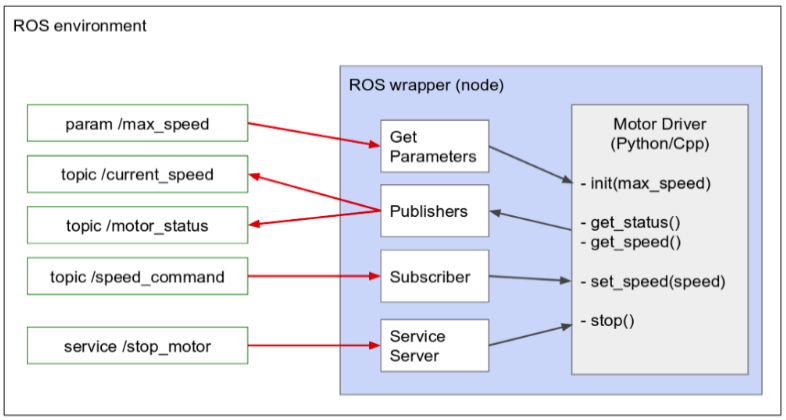


Figure 3.3: ROS Wrapper Features

Here is what the ROS wrapper will allow you to do (as a ROS user of this driver)

- Start the motor driver when you start the node, and stop the motor when you shutdown the node.
- Set ROS params to set default settings in the driver constructor.
- Publish on the *speed_command* topic to send speed commands to the motor.
- Send a service request on the *stop_motor* service, to immediately stop the motor.
- Subscribe to the *current_speed* topic to get current speed, and *motor_status* to get motor temperature and voltage.

Wrapper Pseudocode

```
#!/usr/bin/env python
import rospy
from motor_driver import MotorDriver
from std_msgs.msg import Int32
from std_srvs.srv import Trigger
from diagnostic_msgs.msg import DiagnosticStatus
from diagnostic_msgs.msg import KeyValue
class MotorDriverROSWrapper:
    def __init__(self):
        max_speed = rospy.get_param("~max_speed", 8)
        publish_current_speed_frequency = rospy.get_param("~
            publish_current_speed_frequency", 5.0)
        publish_motor_status_frequency = rospy.get_param("~
            publish_motor_status_frequency", 1.0)
        self.motor = MotorDriver(max_speed=max_speed)
        rospy.Subscriber("speed_command", Int32, self.
            callback_speed_command)
        rospy.Service("stop_motor", Trigger, self.callback_stop)
        self.current_speed_pub = rospy.Publisher("current_speed", Int32,
            queue_size=10)
        self.motor_status_pub = rospy.Publisher("motor_status",
            DiagnosticStatus, queue_size=1)
        rospy.Timer(rospy.Duration(1.0/publish_current_speed_frequency),
            self.publish_current_speed)
        rospy.Timer(rospy.Duration(1.0/publish_motor_status_frequency),
            self.publish_motor_status)
    def publish_current_speed(self, event=None):
        self.current_speed_pub.publish(self.motor.get_speed())
    def publish_motor_status(self, event=None):
        status = self.motor.get_status()
        data_list = []
        for key in status:
            data_list.append(KeyValue(key, str(status[key])))
        msg = DiagnosticStatus()
        msg.values = data_list
        self.motor_status_pub.publish(msg)
    def stop(self):
        self.motor.stop()
    def callback_speed_command(self, msg):
        self.motor.set_speed(msg.data)
    def callback_stop(self, req):
        self.stop()
        return {"success": True, "message": "Motor␣has␣been␣stopped"}
if __name__ == "__main__":
    rospy.init_node("motor_driver")
    motor_driver_wrapper = MotorDriverROSWrapper()
    rospy.on_shutdown(motor_driver_wrapper.stop)
    rospy.loginfo("Motor␣driver␣is␣now␣started,␣ready␣to␣get␣commands.")
    rospy.spin()
```

3.1.4 ROS Topic

A topic is a named bus over which nodes exchange messages. The messages are sent over TCP/IP. The ROS libraries that you will use on your code, will provide you with enough abstraction so you don't have to deal with the TCP/IP layer.

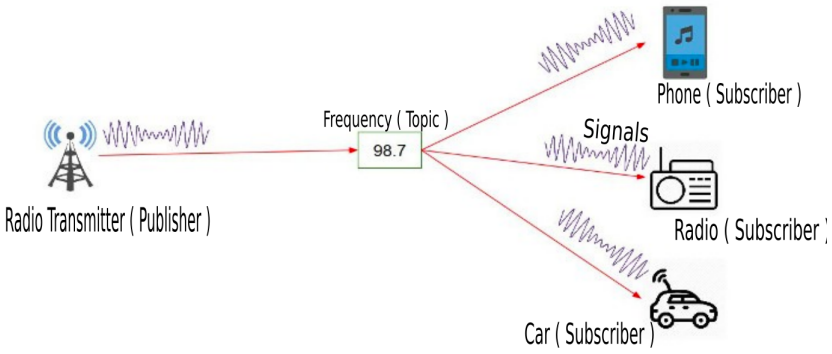


Figure 3.4: ROS Topic

In the green box here, 98.7, is a ROS topic, and the radio transmitter is a publisher of this topic. So for this case, a data stream is sent over the 98.7 topic.

In this case, the phone is a subscriber of the topic (98.7) and the tower is the publisher. To play the music on the phone, from the radio transmitter, you need to send and receive the same kind of message. Both the publisher and subscriber must send messages with the same data structure.

With ROS, you can have multiple subscribers for the same topic. You can see here an example with a topic and 3 subscribers. A subscriber is not aware of the other subscribers, and is not aware of who is publishing the data. It only knows it is receiving data from the 98.7 topic. Thus, we can say that subscribers are anonymous.

When a node wants to publish something, it will inform the ROS master. When another node wants to subscribe to a topic, it will ask the ROS master from where it can get the data. You can see the ROS master as a DNS server for nodes to find where to communicate.

3.1.5 ROS Service

Services are another way to pass data between nodes in ROS. Services are just synchronous remote procedure calls; they allow one node to call a function that executes in another node. We define the inputs and outputs of this function similarly to the way we define new message types. The server (which provides the service) specifies a callback to deal with the service request, and advertises the service. The client (which calls the service) then accesses this service through a local proxy.

Service calls are well suited to things that you only need to do occasionally and that take a bounded amount of time to complete. Common computations, which you might want to distribute to other computers, are a good example. Discrete actions that the robot might do, such as turning on a sensor or taking a high-resolution picture with a camera, are also good candidates for a service-call implementation.

Let's say that you have one node in your application which is controlling a LED panel.

The node is dealing with the hardware to power on and power off the LEDs. Of course, you want this node to be able to communicate with other ROS nodes. For example, other nodes could ask this node to power on or off a specific LED. In this case, you create a ROS service, named "Set Led". Inside the Led panel node, you create a service server for this ROS service.

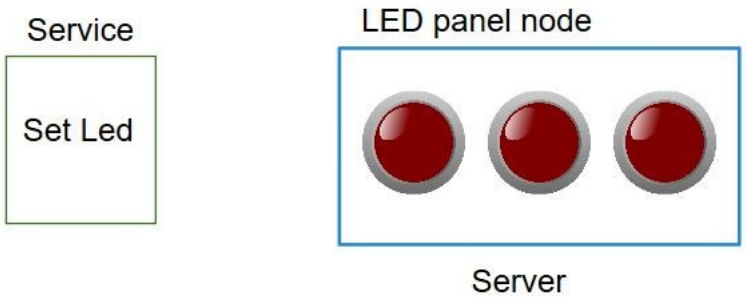


Figure 3.5: ROS Service - set led

Now, you have another node dealing with the battery. One of the functionality is to check if the battery is low, and to notify the user when it's happening. To do that, the battery node will contain a service client for the "Set Led" Service.

Imagine that the battery is going low. When it's detected, the battery node will send a

request to the ROS service. It will send a LED number and a state.

The server, which is the LED panel node, will expect to receive that information. If the data structure is the same as expected, the node can process the information, and as requested, power ON the third LED.

Once this is done, the server will send back a response. This response here, will contain a success flag. During all the ROS service process, the battery node is waiting. Upon reception of this success flag, the battery node knows that the action requested was successfully done.

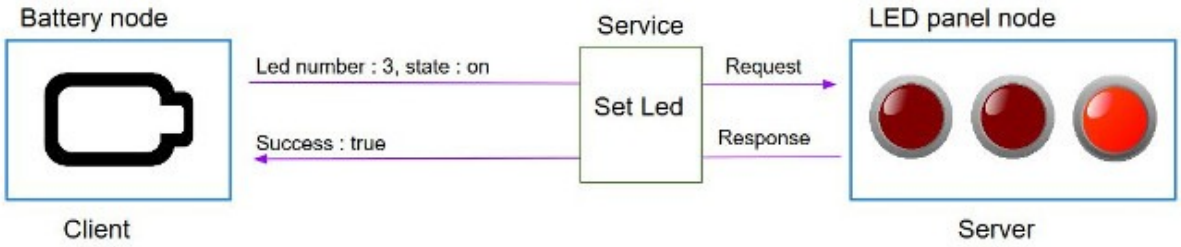


Figure 3.6: ROS Service - battery empty

And that's it, the communication is finished. The server is still up and waiting for new requests.

Later on, after charging the battery, the battery node detects that the battery is now full. It will then decide to send a new request to the "Set Led" service, to power off the third led.

The server receives this request, performs the operation, and sends back a success flag. The communication is done.

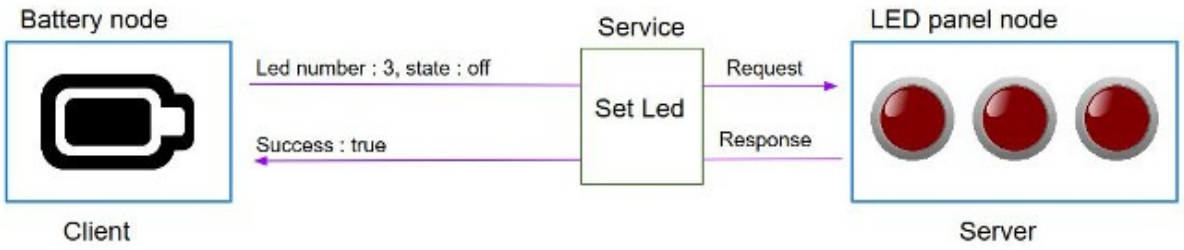


Figure 3.7: ROS Service - battery full

3.1.6 ROS Navigation

To accomplish a task, a robot should "understand" the environment from its sensor measurements. Understanding means, capturing the information at the right level of abstraction. An autonomously navigating robot should know:

- What the environment looks like?
- Where it is in the environment?

Simultaneous localization and mapping (SLAM) is an important approach that allows the robot to acknowledge the obstacles around it and plan a path to avoid these restrictions. This method is a merge of several approaches that allow robots to navigate on unknown or partially known environments. ROS has a package that performs SLAM, named Navigation Stack

One of the many resources needed for completing this task and that is present on the navigation stack are the localization systems, that allow a robot to locate itself, whether there is a static map available or simultaneous localization and mapping is required.

Adaptive Monte Carlo Localizatio (AMCL) is a tool that allows the robot to locate itself in an environment through a static map, a previously created map. The disadvantage of this resource is that, because of using a static map, the environment that surrounds the robot can not suffer any modification, because a new map would have to be generated for each modification and this task would consume computational time and effort. Being able to navigate only in modification-free environments is not enough, since the robots should be able to operate in places like industries and schools, where there is constant movement. To bypass the lack of flexibility of static maps, two other localization systems are offered by the navigation stack: gmapping and hector mapping.

Both gmapping and hector mapping are based on SLAM, a technique that consists on mapping an environment at the same time that the robot is moving, in other words, while the robot navigates through an environment, it gathers information from the environment through his sensors and generates a map.

This way you have a mobile base able not only to generate a map of an unknown environment as well as updating the existent map, thus enabling the use of the device in more generic environments, not immune to changes.

Other indispensable data to generate a map are the sensors‘ distance readings, for the reason that they are responsible in detecting the external world and, this way, serve as reference to the robot. Nonetheless, the data gathered by the sensors must be adjusted before being used by the device.

These adjustments are needed because the sensors measure the environment in relation to themselves, not in relation to the robot, in other words, a geometric conversion is needed. To make this conversion simpler, ROS offers the TF tool, which makes it possible to adjust the sensors positions in relation to the robot and, this way, adequate the measures to the robot‘s navigation.

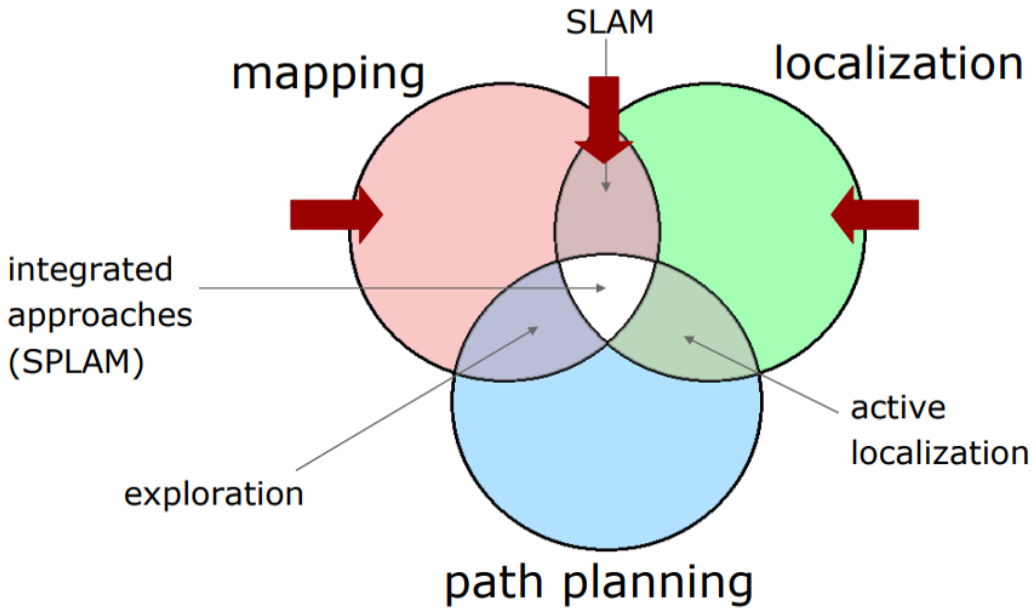


Figure 3.8: Navigation Stack

3.1.7 Simultaneous Localization and Mapping

The Simultaneous Localization and Mapping (SLAM) problem can be defined as a process where a robot builds a map representing its spatial environment while keeping rack of its position within the built map. Mapping is done online with no prior knowledge of the robot's location; the built map is subsequently used by the robot for navigation. SLAM is a key component of any truly autonomous robot. Much recent research has been done tackling the computational efficiency of SLAM and the data association and landmark extraction necessary for a robust SLAM method.

Environmental mapping involves creating a mathematical model of a real environment's spatial information. SLAM extends the requirements of this mathematical model; it must also jointly represent the robot's state and the position of extracted landmarks relative to the robot's location. The robot's state includes information on the robot's position and orientation. The basic SLAM framework involves odometry, landmark prediction, landmark extraction, data association and matching, pose estimation, and map update. These processes are the backbone of every major SLAM method, and are performed in cyclic fashion.

The system is made up of 4 parts:

- **Sensor data**: on mobile devices, this usually includes the camera, accelerometer and gyroscope. It might be augmented by other sensors like GPS, light sensor, depth sensors, etc.

- **Front-end**: the first step is feature extraction. These features also need to be associated with landmarks – keypoints with a 3D position, also called map points. In addition, map points need to be tracked in a video stream. Long-term association reduces drift by recognizing places that have been encountered before (loop closure).

- **Back-end**: takes care of establishing the relationship between different frames, localizing the camera (pose model), as well as handling the overall geometrical reconstruction. Some algorithms create a sparse reconstruction (based on the keypoints). Others try to capture a dense 3D point cloud of the environment.

- **SLAM estimate**: the result containing the tracked features, their locations & relations, as well as the camera position within the world.

Mathematical description of the SLAM problem

Given a series of controls u_t and sensor observations o_t over discrete time steps t, the SLAM problem is to compute an estimate of the agent's location x_t and a map of the environment m_t. All quantities are usually probabilistic, so the objective is to compute:

$$P(mt+1, xt+1 | o_{1:t+1}, u1:t) \tag{3.1}$$

Applying Bayes' rule gives a framework for sequentially updating the location posteriors, given a map and a transition function $P(x_t|x_{t-1})$,

$$P(x_t|o_{1:t}, u_{1:t}, m_t) = \sum_{m_{t-1}} P(o_t|x_t, m_t, u_{1:t}) \sum_{x_{t-1}} P(x_t|x_{t-1}) \frac{P(x_{t-1}|m_t, o_{1:t-1}, u_{1:t})}{Z} \tag{3.2}$$

Similarly the map can be updated sequentially by:

$$P(m_t|x_t, o_{1:t}, u_{1:t}) = \sum_{x_t} \sum_{m_t} P(m_t|x_t, m_{t-1}, o_t, u_{1:t}) P(m_{t-1}, x_t|o_{1:t-1}, m_{t-1}, u_{1:t}) \tag{3.3}$$

Like many inference problems, the solutions to inferring the two variables together can be found, to a local optimum solution, by alternating updates of the two beliefs in a form of EM algorithm.

gMapping

gMapping is probably the most used SLAM algorithm. It is a highly efficient Rao-Blackwellized particle filer to learn grid maps from laser range data.

This approach uses a particle filter in which each particle carries an individual map of the environment. Accordingly, a key question is how to reduce the number of particles. We present adaptive techniques to reduce the number of particles in a Rao- Blackwellized particle filter for learning grid maps. We propose an approach to compute an accurate proposal distribution taking into account not only the movement of the robot but also the most recent observation. This drastically decrease the uncertainty about the robot's pose in the prediction step of the filter. Furthermore, we apply an approach to selectively carry out re-sampling operations which seriously reduces the problem of particle depletion.

1. Each particle = sample of history of robot poses + posterior over maps given the sample pose history; approximate posterior over maps by distribution with all probability mass on the most likely map whenever posterior is needed.

2. Approximate the optimal sequential proposal distribution:

$$P^*(x_t) = P\left(x_t | x^i_{1:t-1}, z_{1:t}, u_{1:t}\right) \propto P\left(z_t | m^i_{t-1}, x_t\right) P\left(x_t | x^i_{t-1}, u_t\right) \tag{3.4}$$

   (a) find the local optimum $argmax_x P^*(x)$

   (b) sample x^k around the local optimum, with weights $w^k = P^*(x^k)$

   (c) fit a Gaussian over the weighted samples

   (d) this Gaussian is an approximation of the optimal sequential proposal p^*

3. Weight update for optimal sequential proposal is

$$p\left(z_t | x^i_{1:t-1}, z_{1:t-1}, u_{1:t}\right) = p\left(z_t | m^i_{t-1}, x^i_{t-1}, u_{t-1}\right) \tag{3.5}$$

   , which is efficiently approximated from the same samples as above by resampling based on the effective sample size S_{eff}

Algorithm

Algorithm 1 Improved RBPF for Map Learning

Require:
$\mathcal{S}_{t-1}$, the sample set of the previous time step
z_t, the most recent laser scan
u_{t-1}, the most recent odometry measurement
Ensure:
$\mathcal{S}_t$, the new sample set

$\mathcal{S}_t = \{\}$
for all $s_{t-1}^{(i)} \in \mathcal{S}_{t-1}$ **do**
$< x_{t-1}^{(i)}, w_{t-1}^{(i)}, m_{t-1}^{(i)} >= s_{t-1}^{(i)}$

// scan-matching
$x_t'^{(i)} = x_{t-1}^{(i)} \oplus u_{t-1}$
$\hat{x}_t^{(i)} = \text{argmax}_x\, p(x \mid m_{t-1}^{(i)}, z_t, x_t'^{(i)})$

if $\hat{x}_t^{(i)} =$ **failure then**
$x_t^{(i)} \sim p(x_t \mid x_{t-1}^{(i)}, u_{t-1})$
$w_t^{(i)} = w_{t-1}^{(i)} \cdot p(z_t \mid m_{t-1}^{(i)}, x_t^{(i)})$
else
// sample around the mode
for $k = 1, \ldots, K$ **do**
$x_k \sim \{x_j \mid |x_j - \hat{x}^{(i)}| < \Delta\}$
end for

// compute Gaussian proposal
$\mu_t^{(i)} = (0,0,0)^T$
$\eta^{(i)} = 0$
for all $x_j \in \{x_1, \ldots, x_K\}$ **do**
$\mu_t^{(i)} = \mu_t^{(i)} + x_j \cdot p(z_t \mid m_{t-1}^{(i)}, x_j) \cdot p(x_t \mid x_{t-1}^{(i)}, u_{t-1})$
$\eta^{(i)} = \eta^{(i)} + p(z_t \mid m_{t-1}^{(i)}, x_j) \cdot p(x_t \mid x_{t-1}^{(i)}, u_{t-1})$
end for
$\mu_t^{(i)} = \mu_t^{(i)} / \eta^{(i)}$
$\Sigma_t^{(i)} = \mathbf{0}$
for all $x_j \in \{x_1, \ldots, x_K\}$ **do**
$\Sigma_t^{(i)} = \Sigma_t^{(i)} + (x_j - \mu^{(i)})(x_j - \mu^{(i)})^T \cdot$
$p(z_t \mid m_{t-1}^{(i)}, x_j) \cdot p(x_j \mid x_{t-1}^{(i)}, u_{t-1})$
end for
$\Sigma_t^{(i)} = \Sigma_t^{(i)} / \eta^{(i)}$
// sample new pose
$x_t^{(i)} \sim \mathcal{N}(\mu_t^{(i)}, \Sigma_t^{(i)})$

// update importance weights
$w_t^{(i)} = w_{t-1}^{(i)} \cdot \eta^{(i)}$
end if
// update map
$m_t^{(i)} = \text{integrateScan}(m_{t-1}^{(i)}, x_t^{(i)}, z_t)$
// update sample set
$\mathcal{S}_t = \mathcal{S}_t \cup \{< x_t^{(i)}, w_t^{(i)}, m_t^{(i)} >\}$
end for

$N_{\text{eff}} = \frac{1}{\sum_{i=1}^{N} (\tilde{w}^{(i)})^2}$
if $N_{\text{eff}} < T$ **then**
$\mathcal{S}_t = \text{resample}(\mathcal{S}_t)$
end if

Figure 3.9: Rao-Blackwellized Particle Filter Algorithm

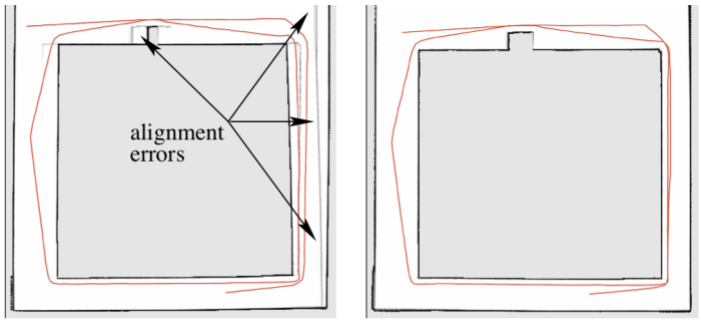


Figure 3.10: Map Output

Different mapping results for the same data set obtained using the proposal distribution which ignores the odometry (left) and which considers the odometry when drawing the next generation of particles (right)

3.1.8 Occupancy Grid

The goal of an occupancy mapping algorithm is to estimate the posterior probability over maps given the data: $p(m|z_{1:t},x_{1:t})$, where m is the map, $z_{i:t}$ is the set of measurements from time 1 to t, and $x_{1:t}$ is the set of robot poses from time 1 to t. The controls and odomtry data play no part in the occupancy grid mapping algorithm since path is assumed known. Occupancy grid algorithms represent the map m as a fine-grained grid over the continuous space of locations in the environment. The most common type of occupancy grid maps are 2d maps that describe a slice of the 3d world.

If we let m_i denote the grid cell with index i (often in 2d maps, two indices are used to represent the two dimensions), then the notation $p(m_i)$ represents the probability that cell i is occupied. The computational problem with estimating the posterior $p(m|z_{1:t},x_{1:t})$ is the dimensionality of the problem: if the map contains 10,000 grid cells (a relatively small map), then the number of possible maps that can be represented by this gridding is 2^{10000}. Thus calculating a posterior probability for all such maps is infeasible.

The standard approach, then, is to break the problem down into smaller problems of estimating.

$$p(m_i|z_{1:t},x_{1:t}) \tag{3.6}$$

for all grid cells *mi*. Each of these estimation problems is then a binary problem. This breakdown is convenient but does lose some of the structure of the problem, since it does not enable modelling dependencies between neighboring cells. Instead, the posterior of a map is approximated by factoring it into

$$p(m|z_{1:t},x_{1:t}) = \prod_i p(m_i|z_{1:t},x_{1:t}) \tag{3.7}$$

Due to this factorization, a binary Bayes filter can be used to estimate the occupancy probability for each grid cell. It is common to use a log-odds representation of the probability that each grid cell is occupied.

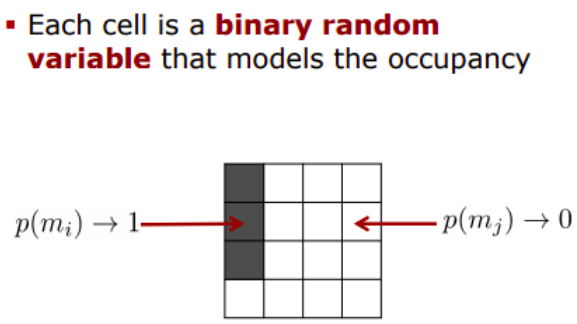


Figure 3.11: Occupancy Grid Representation

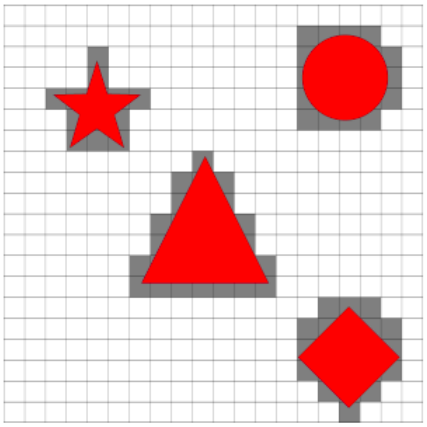


Figure 3.12: Example Occupancy Grid

3.1.9 Light Detection and Ranging

Light Detection and Ranging (LiDAR) uses a pulsed laser beam that it measures the return time of to accurately detect the distance to the object the beam hits as well as the intensity of the returned beam which indicates how reflective the surface is. They typically operate in the near-infrared spectrum and some can work outdoors at ranges varying from a few meters out to over a hundred meters.

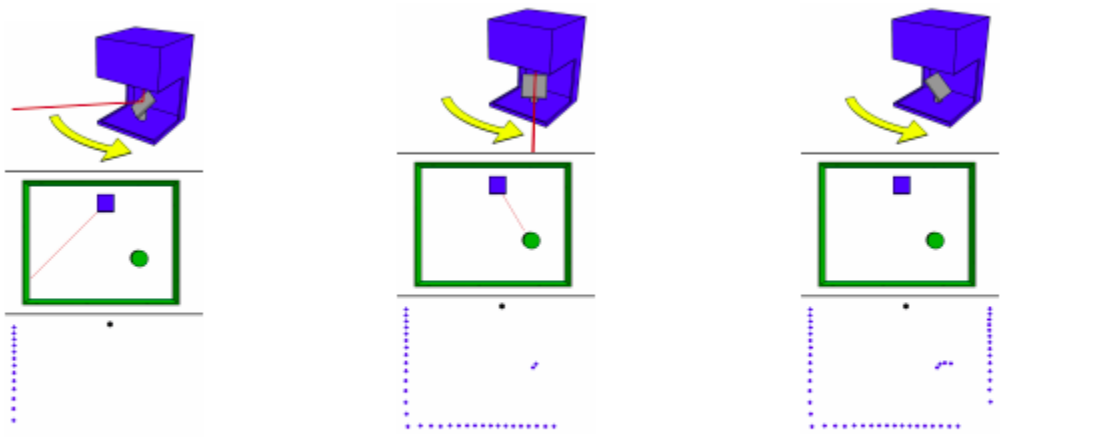
A LiDAR can measure a single point (one dimensional), a plane of points (two dimensional), or measure multiple planes to scan an entire area of points (three dimensional).

While a single point would be useful for backing up a car to alert the driver if there is any obstacles in the way, it is not sufficient for navigational purposes as it does not give enough information. A two dimensional LiDAR gives out all all points along a plane every scan which begins to be sufficient for navigational purposes but does not give a full picture of a scene

which is useful but comes with many more points that have to be analyzed.

The LiDAR works by having a rapidly rotating mirror (around 20Hz) with the laser aimed at it spin around the full scan area and reading back the return beam. Each point is given back as a polar coordinate where the angle is given of the return as well as the distance however it trivial to convert it to cartesian.

Figure 3.13 shows how a 2D LiDAR works where the top image shows the LiDAR, the middle image shows the LiDAR inside of a room with an obstacle, and the bottom image shows what the system ”sees” which is called a point cloud. Three dimensional LiDAR systems can give over one hundred thousand points per scan (also 20 Hz) but the points are in multiple parallel planes to create a three dimensional space. Most work similar to the 2D systems just with more lasers going at the same time however there are some that also have the mirror scan in the vertical direction as well as horizontal at the same time.



(a) Scan starting, laser on

(b) Cylinder hit, blocks laser from seeing wall

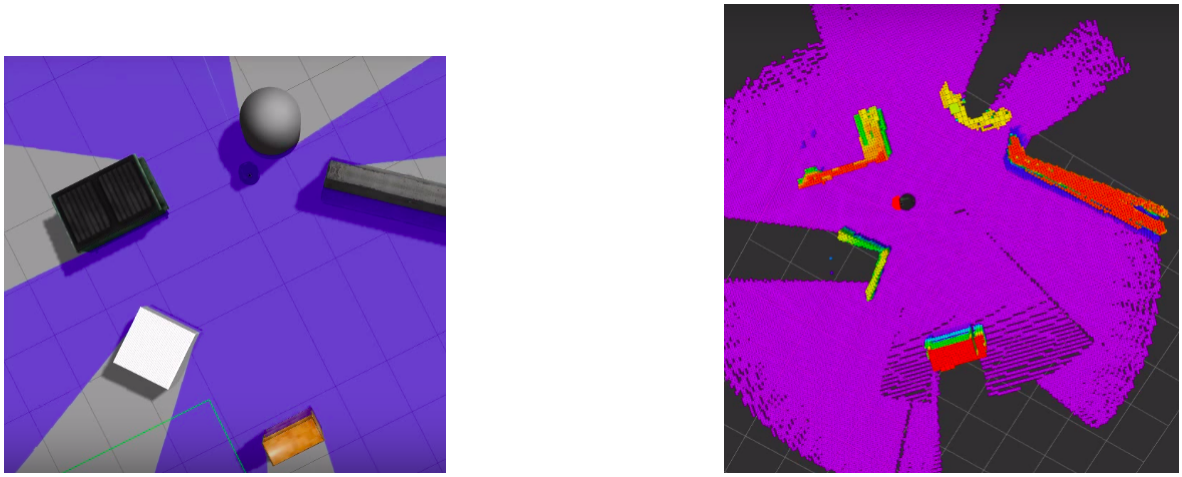
(c) Scan finished, laser off until mirror rotates back to start

Figure 3.13: Lidar scanning a room with an obstacle

Map

A map is a representation of the environment where the robot is operating. It should contain enough information to accomplish a task of interest. A metric map defines a reference frame.

To operate in a map, a robot should know its position in that reference frame and a sequence of waypoints or of actions to reach a goal location in the map is a path.



(a) Obstacle (b) LiDAR Obstacle Detection

Figure 3.14: Working example of LiDAR Map

Localization and Path Planning

Determine the current robot position, the measurements up to the current instant and a map. Determine (if it exists) a path to reach a given goal location given a localized robot and a map of traversable regions.

Mapping

Given a robot that has a perfect ego- estimate of the position, and a sequence of measurements, determine the map of the environment. A perfect estimate of the robot pose is usually not available. Instead we solve a more complex problem: Simultaneous Localization and Mapping (SLAM). It estimates:

- The map of the environment
- The trajectory of a moving device using a sequence of sensor measurements.

To sum up

To navigate a robot we need

- A map
- A localization module
- A path planning module

These components are sufficient if:

- The map fully reflects the environment
- The environment is static
- There are no errors in the estimate

However:

- The environment changes (e.g. opening/closing doors)
- It is dynamic (things might appear/disappear from the perception range of the robot)
- The estimate is “noisy” Thus we need to complement our ideal design with other components that address these issues, namely Obstacle-Detection/Avoidance and Local Map Refinement, based on the most recent sensor reading.

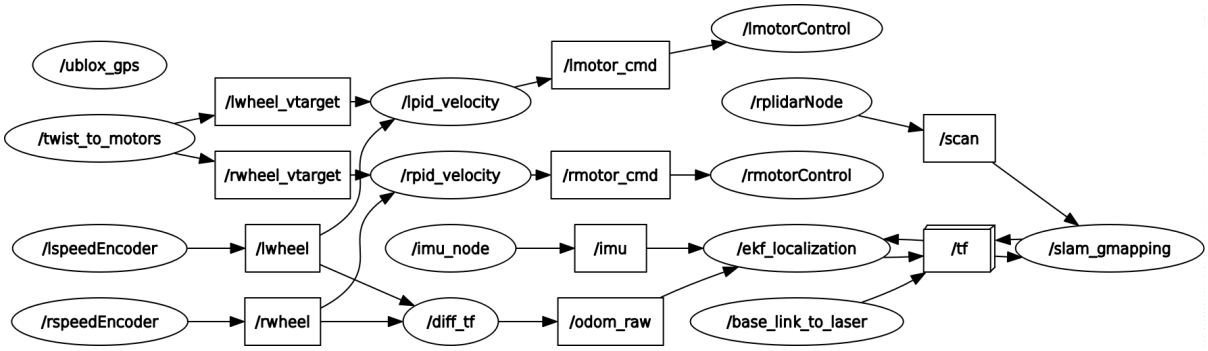


Figure 3.15: SLAM overview

Chapter 4

Observation and Results

4.1 Testing

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include, but are not limited to, the process of executing a program or application with the intent of finding software bugs (errors or other defects).

4.1.1 Black Box Testing

Black-box testing tests functional and non-functional characteristics of the software without referring to the internal code of the software.

4.1.2 White Box Testing

The proposed application contains various different modules and integrated successfully. Testing various cost-map configurations and motion algorithms was imperative to get a good navigation stack. We also tested various navigation algorithms to find the right fit for our robot.

4.1.3 Unit Testing

The unit testing is the process of testing parts of the program to verify whether the program is working correct or not. We had to verify whether the wheels were receiving the correct motion values to ensure that the robot moved correctly. We also had to check and re-check the URDF models to ensure that obstacle avoidance is done correctly.

4.1.4 Integration Testing

Integration testing is also taken as integration and testing is the major testing process where the units are combined and tested. Its main objective is to verify whether the major parts of the program is working fine or not the application includes many and different constraints of functionalities and these modules are integrated and tested. On the final robot tests, we had to check whether we had enough battery power to perform a successful demonstration.

4.2 Results

This delivery robot platform demonstrates how simple mobile robots work and which are the key components they need. These robots can be controlled autonomously or through remote connection. Because delivery robots in reality are meant to drive on streets among people then for safety it is necessary to have a remote control capability as backup option.
The robot could efficiently avoid obstacles using the LIDAR scans and the camera feed enabled us to control the robot remotely with the help of a mobile application or a computer.

4.3 Snapshots

4.3.1 Mobile App

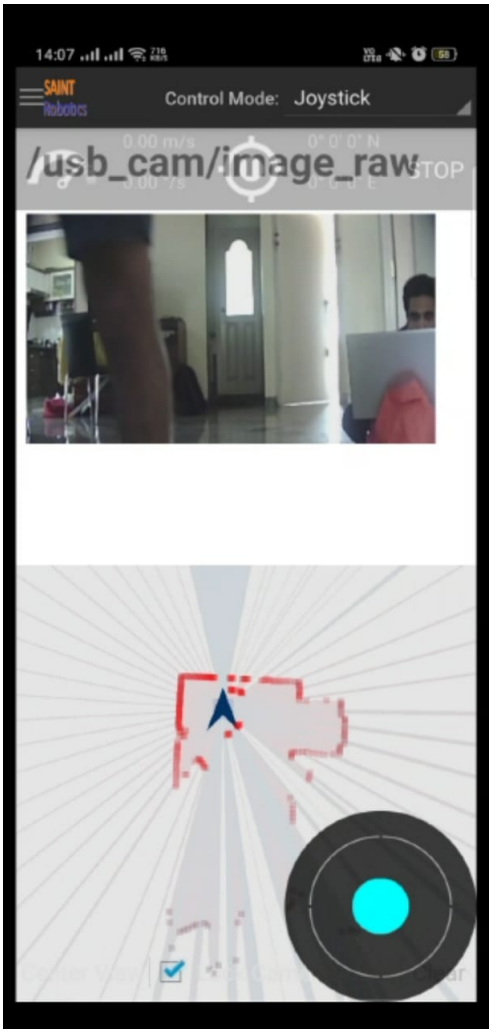


Figure 4.1: Mobile app

The upper portion of the mobile application shows the live-stream from the camera. The lower portion shows the laser scans. It can be observed from the above picture that the laser has detected obstacles and has mapped them relative to its own position. The circle in the bottom right corner of the application is the joystick, this is helpful for manually moving the robot with the aid of the camera feed.

4.3.2 Robot Top

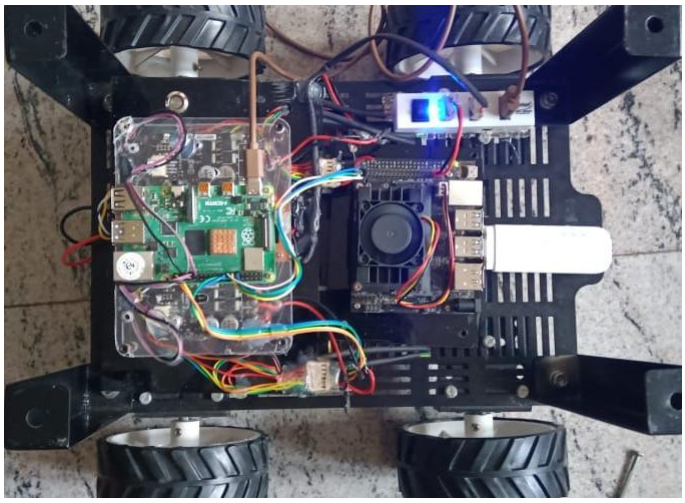


Figure 4.2: Robot top

The above picture shows the top view of the robot. The IMU, raspberry pi, Nvidia Jetson are all present at the base of the frame. The pillars support the upper layer of the robot, which holds the camera, LIDAR and the GPS unit. The wheels restrict the robot usage to indoors only.

4.3.3 Camera

Figure 4.3: Camera

The above picture shows the camera mounted on top of the robot frame, behind the LIDAR. The camera feed is streamed over http to the mobile application. It can also be accessed on the /camera/image topic.

4.3.4 Robot Bottom

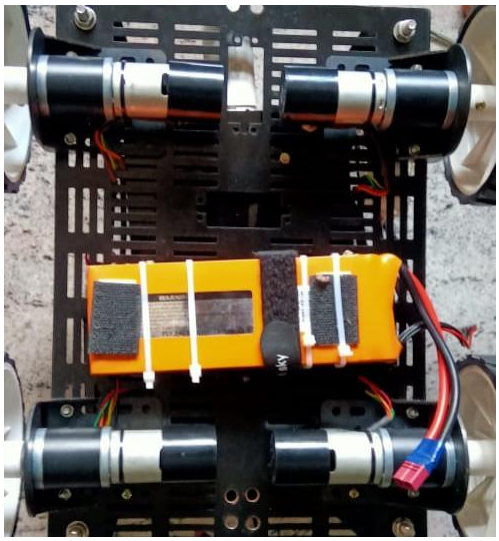


Figure 4.4: Robot bottom

The above picture shows the bottom view of the robot. You can observe the lipo battery which powers the entire robot.

4.3.5 Robot Front

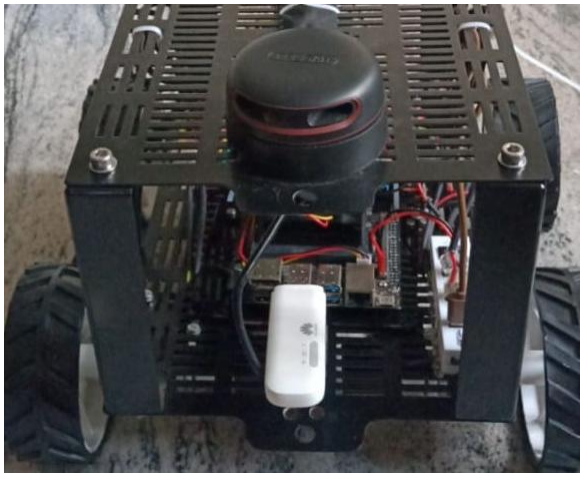


Figure 4.5: Robot front

The front of the robot shows the LIDAR mounted the frame of the robot and also the Internet dongle. The raspberry pi, jetson and the mobile application must be connected to the same wifi network (here, the dongle) in order to work together and communicate with each other.

4.3.6 Visualization

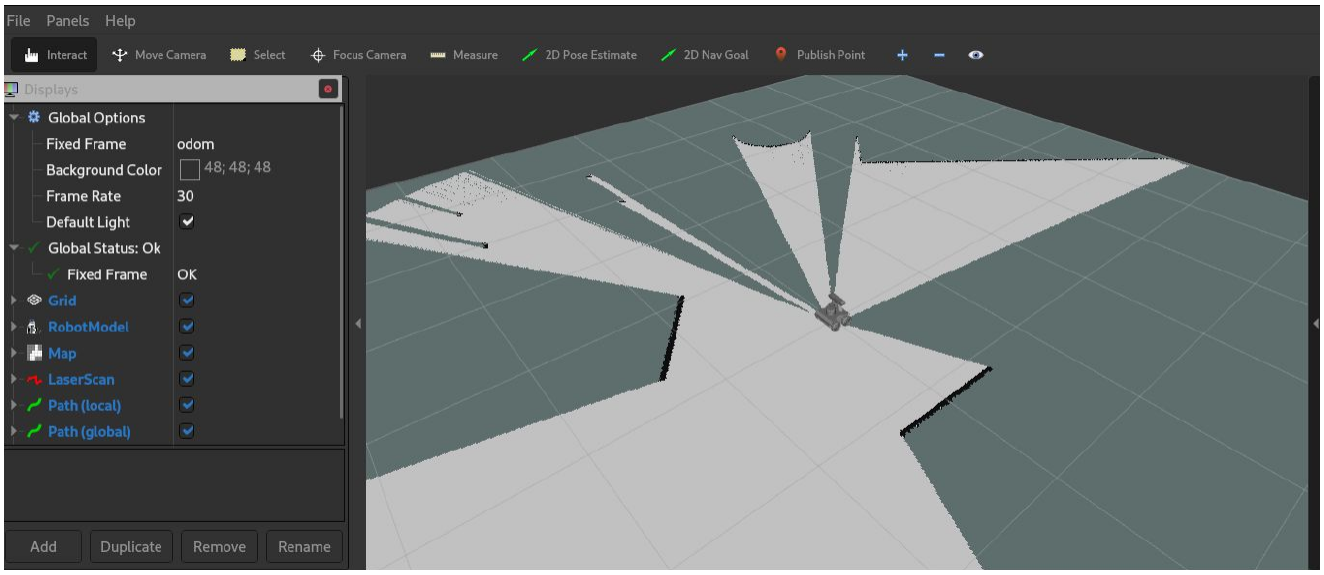


Figure 4.6: Visualization on Rviz

The above visualization shows the robot detecting the obstacles using the LIDAR scans.
The 'Fixed Frame' provides a static, base reference for your visualization. Any sensor data that comes in to rviz will be transformed into that reference frame so it can be properly displayed in the virtual world. Robot model plugin allows you to visualize the Robot Model according to its description from the URDF model. TF plugin allows you to visualize the position and the orientation of all the frames that compose the TF Hierarchy.

Chapter 5

Conclusion and Future Enhancements

We have explored the ROS ecosystem during the course of building this robot. The functionalities implemented here, albeit rudimentary, have performed the task in an appropriate manner. The use of two machines to run the robot was not efficient, we would like to shift all functionalities onto the Nvidia Jetson system, replacing the raspberry pi device and reduce the power consumption. Power consumption had been a major issue with our project, this was largely due to the excessive number of devices being powered with a single battery. In a future edition of this project, we would explore ways to manage power issues in a better way.

The usage of a mobile application to manage the movements of the robot was another problematic area since the apks available were outdated and implementing new functionalities was very hard. In the future we would like to move to a web application, which would solve many issues.

We would also like to implement an *Uber* like functionality that would enable users to summon the robot to serve them.

Bibliography

[1] Ros wiki: http://wiki.ros.org/ROS/Tutorials

[2] Ros answers: https://answers.ros.org/questions/

[3] Stack Overflow: https://stackoverflow.com/

[4] Raspberry-pi pin map: https://www.tomshardware.com/reviews/raspberry-pi-gpio-pinout,6122.html

[5] Learning Wheel Odometry and IMU Errors: https://hal.archives-ouvertes.fr/hal-01874593v2/document

[6] Nvidia Jetson: https://developer.nvidia.com/

[7] SLAM, pid, odometry: http://thecorpora.com/qbo-slam-pid-odometry/

[8] Rodrigo Longhi Guimar˜aes, Andr´e Schneider de Oliveira, Jo˜ao Fabro, Thiago Becker, and Vin´ıcius Amilgar Brenner. ROS Navigation: concepts and tutorial

[9] Arturo Gil, scar Reinoso, Mnica Ballesta, and Miguel Juli. Multi-robot visual SLAM using a rao-blackwellized particle filter. Robotics and Autonomous Systems, 58(1):68 – 80, 2010. ISSN 0921-8890. doi: http://dx.doi.org/10.1016/j.robot.2009.07.026.

[10] H. Lategahn, A. Geiger, and B. Kitt. Visual slam for autonomous ground vehicles. In Robotics and Automation (ICRA), 2011 IEEE International Conference on, pages 1732–1737, May 2011. doi: 10.1109/ICRA.2011.5979711.

[11] Thomas Lemaire, Cyrille Berger, Il-Kyun Jung, and Simon Lacroix. Vision-based slam: Stereo and monocular approaches. International Journal of Computer Vision, 74(3):343–364, 2007.

[12] Xiuzhi Li, Wei Cui, and Songmin Jia. Range scan matching and particle filter based mobile robot slam. In Robotics and Biomimetics (ROBIO), 2010 IEEE International Conference on, pages 779–784. IEEE, 2010.

[13] Michael Montemerlo, Sebastian Thrun, Daphne Koller, Ben Wegbreit, et al. Fastslam: A factored solution to the simultaneous localization and mapping problem. In AAAI/IAAI, pages 593–598, 2002.

[14] F.A. Moreno, J.L. Blanco, and J. Gonzalez. Stereo vision specific models for particle filter-based SLAM. Robotics and Autonomous Systems, 57(9):955 – 970, 2009. ISSN 0921-8890. doi: http://dx.doi.org/10.1016/j.robot.2009.03.002.

[15] Pedro Nu´nez, Ricardo Vazquez-Mart in, and Antonio Bandera. Visual odometry based on structural matching of local invariant features using stereo camera sensor. Sensors, 11(7):7262–7284, 2011.

[16] Taro Suzuki, Yoshiharu Amano, and Takumi Hashizume. Development of a sift based monocular ekf-slam algorithm for a small unmanned aerial vehicle. In SICE Annual Conference (SICE), 2011 Proceedings of, pages 1656–1659. IEEE, 2011.

[17] Hong-jian Wang, Jing Wang, Le Yu, and Zhen-ye Liu. A new slam method based on svm-aekf for auv. In OCEANS 2011, pages 1–6. IEEE, 2011.