Independent Study: Final Report

Ekansh Gupta (egupta@cmu.edu)

Under guidance of David Garlan and Bradley Schmerl

Institute of Software Research
Carnegie Mellon University
Pittsburgh, PA, USA.

August 9, 2016

1 Introduction

1.1 Self Adaptive Systems - Rainbow

With the evolution in software systems, there is an increasing need to minimize the human oversight necessary for operation and support of software applications. To minimize the oversight, the system should be able to detect and fix problems like system errors, varying resources, and changing user priorities. And while doing so, the system should also maintain the goals and systemic properties of the software. Doing this individually for each project requires additional resources and expertise. An architectural approach to this problem can provide solutions that can be used across different types of applications. 'Rainbow' is an attempt to make this possible[1].

Rainbow uses softwares architectural model in its run-time system. The model is used to monitor and reason about the system. The model provides Rainbow the insights on system-level behavior, properties, topographical and behavioral constraints[2]. Rainbow uses this model to establish a scope of change and also ensure the validity of these changes. Doing so, Rainbow can detect changes and faults, and adopt to these effects.

The architecture is divided into three main layers: architecture layer, translation infrastructure and system layer. The system consists of reusable and non-reusable units- Architecture-layer infrastructure, translation infrastructure and system layer infrastructure being reusable. The system APIs created for handling a specific system are non-reusable [1].

For modeling the system and its properties, Rainbow uses architecture which includes components and connectors, constraints, properties, analyses performed on the system.

Rainbow extends this notion of architecture style with dynamic attributes like adaption operators and adaption strategies. The reading goes on to explain two example systems and how the architecture based self adaption can work on them [1].

1.2 Turtlebot/ROS

"TurtleBot is a low-cost, personal robot kit with open-source software. With TurtleBot, a user can build a robot that can drive around your house, see in 3D, and have enough horsepower to create exciting applications" [6]. "TurtleBot combines popular off-the-shelf robot components like the iRobot Create, Yujin Robot's Kobuki, Microsoft's Kinect and Asus' Xtion Pro into an integrated development platform for ROS applications" [7].

ROS stands for robot operating system. It is a meta-operating system for robots that provides services like hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries which give developers the freedom to build, write, and run code across multiple computers [8].

1.3 Goal

The goal of this independent study is to understand Turtlebot, ROS action APIs, instruction graphs and self adaptive systems, and to implement an adaptation of Rainbow on Turtlebot/ROS. The focus will be on creating effectors on Turtlebot, to be used to start/stop a node or swap it with another. Additionally, create an architecture for effectors that can be used in future by the actual self-adaptive system.

2 Important Elements

This section provides details about basic concepts of ROS like nodes, topics and actions. It also explains move_base node and it's working with navigation stack: global planner, local planner and obstacles. Furthermore, it describes instruction graphs, a robot motion language created to interactively instruct a robot to accomplish tasks like move and speak. The section concludes with a description on effectors specific to self-adaptive systems.

2.1 Nodes, Topics and Actions

ROS has three high levels of concept - file system level, computation graph level and community level.Computation graph is a peer to peer network of ROS processes which process data together. Nodes, Topics and Actions are part of this level [9].

A node is a process that performs computation. Nodes are combined together into a graph and communicate with one another using topics, services, and the Parameter Server. A robot control system will usually comprise of many nodes. For example, one node controls the navigation, second node controls wheel motors, third node performs localization, forth node performs path planning, and so on. All running nodes have a graph resource name which uniquely identifies them to the rest of the system [10].

Topics are named buses over which nodes exchange messages. Communication between two nodes happens in publish and subscribe (message bus) architecture, where topics act as message buses. "In general, nodes are not aware of who they are communicating with. Instead, nodes that are interested in data subscribe to the relevant topic; nodes that generate data publish to the relevant topic. There can be multiple publishers and subscribers to a topic. 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" [11].

The 'actionlib' package provides tools to create servers that execute long-running goals that can be changed or cancelled during execution. It also provides a client interface in order to send requests to the server known as the action client. The client and server provide a simple API for users to request goals (on the client side) or to execute goals (on the server side) via function calls and callbacks [12].

2.2 Move_base

'Move_base' is a node and is an important component of the navigation stack.The move_base node provides a ROS interface for configuring, running, and interacting with the navigation stack on a robot [13].

As shown in figure 1, move_base uses global and local planners to accomplish its actions. The local and global planners are plug-ins and can be changed. Also, move_base implements an action server and provides an action client for interactions.

Global planner is responsible for creating a plan that can be executed to achieve a goal. This plan is a path created to reach the target position of a goal.

Local planner is responsible for executing the plan created by the global planner. Local planner continuously evaluates the surroundings and determines the cost of following the plan created by the global plan. When the cost of executing a plan is greater than the number set in the properties, local planner goes in recovery mode and requests global planner for a new plan. Cost in movement refers to the feasibility of executing a path. For example, on introduction of a new obstacle in the planned path, the cost of execution increasing causing the local planner to re-evaluate its path.

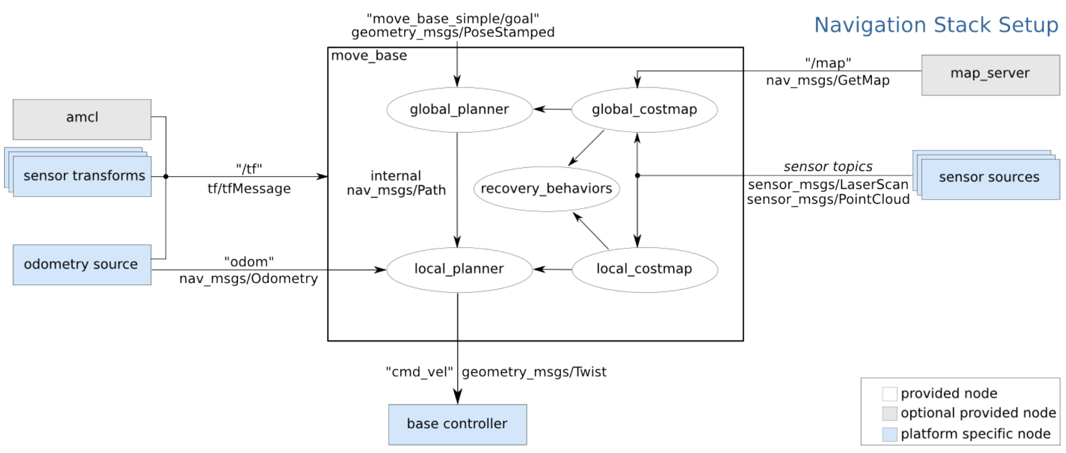

Figure 1: The move base node and various sensors related to it [13].

2.2.1 Recovery Behavior

Recovery behaviors are local planners' ways to acquire information about the surroundings and provide it to the global planner. This allows global planner to re-evaluate the surrounds and create a new plan based on the latest reality. Figure 2 shows the default recovery behavior of the Turtlebot.

Figure 2 shows a sequence of recovery behaviors carried out by move_base in order to find a new path to the target location. It starts with conservative reset, in which case local planner clears the obstacle from the cost map and tries to find it again in the background. If the behavior works move_base resumes the execution, otherwise it moves to the second behavior i.e. clearing rotation.

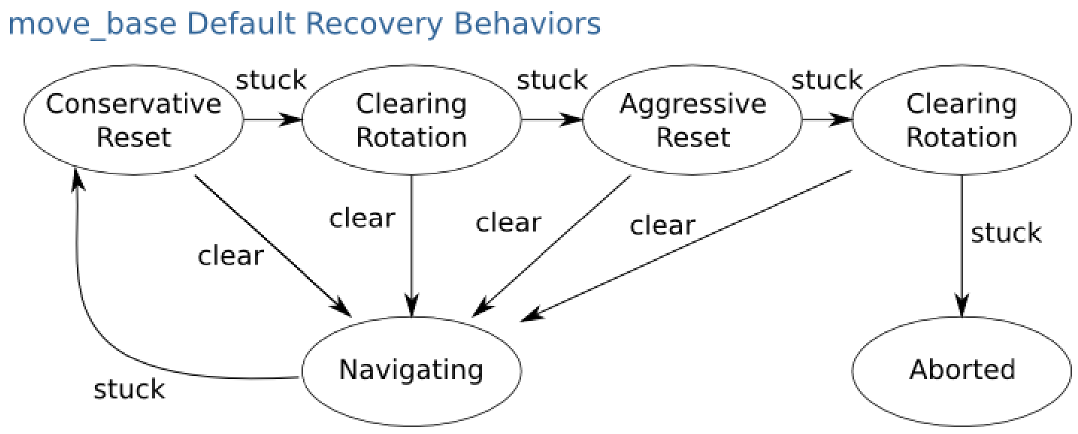

Figure 2: The default recovery behaviors for move_base [13].

2.2.2 Move_base goal execution

To explain how a goal is executed in move_base, lets have a look at figure 3. A goal is a task created by a programmer or application to be executed on an action server. Move_base goals can be of two types: going to a specific position on the map or moving around with respect to the current position.

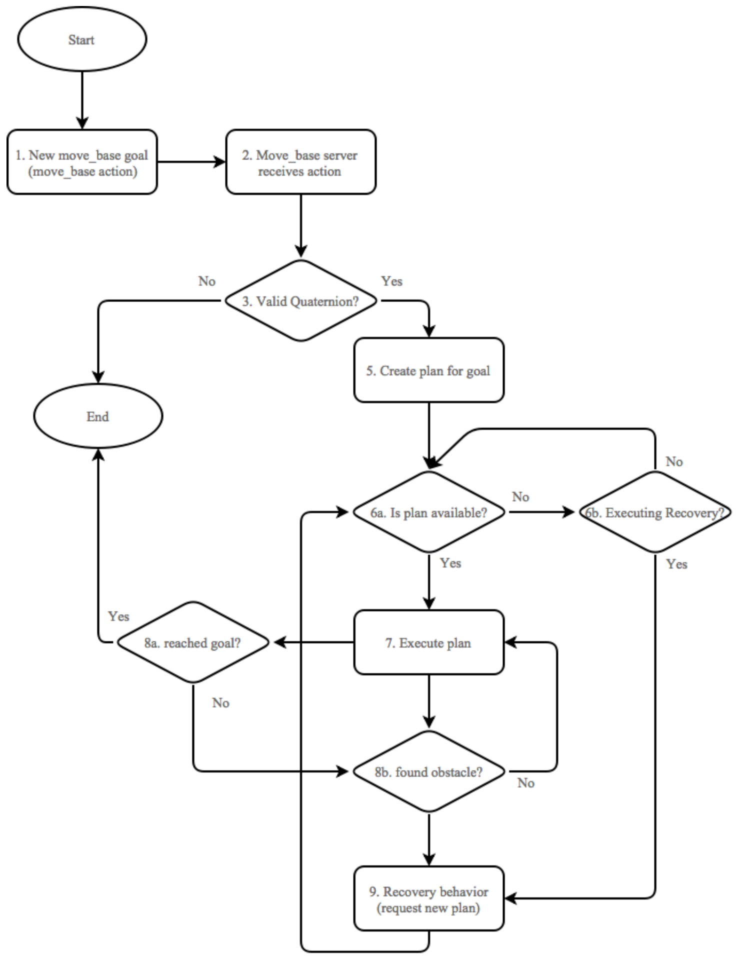

Figure 3: The flow of events that take place to execute a goal.

Note: Figure 3 shows a simplified version of the tasks taking place in move_base and is not a real representation of execution. The diagram abstracts away other processes running in the actual execution.

Figure 3 shows the process where a programmer or code sends a new goal to the move_base server. We use move_base client to send this goal. The python file in citation provides an example for creating an action client [14]. On receiving a new goal, the action server validates the goal (as shown in bullet 3, quaternion is a number system used to refer mechanics in three dimensions [15]). On validation, the target positions from the goal are sent to the global planner to create a new plan for the bot to move to the target location. This process is executed by make_plan service. Once a new plan is available, the plan is executed by local planner. The local planner continues to execute the plan till it has reached its target position or is stuck because of an obstacle. In case of an obstacle, local planner waits for the global planner to create a new plan, meanwhile executing the recovery behaviors as discussed above.

Global planner, local planner and the sensors work in synergy to generate and follow a plan. The behavior of these components depends on the distance of the obstacle from the robot. Here are a few scenarios to explain the same.

2.2.3 Obstacle avoidance

Consider a map as shown in figure 4; the objective for the bot is to reach from the start position in blue to the end position in green. The global planner creates a plan (shown in dotted line) to be executed by local planner. Here are a few cases in which an obstacle can be introduced.

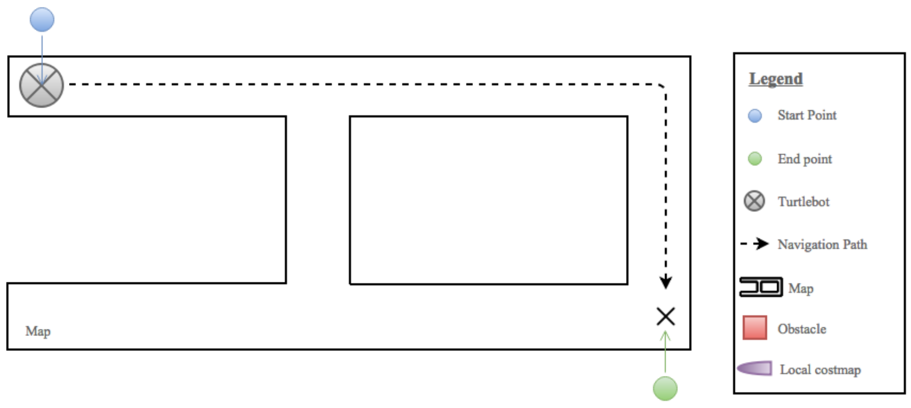

Figure 4: Top view of a map with Turtlebot at the start position and starting execution of a plan to reach end position.

Case 1 - An obstacle is detected at a reasonably far distance from the bot
We call any distance outside the range of the local planner as far distance. In this case the the local planner range is shown by the purple triangle.

- Sensors detect the obstacle introduced.

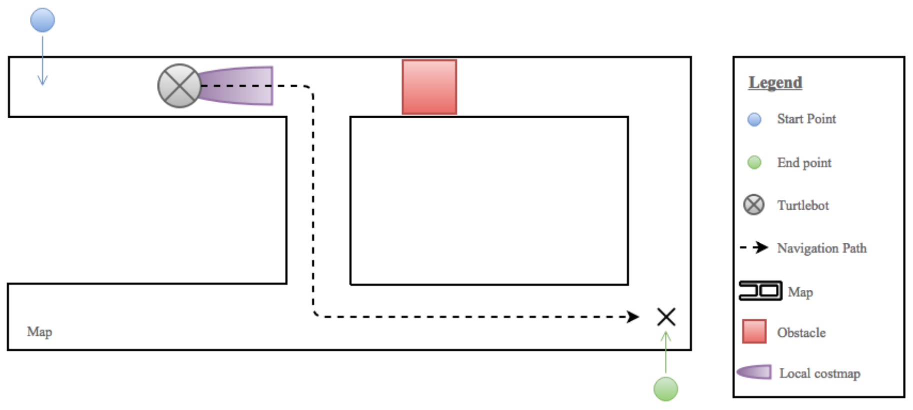

Figure 5: Showing top view of a map with an obstacle introduced at a far distance from the bot

- As the distance of the obstacle is outside the range of the local planner, the obstacle's cost is not added to the local cost-map.
- The cost of getting through the obstacle is added to the global cost-map. Global planner re-evaluates the plan and creates a new plan to be executed. This is evident from the change in the dotted line.

Case 2 - An obstacle is detected at a very small distance from tbot.

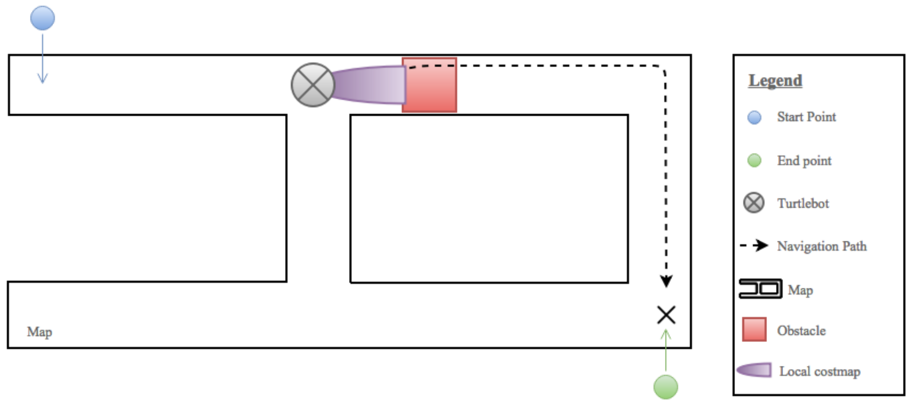

Figure 6: Showing top view of a map with an obstacle introduced close to the turtlebot

In this case, the distance from the obstacle is within the range on the local planner. The steps taken by move_base are as follows:

- Sensors detect the obstacle introduced.

- As the distance of the obstacle is within local planner, the cost is updated in both global and local cost-map.
- The global planner detects the change and tries to find a new plan.
- The local planner detects the change and starts recovery behaviors.
- The local planner continues recovery behaviors till a valid plan is available to be executed.

In this case, as the bot sensors cannot see the whole obstacle, the global planner provides a plan that is actually not feasible. As the global planner is unaware of the size of the obstacle, it provides a bad plan.

Case 3 - Global planner provides an unfeasible plan.

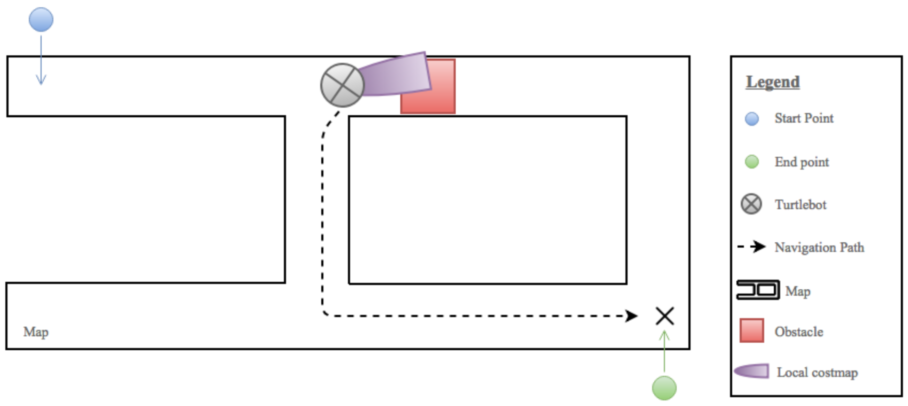

Figure 7: Showing top view of a map with turtlebot trying to execute an unfeasible plan.

In this case, the global planner is not aware of the actual size of the obstacle and so provides a plan that is not feasible.

- Local planner tries to execute the new plan.
- As it is not able to execute the plan, the local planner goes back in the recovery mode.
- As part of recovery, the bot rotates around its axis to explore the surroundings. This gives global planner a better chance of creating a feasible plan.

During the recovery behavior, the global planner will be updated with the actual dimensions of the obstacle and it will create a new feasible plan for getting to the target.

2.3 Instruction Graph

Instruction graph is a language created to convert sequence of primitives carried out by the bot into human readable graphs [16].

2.3.1 Language Representation

Tuple representation

$G = \langle V, E \rangle$
Here, V is a vector and E is the connection between two V.

$V = \{v_i | \forall i \in [0, n]\}$ and
$E = \langle V_1, V_2 \rangle$.

and v_i is defined as:

$v_i = \langle ID, ActionType, Action \rangle \; where, \forall v \in V, \; ID = i.$

The ActionTypes for vectors are:

- **Do** - Single statement
- **DoUntil** - Loop
- **Conditional** - If/else
- **Goto** - To go back to a specific vector.

And actions are primitive actions for the robot like move and speak.

2.3.2 Instruction Graph Interpreter

Based on these primitives created by the Robotics Institute at Carnegie Mellon University, Andrew Benson from Institute of Software Engineering created an interpreter for Turtlebot [17].

The main components from figure 8 are described as follows:

- **Main** - Starts execution, reads instruction graphs
- **Parser** - It is used to parse the instruction graph into graph object.
- **Lexer** - It is used to validate the correctness of the syntax of instruction graph.
- **Dynamics** - Divides the graph object into vectors and runs them.

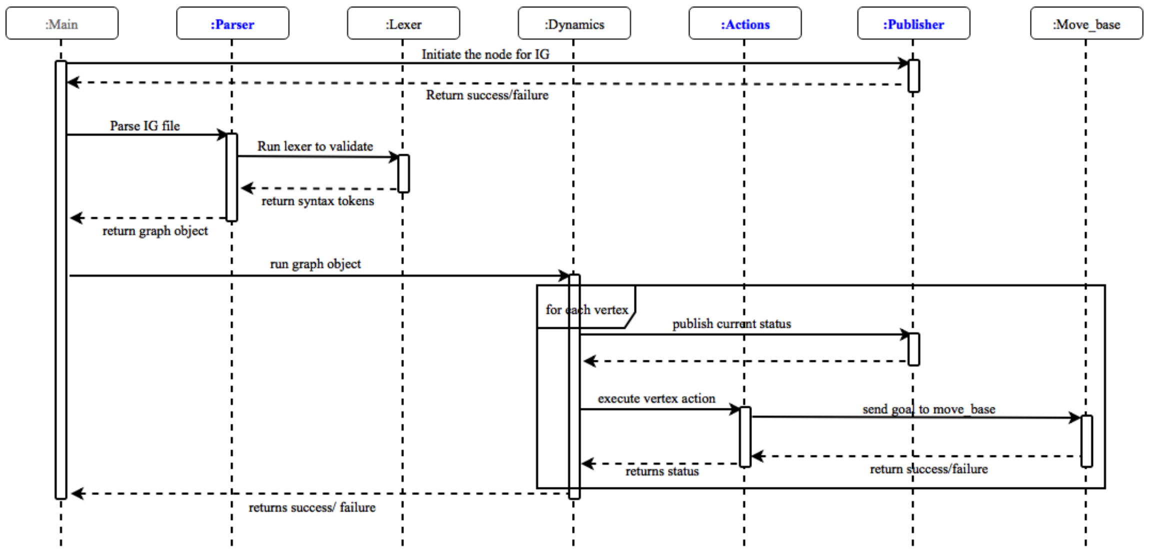

Figure 8: Sequence diagram showing the sequence of events in the execution of instruction graphs.

- **Actions** - Used to interact with move primitives on turtlebot. In this case we use move_base.
- **Publisher** - It is used to start a node and publish status messages.
- **Move_base** - Used to move the turtlebot.

Changes made to the original interpreter (figure 8 shows them in blue) [19]:

- **Parser** - The original parser for the interpreter worked on forward movement and turn. This two tuple representation cannot capture all the movement of the bot. To made it robust, the representation is now changed to a five tuple, i.e., {Δx,Δy, linear speed, angular speed, rotation}. For example, {1, 0, 1, 0 ,0} means move the bot 1 meter forward at a speed of 1 m/s.
- **Actions** - The action implementation for turtlebot was based on cmd_vel topic. Also, the implementation was developed for a two tuple representation. This implementation is now modified to work with five tuple representation using cml_vel topic. Furthermore, a new move_base based implementation has been added.
- **Publisher** - A new publisher module has been added to the implementation. This module can be used to initiate the node and publish status message on a topic.

Deciding what to use: Move_base or cmd_vel

Both the options can be used to move the bot. However, cmd_vel does not use the navigation stack. Hence, while running cmd_vel actions, the bot does not provide any safety guarantees. On the other hand, move_base uses navigation stack, which provide safety and recovery behaviors.

Both the options can valid can be used in different situations. The implementation uses move_base because safety is an important attribute for adaption.

2.4 Effectors

In Rainbow, effectors are used to perform adaptions on the target system.The decision for a change is evaluated in the architectural layer and passed on has commands to the effectors. For Turtlebot, these changes are switching nodes, changing algorithm etc. These changes help the bot save power, graceful complete/fail task and adapt to the situations. Currently, there are a few scripts running as a node and subscribed to a topic. Based on the publications on the topic, effector node can switch two nodes.

3 Approach and Architecture

3.1 Obstacle avoidance with Instruction graphs and Move_base

As we have seen in section 2.2, move_base can be used to avoid obstacles. We apply this concept to instruction graphs. A typical instruction graph contains information about movement and turns. For example, move 2 meters forward, turn 90 degrees, move 1 meter etc. We individually execute each of the vectors as goals in move_base. So, we provide move_base a goal to move 2 meters forward and when completed we provide another vector.

3.1.1 Avoiding recovery behavior on encountering an obstacle

As seen before on encountering an obstacle move_base will execute recovery behavior. We do not want to execute recovery behavior because recovery behaviors are unpredictable. And triggering recovery can cause two problems:

- We cannot ensure a specific position in which the recovery will end. Also, as we are executing the Instruction graph in parts, executing a recovery will result is synchronization issues between the vectors in the graph.
- To power consumed during the recovery process can vary on the time it takes the bot to find a new path. It can be a difficult task to estimate the power consumption during recovery.

Hence, to avoid this problem we want to avoid move_base executing recovery behavior. Recovery behavior in move_base can be disabled using a few parameters in move_base [18].

In this case, on encountering an obstacle, move_base will still stop, but it will not execute recovery behaviors.

3.1.2 Moving short distances to avoid redirection

Imagine the case discussed in figure 5, an obstacle is introduced at a far distance from the bot (let consider this distance to be 4 meters). If the bot has to go forward more than the distance away from the obstacle (like 5 meters), the global planner will redirect this path as the current path is not feasible. This causes two main problems:

- This will render instruction graph execution useless as no guarantees about the bot's orientation can be made, and hence, the next vertex cannot be executed.
- Also, this is bad for power estimations. Instructions in the graph can be used to make power estimations before starting execution. But, as the bot goes off script from following the graph, the power estimations are no longer valid.

To avoid the situation, the goals sent to move_base are short (like 1 meter). The approach assumes that an obstacle is not smaller than this measure, and so, the turtlebot will not attempt to find a way around it.

3.2 Using effectors

Section 2.3 demonstrates moving turtlebot using instruction graphs and move_base. Section 3.1 demonstrates how to stop recovery to maintain state and avoid obstacle. We integrate all the learning from before to swap instruction graphs on detecting an obstacle.

Figure 9 shows the steps involved in the demonstration. Following are the description of the main components involved.

- **Instruction Graph 1** - The node for the first instruction graph.
- **Instruction Graph 2** - The node for the second instruction graph.
- **Move_base** - Used to run move actions and detect obstacles.
- **Move_base status** - The status messages published by move_base. It publishes status 4 for obstacles/error in execution.
- **Effector** - Used to swap nodes, the effector kills instruction graph 1 and spawns instruction graph 2. The effector listens to the effector topic (message type string), upon receiving the message to swap, it performs the action.
- **Effector topic** - Used by the monitor to publish messages for effector.
- **Monitor** - Monitors move_base status, in case of status = 4, instructs effector to swap instruction graphs.

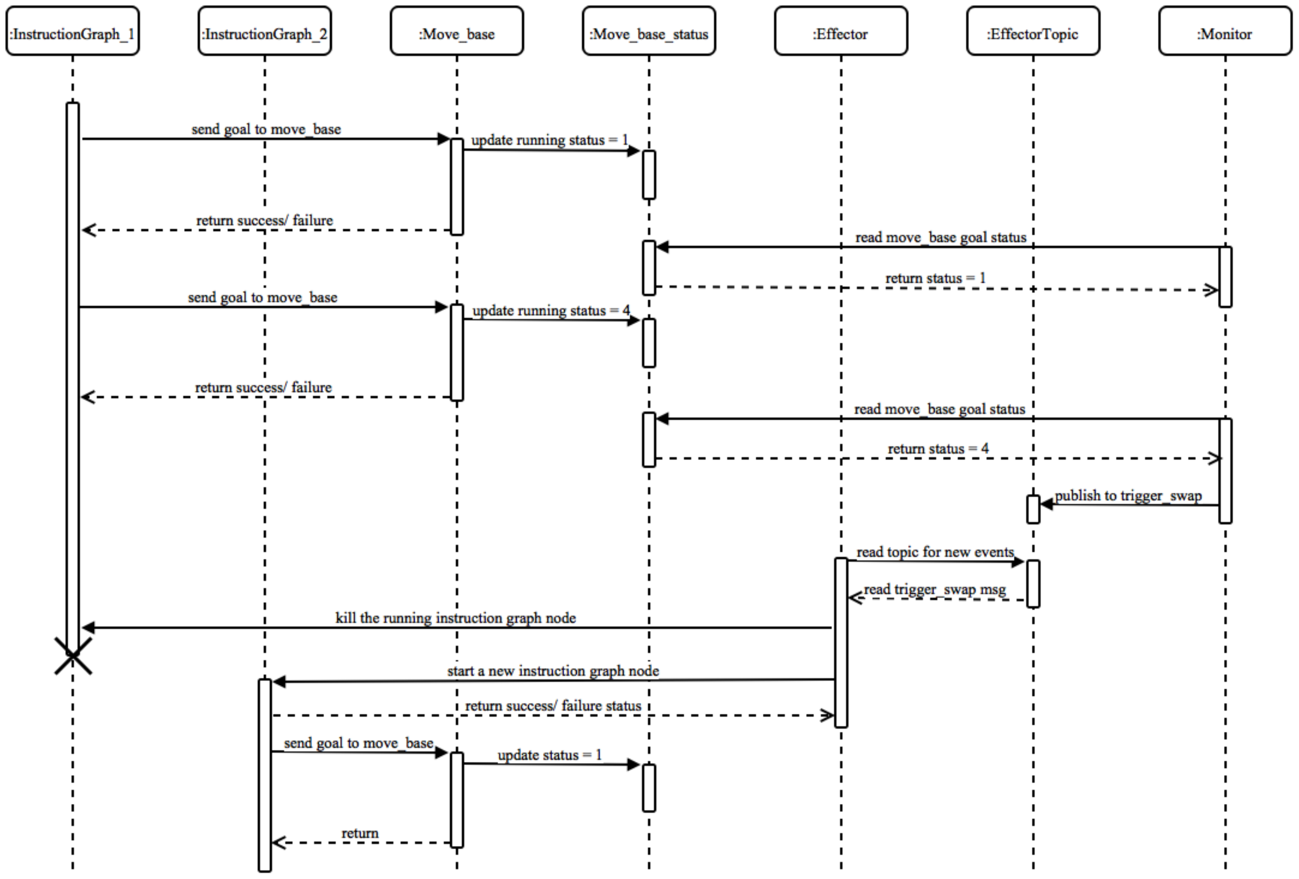

Figure 9: Sequence diagram showing the sequence of events involving in replacing instruction graphs using effectors on encountering an obstacle.

Essentially, the architecture is divided into three parts: Work, monitor and administer. The actual work is performed by instruction graph and move_base. Monitoring is carried out by the monitor continuously querying move_base status. Effector node does administration, based on the situation it can start or stop nodes.

4 Future Research

This research is a preliminary step towards adaption of the Turtlebot/ROS. The demonstration only works for a specific map and can only swap between the two instruction graphs provided.

4.1 Instruction Graph

In the future state of the system, the self adaptive system should be able to create instruction graphs on run-time. This instruction graph can be provided as input to the instruction graph interpreter.

Creation of a new instruction graph should be done in the architectural layer of the self-adaptive system. This graph created can now be provided to the effector by publishing on the effector topic. Currently, the effector topic only reads a string, however, in the future effector topic can have a complex message type and read an instruction graph as input. This input can now be provided to the interpreter which is to be executed.

Currently, the interpreter only takes files inputs- it needs to be modified appropriately to enable input of a complex message or a string.

Alternatively, another option to implement instruction graphs is creating an action server. Action server is a ROS component which has five fixed topics:

- **Status** - The action server publish the progress on the status.
- **Goal** - The action server subscribes to this topic. Action client can send new goals on this topic.
- **Feedback** - The action server publishes feedback on this topic. Feedback can be used to display details about the execution.
- **Result** - The action server publish results on this topic.
- **Cancel** - The action server subscribes to this topic. Action client can send a goal cancel request on this topic.

At present, instruction graph publish module uses status and goal topics to publish current status and current goal. However, to stop an executing instruction graph, the effector has to kill the node and spawn another. An action server implementation solves the problem. Moreover, it provides a ROS standard solution. We can reuse status and goal topics and introduce cancel goal instead of killing a node.

4.2 Modelling power consumption

Instruction graphs can be used to move a bot from one position to another. Instruction graphs use primitive actions like moving forward and turning. Power consumption for moving forward and turning can be recorded for different surfaces. Now, these recordings can be mapped to an instruction graph to estimate the total power required to conduct a task. These estimates can be further used to evaluate the feasibility of a task.

However, for doing so we need to be careful about the following things:

- Recovery behavior should be turned off. This will ensure that in case of encountering an obstacle, the bot does not go off script.
- Instruction graph should execute short distance goals. This will ensure that in case of an obstacle the global planner does not redirect the bot follow an alternate path.

5 Reflections

As a part of the independent study, I have gained in-depth understanding on Robot operating system and the major components of ROS such as Nodes, Topics, Services, Actions and Parameter server. I have also attained knowledge of instruction graphs and how five dimension notation (Δx, Δy, angular speed, linear speed, rotation) can be used to move robots. While working on navigation, I was also introduced to the many ways a robot can be made to move, for example, autonomous navigation using maps, instruction graphs, manual movements using keyboard and the complexity involved in the different types of interactions.

I also learnt how the global planner, the local planner and the sensors work together to avoid obstacles and provide the bot alternate path to reach its goal. As part of self adaptive system I got to know how monitors and effectors can be used to achieve self-adaption. I learnt two new languages during the course of the study: Python (rospy) and Instruction graphs (IG). The learning from self adaptive systems and cyber-physical systems can be extended to solve problems like managing traffic lights, managing self driving cars, and creating smart cities etc.

During the research, I worked on a simulator (Gazebo). Many a times using the simulator was difficult and it caused delays in finding results. Eventually, I made it a point to double check the results as simulators could provide deceiving results. Also, testing the code on the actual bot was different than that on the simulator and took a lot of time as the actual bot's motion can be biased. For example, while creating 90 degree turn for the bot, the turn worked perfectly in the simulation but did not work well in the actual bot.

As ROS is open-source, the documentation is minimal. Initially, this caused issues in understanding ROS concepts. For some details about the system, I had to explore and study the code and infer the meaning. Bad documentation caused a lot of inefficient work. Properly documenting the concepts will make understanding them easier. As part of the initiative, I have tried to document the part of the system that I worked on.

References

[1] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl and Peter Steenkiste. *Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure.*

IEEE Computer, Vol. 37(10), October 2004.

[2] Shang-Wen Cheng, David Garlan, Bradley Schmerl. *Evaluating the Effectiveness of the Rainbow Self-Adaptive System.* Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA - 15213, USA.

[3] Bradley Schmerl, Jonathan Aldrich, David Garlan, Rick Kazman and Hong Yan. *Discovering Architectures from Running Systems.* Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA - 15213, USA.

[4] Shang-Wen Cheng, David Garlan and Bradley Schmerl. *Stitch: A Language for Architecture-Based Self-Adaptation.* Journal of Systems and Software, Special Issue on State of the Art in Self-Adaptive Systems, Vol. 85(12), December 2012.

[5] David Garlan, Bradley Schmerl, and Shang-Wen Cheng. *Software Architecture-Based Self- Adaptation.* Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA - 15213, USA.

[6] Open Source Robotics Foundation, Inc. `http://www.turtlebot.com`

[7] Open Source Robotics Foundation, Inc. `http://wiki.ros.org/Robots/TurtleBot`

[8] Open Source Robotics Foundation, Inc. `http://wiki.ros.org/ROS/Introduction`

[9] Open Source Robotics Foundation, Inc. `http://wiki.ros.org/ROS/Concepts`

[10] Open Source Robotics Foundation, Inc. `http://wiki.ros.org/Nodes`

[11] Open Source Robotics Foundation, Inc. `http://wiki.ros.org/Topics`

[12] Open Source Robotics Foundation, Inc. `http://wiki.ros.org/actionlib`

[13] Open Source Robotics Foundation, Inc. `http://wiki.ros.org/move_base`

[14] Rainbow Effectors for BRASS `https://github.com/ekanshgupta90/rainbow_effector/blob/master/move_base/move_base_action.py`

[15] Open Source Robotics Foundation, Inc.
`https://en.wikipedia.org/wiki/Quaternion`

[16] Cetin Mericli, Steven Klee, Jack Paparian, and Manuela Veloso.
An Interactive Approach for Situated Task Specification through Verbal Instructions. In Proc. of AAMAS, 2014.

[17] Andrew Benson: Instruction Graphs
`https://github.com/anbenson/instructiongraph`

[18] Rainbow Effectors for BRASS
`https://github.com/ekanshgupta90/`
`rainbow_effector/`

[19] Instruction graph modified for robustness
`https://github.com/anuragkanungo/instructiongraphs`