MSci Project

Maciej Musialek

February 11, 2016

Contents

Chapter 1

Introduction

This project will be based on a game of finding the treasure. There will be a set amount of robots(One of two to start with) and they will be placed in an environment with limited view although they will know where they are. The human cannot see the robots nor the surroundings. It will be the robots job to relay information to the human. This information can range from paths to take, pictures of potential treasure, and questions regarding dangers and priority. There is a catch though. Each robot has limited energy and each task draws a bit of that energy from it.

The project will focus on retrieving the code from the supervisors that already has been set up for the server and improving it with a GUI, communication between human and robots, teaching the robots what information is useful to show and what isn't. Later on, it will be ideal to make the robots communicate as to taking the paths that would be most beneficial i.e. it wouldn't be ideal for the robots to approach each other given a path simply because it would be a waste of energy for them to pass each other. Also, if they shared a map, they would avoid going into the same room hence discarding excessive use of energy. In the later stages of the project I will also test whether or not there is a difference between guided directions and random actions taken by robots to find the particular treasure.

The project will be based on building robots around the ROS(Robot Operating System) architecture which can be found on: http://wiki.ros.org/. It will be the mainframe on which the robots operate. The server itself will be made in Java and the GUI will be decided during the project with considerations taken into account for: usability, availability and traffic handling.

It is important to note that this game is more applicable than it seems. Searching for treasure can be treated as a synonym for finding people in catastrophe weighing danger of losing machinery to find humans, ease of extraction and value of the individual that is being found. Rescue missions in the avalanches, fiery buildings and tsunamis are a great example of how this game could be applied in real life to rescue drones on missions.

Chapter 2

The plan

The plan will be a rough outline for the project. It will explain what each item in the agenda means and its relevance to the project.

2.1 Learning ROS - Finished by 13th November

Since the mainframe for robot handling is ROS it will be important to learn and discover what it does and how it does it. As I learn I will map these functions down and make rough sketches onto how to fit this into the project to make it accessible for everyone. ROS is a very complex operating system that requires multiple terminals to run. It has a core that handles topics, nodes and packages to create not only 2-3 and 3-d planes for the robots but allows training them to take paths, keyboard input and possibly instructions from other programs.

2.2 Studying the server code - Finished by end of November

Once I learn ROS, it will be time to start studying the server code pre-made by PhD students in the past that were involved in this game. This server code is held by one of the supervisors and I will be working alongside them to create as much of the game as is possible to their requirements. This will allow me to establish the possibilities and limitations of the current design and tailor my future choices to it. Whether or not this code will be extended or left as is will also depend on the current requirements set by the supervisor.

At this point in time it is important to mention that not all the code that is created now will be the only code provided by the supervisors. As the project progresses, it will be my job to create implement additional pseudo code for both the robots and the GUI so the code I study now might change with time.

2.3 Researching the right tool for GUI - Finished by 11th December

Researching the right GUI will be crucial in this project. The game is to go live and be presented to whoever is interested. The GUI will need to be robust, accessible and allow whoever wants to use it, to use it from the start to create a pleasant experience and a playful environment.

At the start I will be meeting with the supervisor to receive sketches on the GUI and agree on what the system should look like, I will then research different platform possibilities and study their pros and cons against what the game needs to be implemented in. Upon choosing the most desirable platform, it will be the time to research the ease of implementation and how that can be done, eventually leading to possible implementation alongside the ROS server.

2.4 Implementation of GUI - Finished by Mid January

Following the research of the right tool for GUI comes its implementation. This chapter will involve creating the GUI and the process of working on it so that an agreement can be reached between me and the supervisor for how the GUI looks. Information feed will need to be generated that accepts information sent by the Java server from the ROS architecture and a decision log will need to be created to send the information back through the same route.

At this point in the project I will be introducing formative testing to test for all the possible faults of the system and correcting them until the GUI is of sufficient quality to start working properly on the project and the robots.

2.5 Implementing two robot one human environment - Finished by mid February

To even start thinking of multi communication, an environment needs to be created to allow this. This will involve creating two robots and one player environment with two robots searching possible paths and asking the player about possible decisions. They will work independently of each other and firsts actions will in no way influence the other. This could mean that the player will have to keep a map in their head of where each robot visited so no place is visited twice. In an ideal design robots will not visit the others places and not suggest a path that has been visited. At this point in the project though, creating a full environment comes before planning that.

2.6 Researching possible implementations of robot collaboration - Finished by end of February

At this point the GUI has been chosen and implemented and an environment where two robots and one player can interact and play the game has been created. Now the time has come for creating the right kind of information handling so that the robots can start making smart decision based on past actions taken by the rest of the group. Working as far away from each other can be one of them, although there are situations where a high concentrations of treasures in a certain area are present meaning one robot could run out of energy before reaching them all. This too will need to be taken into account. It will probably be the most challenging part of the project, that involves a lot of decision making to finally agree on how the robots should handle and react to information received. With this completed the project will either be halted and finished or expanded if time allows.

2.7 Formative testing - Finished by end of March

Once everything has been implemented, the time has come for an exhaustive testing of the full system and testing how it works as a whole, how it handles traffic, how it can be improved, whether the design can support more than two robots and still manage to make intelligent decisions. With that in mind, I will also be testing the effectiveness of the human making guiding the robots in comparison with how the robots would handle if they chose what to do at random. The effectiveness will be weighed based on treasures found, wasted energy and useful actions taken.

2.8 Conclusion - Finished by end of April

The conclusion will be a reflection of how the project was handled, researched and implemented. It will provide constructive criticism on the current implementation and decisions made to guide the project, possible improvements to the current design, possible restructuring to make it either more user friendly or scalable and where this project could go given more time.

Chapter 3

Learning ROS

Robot Operating system (ROS) is a complex system used to simulate robots from real life. It has a broad amount of possibilities and allows simulation of environments including 2-D and 3-D planes. It does it via a hierarchical based structure of its core, nodes, topics and packages that altogether create an outstanding environment for expansion and already built tools like path searching allow for robots to already be impressively implemented through this system without any additional expansion on the programmers side.

Having said that, the program does come with a few handicaps that can be very easily overcome with some intermediate knowledge of Scripting and Object Oriented languages like Java for server writing or Python and C++ for creating and instructing the robots on what to do.

Even in the tutorials the power of ROS is easily shown when after getting the core up and running the user can already manage a robot that is able to be moved via keyboard commands. A function that is otherwise not very useful in this project but one that shows how incredibly developed this system is which is why the supervisors decided to work with it as the main operating robot system for ROS.

In the following sections I am going to discuss the basic structure of ROS followed by an extensive research into its possibilities in path finding, server connection and virtual machine running to establish a stable server that the GUI can be connected to in the future.

3.1 Structure

This section will be focused on the structure of ROS. That means it will explain all the core subjects that go about the system and make it tick. There are quite a few to talk about and I will do my best to infer how important each is to the project in its own way and how it could possibly improve the overall use of the system.

As stated before, ROS is a hierarchical system which can be highly customized to the needs of the creator of the program and the user. There are set practices which revolve around it to ensure that whenever the code is sourced, everyone else can understand the intention of the programmer hence allowing everyone to collaborate more easily and inject necessary sourced code into their systems with ease.

3.1.1 Packages

Software in ROS is organized in packages. It is to my understanding that topics are built to be light weight modules that have the primary goal of being re-used. A quote taken directly from the website describes packages as:

> "A package might contain ROS nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module."

This indicated that packages are the main building blocks of ROS that follow the principal of minimalistic coding by creating smaller pieces of software that provide functionality to otherwise dumb clients that take advantage of them. This also means that it is vital to follow the developers guide that ROS has set up to make the whole process of injecting packages much easier.

Regarding their creation, ROS has went out of their way to make the process as easy as possible by providing the user with possibilities to create packages by hand or by using software that ROS has developed to start the package off. These packages can then be implemented using the following two languages: Python and C++. This project will definitely involve a lot of packages that will most likely be created by hand to keep customizability to the maximum.

With regards to the choice of language between Python and C++ it seems like it will be quicker to work with Python as the language isn't as strict and can easily be interpreted by software rather than built and compiled. Choosing this option will also spare trouble with memory management that Python can handle very well.

3.1.2 Nodes

Nodes are computational processes in ROS that are combined together into a graph and are able to communicate. They can do so using streaming topics, RPC services and a parameter server which will all be discussed in future sections and chapters of this project. These nodes are supposed to operate on fine grained scale meaning that a single robot control system will be a combination of a lot of nodes that each handle execution and calculation separate of each other. An example taken directly from ROS website states:

> For example, one node controls a laser range-finder, one Node controls the robot's wheel motors, one node performs localization, one node performs path planning, one node provide a graphical view of the system, and so on.

It is later mentioned that nodes work independently of each other which means that a crash of one node will not compromise the system at all unless that specific node is crucial to all the rest of computations, meaning that any crucial calculations should be split among nodes to reduce a possibility of that happening. This means that fault tolerance among ROS packages is very high and allows for pin pointing errors much faster than in a monolithic system.

A node consists of a graph resource name, node type and a package resource name both of which are crucial to understand. On a regular graph, a graph resource name is a unique identifier of a particular node running. A node type is responsible for finding the right executable in the package to run that particular node properly which leads to package resource names which holds the name of the package a particular type belongs to as well as the executable file name used to run that node. The way ROS deals with those is that it searches the package and the first executable it finds that matches the name will be chosen to run that node. This in itself should make the user wary about how they choose to name their packages as conflicts may result in unsightly errors however it also opens a door to filename manipulation which could swap filenames on the run in the future; although this kind of feature would most likely over complicate the system.

These nodes are produced using ROS client libraries.

3.1.3 Topics

Topics are named buses over which Nodes can communicate anonymously. It does so in a MQ design pattern way. It allows a node to become a broadcaster of information i.e. **Publisher** and other nodes to subscribe and receive information when such information is presented; these nodes are called **Subscribers**. In general nodes are not aware of where the information comes from but they are aware of the topic name and what type of broadcast they will be interested in. Of course ROS allows for multiple nodes to be Publishers as well as Subscribers. Following the websites definition:

> Topics are intended for unidirectional, streaming communication. Nodes that need to perform remote procedure calls, i.e. receive a response to a request, should use services instead. There is also the Parameter Server for maintaining small amounts of state.

Following this quote I can infer that the future chapter about researching implementation of multi collaboration will benefit greatly from a Parameter Server to extract information based on previous visits without an overwhelming amount of broadcast data. This will save a lot of time by allowing to create an environment requiring broadcast handling is necessary and also save time during execution of each node.

3.1.4 Services

When one-way communication that doesn't allow requests fails, services come into play. They are a two-way communication system of request/reply. It is possible to create a sustainable connection but it will be at a cost of robustness of the system for many reasons like: Depending on one node and one connection to provide the necessary information when either the connection crashes, or the node fails.

This works by a node providing a service under a certain name, and another sending requests to provide it with a message inside it. After doing some research, only one node can provide a specific service meaning that if I would want to create a more robust system, I would have to create a naming convention for same services being provided if I want the system to be robust.

3.1.5 Messages

After discussing the Topics and Services, it is important to mention how the Messages work. Messages are file of type msg that can either be broadcasted across the nodes in a Topic or requested and provided by nodes using the Services. Messages are based on a set of primitive types information like integer, boolean, char and arrays of those are supported which very closely resembles a syntax of C.

Messages have types that distinguish them in between each other. Those types are stored as .msg files and define the data structure for the method so that each message is consistent and is easily extractable and read by other nodes without changing the format of data. This also means that nodes only recieve messages of the type they need. When a message is broadcasted, its md5 is also computed to ensure that the data is consistent and hasn't been tampered.

When it comes to converting the messages from msg to source code, ROS Client libraries come to help with a simple *rosbuild_genmsg*() command that can be added to CMakeList.txt to instruct CMake about installing the ROS packages.

An additional note is that the messages can have a header type set to them that will set the timestamp and FrameID to the message that the ROS can handle for the user making them very useful for debugging and creating statistics to analyze how the code is running.

3.1.6 Parameter Server

> "A parameter server is a shared, multi-variate dictionary that is accessible via network APIs."

This means that it is a server for nodes to use to both share information and hold data for themselves privately. According to ROS specification, it is not designed to be high-performance and should only be used to store state parameters rather than complex information.

Parameters in this server follow the standard ROS naming convention that is used in types as well as services to prevent name collision. It does so by using a hierarchical approach for nested dictionaries meaning that if a user wanted to retrieve a specific parameter, they would have to write down the whole tree. If they, however wanted to retrieve a dictionary of certain information all they would have to do is walk down the tree and a dictionary will be retrieved. In that sense, this sort of structure is not very much different from JSON or XML and makes the Server this much easier to work with.

As for the types supported by the Parameter Server, these are: 32-bit Integers, Strings, Lists, Doubles, Booleans, iso8601 dates, base64-encoded bit data. This actually provides almost all types that Java natively provides without more complex data structures and should easily be sufficient to create data structures in packages using some naming convention manipulation for parameter holding if it becomes necessary.

Parameter Server also supports private naming by using the ˜name as a private name for the node. This precedes whatever name is set to that node and then adds the set name to that creating a unique parameter that is based off the nodes id. Having said that, the parameter is still accessible from other parts of the system but it is much more difficult to locate the desired part and it is more effective when it comes to data collision handling.

Parameter Server will also play a crucial role in this project in the future when possible implementations of multi collaboration will be developed. Holding states of visited states in a list or just generally have a 2-D array of visited arrays will be extremely useful for robots making their decisions.

3.1.7 The Master

The ROS master is a provider of naming and registration services for the rest of the nodes in the system. It holds data on topics and services and allows notes to subscribe and publish messages. It's role is to allow nodes to find one another. However when nodes do finally find each other they communicate in a peer to peer fashion. It also provides the parameter server to the system. It is mostly executed by the $*roscore* command.

A good example taken from the ROS website explains how the master handles the Advertising of data in the ROS environment. We have two nodes: A camera and an Image Viewer. The camera Advertises a topic to the master called images. The Master then is contacted by the Image Viewer to subscribe to the images topic. Once that is completed the master subscribes the Image Viewer and from this point onwards the Image Viewer has access to images whenever Camera node publishes any without any interaction from the master. This was done because the master informed each node of its existence and allowed them to communicate with each other.

3.1.8 Bags

Bags are data loggers for the ROS system. They subscribe to topics and store the serialized message data in a file as it is received. These bags can be played back in ROS to the same topic they were recorded from or remapped to others which makes them extremely useful on occasions where data needs to be replayed to topics so that they can keep consistency in time stamps and messages depend on what was being sent previously.

Data could be replayed by nodes however this could pose issues with timestamped data stored in the message data which could cause conflicts or they data to not be consistent with their md5 sum causing it to not be a desired solution. To deal with this, bags are provided with an option to publish a simulated clock that corresponds to the time the data was recorded in the file creating a more robust solution than using data.

For offline uses of bags one very major advantage is its ability to replay data and sorting it in a timely manner making it a perfect debugging tool for long-term diagnosis. There are also tools for bags that allow visualization of data in the bag file, including plotting fields and displaying images.

On top of that, we are also presented with programmatic APIs in ROS that give C++ and Python packages the ability to iterate over stored messages. For example for quicker manipulation of bag files, there is a tool in rosbag that supports re-bagging a bag file with only information that satisfied a certain filter.

To add robustness to Bags, the rosbagmigration tool allows them to be updated inside the bag based on standards specified before keeping the integrity of the msg files throughout the project without having to discard previous messages that would otherwise be unusable due to a new standard for the message type.

Chapter 4

Developing One Human One Robot Environment

When first started the development I had to decide which would be the right path to take, that is to use Gazebo[2] or to use Rviz[3] with my project. The benefits of Gazebo are that they allow to create a robot from a simple .yaml file description and emulate the environment[4] in a fast manner that Rviz won't allow since it's working lower with the navigation stack. Videos on YouTube have also presented Gazebo[5] to be the simpler environment.

Rviz was a much lower level solution for the navigation stack as it only emulated the navigation stack as a tool and was a more robust tool to creating an environment that allowed customizations[7] to be made and provided an interface for a more programmable perspective through coding. Both had a disadvantage of setting the goal for a singular robot which will be a point of concern in the future when developing more than two robots.

4.1 Developing the skeleton of the code

The first-order priority was using the tutorials provided by the ROS wiki pages to[7] incorporate the skeleton of Navigation stack with Rviz and to start the basic set goal for the robot which started out with a variety of issues regarding the wiki pages and tutorials from the website.

The foremost issue with the codes provided by the wiki pages were that they didn't entirely work together in creating an environment that worked from the user on the go[8], hence requiring the user to branch out into multiple forums to find the right answer and understand the environment by trial and error.

It seemed that the code was so disagreeable that it turned out to make the environment unusable unless sufficient research was made to have a much deeper understanding of the project. To reach my goal I have followed the YouTube video that demonstrated the use of Rviz with tf Navigation stack[9][10] has popped up problems from the start.

The first issue was setting up the map with the server which was later resolved using .launch files to execute commands consecutively and creating a map to odometry broadcaster in the code that constantly created a mapping sequence from the map(Global source) to odometry(Robot control). It was at that time that I found out that singular mapping does not work in navigation stack and to have a successful product I will have to create a broadcaster that will constantly map these topics together.

Following that I was able to launch Rviz and see a map of the environment that the robot will operate on. This environment is 2-D right now but already presents promising possibilities in future development for the robot in 3-D environments. However the robot was not at all present on the map.

After considerable amount of research I have tried to emulate a circular robot to avoid issues with the robot outline to no success eventually finding out correct coordinates in some wiki answer pages[11] and use those to create the robot which resulted in the robot finally showing up on the map with the server without moving at all. Changing its position also didn't seem to hinder the robot which is where I found out that Rviz does not automatically implement "Initial Pose" click button but rather works as a broadcaster itself sending that initial pose to Odometry which should handle the re-positioning of the object itself. This resulted in a realization that there were mapping issues between odometry in a robot that required mapping from Odometry to the robot itself for the function to work properly and to realize that the code from the website had set velocities for the robot causing it to move from the start of the application. Careful code reconstruction allowed to undo that problem and move on with the project having the initial pose function set.

I decided that since I have a working robot and a the initial pose setting, it is time to start setting the set goal functionality with Rviz however that function has created issues of its own, relying on laser data[13], and robot positioning with amcl[12] that would allow creating a map of where does amcl think the robot is for controlling the velocities. This called for extra mapping in odometry and tf_broadcaster topics to allow LaserScan and LaserCloud data to be submitted and for amcl to finally be able to find the position of the robot which eventually led to realization that publishing frequencies along codes was also off and required to be set simultaneously in order to achieve a strong non error probing architecture.

Amcl also required odometry to provide a velocities function which in the wikis was underdeveloped and didn't take into account setting the initial pose orientation or setting velocities that amcl could manipulate.

Having developed those the robot was able to start moving from initial pose to goal pose, however two issues are still unsolved: The robot positioning is more and more uncertain as it moves towards goal phase which creates issues for setting consecutive goals(Provide screen shots of the particle cloud branching out more and more over the screen). This however can be solved with setting initial pose for whenever the robot reaches its goal.

A more serious issue is that the wikis provide code for laser scans and laser cloud that aren't specific to Rviz and server-map which means that mapping for obstacles is almost non-existent and is mostly handled by amcl by itself. This causes the robot to bump into objects much easier but inflating the map to make the robot smaller and swifter on a map could provide the solution. Inflating obstacle avoidance is also a useful tool however it hasn't proved to solve much of the issues with the robot and that robot needs space to maneuver otherwise bumping into walls themselves.

4.2 Future considerations

For the environment to be fully multi-robot friendly there will need to be careful considerations made in order to support a multi-robot architecture in the navigation stack and additional research carried out in order to restructure the code in the future. This means that the Odometry code will have to be restructured in a way that will allow a spawning function to operate and create broadcasting topics for each robot, which could eventually lead to a very careful data structure development along with threaded operations that will work for broadcasting topics and odometry for individual robots, in turn leading to a more scalable approach in the future allowing the users to add robots as they see fit to the environment and controlling them by clicking on individual robots in the environment. This kind of OOP approach will allow a universal robot architecture along with an opportunity to add separate energy levels, photos and reports sent by different robots.

Another important task is development of listening and implementation functions in the environment that will listen for requests and carry them out in a RESTful[REST rules website] manner to support the website GUI and allow future connections. It is of important note that the map in the server is set from a certain corner at (0,0) coordinate and allows coordinate calculations to be made but due to different scalings it might be better to preset coordinates for each room to be able to give the users a choice of where the robots should go rather than right-clicking as each map and each scale would need different coordinate translation or a complex mathematical function which could mean a lengthy process of developing one to fit the needs. An easier solution would be presetting room coordinates at certain scales for the map and telling the user where the robot is, so that they can choose the right path.

For the project to be completed, a future implementation will be required for new listeners to listen to the middle-ware between the website GUI and the navigation stack to allow data exchange and execution. This will call for additional functions to be created in the future.

Chapter 5

Studying server code

Chapter 6

Researching the right tool for GUI

Chapter 7

Implementation of GUI

Chapter 8

Implementing two robot one human environment

Chapter 9

Researching possible implementations of multi collaboration

Chapter 10

Formative testing

Chapter 11

Conclusion

Bibliography

[1] ROS main page, *http://www.ros.org/*.

[2] Gazebo main page, *http://wiki.ros.org/gazebo*.

[3] Rviz wiki page, *http://wiki.ros.org/rviz*.

[4] Helping with setting up yaml files, *http://answers.ros.org/questions/query:yaml%20gazebo/*.

[5] Gazebo YouTube presentation, *https://www.youtube.com/watch?v=0uifKcF0TRU*.

[6] Navigation stack wiki, *http://wiki.ros.org/navigation*.

[7] Navigation stack tf setup, *http://wiki.ros.org/navigation/Tutorials*.

[8] My user account used for help in development, *http://answers.ros.org/users/25237/maciejm/*.

[9] Setting up robot for navigation stack, *http://wiki.ros.org/navigation/Tutorials/RobotSetup*.

[10] Video demonstrating navigation stack with Rviz working together, *https://www.youtube.com/watch?v=0CsSok3QgZk*.

[11] A question holding an extensive footprint description, *http://answers.ros.org/question/221380/global_plan-goes-through-the-wall/*.

[12] A wiki page for AMCL, *http://wiki.ros.org/amcl*

[13] A tutorial holding inadequate LaserScan code, *http://wiki.ros.org/navigation/Tutorials/RobotSetup/Sensors*