



SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING

AI-Enabled Information Technologies for Multi-Robot Coordination

Nha Nam Nguyen Nguyen

B.E.(E&E)(Hons.)

Supervisor: Professor Cheng-Chew Lim

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENT FOR THE DEGREE OF
BACHELOR OF ENGINEERING

June 13, 2020

CONTENTS

Abstract	v
Acknowledgement	vii
List of Acronyms and Abbreviations	ix
List of Symbols	xi
List of Figures	xiv
1 Introduction	1
1.1 Motivation	1
1.2 Research Objectives	2
1.3 Technical Challenges	2
1.4 Contribution	3
1.5 Thesis Structure	3
2 Background	5
2.1 Distributed System	5
2.2 Ad Hoc Network	6
2.3 Robot Operating System	8
2.4 Turtlebot3 Burger	11
2.5 Literature Review	13
2.5.1 Robot Localisation	13
2.5.2 Navigation Planning	14
2.5.3 Simultaneous Arrival	16
2.5.4 Multi-Robot System	16
2.5.5 Wireless Network Connectivity	17
3 Cooperative Navigation for Multi-Robot System	19
3.1 Simultaneous Arrival using Dynamic Window Approach	19
3.1.1 Dynamic Window Approach	19
3.1.2 Problem Formulation	21
3.1.3 Cost Function for Simultaneous Arrival	22
3.1.4 Distributed Max-Consensus Algorithm	23

CONTENTS

3.1.5	Event of Member Failure	24
3.2	Distributed Multi-Robot Coordination Framework	25
3.2.1	Task Synchronisation	26
3.2.2	Member Failure Handling	28
4	Distributed Multi-Robot Testbed	31
4.1	Testbed Requirements	31
4.2	Distributed Communication	33
4.2.1	Ad hoc Network	33
4.2.2	Multi-Master Stack	34
4.3	Multi-Robot Cooperative Navigation	36
4.3.1	Navigation Stack	36
4.3.2	Cooperative Navigation Stack	38
4.4	Software Integration	40
5	Results and Discussion	45
5.1	Experimental Setup	45
5.2	Localisation Stack	46
5.2.1	Localisation Strategy	47
5.2.2	Discussion	47
5.3	Simultaneous Arrival using Dynamic Window Approach	49
5.3.1	Simulation	49
5.3.2	Physical Environment Testing	53
5.3.3	Discussion	54
5.4	Distributed Multi-Robot Coordination Framework	56
5.4.1	Task Synchronisation	56
5.4.2	Team Resilience	57
6	Conclusion	61
6.1	Research Summaries	61
6.2	Recommendations of Future Work	62
Appendix A	System User Manual	63
A.1	Prerequisites	63
A.1.1	Hardware Requirements	63
A.1.2	Software Installations	63
A.1.2.1	On the Commander	63
A.1.2.2	On the Robot	65
A.2	System Configuration	66
A.2.1	On the Commander	66

CONTENTS

A.2.2	On the Robot	68
A.2.3	On the Bridge Device	71
A.3	Quick Start	73
A.3.1	Simulation	73
A.3.2	Physical Environment	76
A.4	Testbed Software Parameters	79
A.4.1	Cooperative Navigation Stack	79
A.4.1.1	Simultaneous Arrival using DWA	79
A.4.1.2	Localisation Stack	80
A.4.2	Distributed Multi-Robot Coordination Framework	80
A.4.3	Mission Clients	81
	Bibliography	83

ABSTRACT

Coordinated behaviours in a multi-robot system allow missions such as search and rescue, exploration or transportation to be executed in a robust, reliable and efficient manner. Being coordinated requires information exchange among team members, which leads to various potential issues where the failure of a single member could cause the mission failure. This research focuses on cooperative navigation under disturbances where a team of robots is tasked with a navigation mission to arrive at the targets simultaneously.

In this work, a new technique is proposed to solve the problem of simultaneous arrival in a cooperative navigation mission. It modifies the Dynamic Window Approach - a widely-used motion planning technique - to incorporate simultaneous arrival with object avoidance capability. This research introduces a coordination framework for task synchronisation in a distributed multi-robot system which can handle events of team member failure, yet still ensure mission achievement. A ROS-based distributed multi-robot testbed enabled by an ad hoc network is then developed to verify all proposed techniques in a unified system.

The proposed techniques were tested in simulation and deployed in physical environment. Simultaneous arrival using Dynamic Window Approach performed well in different complex scenarios under the presence of dynamic obstacles and events of member failure. It also showed promising results where the average time difference of arrival was 0.45 second. Task synchronisation produced desired behaviours with the average synchronisation latency was 0.4 second. Team member failure handling was proven to behave as expected where the team completed the assigned missions under various events of failure. The ROS-based distributed multi-robot testbed deployed successfully an ad hoc network whose routing was enabled by the BATMAN-adv protocol. This method established a reliable and robust communication medium for the coordinated multi-robot system.

ACKNOWLEDGEMENT

Firstly, I wish to express my deepest gratitude for my supervisor - Professor Cheng-Chew Lim. The research would have been impossible without your persistent guidance throughout the journey. I, professionally and personally, have learned a lot from you; especially the way you transform daily life problems into constrained - or sometimes, unconstrained - optimisation ones.

Special thanks to Dr Tan Minh Doan, Mr Paul Solomon, and Dr Andrew Walters for always encouraging me, especially during the time I was getting lost in the research. You all have really turned a tough project into an interesting journey.

I also had great pleasure of working with Dr Syed Imranul Islam. Without your consistent help, the ad hoc network deployment would not have been achievable.

Thanks should also go to Mr Danny Di Giacomo for the support to prepare the testing environment. I very appreciate your time and your patience when we had been working late at night for several weeks.

The testbed can not be fully tested without the help of my project partner, Zeping Zhao. Thanks for driving me to the university, and never complaining, regardless of the weather.

To my parents, I am forever grateful for your love, care and continued support through thick and thin. Mom and Dad, I would not be where I am today without your education and opportunities you gave me.

To Ha Hanh Duyen, who inspired me and reinforced my belief to keep working on SADWA, thank you so much for always knowing how to make me smile even when I was at my lowest.

Last but not least, I want to say thanks to my friends in the Cave, who make sleepless nights doing research so meaningful.

LIST OF ACRONYMS AND ABBREVIATIONS

ACK	acknowledgement. 27
B.A.T.M.A.N	Better Approach To Mobile Ad hoc Network- ing. 17
CNS	Cooperative Navigation Stack. 38
DMRC	Distributed Multi-Robot Coordination. 25
DNS	Domain Name System. 8
DWA	Dynamic Window Approach. 15
ETA	estimated time of arrival. 51
IMU	Inertial Measurement Unit. x
KLD	Kullback-Leibler Distance. 14
LiDAR	Light Detection and Ranging. 12, 13, 39
LTS	Long Term Support. x, 62
MCL	Monte Carlo Localisation. 14
OLSR	Optimised Link State Routing. 17
OS	Operating System. 8
ROS	Robot Operating System. 8
RViz	ROS Visualization. 10

LIST OF ACRONYMS AND ABBREVIATIONS

SADWA Simultaneous Arrival using Dynamic Window
Approach. 19

TDA time difference of arrival. 53, 54

LIST OF SYMBOLS

D	associated array.
$G(\cdot)$	objective function of Dynamic Window Approach.
$J(\cdot)$	simultaneous arrival's objective function.
K_{ij}	covariance between the dataset i and j .
$L(\cdot)$	path length function (m).
N	number of robots in a team.
T	estimated time of arrival (sec).
V_a	set of admissible velocities.
V_d	set of velocities within dynamic window.
V_r	resulting velocity search space.
V_s	initial set of feasible velocities.
Σ	covariance matrix.
$\Theta(\cdot)$	tight bound on algorithm's complexity.
$ A $	cardinality of a set A .
$ f(x) $	absolute value of function $f(x)$.
$\mathcal{O}(\cdot)$	upper bound on algorithm's complexity.
$\mathcal{P}$	set of collision-free paths to target.
$\mathcal{R}$	set of a robot's neighbours.
$\mathcal{U}$	decision vector.
ϕ	consensus time (sec).
ω_l	localisation's rotational velocity (rad/sec).
ω	rotational velocity (rad/sec).
σ^2	variance.
ε	localisation tolerance (m ²).
u_t	decision variable at time t .
v	translational velocity (m/sec).

LIST OF FIGURES

2.1	Connectivity of a centralised system and a distributed system	6
2.2	Dynamic topology under an ad hoc network.	7
2.3	ROS topics communication.	9
2.4	ROS service communication.	9
2.5	ROS action communication.	10
2.6	An RViz view in a navigation task.	11
2.7	A simulated operating environment in Gazebo.	11
2.8	Turtlebot3 Burger.	13
4.1	A state machine of testbed operations.	32
4.2	Network architecture of the proposed testbed.	33
4.3	Components and data path in multimaster_fkie.	36
4.4	Components and data paths in the navigation stack.	37
4.5	A navigation mission executed by the Navigation Stack.	38
4.6	Components and data paths in the Cooperative Navigation Stack. . . .	39
4.7	Server side architecture.	41
4.8	Client side architecture.	41
4.9	Network architecture of the proposed testbed.	42
4.10	Software architecture for each robot.	43
5.1	A simulation environment of 70 m ² with 5 robots.	46
5.2	A physical operating environment of 36 m ² with 3 robots.	47
5.3	Localisation strategy developed in the CNS (simulation).	48
5.4	Localisation with different values of tolerance (simulation).	48
5.5	A navigation task with static and dynamic obstacles.	49
5.6	Trajectory generated from the team of 5 robots.	50
5.7	Handling a complex situation where crossing paths exists during navigation.	51
5.8	ETAs over time of 5 robots.	52
5.9	Simultaneous arrival in the event of team members' failure.	52
5.10	Simultaneous arrival's data points and histogram.	53
5.11	Operations in a complex environment with static, unexpected obstacles, and people.	54

LIST OF FIGURES

5.12	A trade-off between simultaneous arrival reliability and position tolerance.	55
5.13	The distribution of the TDAs in simultaneous arrival tasks.	56
5.14	Maximum synchronisation latencies data points.	57
5.15	A histogram of synchronisation latencies.	57
5.16	Operation of expiry removal in a cooperative navigation task.	58
5.17	Program log of a robot under the event of another member's failure. .	59
A.1	RViz commander interface in simulation.	76
A.2	RViz commander interface in physical environment's deployment. . .	78

CHAPTER 1

INTRODUCTION

1.1 MOTIVATION

Human beings are gifted with such a wonderful ability that one might subconsciously take for granted - the ability of *collaboration*. There exist tasks naturally-rooted in collaboration such as construction, transportation, or production. Such tasks require each individual to perform collaborative activities to satisfy the team goals in a more productively and efficiently manner. For years, coordination between autonomous systems is motivated by such nature. Coordination aims to efficiently carry out complex missions by the employment of a *multi-robot system*. Coordination in a multi-robot system exhibits system flexibility and reliability where the failure of one or more robots does not affect the performance of the whole system. A task executed by a multi-robot system takes less execution time, computational resource, or energy consumption than a task executed by a single robot.

Executing a task, regardless its complexity, requires a robot to make proper decisions to optimally satisfy the requirements. Decisions are usually made by analysing data extracted from the sensors equipped in a robot. Recent advancements in the field of Artificial Intelligence (AI), along with the commercialisation of various high-end types of sensors, provide a robot with the ability to make complex decisions on its own in response to the surrounding environment. Furthermore, a joint-mission executed by a multi-robot system also requires information to be exchanged frequently among team members. The information exchange process allows one robot to make decisions in harmony with other team members to satisfy the team's goal. Thus, it is required to establish a reliable communication channel within the team. Thus far, emerging technologies in computer network - such as the Internet of Things - have enabled robust, efficient and scalable protocols for internode communications.

This research primarily focuses on the question: can the combination of AI, and information technology's advancements, enable a multi-robot system to execute co-operative missions under disturbances in a robust, reliable and resilient manner?

1.2 RESEARCH OBJECTIVES

In this research, indoor navigation is selected as the primary cooperative task. To answer the research question in the context of cooperative indoor navigation, specific scenarios are set out. In a rapidly-changing contested environment, a team of robots navigates, individually and collectively, through an area and arrives at a set of targets simultaneously. The team is loaded with a priori relevant to the contested operating environment. Each robot relies on its sensors to collect data on the surrounding environment, and uses AI techniques to process the data so that the priori is updated periodically to cope with the unpredictable changes in the operating environment. Throughout the mission, information exchange is required to collaborate each robot's local information so that the team's navigation mission can be achieved. In the event of member failure, the team coordinates the information from the active robots in a way that is adaptively reorganised to achieve the mission. To add another layer of guarantee to simultaneous arrival, every new navigation task must be synchronised - or in other words, be activated at the same time - across the team members.

From the scenario, several research objectives can be identified. The research aims to study various navigation techniques and apply the domain knowledge to simultaneous arrival missions. It also outlines different computer networking approaches to establish a reliable, robust and scalable communication medium for a multi-robot system. This research also aims to develop a multi-robot coordination framework which serves as an abstract interface to handle task synchronisation and team member failures. With the objectives being stated, the research outputs a multi-robot testbed where all research's studies and proposed techniques can be verified in a unified system.

1.3 TECHNICAL CHALLENGES

Being able to achieve autonomous and collaborative behaviours requires a team of robots to make appropriate decisions throughout the assigned missions. In a constantly changing real-world environment, it is impractical to provide the robots all possible operational information, which poses uncertainties to each robot's decision making process.

Human beings often take the navigation capability for granted as it is attached to our subconsciousness. One might not notice how large the amount of data a brain must collect and process in order to successfully perform a simple navigation task. Nevertheless, navigation is considered as a complicated task in robotics. Given a set of priori knowledge, such as a static map of an environment or coordinates of a target, a robot must know its current position relative to the map, then plan a collision-free path to the specified target in a way that any obstacles are avoided. These processes

are carried out by the aid of sensors to sense any nearby objects. Even though various types of sensor have been developed and refined for years, no such one is perfect. The presence of noise in the sensor adds more uncertainties to the carrying tasks.

Simultaneous arrival and task synchronisation strictly require information exchange among the team. Thus, a well-established and robust network is critical to provide the team with a reliable communication medium. However, maintaining such a network in a contested environment is complex and costly, especially when the components in the network (the robots) are free to move in different directions. Scenarios, such as one or more robots move out of the effective range of communication or networking infrastructures (e.g., router) are malfunctioned, can potentially subject the team to connection loss. Identifying a suitable networking architecture depends on different criteria, and would require extensive research.

Maintaining the network infrastructure leads to another interesting problem - the *single point of failure* problem. A system is considered to be exposed to a single point of failure if there introduce multiple components whose operations heavily depend on another single component. In such case, the failure of the single component would stop others from functioning. Wireless router is a typical example of a single point of failure. To address this problem, the system must be designed in a manner that all components are as less dependent on each other as possible.

1.4 CONTRIBUTION

In this thesis, we propose a new technique for simultaneous arrival by modifying the Dynamic Window Approach - a commonly-used motion planning technique for obstacle avoidance (Section 3.1). A ROS implementation for the proposed technique has been developed and deployed on physical robots. We also contribute a ROS-based distributed multi-robot coordination framework which provides the ability of task synchronisation and team member failure handling in a cooperative navigation mission (Section 3.2).

1.5 THESIS STRUCTURE

The remainder of this thesis is organised as follows:

Chapter 2 introduces background concepts relevant to this research. Many techniques related to the research, such as navigation techniques or networking methods, have also been summarised and compared in the literature review to select the most suitable ones out.

Chapter 3 covers the topic of cooperative navigation in a multi-robot system. In this chapter, we propose a new technique called SADWA to solve the problem of simultaneous arrival. We also propose and present in detail a distributed multi-robot

coordination framework called DMCF framework which handles task synchronisation as well as team member failure.

Chapter 4 presents in detail the architecture of the distributed multi-robot testbed developed to verify the proposed techniques in this research. Various aspects of the implementation such as testbed requirements, network architecture, software design and system integration are covered and discussed.

Chapter 5 evaluates and verifies the proposed techniques under different scenarios in both simulation and physical environment. The results are presented and discussed in detail. It also identifies and suggests future directions to several gaps in the proposed techniques.

Chapter 6 summarises the research for the purpose of answering the research question. This chapter also suggests and discusses several general research's future directions.

BACKGROUND

This chapter aims to provide the reader with the background knowledge to be used in the research. Many terminologies and frameworks will be introduced to familiarise the reader with the research context. It also provides a literature review where techniques in the research-related fields are explored and analysed to lay foundations for the methodologies proposed in later chapters.

2.1 DISTRIBUTED SYSTEM

A *distributed system* is a system which consists of computational entities (e.g., processors, computers) interacting with one another to achieve a common goal [1]. An entity can not directly communicate with another by sharing variables as each entity in a distributed system has its own private memory. Instead, *message-passing* through a communication network is used as an information sharing mechanism between entities. In a distributed architecture, the system is able to tolerate the failures of individual entity. Each entity in a distributed system does not have global information of the whole system; instead it achieves the system's goal by processing incomplete input of its own and from other entities.

A program distributed over the system entities and running to achieve a common goal is called a *distributed program*. A distributed program is implemented on top of a *distributed algorithm*. In a distributed system, all entities run the same distributed program and operate correctly regardless of the network's structure.

For many applications, it is possible to use a *centralised system* to achieve system's common goal. In the centralised system, a single entity - acts as a centralised-coordinator - processes all information from all other entities to decide common system's behaviours. However, in a practical point of view, such mechanism is subject to the *single point of failure* where the failure of the centralised-coordinator could stop all entities from operating. In a distributed system, as each entity operates on its own, there is no single point of failure existing; thus, it provides a more reliable and robust system than a centralised one. Figure 2.1 illustrates a graph representation of a centralised system and a distributed system.

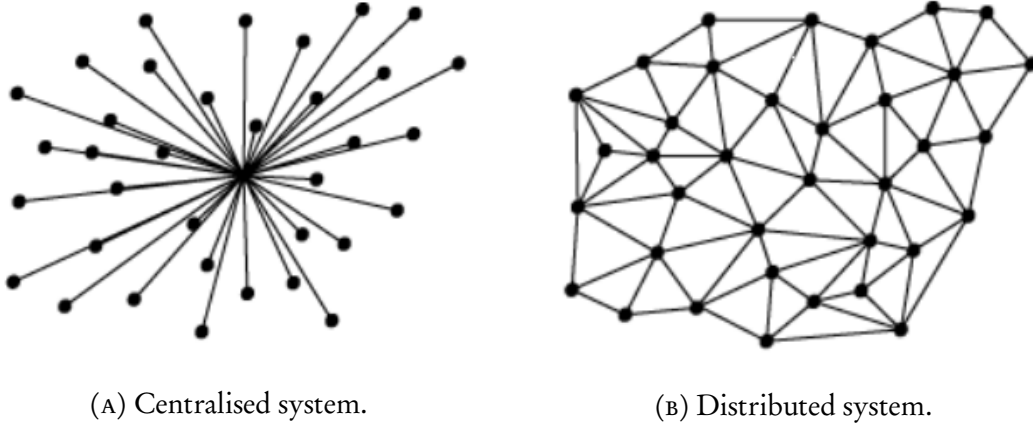


FIGURE 2.1: Connectivity of a centralised system and a distributed system.¹

Even though there is no single point of failure, distributed entities require the information from each other to decide its own local behaviours. In practice, the information is not always ensured to be arrived at the in-need entity. Lynch classified entity failure in a distributed system into *link failure* and *stop failure* [2]. Link failures occur when communication link(s) between entities are lost. Such communication link's loss results in losing messages to be passed among entities. *Stop failures* occur when one or more entities stop in the middle of its execution. The passing messages in such case, thus, are incomplete and likely to be incorrect. A distributed algorithm not only provides solutions for a distributed system, but also ensures in events of failure, all active entities can still operate and achieve the common system's goals.

2.2 AD HOC NETWORK

In a distributed system, entities may join or leave the system unpredictably. Setting up the infrastructure and administration to handle such dynamic nature of a distributed system can be costly and may not be reliable. To realise a distributed system, an *ad hoc network* is utilised. Formally, "an ad hoc network is a collection of wireless mobile node dynamically forming a temporary network without the use of any existing network infrastructure or centralised administration"[4]. Put simply, an ad hoc network allows nodes² to join the network without pre-existing infrastructure such as routers or access points. As the complexities of infrastructure set-up are removed, an ad hoc network enables nodes to create and join the network "on the fly".

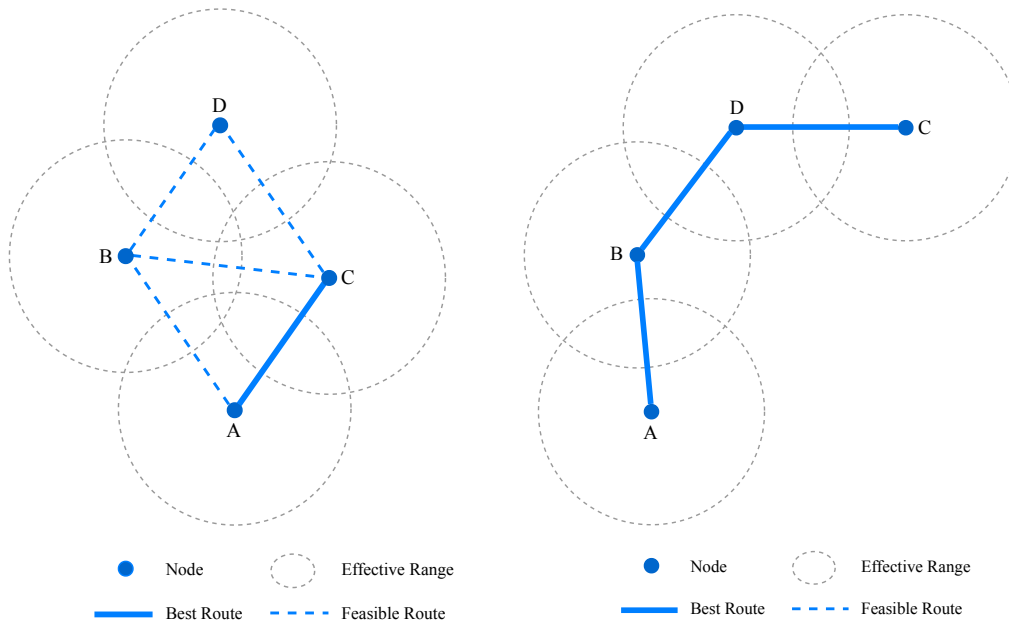
Sarkar [4] introduces how communications are managed in an ad hoc network. Communications in an ad hoc network are handled in a *multi-hop* fashion in which information is relayed among nodes. The message-relaying mechanism is implemented

¹Image adopted from [3]

²In this section, the term "node" will be used, instead of "entity", to fit with the networking terminology.

by a *routing algorithm* (or *routing protocol*). To direct a packet from a source to a destination with message-relaying, a routing algorithm running at each node identifies the next best node to forward the packet to. Topology is determined dynamically on the basis of the criteria defined in the routing algorithm such as distance to node or signal quality. Thus, an ad hoc network is a self-configured and self-healing network.

The message-relaying capability of an ad hoc network is critical for a distributed multi-robot system where one robot may move out of the effective communication range of another. Figure 2.2 illustrates the process of routing a communication link between Node A to Node C. Assume that Node A wants to transfer a packet to Node C. In Figure 2.2(a), Node A and Node C are in the effective communication range of each other, thus the packet can be transferred directly. Figure 2.2(b) illustrates the scenario in which Node A and Node C move out of each other's effective communication range. The topology, in this case, adjusts itself using a routing algorithm running at each node. The packet is forwarded through Node B and Node D, before arrives at the destination - Node C.



(A) Node A can forward information directly to Node C as they are in the effective communication range of each other. (B) As Node A and Node C move out of the effective range of each other, the ad hoc network's topology changes.

FIGURE 2.2: Dynamic topology under an ad hoc network.

Its obvious strengths, however, pose several challenges in the network implementation. As all nodes joining an ad hoc network may be mobile, may connect or disconnect at any time, a highly-dynamic network topology is required to cope with the restructure in a timely and reliable manner [4]. System scalability also introduces more time delays in internode communications.

2.3 ROBOT OPERATING SYSTEM³

Robot Operating System (ROS) is an open-source framework used for robotics' software development. Although being named as an Operating System (OS), ROS is not an actual OS. Instead, it serves as a platform built on top of a host operating system such as Linux distributions, Windows, OS X, etc. Such a characteristic allows ROS to not only inherit features of an operating system, but also present its own functionalities tailored for robotics. Similar to a traditional operating system, ROS offers "hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management". ROS aims to simplify the process of robotics' software development by providing a fundamental platform for robotics software developers to collaborate and inherit others' work.

One of the most powerful features of ROS lays in its ecosystem with various tools, functionalities and packages. Several applications can be listed such as navigation, simultaneous localisation and mapping, robotics vision, 3D visualisation, simulation tools, etc. Majority of the packages have been well-documented and supported by the ROS community. Another appeal is that ROS supports a wide range of programming languages. At the time of writing this thesis, the primary languages supported are C++, Python, and LISP; beside many experimentally-supported languages such as MATLAB, Java, Javascript, Go, etc.

To familiarise oneself with ROS, several fundamental ROS concepts are introduced in this section.

In the context of ROS, a *node* represents a single process running in the application. Typically, a ROS program consists of multiple nodes communicating with each other through *message-passing*. Communications take place in a peer-to-peer manner. A message - which contains information to be sent/received - is constructed by primitive data types (e.g., int, bool, or string, etc.) and/or array of data types. The communication from a node to another is managed by a *ROS master*. Each node registers its name with the ROS master so that the master can enable an individual node to locate another one. Once a node can locate another, a peer-to-peer communication can be established. For a networking analogy, the ROS master can be thought of as a Domain Name System (DNS) server that provides address-lookup for nodes in a ROS application. A node can *publish* (transfer) or *subscribe* (receive) message(s) through relevant *topics*. Topics, typically, are many-to-many connections that provide continuous data flow for a node. Figure 2.3 describes the process of topic communication managed by a ROS master. It should be noted that, for all ROS's model diagram

³This section uses the information from the ROS Wiki. All concepts described are available at <http://wiki.ros.org>.

in this section, the dashed arrows are only considered as a communication link only if both Node A and Node B have completed the registration process.

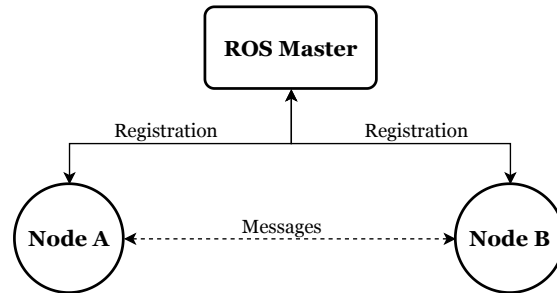


FIGURE 2.3: ROS topics communication.

Less formally, one can think of a ROS node as a house where each person lives in represents a ROS topic. Consider a person in the House A (publisher) wants to send a parcel to another one in the House B (subscriber). In this case, the parcel represents the ROS message to be sent. Each house must register with the local government - an analogy for the ROS master - to be assigned an address. Being aware of the House B's address, the person in the House A can directly drive to House B, and give the parcel to the receiver.

Although the publishing/subscribing paradigm offers a continuous streams of communication between topics, it does not provide the flexibility of implementing a request/reply interaction. Such interactions are often required by a distributed robotics system. *Service* addresses this problem by providing a client node the ability to send a request to a service server. The server - after processing the client's request - replies the results back to the client. The request/reply interactions are also managed by a ROS master. Figure 2.4 describes a request/response model of a ROS service.

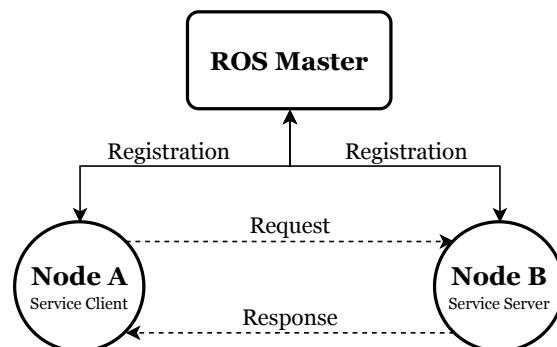


FIGURE 2.4: ROS service communication.

Services are often suitable for remote procedure calls that terminate quickly. However, in various complex tasks, the server may take more time, or worst, fail to process the request. Under such case, an *action* is the recommended implementation. For a time-consuming task, an action server allows the client to receive feedbacks as the

task progresses over time. Under the event of task's failure, the client can preempt the current task being processed by the action server, and request a new one. Figure 2.5 describes the functionalities of a ROS action. In ROS implementation, a client can send goal/preemption request to an action server, and receive the task status/feedback or the results (if the task can be completed). It is recommended that the action should be used for any discrete behaviours that move a robot or that run for a longer time but provide feedback during execution.

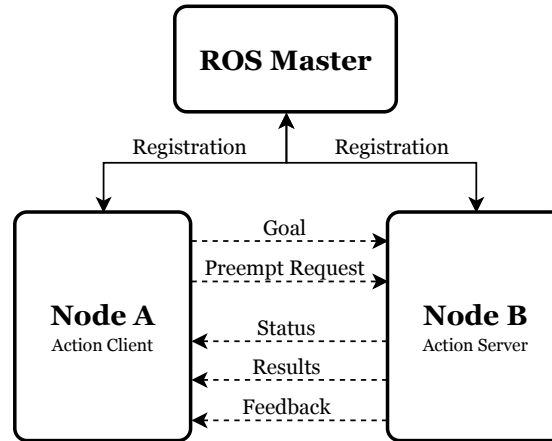


FIGURE 2.5: ROS action communication.

Typically, a ROS program contains parameters to be configured in run-time. Being able to configure parameters in run-time, instead of in compile-time, reduces the time of rebuilding the whole application. Such implementation can be realised by using a *parameter server*. A parameter server provides ROS nodes a shared dictionary to be accessed remotely at run-time. It is very useful in case a tool is required to inspect or modify the global configuration state of the system.

All these concepts are the main components of a *ROS package*. ROS packages are the most basic units and the lowest-level building block for a ROS application. Thematically-similar ROS packages are grouped into a *stack*. A stack ultimately is a collection of packages developed together and mutually interdependent executing the same behaviours.

One of the most powerful ROS debugging tools is ROS Visualization (RViz). RViz is a ROS 3D visualisation tool which provides robotics developers the ability to view the world through the robot's perspective. RViz can visualise 2D/3D point cloud, navigation paths, or robot models, etc. Figure 2.6 shows an RViz view of 5 robots in an environment. The red and green dots are the visualisation of each robot's sensor values.

Gazebo is a 3D simulator to simulate complex operational environment. Although Gazebo is not a component in the ROS ecosystem, it is widely used together with ROS for testing robotics algorithms in various scenarios without costly and complex setup.

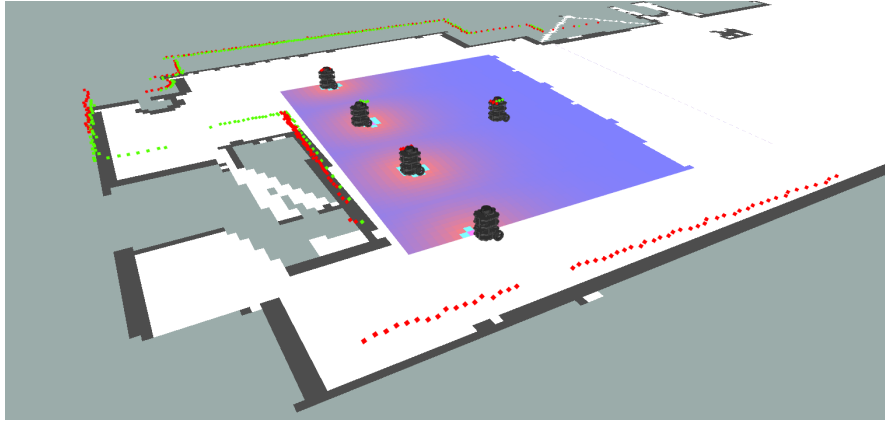


FIGURE 2.6: An RViz view in a navigation task.

Gazebo is able to accurately model sensors such as laser range finders, cameras, depth-cameras, etc., from which the robot can sense and react in the simulated environment. Figure 2.7 shows an environment built by Gazebo.

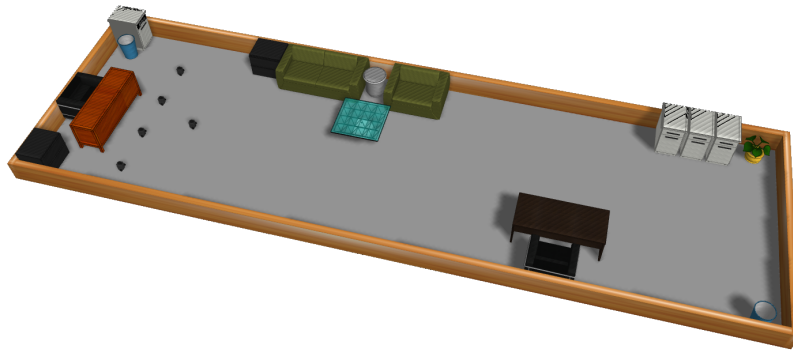


FIGURE 2.7: A simulated operating environment in Gazebo.

ROS plays a critical role in this research as being the main platform to develop the testbed.

2.4 TURTLEBOT3 BURGER

The main robot platform used in this research is the Turtlebot3 Burger. TurtleBot3 Burger is a small, inexpensive, ROS-based mobile robot used in education, research, hobby, and product prototyping. The goal of TurtleBot3 is to dramatically reduce the size of the platform and lower the price without any functionality and quality trade-off. The TurtleBot3's core technology is Simultaneous Localisation and Mapping (SLAM), navigation and manipulation, making it suitable for home service robots. Table 2.1 summarises all important specifications of the Turtlebot3 Burger.

Figure 2.8(a) shows the Turtlebot3 Burger dimensions. Such a small size along with two-wheel *differential-drive* offers the Turtlebot3 Burger the flexibility in navigation tasks. A robot with differential drive means that it has two wheels driven by

TABLE 2.1: Turtlebot3 Burger specifications.

Item	Specification
Maximum translational velocity	0.22 m/sec
Maximum rotational velocity	2.84 rad/sec (162.72 deg/sec)
Size	138 mm x 178 mm x 192 mm
Total Weight	1 kg
Expected operating time	2h 30m
Expected charging time	2h 30m
Single Board Computers	Raspberry Pi 3 Model B+
MCU	32-bit ARM Cortex [®] -M7 with FPU
Laser Distance Sensor	360 Laser Distance Sensor LDS-01
IMU	Gyroscope 3 Axis Accelerometer 3 Axis Magnetometer 3 Axis
Battery	Lithium polymer 11.1V 1800mAh/19.98Wh 5C

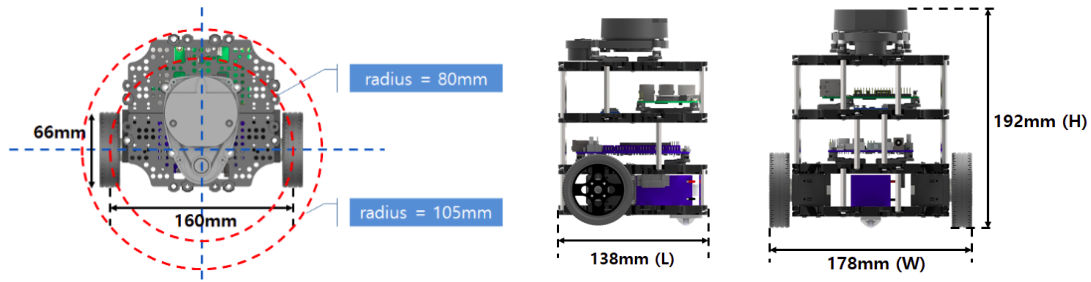
TABLE 2.2: 360 Laser Distance Sensor LDS-01 specifications.

Item	Specification
Distance Range	120 mm - 3500 mm
Distance Accuracy (120 mm - 499 mm)	± 15 mm
Distance Accuracy (500 mm - 3500 mm)	$\pm 5.0\%$
Distance Precision (120 mm - 499 mm)	± 10 mm
Distance Precision (500 mm - 3500 mm)	$\pm 3.5\%$
Scan Rate	300 ± 10 rpm
Angular Range	360°
Angular Resolution	1°

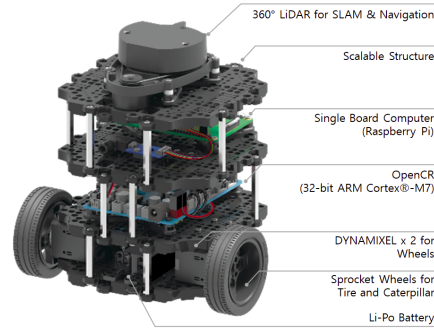
independent motors. The motors rotate around the centre point of the robot with one or more additional ground contact point for the purpose of stability [5].

All components of a Turtlebot3 Burger are shown in Figure 2.8(b). The Turtlebot3 Burger is equipped with a *Light Detection and Ranging* (LiDAR) for object recognition. LiDAR is a laser-based sensor for distance-to-object measurement. As being able to rotate 360° to scan the environment, the LiDAR on the Turtlebot3 Burger can generate 2D point cloud. Table 2.2 provides the specifications of the LiDAR equipped in the Turtlebot3 Burger. The LDS-01 LiDAR can measure the distance to an object of up to 3.5 meters. The minimum distance it can measure is 12 cm. As can be seen from the Table 2.2, the accuracy of the scan values depends on the actual distance to objects.

There exist several limitations introduced on the object recognition capability of the LiDAR equipped in the Turtlebot3 Burger [6]. As the LiDAR measures the return time of the laser source, it is useless if the laser source is scattered, or does not



(A) Dimensions of the Turtlebot3 Burger.



(B) Components of the Turtlebot3 Burger.

FIGURE 2.8: Turtlebot3 Burger.

TABLE 2.3: Raspberry Pi 3B+ specifications.

Item	Specification
Processor	Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz
RAM	1GB
Wireless LAN	2.4GHz and 5GHz IEEE 802.11.b/g/n/ac
Camera Port Supported	Yes
Power Input	5V/2.5A DC

bounce back. Transparent and mirror-like objects result in significantly inaccurate measurements. Another limitation lays in the robot's height. As the LiDAR - which is placed on the top of the Turtlebot3 Burger - can only generate 2D point cloud, the Turtlebot3 Burger can not recognise any objects lower than its height (19 cm).

The Turtlebot3 Burger is powered by the popular Raspberry Pi 3 Model B+. Table 2.3 summarises the specifications of the Raspberry Pi used in the Turtlebot3 Burger.

2.5 LITERATURE REVIEW

2.5.1 ROBOT LOCALISATION

Localisation is the process of estimating the position of a robot relative to a *reference* in an environment. Localisation is a critical process in a navigation task, as it keeps track

the robot's locations over time which will be input to navigation planning. Thrun, Burgard, and Fox [7] discussed two primary problems in localisation: (1) how to estimate the current pose when the initial pose is unknown - known as *global localisation*, and (2) how to perform position tracking when the initial pose is made available - known as *local tracking*. Once the global localisation is solved, the local tracking is just the matter of keeping track the pose estimated from the global localisation over time [7].

Kalman Filter was proposed by Kalman to estimate the state of a linear system [8]. To deal with state estimation in a non-linear system, Extended Kalman Filter (EKF) was proposed by Julier and Uhlmann [9]. Both Kalman Filter and EKF assumes the noise is Gaussian.

To deal with non-linear and non-Gaussian noise position estimation problem, Dellaert *et al.* [10] proposed Monte Carlo Localisation (MCL) - a localisation technique using particle filter applicable for both the global localisation and local tracking. MCL maintains a set of random-generated *samples* (or *particles*) drawn from a probability density function to describe and update uncertainties using Monte Carlo methods. MCL claimed to efficiently localise globally and track the positions of a robot for a long time at high-speed velocity by using a fixed number of samples (5000 samples) [10]. This fixed set of samples, even though claimed to be well-tracking, can be a waste for computational resource as the algorithm maintains the whole set during localisation. Fox presented a technique called KLD-sampling (Kullback-Leibler Distance-sampling) which adapts the number of particles needed for localisation using the principle of the Kullback-Leibler distance. In his paper, Fox claimed that the KLD-sampling approach uses only 6% of the samples required by the fixed-particles technique after global localisation is completed [11].

This project will solely use the state-of-the-art KLD-sampling Monte Carlo Localisation for robot localisation.

2.5.2 NAVIGATION PLANNING

Navigation planning is the process of identifying optimal paths and velocities from a starting position to a target. The current pose of the robot is updated over time by the localisation. A good planning strategy can optimise the robot operations in terms of energy, time, or computational resources, etc. Navigation planning can be classified into (1) *global planning* in which a pre-calculated path from a source to a destination is generated, and (2) *motion planning* in which the optimal velocity is selected to reach the destination - following the global path - in a collision-free manner.

Dijkstra's algorithm is a global planner presented originally by Dijkstra. It is able to plan a shortest path to the destination by minimising the cost function during the planning process [12]. Although Dijkstra algorithm always guarantees a shortest

path to the target, it exhaustively searches through all neighbour nodes to find the next best node. To improve the Dijkstra's searching time complexity, Hart, Nilsson, and Raphael introduced A* algorithm which searches heuristically to find the next best node [13]. As being heuristic, A* may not guarantee the shortest path is found. Instead, it provides a trade-off between path-searching time and path optimality. Dijkstra's algorithm is a special case of A* algorithm when no heuristic function is used.

As the robot may operate in a complex environment where dynamic obstacles may be present, a fixed optimal path to the goal will not be sufficient to cope with the unpredictable changes in the operating environment. Usually, a *grid map* generated by mapping algorithms⁴ provides the robot with only the static obstacles' information. The robot may not be able to handle the scenario in which unexpected obstacles appear. Moreover, only binary information - which indicates definite obstacle or definite free space - is represented in a grid map. This does not provide the global planning algorithm with a flexible path generation. Lu, Hershberger, and Smart proposed an approach called *layered costmaps*. Instead of using the grid map directly, a layered costmap keeps multiple ordered layers - each updates the cost related to a specific functionality - accumulated into the grid map [14]. One might use a layer for dynamic obstacle detection, another layer for caution zone indication where the robot tries not to navigate to, or another layer for buffering a zone around an obstacle. All, even though are separated, provide the global planning a common map to perform highly-flexible path planning.

For action planning, Fox, Burgard, and Thrun presented Dynamic Window Approach (DWA). DWA is a *sample based optimization* algorithm. DWA selects a translational velocity and a rotational velocity by sampling from a *velocity search space* such that an objective function is minimised. DWA maintains and calculates the objective function regularly so that the robot is able to head to a destination without any collisions [15].

Another approach for obstacle avoidance is the Time Elastic Band (TEB) presented by Rsmann, Hoffmann, and Bertram. TEB seeks for the time-optimal solution. The approach discretises the predicted trajectory in terms of time and applies a *continuous numerical optimization scheme* [16]. Due to the continuous optimisation, depending on the discretisation resolution, TEB can generate flexible paths. In the paper, TEB transforms a constrained optimisation problem into an unconstrained optimisation problem, which implies that the constraints pose on a robot such as obstacle avoidance, velocity bound, etc. can not be guaranteed [16].

This project will use A* algorithm, along with layered costmaps, as the global path planner. This combination enables the robot the ability to avoid both static

⁴This project will not focus on the mapping techniques. Interested readers can find a more thorough information under Part III of the book Probabilistic Robotics [7].

and dynamic obstacles. Motions will primarily be planned using DWA. All these techniques are provided in the *ROS Navigation Stack* - which will be introduced in Subsection 4.3.1.

2.5.3 SIMULTANEOUS ARRIVAL

Simultaneous arrival (or *rendezvous*) is an application of cooperative timing mission which refers to the capability of the robots in a multi-robot system to reach pre-specified targets simultaneously [17]. Simultaneous arrival can be applied in many civil and military application, such as surveillance, rescue operations, time-synchronised attack, package delivery, etc.

The idea behind simultaneous arrival is to have all robots in the network reached a consensus on the time of arrival [18]. As such, communications among the robots are required. There are several approaches proposed to obtain simultaneous arrival under the assumption of network connectivity preservation such as [19], [20]. Kan *et al.* proposed a leader-follower approach where only the leader is aware of the arrival time, and the followers try to achieve the consensus with the leader [21]. Li and Shi proposed an approach for rendezvousing at different targets and avoiding obstacles [17]. However, the work introduced in [17] did not take into account dynamic obstacles.

2.5.4 MULTI-ROBOT SYSTEM

A *multi-robot system* is a system consists of multiple robots coordinated with each other to achieve a common goal. Each robot communicates with another through a communication network. By coordination, a multi-robot system can solve complex tasks in an efficient manner. Several typical approaches to robot coordination was summarised by Dias *et al.* [22] and are introduced in this subsection.

In *centralised approach*, a single robot acts as a centralised-coordinator that gathers information and plans coordinated tasks for the entire team. Even though this approach eases the implementation, it easily suffers from bandwidth overflow due to all information goes through a single centralised-coordinator, or being subject to a single point of failure. This approach best suits with simple coordination where a small number of robots are required to collaborate in a static environment.

Distributed approach provides a more robust and fast solution as each robot operates and processes local knowledge on its own. In a distributed approach, no single point of failure exists. It enables a multi-robot system to be reliable and robust in the face of member failures. However, suboptimal task plans are often produced due to

the lack of global information exchange incorporation. Distributed approach is applicable for the coordination between a large number of robots carried simple tasks which do not require efficiency.

Market-based approach is a hybrid approach that incorporates both centralised and distributed approach. It presents the concept of *auction* which collects data and plans distributed resources in a single robot. Then, *market mechanisms* is used to distribute resources and tasks over the team. The market-based approach takes advantages of the centralised approach where global information is collected and planned, yet remains the distributed nature of the team when carrying out a coordinated tasks. The biggest challenge of the market-based approach lays in its complicated implementation.

This research employs the distributed approach for robot coordination. The distributed approach is easier to implement than a market-based approach, yet still provides a more robust and reliable coordination mechanism than the centralised one.

2.5.5 WIRELESS NETWORK CONNECTIVITY

In a multi-robot system, it is required to provide a communication medium for the members to communicate with each other. As such, wireless network connectivity between team members is critical for a coordinated task in which information exchanges among robots are strictly required. Kurose and Ross introduced several types of wireless networks [23]. Networks feasible to be deployed in this research are summarised in this subsection.

The widely-used and popular Internet is categorised as a *single-hop and infrastructure-based* network where nodes communicate with others through a single wireless hop. Base stations are required to connect the nodes to the Internet. *Multi-hop, infrastructure-based* is a type of network where a centralised station wired to a larger network is required. Nodes in the network, however, still relay communications among themselves to reach the centralised station. An example of a multi-hop and infrastructure-based network is the wireless *mesh network*. Mobile nodes with connectivities changing unpredictably are usually handled by a *multi-hop, infrastructure-less* network. In this type of network, mobile nodes self-form a network without any base stations. Communications between nodes take place by information relaying among themselves. The ad hoc network presented in Section 2.2 falls into this category.

Due to the mobile and distributed nature present in a multi-robot system, the multi-hop and infrastructure-less network, specifically the ad hoc network, will be deployed in this project.

In an ad hoc network, it is important to select a suitable routing protocol. The two most popular routing protocols in the open source world are *Optimised Link State Routing* (OLSR) [24] and *Better Approach To Mobile Ad hoc Networking* (B.A.T.M.A.N) [25]. Murray, Dixon, and Koziniec suggested that the performance of OLSR and

BACKGROUND

B.A.T.M.A.N is similar [26]. In this research, BATMAN-adv - an implementation of B.A.T.M.A.N routing protocol - is selected as it has been a part of the official Linux kernel, which eases the process of protocol installation in the testbed.

COOPERATIVE NAVIGATION FOR MULTI-ROBOT SYSTEM

This chapter proposes techniques to obtain multi-robot coordination in navigation tasks. It firstly introduces a new technique for simultaneous arrival using Dynamic Window Approach as the obstacle avoidance mechanism. A systematised framework to distributed multi-robot coordination - including task synchronisation and member failure handling - is also proposed and discussed thoroughly.

3.1 SIMULTANEOUS ARRIVAL USING DYNAMIC WINDOW APPROACH

In a complex environment under the presence of dynamic obstacles, simultaneous arrival tasks are required to reliably perform object avoidance to cope with the uncertainties in the operating environment. All members must response quickly to the unanticipated changes in the environment, yet remain cooperation with others to guarantee a synchronised time of arrival.

Dynamic Window Approach (DWA) - in combined with layered costmaps - is a commonly-used and well-documented algorithm as a motion planner for ROS navigation tasks. To the best of our knowledge up to the thesis submission time, DWA - despite its reputation - has not been well-applied to the field of simultaneous arrival. In this section, a modified DWA is proposed to incorporate simultaneous arrival as a criterion in navigation tasks. The proposed technique is called *Simultaneous Arrival using Dynamic Window Approach* (SADWA).

This section firstly introduces the principle of the Dynamic Window Approach. The problem of simultaneous arrival is then formulated and incorporated to the DWA. The rest of this section discusses the distributed algorithm to obtain simultaneous arrival, and to handle events of member failure.

3.1.1 DYNAMIC WINDOW APPROACH

As proven by Fox, Burgard, and Thrun [15], a trajectory can be approximated by a sequence of circular arcs (also referred to as *curvatures*). Those curvatures can be

modelled by a set of feasible translational and rotational velocities $V_s = (v, \omega)$. The pair of translational and rotational velocities, in the paper, is simply referred as a velocity. Put simply, the set V_s contains all possible velocities a robot can generate. This results in an initial *velocity search space* V_s . Dynamic Window Approach, based on several criteria, reduces the space of this initial velocity search space.

The ultimate goal of the Dynamic Window Approach is to optimally select a velocity in the robot's constrained velocity search space, such that the robot can move to the destination in a collision-free manner.

Let V_a be a set of *admissible velocities*. The set of admissible velocities contains feasible velocities (v, ω) which allows the robot to stop before the nearest collisions. Let $\dot{v}_b$ and $\dot{\omega}_b$ be the acceleration for breakage. The set of admissible velocities V_a is defined as

$$V_a = \{(v, \omega) \mid v \leq \sqrt{2 \text{dist}(v, \omega) \dot{v}_b} \wedge \omega \leq \sqrt{2 \text{dist}(v, \omega) \dot{\omega}_b}\}, \quad (3.1)$$

where $\text{dist}(v, \omega)$ is the distance to the closest obstacle with the current pair of velocities (v, ω) . The admissible velocity constraints ensure only collision-free trajectories are generated.

Let V_d be the *dynamic window*. A dynamic window is a set of restricted admissible velocities that can only be reached within the next time interval, given the robot acceleration. Let t be the time interval that the accelerations $\dot{v}$ and $\dot{\omega}$ are applied within. At the actual velocity (v, ω) , the dynamic window V_d is defined as

$$V_d = \{(v, \omega) \mid v \in [v_a - \dot{v} t, v_a + \dot{v} t] \wedge \omega \in [\omega_a - \dot{\omega} t, \omega_a + \dot{\omega} t]\}, \quad (3.2)$$

As the dynamic window represents the velocities that can be reached within the next time interval, all curvatures outside the dynamic window are not considered for obstacle avoidance.

The above criteria pose constraints on the velocity search space. The resulting velocity search space V_r within a dynamic window is the intersection of (a) initial velocity search space V_s , (b) admissible velocity V_a , and (c) dynamic window V_d described as

$$V_r = V_s \cap V_a \cap V_d. \quad (3.3)$$

An optimal velocity (v, ω) is selected from the set V_r such that the objective function

$$G(v, \omega) = \sigma(\alpha \text{ heading}(v, \omega) + \beta \text{ dist}(v, \omega) + \gamma \text{ velocity}(v, \omega)), \quad (3.4)$$

where α is the weight for the attempt of staying aligned with the target position,
 β is the weight for the attempt of keeping a clearance distance to the nearest obstacles,
 γ is the weigh for fast movements,
 σ is the normalisation factor,

is minimised.¹ The optimal velocity at which the objective function is minimised can be identified by calculating the objective function over V_r . The calculation is done by discretisation the velocity search space V_r .

The function $\text{heading}(v, \omega)$ calculates the alignment of the robot to a target direction. The function $\text{dist}(v, \omega)$ measures the distance to the closest obstacles. The function $\text{vel}(v, \omega)$ represents the robot's forward velocity. These three components are equally-critical criteria that seek to obtain the heading closest to the goal, maximise the robot clearance to obstacles, and maximise the velocity. Fox, Burgard, and Thrun [15] noticed that, by maximising only the $\text{dist}(v, \omega)$ and the $\text{velocity}(v, \omega)$, the robot would always move to the free space, but will not navigate towards the goal. By maximising only the $\text{heading}(v, \omega)$, the robot will be unable to avoid obstacles. One should tune the weights such that the objective function is minimised in a way that fits with the desired robot's behaviours.

3.1.2 PROBLEM FORMULATION

Under the assumption that a team of N robots is operating in a complex environment where obstacles may be present, each agent must make decisions to not only cope with its current state x_t , but also coordinate with others to satisfy the team objectives.

Let $\mathcal{U}_i(x_t)$ be the decision vector of the robot $i \in N$ at state x_t . A decision vector contains a set of feasible decisions from which a decision variable $u_t \in \mathcal{U}_i(x_t)$ is selected to optimally react to a state x_t . In the context of simultaneous arrival that takes into account object avoidance capability, the decision vector contains a set of feasible translational velocities v_t , a set of feasible rotational velocities ω_t and a set of collision-free paths P_t to targets. Thus, a decision variable has the form of

$$u_t = \begin{bmatrix} v_t \\ \omega_t \\ P_t \end{bmatrix}, \quad (3.5)$$

¹In the paper [15], the velocity is selected such that the objective function is *maximised*. However, the ROS implementation of DWA indicates the contrary - the objective function is minimised at the chosen velocity. As maximisation or minimisation depends on how the heading, dist, and velocity are implemented, and SADWA relies heavily on the DWA's ROS implementation, SADWA chooses minimisation to ease the process of deriving the simultaneous arrival's objective function later.

where $v_t, \omega_t \in V_r$ is a feasible translational and rotational velocity in the resulting velocity search space defined in (3.3),
 $P_t = \{p_k : k = 1 \dots M\}$ is a set of M points in 2D space forming a path to target.

The basic idea behind a simultaneous arrival navigation task is to have all robots come to an agreement on the *estimated time of arrival* (ETA). This value of time agreement, in this section, will be referred to as the *consensus time*. A team of N robots is stated to simultaneously arrive at the targets if

$$T_i = T_j = \psi, \quad \forall i, j \in N, \quad (3.6)$$

where T_i, T_j are the ETA of the i -th and j -th robot respectively,
 ψ is the consensus time.

One of the challenges to achieve simultaneous arrival is to choose a unique consensus time ψ which all robots agree on at every time instance t . Each robot will then adjust its velocity to fit with the consensus time.

Let $J_i(u_t, \psi)$ be the team objective function of a robot i . The objective function defines the capability of a robot to avoid obstacles and arrive at the target simultaneously with other robots in the team. To achieve such objectives, the robot i must choose a decision u_t in response to the consensus time ψ such that the cost of the team objective function $J_i(u_t, \psi)$ is minimised. The problem of simultaneous arrival in this work is formalised as

$$\arg \min_{u_t \in \mathcal{U}_i} J_i(u_t, \psi), \quad \forall i \in N. \quad (3.7)$$

In SADWA, several assumptions are made to reduce the problem complexity. Firstly, SADWA in this research does not take into account the rotational velocities during navigation. This means all robots in the team may not arrive simultaneously at their specified orientations. Secondly, to reduce the dimensional complexity of the optimisation problem, the collision-free path P_t in response to a state x_t is assumed to be selected and updated over time from a global path planning algorithm such as Dijkstra or A*. With the two assumptions made, SADWA is only required to select an optimal translational velocity to guarantee simultaneous arrival.

3.1.3 COST FUNCTION FOR SIMULTANEOUS ARRIVAL

From (3.5), at each decision u_t , the ETA of a robot j is calculated as

$$T_j(u_t) = \frac{L(P_t)}{v_t}. \quad (3.8)$$

where $L(P_t) = \sum_{n=0}^{M-2} \|p_{n-1} - p_n\|$ is the length of the path P_t .

The decision u_t to be made at each state x_t will affect the robot's ETA on targets. To guarantee simultaneous arrival is always achieved, at every time instance t , it is safe to choose the largest ETA among all robots as the consensus time. This value of consensus time will be updated over time. The consensus time is formally defined as

$$\psi = \max_{j \in N} T_j(u_t). \quad (3.9)$$

Having the consensus time chosen, at every state x_t , each robot must choose a decision such that its own ETA is as close to the consensus time ψ as possible. In other words, a robot i must select a decision u_t such that its cost function

$$F(u_t, \psi) = |T_i(u_t) - \psi|, \quad (3.10)$$

where $T_i(u_t)$ is the ETA of the i -th robot as defined in (3.8),
 ψ is the consensus time chosen in (3.9),

is minimised. This cost function can be incorporated to the DWA's objective function in (3.4) as another criterion during navigation. Thus, the objective function $J(u_t, \psi)$ in (3.7) is defined as

$$J(u_t, \psi) = \sigma(\alpha \text{ heading}(v_t, \omega) + \beta \text{ dist}(v_t, \omega) + \gamma \text{ velocity}(v_t, \omega) + \delta (T_i(u_t) - \psi)) \quad (3.11)$$

where δ is the weight for the attempt of arriving simultaneously with other robots. The larger value of δ gets, the more the robot attempts to achieve consensus with others. As obstacle avoidance remains critical in a navigation task, it is recommended that the weight α , β , and γ should only be changed slightly from the original DWA.

3.1.4 DISTRIBUTED MAX-CONSENSUS ALGORITHM

This section introduces a distributed algorithm to achieve a unique consensus time with other robots at every instance time t .

Let $\mathcal{R}(i)$ be a set of all neighbours of a robot i . Let D denote an *associated array* (also known as *map*, or *dictionary*). An associated array, in computer science, is an abstract data type that contains collections of (key, value) pairs. In an associated array, each key only appears at most once. When inserting a pair of (key, value) to an associated array, if the key exists, its corresponding value will be updated; in case the key does not exist, the pair will be inserted to the associated array as a new entry. An associated array allows each element to have multiple values, but only one key. Inserting a pair of key-value (x_i, y_i) to an associated array D is represented as $D.\text{insert}(x_i, y_i)$. To remove a key x_i and its corresponding value, $D.\text{remove}(x_i)$ is

used. In the scope of this section, $D(k)$ refers to a set of keys, whilst $D(v)$ refers to a set of values in the associated array D .

Algorithm 1 presents the pseudo-code for a distributed algorithm running in an arbitrary robot i to achieve the consensus time ψ . The input to the algorithm is a decision u_t which the robot i selects at time t . The output is a unique consensus time ψ at each time instance t , or in the context of distributed algorithm, at each *communication round*. Each robot requires a set of its neighbours $\mathcal{R}$ to ensure the algorithm be functioned.

Each robot constantly calculates its own ETA and sends to its neighbours (together with its ID). Upon the event of information reception at a robot i , data from its neighbours $\mathcal{R}(i)$ are inserted to the associated array D . The consensus time ψ is identified as the maximum ETA from the associated array D . As Algorithm 1 is a distributed algorithm, the same process applies for other robots. In every communication round, as each robot has a complete information on the ETAs, a unique consensus time will always be guaranteed.

Algorithm 1 Distributed Max-Consensus Algorithm

Require: $\mathcal{R}(i)$

Ensure: $D \leftarrow \emptyset$

- 1: **upon event** received data from $j \in \mathcal{R}(i)$ **do**
 - 2: $D.insert(id_j, T_j(u_t))$
 - 3: **procedure** MAX_CONSENSUS(u_t)
 - 4: $T_i(u_t) \leftarrow \frac{L(P_t)}{v_t}$
 - 5: send $[id_i, T_i(u_t)]$ to all neighbours $j \in \mathcal{R}(i)$
 - 6: $\psi \leftarrow \max\{D(v), T_i(u_t)\}$
 - 7: **return** ψ
-

3.1.5 EVENT OF MEMBER FAILURE

Even though a unique consensus time is guaranteed, the event of member failure is not handled in Algorithm 1. During a cooperative mission, assume that link failure occurs at an arbitrary robot i . This results in a sudden information loss in the team. Even though the information is lost completely, the associated arrays D in other active robots still keep the latest pair (key, value) of the robot j from the last communication round before it failed. This redundant value results in incorrect consensus time among the robots.

This problem can be solved by adding an *expiry removal* mechanism. Expiry removal refers to a mechanism to remove expired data out of an associated array. An entry of data is considered to be expired if no new information is received after an amount of pre-specified timeout. Algorithm 2 describes the implementation of an

Algorithm 2 Distributed Max-Consensus Algorithm - Member Failure Handling**Require:** $\mathcal{R}(i)$ **Ensure:** $D \leftarrow \emptyset$

```

1: upon event received data from  $j \in \mathcal{R}(i)$  do
2:    $D.insert(id_j, T_j(u_t), time\_stamp_j)$ 
3: procedure MAX_CONSENSUS( $u_t, timeout$ )
4:    $T_i(u_t) \leftarrow \frac{L(P_i)}{v_t}$ 
5:    $time\_stamp_i \leftarrow NOW()$ 
6:   send  $[id_i, T_i(u_t), time\_stamp_i]$  to all node  $j \in \mathcal{R}(i)$ 
7:   for each (id, eta, time_stamp) in D do
8:     duration  $\leftarrow NOW() - time\_stamp$ 
9:     if duration  $\geq$  timeout then
10:      D.remove(id)
11:    $\psi \leftarrow \max\{D(v_1), T_i(u_t)\}$ 
12:   return  $\psi$ 

```

expiry removal mechanism. The max-consensus algorithm's principle remains unchanged. In the expiry removal implementation, a time stamp - at which the ETA is sent - is added and sent to the neighbour set in Line 6. The $NOW()$ function returns the current system timestamp at which the statement is executed. Each robot keeps checking the duration of an entry existed in the associated array D. This checking process occurs in Line 8 and Line 9. If the duration of an entry exceeds an amount of timeout, such robot is considered as expired. The whole entry of the expired (or failed) robot will be removed. The consensus time is updated at each robot without the failed one. The same procedure applies for the event of stop failure.

The mechanism of expiry removal introduced in this subsection is one simple approach to guarantee team resilience. It is a predecessor for a more systematic coordination framework which will be presented in the next section.

3.2 DISTRIBUTED MULTI-ROBOT COORDINATION FRAMEWORK

In this section, a framework named *Distributed Multi-Robot Coordination* (DMRC) is proposed to provide a systematic framework for coordination tasks. It primarily addresses two functionalities in a coordination task: *task synchronisation* and *member failure handling*. This section firstly describes the implementation of task synchronisation, and later, discusses member failure handling to ensure the assigned missions are achieved despite events of member failure.

3.2.1 TASK SYNCHRONISATION

Task synchronisation, in the context of this research, refers to the capability of multiple robots in a team executing tasks concurrently. In other words, task synchronisation ensures the whole team starts a new task at the same time. Task synchronisation is critical when a sequence of tasks are required and a new task can not be executed without the operations of multiple robots.

Algorithm 3 Task Synchronisation

Require: $\mathcal{R}(i)$
Ensure: $D \leftarrow \emptyset$

```

1: function IS_ACK( $\mathcal{R}(i)$ )
2:   for each (id, status, ack) in D do
3:     if status = false or D.size  $\neq |\mathcal{R}(i)|$  then
4:       return false
5:   return true
6: function IS_NEIGHBOUR_READY( $\mathcal{R}(i)$ )
7:   for each (id, status, ack) in D do
8:     if status = false or ack = false or D.size  $\neq |\mathcal{R}(i)|$  then
9:       return false
10:  return true
11: upon event received data from  $j \in \mathcal{R}(i)$  do
12:   D.insert(idj, statusj, ackj)
13: procedure TASK_SYNC( $\mathcal{R}(i)$ )
14:   team_status  $\leftarrow$  false
15:   statusi  $\leftarrow$  false
16:   acki  $\leftarrow$  false
17:   if IS_ACK( $\mathcal{R}(i)$ ) then
18:     acki  $\leftarrow$  true
19:   send [idi, statusi, acki] to all node  $j \in \mathcal{R}(i)$ 
20:   execute the current task (non-blocking)
21:   if current task of node  $i$  is done then
22:     statusi  $\leftarrow$  true
23:     if IS_NEIGHBOUR_READY( $\mathcal{R}(i)$ ) then
24:       send [idi, statusi, acki] to all node  $j \in \mathcal{R}(i)$ 
25:     team_status  $\leftarrow$  true
26:   if team_status = true then
27:     execute the new task

```

Let $\mathcal{R}(i)$ be a set of all neighbours of a robot i . The number of elements (*cardinality*) in the set $\mathcal{R}(i)$ is denoted as $|\mathcal{R}(i)|$. Let D denote an associated array to store the information from other robots. Algorithm 3 shows the pseudo-code for a distributed task synchronisation algorithm to be run in a robot i . The main function is the procedure TASK_SYNC. The input to the algorithm is the set of all neighbours of a robot i . The output is a synchronised value of team_status which indicates the

team readiness to execute a new task. In other words, if the `team_status` is `true` in all robots at the same time, the new task can be guaranteed to be synchronised. The algorithm proposed in this framework does not guarantee the first task in the task sequence to be synchronised, as the algorithm strictly requires the current task's status to identify the team readiness. This limitation can be worked around by assigning a short-term and non-trivial task as the first task for the team.

In a distributed algorithm, from time to time, each robot sends its information to each of its neighbour. The information to be sent - in task synchronisation - includes a robot ID (`id`), a current task status (`status`), and an *acknowledgement* (`ack`). The current task status can be either `true` or `false` to indicate completed, or uncompleted task respectively. The acknowledgement is `true` if a robot has been sufficiently received information from its neighbours; and is `false` otherwise. The robot ID acts as a key, whilst the current task status and the acknowledgement act as the values, in the associated array `D`. Thus, in task synchronisation, the associated array `D` has the key-value pair under the form of (`key`, `value1`, `value2`).

Before going through the details of the algorithm, several assumptions are made to define the concepts of *information acknowledgement* and *neighbour readiness*. A robot is considered as sufficiently received information if (a) all task statuses received from its neighbours are `true` (all tasks completed), and (b) the number of entries in its associated array `D` equals to the cardinality of its neighbour set ($D.size = |\mathcal{R}|$).² This assumption is made due to the fact that, in an associated array, each key only appears at most once; thus, this ensures one robot ID is associated with only one entry. A robot i can acknowledge that it has received sufficient information by setting `acki` to `true`. Information acknowledgement is implemented under the function named `IS_ACK` in the Algorithm 3.

At an arbitrary robot, its neighbours are considered to be ready if (a) its neighbours completed their current tasks (`status` $\leftarrow$ `true`), (b) sufficient information are acknowledged to be received (`ack` $\leftarrow$ `true`), and (c) the number of entries in its associated array `D` equals to the cardinality of its neighbour set ($D.size = |\mathcal{R}|$). Neighbour readiness is implemented under the function named `IS_NEIGHBOUR_READY`. At every communication round, the function iterates through the associated array; if one condition does not hold, it returns `false` immediately. Once all entries in the associated array are iterated with all conditions hold, all neighbours are considered to be ready, the function returns `true`.

²In theory, acknowledgement of sufficiently-received information only requires the condition (b) to be held. However, practically, as the information is transmitted under various disturbances, condition (b) may be false positive. An example can be communication loss in the middle of a task which causes incorrect values in the associated array. In such case, even though condition (b) holds, information are not received sufficiently. Thus, it is safer to acknowledge the information only when its neighbours complete their tasks.

As the main function is the procedure `TASK_SYNC`, the algorithm explanation starts from Line 13. Each robot constantly checks for information acknowledgement and sends its information to its neighbours as in Line 19. After sending the information, an arbitrary task - which, from this sentence on, will be referred to as the current task - is executed.³ It should be noted that the current executing task must be a non-blocking process. This allows the information transmission to take place concurrently with the executing task.⁴ Upon the event of information reception at a robot i , data from its neighbours $\mathcal{R}(i)$ are inserted to the associated array D . Once the current task executing at the robot i is completed, its own task status is set to true. If its own current task together with its neighbours' current tasks are all completed, the team is considered as being ready to execute a new task (`team_status` $\leftarrow$ true). The new task now is synchronised across the robots.

3.2.2 MEMBER FAILURE HANDLING

Team resilience is one of the most critical requirements in a cooperative mission. Although failures sometimes may not be inevitable, it is important that the team can recover and still achieve the mission. Consider a team of robots running Algorithm 3. During a cooperative mission, assume that one robot fails before it completes the assigned task. Under this scenario, the last data that the failed robot sent were `status` $\leftarrow$ false and `ack` $\leftarrow$ false, which prevents the function `IS_NEIGHBOUR_READY` from returning true; thus, new task will never be executed. This subsection introduces another algorithm for task synchronisation, but with member failure handling in the design, as shown in Algorithm 4.

The principle of task synchronisation with member failure handling remains unchanged from the normal task synchronisation; but with the addition of an expiry removal mechanism. The input of the Algorithm 4 is a set of neighbours of a robot i , and a value of `timeout` which later will be used for expiry removal. The output is a synchronised value of `team_status` as discussed in Subsection 3.2.1.

The main function in Algorithm 4 is the procedure `TASK_SYNC` in Line 20. At each robot, beside a robot ID, a current task status, and an acknowledgement, a timestamp - at which the information is packed - is also sent to its neighbours. Each robot uses this timestamp to identify the expired information being present in an associated array. Any entry exists in an associated array without any new information being updated - or in other words, the value of `timestamp` is unchanged - longer than a specified amount of `timeout` will be removed. This expiry removal mechanism is implemented

³As being discussed, the first task in the sequence will not be synchronised.

⁴Such capability can be achieved by the use of *scheduling algorithms*. Interested reader can find more information at [27].

Algorithm 4 Task Synchronisation - Member Failure Handling

Require: $\mathcal{R}(i)$ **Ensure:** $D \leftarrow \emptyset$

```

1: function IS_ACK( $\mathcal{R}(i)$ )
2:   for each (id, status, ack, time_stamp) in D do
3:     if status = false or D.size  $\neq |\mathcal{R}(i)|$  then
4:       return false
5:   return true
6: function IS_NEIGHBOUR_READY( $\mathcal{R}(i)$ , timeout)
7:   EXPIRY_REMOVAL( $\mathcal{R}(i)$ , timeout)
8:   for each (id, status, ack, time_stamp) in D do
9:     if status = false or ack = false or D.size  $\neq |\mathcal{R}(i)|$  then
10:      return false
11:   return true
12: function EXPIRY_REMOVAL( $\mathcal{R}(i)$ , timeout)
13:   for each (id, status, ack, time_stamp) in D do
14:     duration  $\leftarrow$  NOW() — time_stamp
15:     if (duration  $\geq$  timeout) and
16:       (status = false or ack = false or D.size  $\neq |\mathcal{R}(i)|$ ) then
17:       D.remove(id)
18: upon event received data from  $j \in \mathcal{R}(i)$  do
19:   D.insert(idj, statusj, ackj, time_stampj)
20: procedure TASK_SYNC( $\mathcal{R}(i)$ , timeout)
21:   team_status  $\leftarrow$  false
22:   statusi  $\leftarrow$  false
23:   acki  $\leftarrow$  false
24:   time_stampi  $\leftarrow$  NOW()
25:   if IS_ACK( $\mathcal{R}(i)$ ) then
26:     acki  $\leftarrow$  true
27:   send [idi, statusi, acki, time_stampi] to all node  $j \in \mathcal{R}(i)$ 
28:   execute current task (non-blocking)
29:   update  $\mathcal{R}(i)$ 
30:   if current task of node  $i$  is done then
31:     statusi  $\leftarrow$  true
32:     if IS_NEIGHBOUR_READY( $\mathcal{R}(i)$ , timeout) then
33:       send [idi, statusi, acki, time_stampi] to all node  $j \in \mathcal{R}(i)$ 
34:       team_status  $\leftarrow$  true
35:   if team_status = true then
36:     execute the new task

```

in function EXPIRY_REMOVAL from Line 12 to Line 17. At every communication round, the neighbour set $\mathcal{R}$ will be updated to check for events of failure.

DISTRIBUTED MULTI-ROBOT TESTBED

As having the methodologies developed, a distributed multi-robot system testbed is required to verify the operations of the proposed methodologies. We designed a testbed consists of 3 Turtlebot3 Burgers and is developed using ROS as the primary software platform. This chapter presents in detail the architecture of the distributed multi-robot testbed. Various aspects of the implementation such as testbed requirements, network architecture, software design and integration are covered and discussed.

4.1 TESTBED REQUIREMENTS

This section describes the required operations of the testbed to provide the basis for any functionalities to be implemented.

Due to the distributed nature of the developing testbed where new robots may join the team at any time, the testbed must be designed such that it is as less dependent on the external infrastructure as possible. As being desired to be present no single point of failure, each robot in the testbed must be able to operate on its own without any centralised coordinator. The robot communications must be managed by a reliable and robust network. Adding or removing robots out of the team should require no to minor configurations.

The major functionality to be verified is the cooperative navigation where the team of robots autonomously navigates and arrives at the pre-specified targets simultaneously. The team must be able to avoid any static, moving and unforeseeable obstacles. In a navigation task, it is assumed that each robot is provided with a map of environment as a priori. All new tasks must be synchronised. In other words, all robots in the team must start a new task at the same time.

It should be noted that the most critical testbed requirement is being resilient. In other words, no matter what event(s) - that may be subject the robot(s) to failure - occur, the active robots must still complete the mission without the failed ones. In case failure occurs, the mission at the failed robot should be aborted either manually or autonomously, which makes it stop and cancel all tasks executing.

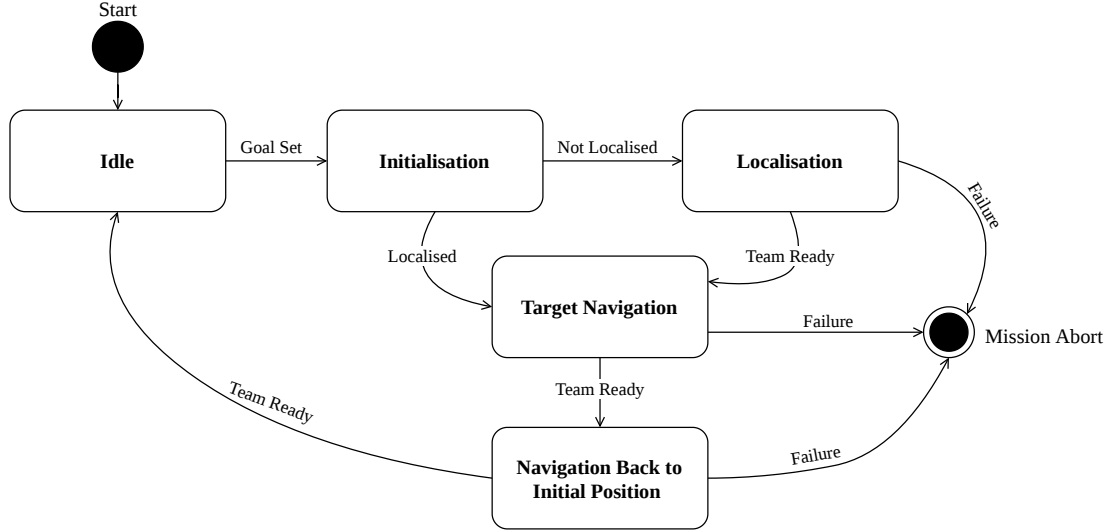


FIGURE 4.1: A state machine of testbed operations.

Figure 4.1 shows a state machine representing a set of tasks to be executed to verify the testbed requirements. Assume that, at each robot, the map has been provided and a target has been specified, the team operations under no events of failure are summarised as follows:

1. In the first task, all robots localise themselves in the operating environment. Each robot waits for all others to complete localisation. Once all robots successfully estimate their poses, the navigation task is prepared to be executed.
2. In the first cooperative navigation task, each robot navigates to its assigned target in a collision-free manner. All robots must arrive at the targets simultaneously. Once all robots reach their targets, the next navigation task is prepared to be executed.
3. In the last cooperative navigation task, each robot navigates back to its initial pose. All robots must arrive at the targets simultaneously. The completion of all robots marks the end of the mission.

Under the events of member failure, the rest of the team must wait the failed robot(s) for a pre-specified amount of time. After timeout occurs, the active robots present in the team must keep executing the mission as normal without the failed ones. This behaviour exhibits the expiry removal proposed in Subsection 3.2.2.

4.2 DISTRIBUTED COMMUNICATION

4.2.1 AD HOC NETWORK

Being a distributed system requires a common network to provide the robots with a communication medium for information exchange. As discussed in Section 2.2, an ad hoc network - with BATMAN-adv as its routing protocol - is deployed in the testbed.

Robots in the system form an ad hoc network. Message-relaying among robots is handled by BATMAN-adv routing protocol. The ad hoc network uses WiFi (IEEE 802.11) [28] as its communication medium. There introduce two frequency ranges for 802.11 devices: 2.4 GHz range and 5 GHz range [23]. The 2.4 GHz frequency range offers a wider WiFi coverage but less bandwidth than that of the 5 GHz frequency range. As the testbed is expected to operate in-door, a wide coverage of WiFi is not vital. This results in a 5 GHz frequency being selected in the ad hoc network deployment to provide the testbed with more communications bandwidth.

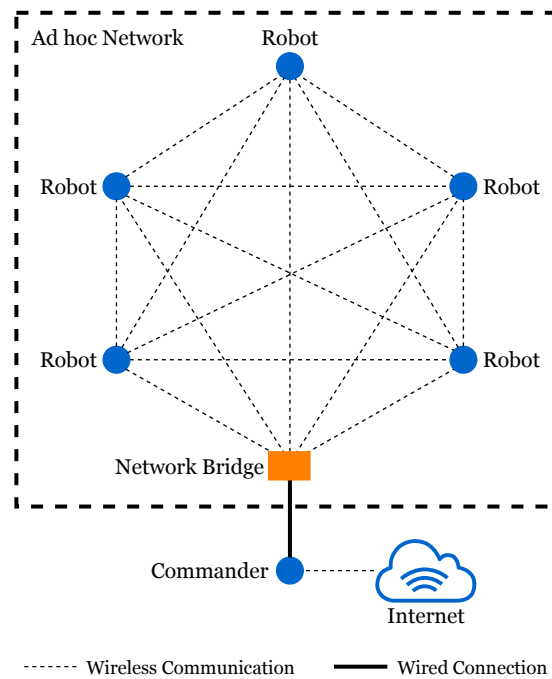


FIGURE 4.2: Network architecture of the proposed testbed.

Figure 4.2 shows the network architecture of the testbed. After having an ad hoc network self-formed, a commander (usually a PC or a laptop to control the testbed remotely) seeks to join the ad hoc network to establish communications with, and if necessary, provide the Internet for the robots. As both the ad hoc network and the Internet require access to the commander's wireless network interface, either only joining the ad hoc network or providing the Internet can be achieved. This problem requires the help of a *network bridge*. A network bridge is a networking device that

bridges multiple communication networks into a single one [29]. In the case of the distributed multi-robot system developing in this research, the commander may connect to the Internet using a wireless network interface, and be wired to the bridge using an Ethernet interface. The bridge, as having the wireless network interface available, can join the ad hoc network and establish communications from the commander to the robots. The network bridge can also be configured as a gateway which forwards packets from the robots to the commander. This configuration enables Internet access to all robots in the ad hoc network.

As the Raspberry Pi does not provide any external real time clock (to reduce its price), system time among the robots may be different. Being a distributed system where information exchange takes place frequently, it is required that the timestamp embedded in the sent message must not differ significantly from the receiver's system time for the purpose of message processing. Being enabled the Internet access may be useful for time synchronisation across the robots. Each robot can send a time-update request to a *Network Time Protocol* (NTP) server and receive the reply as the correct wall time. The time synchronisation process is handled by a Network Time Protocol [30].

4.2.2 MULTI-MASTER STACK

Having a common network configured, ROS is employed to provide the robots with the communication capability by message-passing mechanism.

A traditional ROS network only allows to have one ROS master that manages node registration and address lookup. Even though this mechanism is ROS built-in, such implementation exposes the multi-robot system to a single point of failure - the ROS master. Theoretically, as messages are passing between nodes in a peer-to-peer manner, the failure of the ROS master will not affect the operations of the multi-robot system as long as *no new communication is established*. Practically, in case a node tries to subscribe to another node's topic, the service for a node to register itself will not be available as no ROS master is present. The new communication ends up being timeout, which results in unexpected node behaviours.

Another weakness of the single ROS master lays in the system scalability. Increasing the number of robots in the network would cause the single ROS master to be overloaded, and may introduce delays in the communications. This subsection presents the concept of *topic complexity* which analyses the number of topics a ROS master has to process. Similar to algorithm analysis, the Big-Oh notation represents an upper bound for the topic complexity, and is denoted as $\mathcal{O}$.

Consider the single ROS master mechanism, let N be the number of robots in the network and M be the number of topics at each robot. Under this case, a single ROS

TABLE 4.1: Topic complexity's comparison between a single-master and a multi-master mechanism.

Topic analysis	Single master	Multi-master
Best case scenario	$\mathcal{O}(N^2M)$	$\mathcal{O}(NM)$
Average case scenario	$\mathcal{O}(N^2M)$	$\mathcal{O}(N(M + s))$
Worst case scenario	$\mathcal{O}(N^2M)$	$\mathcal{O}(N^2M)$

master has to handle the registration all topics in the network, which is of $N \times N \times M$ topics. The topic complexity is $\mathcal{O}(N^2M)$.

In the case of single master, the topic complexity remains unchanged in the best case, the average case, and the worst case.¹ The topic complexity always grows in polynomial time to the number of robots present in the network.

These limitations of the single master mechanism make it not an ideal candidate for a multi-robot system. The testbed, instead, employs another mechanism called *multi-master*. In the multi-master approach, each robot operates its own master. Having multiple redundant masters in the network solves the problem of a single point of failure. Furthermore, as the ROS masters are distributed across the robots, the number of topics that a ROS master has to process are reduced significantly. At an arbitrary ROS master in the network, let s be the number of topics from another robot that the ROS master has to be processed. The number of topics to be processed are $M + s$. As there are N robots present in the team, the topic complexity is $\mathcal{O}((M + s)N)$.

In the multi-master mechanism, the worst case occurs when $s = (N - 1)M$, or in other words, when all topics from other robots are required to be processed. In such case, the topic complexity is $\mathcal{O}(N^2M)$ which is equal to the topic complexity of the single master approach. The best case occurs when there is no topics from other robots required to be processed (or $s = 0$). The topic complexity in the best case is $\mathcal{O}(NM)$. It can be said that, at an arbitrary ROS master in a multi-master mechanism, the topic complexity - on average - grows linearly with the number of robots in the network, under the assumption that a few topics are required to be processed by the ROS master. Table 4.1 summarises the topic complexity in both the case of a single master mechanism and a multi-master mechanism.

In ROS community, the multi-master mechanism has been developed and supported by Tiderko, Hoeller, and Röhling under the package named `multimaster_fkie` [31]. The testbed primarily employs this package to establish and manage a multi-master ROS network.

¹In the single-master mechanism, a more accurate notation of the topic complexity should be $\Theta(N^2M)$ which provides a *tight bound* on the complexity. However, for the purpose of comparing with the multi-master approach, Big-Oh notation is solely used to analyse the topic complexity.

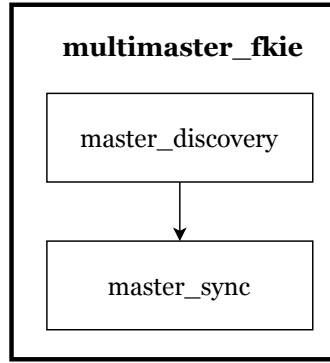


FIGURE 4.3: Components and data path in multimaster_fkie.

multimaster_fkie consists of two main nodes: master_sync and master_discovery as shown in Figure 4.3.

Sergi Hernandez Juan introduced the operational principles of the multimaster_fkie package in their report [32]. The master_discovery, as suggested by its name, discovers other masters in the network. Each master periodically sends *multicast messages*² to the common network to notify about its presence and discover other available ROS masters. The masters discovered from the master_discovery are provided for the master_sync for the purpose of remote topic/service registrations to the local ROS master. These registrations at every ROS master are updated in accordance with the changes from other ROS masters. This process, in multimaster_fkie is called *master synchronisation*. master_sync can be configured to synchronise only specific topic(s), service(s), or even host(s). By default, all topics and services of all hosts are synchronised. However, it is recommended to synchronise as less topics/services as possible to reduce the network bandwidth. It should be noted that multi_master does not support the functionality of remote parameter synchronisation.

4.3 MULTI-ROBOT COOPERATIVE NAVIGATION

4.3.1 NAVIGATION STACK

The Navigation Stack is the most widely-used stack to perform a navigation task. The aim of the Navigation Stack is to reliably move a robot from a starting position to another goal without any collisions made in the environment. The Navigation Stack takes the input from the robot odometry, sensor values, the starting positions, and the goal position. The output is the velocity to safely move the robot to the target.

In the Navigation Stack, a map is not compulsory for navigation. Without a map, a robot would use its local odometry as a reference during navigation. The position

²Multicast is a type of communication where data are transmitted to a groups of destination simultaneously. It can be one-to-many or many-to-many transmission. An introduction on multicast can be found at [33].

of the robot, in this case, is relative to the odometry reference frame. Despite the obvious advantage of operating without a map, odometry is not reliable for a long-term global reference as it is subject to accumulated error over time [34]. As being a multi-robot system performing navigation tasks, all robots must have a common global reference to relate the position of one robot to another. A map can provide a fixed frame - so-called *map frame*. The map frame is useful as it provides a reliable long-term global reference for the team of robots to perform navigation, which explains the reason why the testbed in this research will require a map to operate.

There exist several constraints on the hardware required to use the Navigation Stack [35]. Only differential drive and holonomic wheeled robots are supported in the Navigation Stack. A planar laser - used for localisation, and obstacle avoidance - is required to be mounted on the robot base. The Navigation Stack will perform best on "robots that are nearly square or circular". Even though the "Navigation Stack does work with robots with arbitrary shapes and sizes, it may have difficulty with large rectangular robots in narrow spaces like doorways".

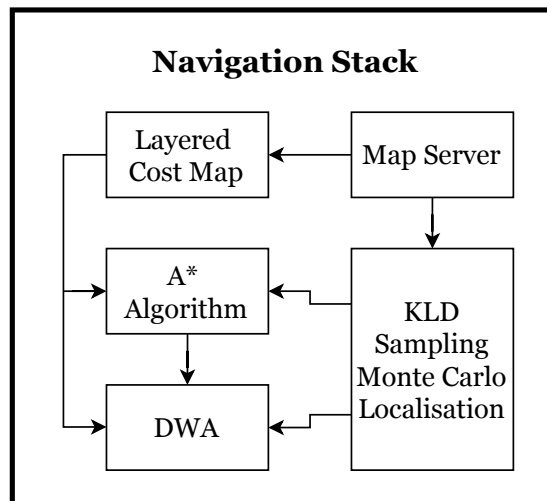


FIGURE 4.4: Components and data paths in the navigation stack.

TABLE 4.2: Correspondent algorithms with their ROS implementation.

Algorithm	ROS implementation
Map Server	map_server
KLD-Sampling MCL	amcl
Layered Cost Map	costmap_2d
A* Algorithm	global_planner
DWA	dwa_local_planner

Although the Navigation Stack consists of various packages, only the primary ones will be presented in this section. A more complete details on the packages of the

Navigation Stack can be found at [35]. Figure 4.4 introduces the primary components and the data path in the Navigation Stack. The actual package names in the Navigation Stack are modified in the scope of this thesis to match with the techniques being discussed in Chapter 2. One can find the corresponding techniques with the actual ROS package names in Table 4.2.

KLD-Sampling Monte Carlo Localisation estimates the robot pose using the grid map published from the Map Server. The robot pose is input to the A* Algorithm and Dynamic Window Approach to perform global planning and action planning respectively. During navigation, the Layered Cost Map keeps the information of static/dynamic obstacles being updated by the LiDAR scanning the environment. The planned path and the movements change to cope with the costmap updates.

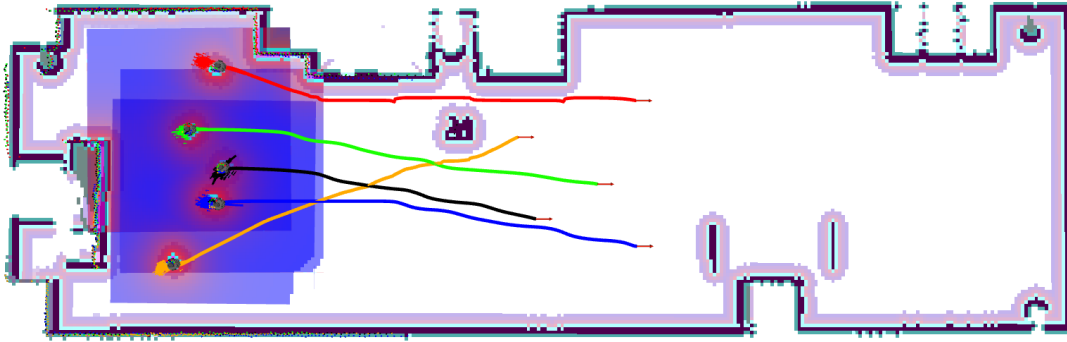


FIGURE 4.5: A navigation mission executed by the Navigation Stack.

Figure 4.5 illustrates a navigation mission managed by the Navigation Stack. The coloured lines are the path to targets generated by A* algorithm and DWA. The blue squares represent layers of the costmaps generated from the robots - named as obstacle layer. The purple area buffered around the map is another layer of costmap - called static layer.

4.3.2 COOPERATIVE NAVIGATION STACK

As the Navigation Stack is not designed specifically for multi-robot cooperative navigation, it - without any modifications - will not be suitable for the testbed. This subsection introduces a modified Navigation Stack - called *Cooperative Navigation Stack* (CNS) - to support multi-robot cooperative navigation.

Figure 4.6 introduces the components and the data paths in the Cooperative Navigation Stack. The CNS divides the original Navigation Stack into (a) the Localisation Stack, and (b) the Co-Navigation Stack.

In the original Navigation Stack, robot localisation - to the best of our knowledge - requires an operator to perform several repetitive steps. For example, to perform global localisation, the operator firstly spreads the particles evenly across the provided map, then intuitively drives the robot around until the particles converge to the

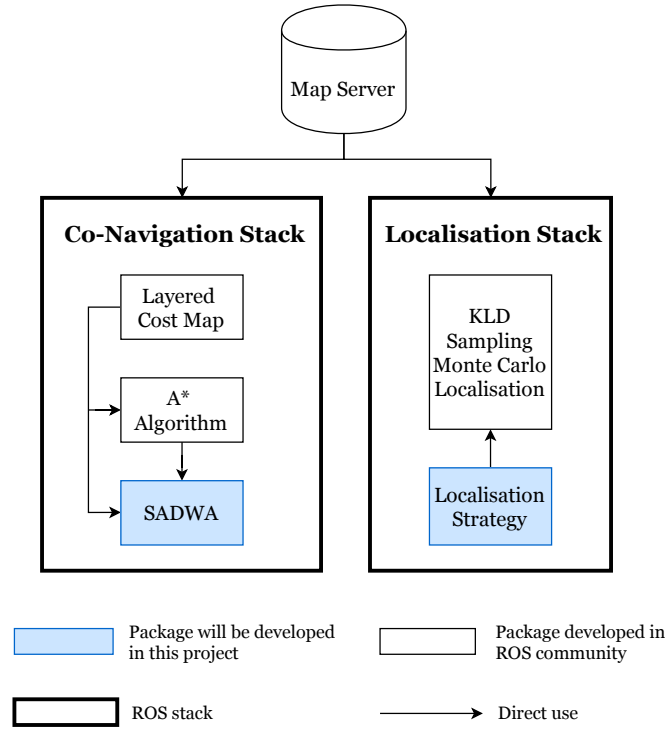


FIGURE 4.6: Components and data paths in the Cooperative Navigation Stack.

true robot pose. The Localisation Stack in the CNS is meant to replace such manual steps. It provides the basic functionalities for autonomous robot localisation. In the Localisation Stack, a map is provided to the KLD-Sampling Monte Carlo Localisation. The Localisation Strategy, as suggested by its name, implements a set of steps (or strategy) for robot localisation.

In this testbed, the simplest strategy is implemented to verify the initial operations. The autonomous strategy firstly spreads the particles uniformly across the provided map. The robot then performs in-place rotations to scan the environment. The in-place rotation is chosen as it is the safest movement which guarantees no collisions with any obstacles in the surrounding environment. KLD-Sampling Monte Carlo Localisation takes in the LiDAR scanner values and updates the likely pose as the robot rotates. The ROS implementation of the KLD-Sampling Monte Carlo Localisation returns the robot pose under the form of $[x, y, z, \alpha, \beta, \gamma]$; along with a 6×6 covariance matrix of the form

$$\Sigma = \begin{bmatrix} \sigma_x^2 & K_{xy} & K_{xz} & K_{x\alpha} & K_{x\beta} & K_{x\gamma} \\ K_{yx} & \sigma_y^2 & K_{yz} & K_{y\alpha} & K_{y\beta} & K_{y\gamma} \\ K_{zx} & K_{zy} & \sigma_z^2 & K_{z\alpha} & K_{z\beta} & K_{z\gamma} \\ K_{\alpha x} & K_{\alpha y} & K_{\alpha z} & \sigma_\alpha^2 & K_{\alpha\beta} & K_{\alpha\gamma} \\ K_{\beta x} & K_{\beta y} & K_{\beta z} & K_{\beta\alpha} & \sigma_\beta^2 & K_{\beta\gamma} \\ K_{\gamma x} & K_{\gamma y} & K_{\gamma z} & K_{\gamma\alpha} & K_{\gamma\beta} & \sigma_\gamma^2 \end{bmatrix},$$

where Σ is the covariance matrix,
 K_{ij} is the covariance between the data set i and j ,
 σ_i^2 is the variance of the data set i ,
 x, y, z are the Cartesian coordinates,
 α, β, γ are the roll, pitch, yaw respectively in the Euler angles.

Euler angles together with the Cartesian coordinates are used for 3D localisation such as for drones or robot arms. As the Turtlebot3 Burger is a ground vehicle, only the x, y Cartesian coordinates with the yaw γ in the Euler angles are used. As such, the localisation strategy is only interesting in the variance of x, y , and γ . As the robot rotates with a rotational velocity ω_l , the variance σ_x^2 , σ_y^2 , and σ_γ^2 are compared with a value of localisation tolerance. Once the pose variances are less than the tolerance, the particles are considered to be converged, and the robot can stop rotating. This marks the end of the localisation process. Algorithm 5 describes the localisation strategy's operations using pseudo-code.

Algorithm 5 Localisation strategy for KLD-Sampling Monte Carlo Localisation

Require: Rotational speed for localisation ω_l , localisation tolerance ε

```

1: procedure LOCALISATION( $\omega_l, \varepsilon$ )
2:   spread particles uniformly over the map
3:   repeat
4:     rotating in place with rotational velocity  $\omega_l$ 
5:     update likely pose using KLD-Sampling MCL
6:   until  $\sigma_x^2 \leq \varepsilon$  and  $\sigma_y^2 \leq \varepsilon$  and  $\sigma_\gamma^2 \leq \varepsilon$ 
7:   stop rotating in place
8:   return  $[x, y, \gamma]$ 
    
```

In the Co-Navigation Stack, the DWA is replaced by SADWA - the algorithm proposed in Section 3.1 for simultaneous arrival. The operations remain unchanged from the original Navigation Stack; except that an additional simultaneous arrival function is added as another criterion for navigation.

4.4 SOFTWARE INTEGRATION

With the required operations discussed in Section 4.1, this section provides the reader with a ROS high-level software design where the proposed methodologies are integrated to form a complete testbed. The testbed's software architecture can be divided into (a) the server side - where ROS action servers are implemented, and (b) the client side - where ROS action clients and mission controller are implemented.

In the server side architecture, the `multimaster_fkie` stack plays a critical role as it handles the communication between all robots in the testbed. At each robot, the `master_discovery` discovers and establishes communication links to all other

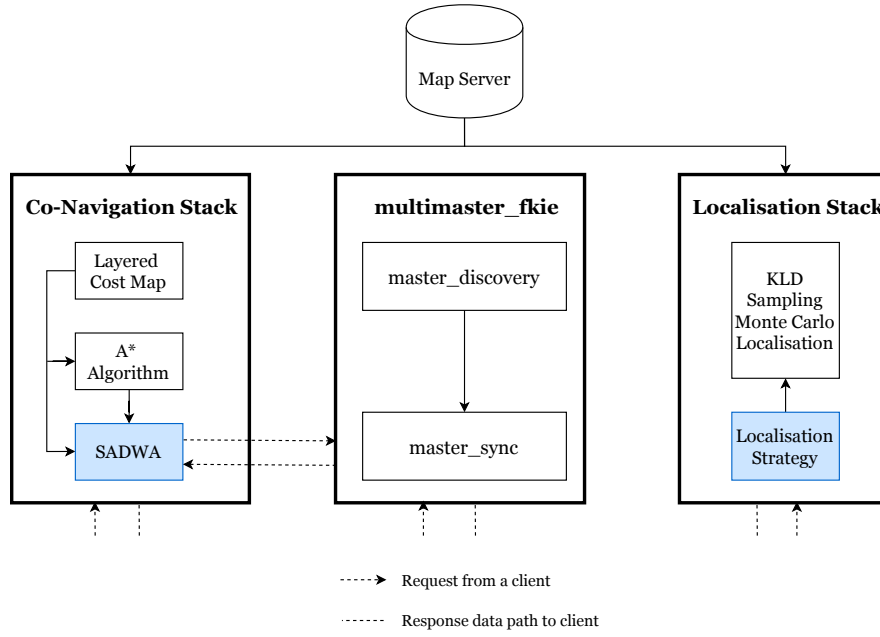


FIGURE 4.7: Server side architecture.

robots joining the network. Required topics and services are synchronised through `master_sync`. This forms a complete ROS multi-master network. The Co-Navigation Stack and the Localisation Stack are parts of the CNS. The operations of the CNS are discussed in Subsection 4.3.2. Notably, for simultaneous arrival in SADWA to achieve the max-consensus as presented in Subsection 3.1.4, communication links with other robots in the team must be established to ensure sufficient information are received for Algorithm 1 to function. These communication links are established through the `multimaster_fkie` stack.

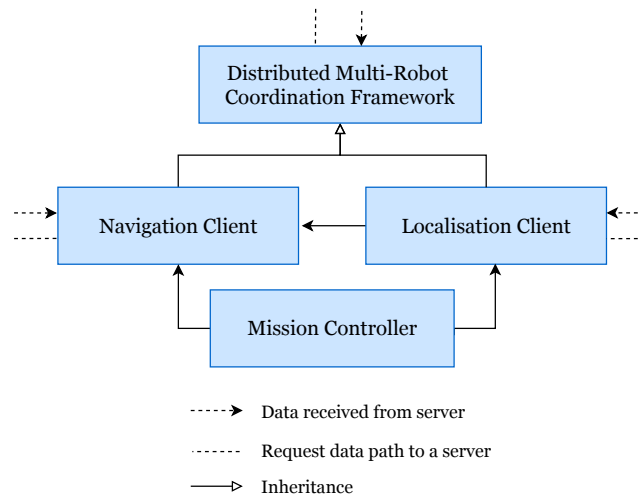


FIGURE 4.8: Client side architecture.

The Distributed Multi-Robot Coordination Framework (DMRC framework) provides an abstract interface for all tasks to be inherited from. Tasks - which inherit from

the DMRC - will have the function of task synchronisation, and are handled events of member failure. The DMRC indirectly establishes communication links and receive information with its neighbours through the `multimaster_fkic`. To satisfy the required testbed's operations, Mission Controller, Localisation Client, and Navigation Client implement the state machine in Figure 4.1. The Mission Controller allows the operator to specify a set of targets for the team to navigate to. To perform localisation, the Localisation Client can request a localisation task, and receive the localisation results as the response from the Localisation Stack. For target navigation, Navigation Client can request for a navigation task in the Co-Navigation Stack. The actual navigation process is managed by the Co-Navigation Stack. After completing a navigation task, the Co-Navigation Stack will send the navigation status, such as cancelled, aborted, or completed to the Navigation Client.

Figure 4.9 shows the network architecture for the testbed. The communication links between robots are managed by `multimaster_fkic`. Figure 4.10 presents the complete software architecture for each robot in the testbed. As the testbed is a distributed multi-robot system, the same software architecture can be implemented and deployed on every robot, yet still ensures the desired team behaviours.

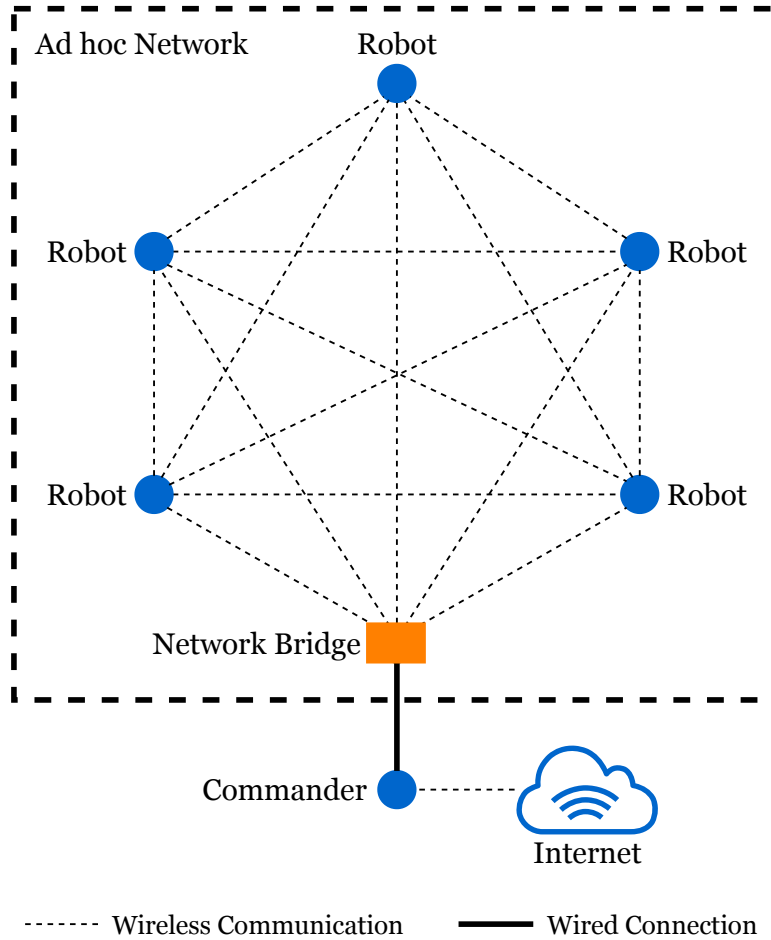


FIGURE 4.9: Network architecture of the proposed testbed.

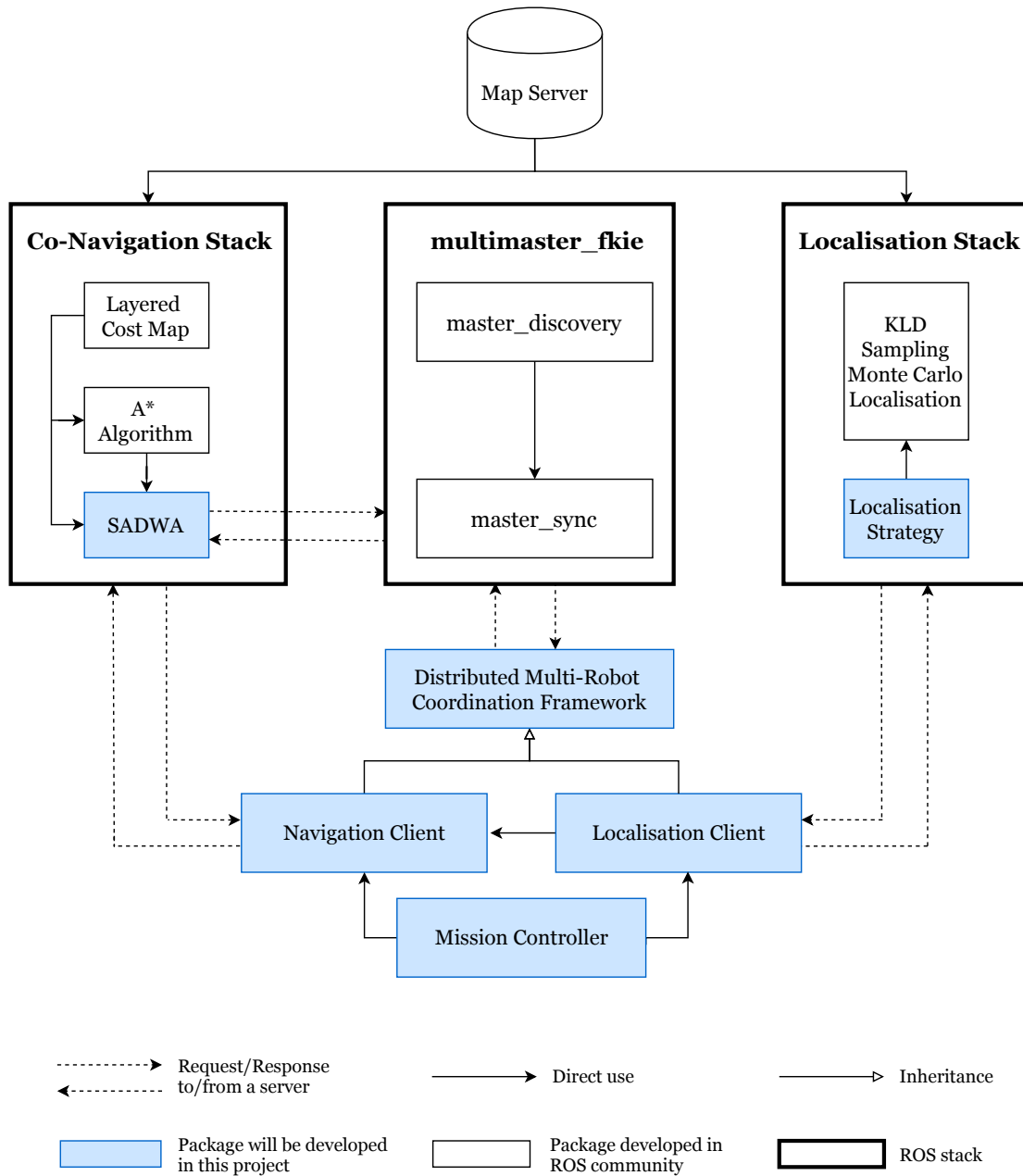


FIGURE 4.10: Software architecture for each robot.

RESULTS AND DISCUSSION

This chapter evaluates and verifies the proposed techniques under different scenarios in both simulation and physical environment. The results are presented and discussed in detail. It also identifies and suggests future directions to several gaps in the proposed techniques.

5.1 EXPERIMENTAL SETUP

This section provides the reader with a detailed description on the operating environment where the experiments were conducted to verify the testbed operations. SADWA and the DMRC framework proposed in Chapter 3 were firstly tested in simulation environment. The testbed proposed in Chapter 4 was then deployed and verified in physical environment.

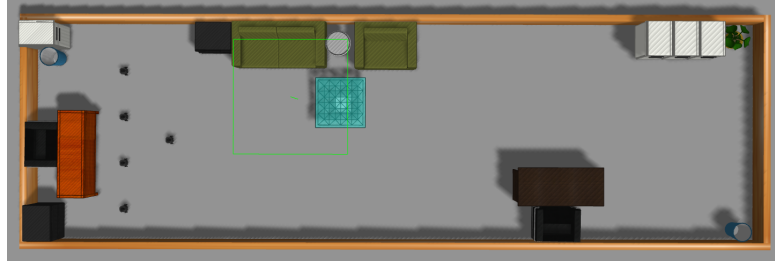
Several assumptions are made to ensure proper operations of the testbed. Firstly, the operating environment, regardless simulation or physical environment, should have distinctive features to support localisation. Symmetric environments such as hallways will significantly reduce the localisation accuracy. As the robot uses the LiDAR scan values to estimate and track its pose, the LiDAR's laser source should always be reflected back. From our practical experience, it is recommended that the operating environment, for every 12 m^2 should have at least one object.¹ The further away, the less accurate the LiDAR is. Under this research, it is assumed that a map of the operating environment has been generated by a mapping algorithm.² The map acts as a priori for navigation. It also provides the team of robots a common reference for position tracking.

Figure 5.1(a) presents a bottom-down view of the simulation environment with 5 robots forming a team. The testing simulation environment has an area of approximately 70 m^2 . The corresponding map of this environment is shown in Figure 5.1(b). In simulation, both ad hoc network and multi-master mechanism were not employed.

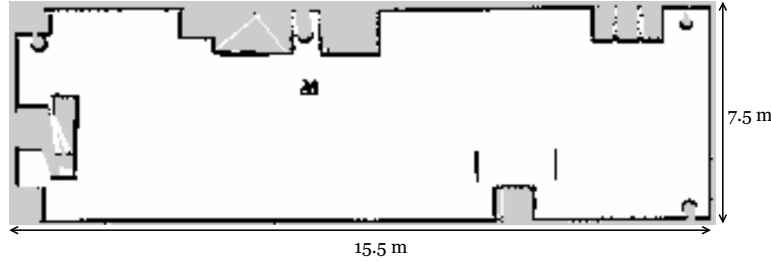
¹The maximum range the LiDAR can scan is 3.5 m.

²Mapping is an application of Simultaneous Localisation and Mapping (SLAM). As SLAM is not a focus of this thesis, it will not be discussed in details. Interested reader can find more information on the mapping algorithm for map generation in this project at <http://wiki.ros.org/gmapping>.

Ad hoc network and multi-master mechanism required multiple devices to form a network; however, simulation only ran on a single machine. Thus, instead of an automatic robot discovery managed by `master_discovery`, a list of all robots in a team was manually provided to each robot. This list could be used to establish communication links from one to others, just as what `multimaster_fkie` does, but in a single-master manner.



(A) Gazebo environment: Bottom-down view.



(B) Map generated by mapping algorithm.

FIGURE 5.1: A simulation environment of 70 m² with 5 robots.

Figure 5.2(a) and Figure 5.2(b) presents a testing physical environment whose area of approximately 36 m². The experiments were conducted over 3 Turtlebot3 Burgers. The corresponding map generated by a mapping algorithm is shown in Figure 5.2(c). In the physical environment, an ad hoc network (discussed in Section 2.2 and Subsection 4.2.1) together with a multi-master mechanism (discussed in Subsection 4.2.2) were deployed on the team of 3 robots.

From this section on, simulation testing refers to the experiments conducted in the environment as shown in Figure 5.1; while physical environment testing refers to the verification in the environment as shown in Figure 5.2.

5.2 LOCALISATION STACK

In this section, the results of the Localisation Strategy (discussed in Algorithm 5) was presented to verify the proposed Localisation Stack. This section also discusses the insights over the results, and suggests several future directions. As the main research theme does not focus on localisation, this section is not meant to provide the results on the performance of KLD-Sampling Monte Carlo Localisation. One can find a thorough and detailed evaluation in the paper of Fox [36].

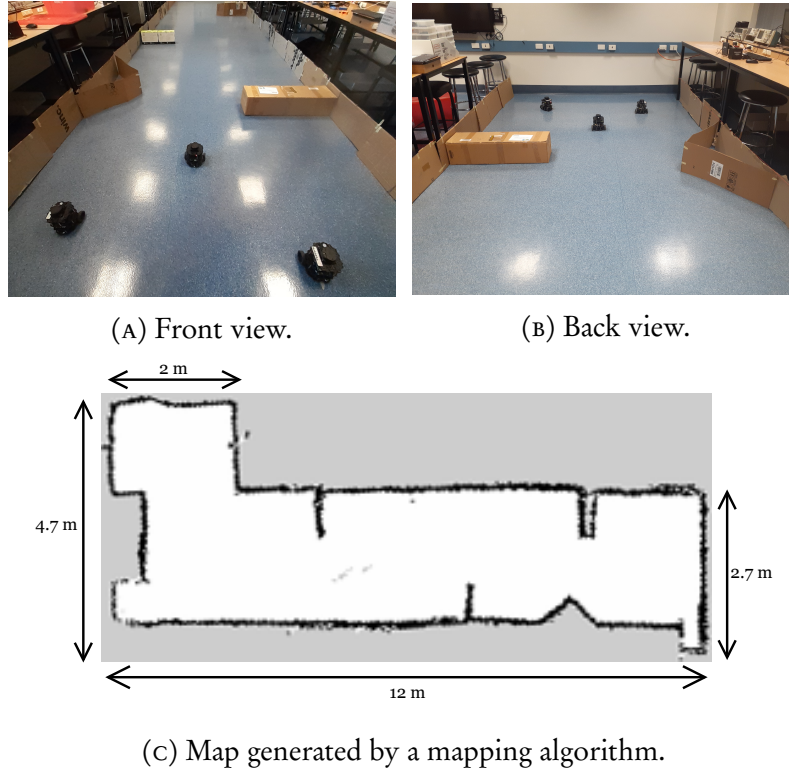


FIGURE 5.2: A physical operating environment of 36 m² with 3 robots.

5.2.1 LOCALISATION STRATEGY

Figure 5.3 illustrates the simulation of localisation process managed by the Localisation Strategy in the CNS. The localisation was performed with maximum 5000 particles, a rotational velocity of $\omega_l = 0.7$ rad/sec and a localisation tolerance of $\varepsilon = 0.03$ m². Simulation ran on an Intel Core i7's laptop with 8 GB of RAM. As introduced in Algorithm 5, firstly, particles were distributed evenly across the map as shown in Figure 5.3(a). Each robot collected more surrounding information as it rotated in-place. Figure 5.3(b) and Figure 5.3(d) illustrates that the particles, after several rotations, concentrated on several most likely areas to be the actual position of the robots. Localisation was completed when the particle variance was less than the localisation tolerance $\varepsilon = 0.03$ m². The robot stopped, prepared for the new navigation task.

Figure 5.4 demonstrates the effect of modifying the localisation tolerance ε . For the localisation tolerance of 0.5 m², the particles tend to be more spread out than in the case of localisation tolerance of 0.05 m². From practical experience, we argue that the less the localisation tolerance, the more time it takes to localise.

5.2.2 DISCUSSION

Even though the Localisation Stack, specifically the KLD-Sampling MCL, demonstrated promising results in simulation, testing the algorithm using the testbed in

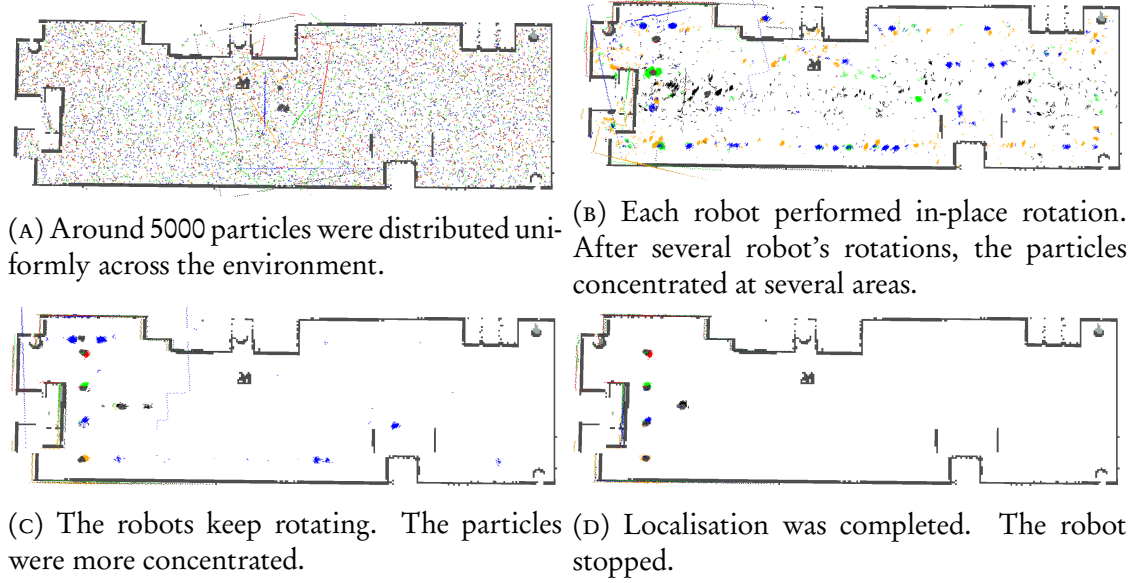


FIGURE 5.3: Localisation strategy developed in the CNS (simulation).

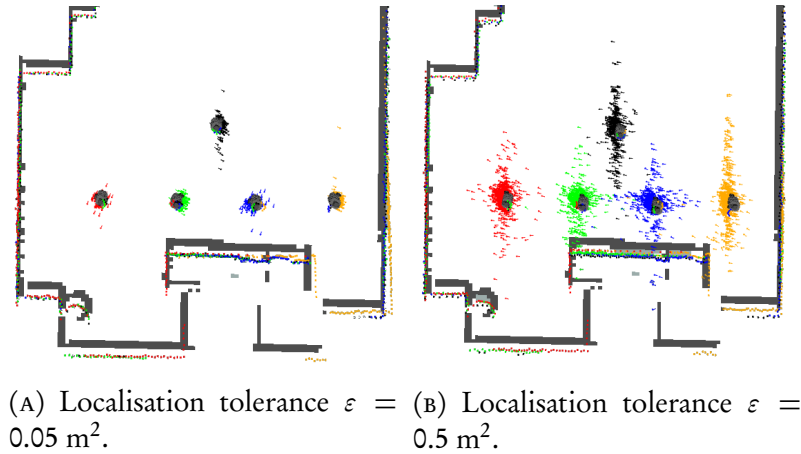


FIGURE 5.4: Localisation with different values of tolerance (simulation).

physical environment indicated another different story. From trial and error, in KLD-Sampling MCL, we estimated that the maximum number of particles which the Raspberry Pi 3 Model B+ could handle was 2800 particles. Specifying more than 2800 particles results in excessive use of the Raspberry Pi's CPU. From experiments, CPU overloaded could make the particles not being converged. Testing in an area of 100 m^2 with 2900 particles led to CPU being overloaded; however, reducing down to 2700 particles did not guarantee localisation accuracy. We also tuned other parameters such as `kld_err`, `laser_z_rand`, `laser_z_hit`, `odom_alpha` which followed the guidelines of Zheng [37], but no positive results returned.³ This limitation poses a huge constraint on the testbed operations in a larger environment.

³Complete parameters' descriptions can be found at <http://wiki.ros.org/amcl>.

For localisation strategy, due to the time constraint put in the research, we implemented a simple strategy which only incorporated rotational movement. One can devise a more complex and flexible strategy for better localisation results. Strategy that incorporates translational movements or drives a robot through a long distance, can provide the algorithm with more data on the environment, which makes the localisation more accurate.

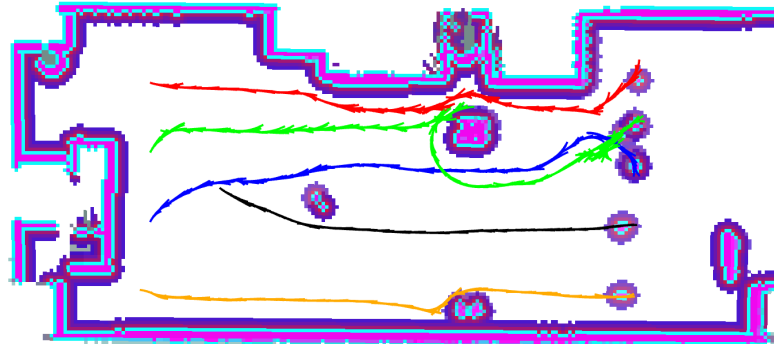
5.3 SIMULTANEOUS ARRIVAL USING DYNAMIC WINDOW APPROACH

In this section, Simultaneous Arrival using DWA (SADWA) was tested in simulation with a team of 5 robots and was verified in a physical environment with 3 robots. Obstacle avoidance and simultaneous arrival were the two critical functions to be tested. The team of robots must be able to avoid static/dynamic moving obstacles and arrive at a pre-specified set of targets simultaneously. The team executed various navigation missions to different targets and was exposed to disturbances such as obstacles, or events of member failure. This section provides the reader with the SADWA's results and discusses several insight over the results.

5.3.1 SIMULATION



(A) A simulation environment with dynamic obstacles. The trash acted as an unexpected obstacles; the person acted as a moving obstacle. The team was assigned a mission to navigate to the Area of Targets.



(B) Trajectories generated by 5 robots. The person moved during the navigation task causing a robot to re-plan its trajectory.

FIGURE 5.5: A navigation task with static and dynamic obstacles.

A team of 5 robots were tested by assigning a navigation mission to 5 different targets. In the mission, static, unexpected and moving obstacles were added as the disturbances. Figure 5.5(a) presents the testing environment in Gazebo where a trash can and a person were added as dynamic obstacles. The trash can, since not being present in the provided map, acted as an unexpected obstacle. While the team of robots were proceeding to the targets, the person was moved in a way that he stood in the pre-planned path of one robot. Figure 5.5(b) shows the trajectories generated by the team in RViz.

The performance of obstacle avoidance can be seen more clearly by comparing the baseline case in Figure 5.6(a) where no dynamic obstacles present, with the case in Figure 5.6(b) where dynamic obstacles appeared in the environment. Robot 4 and Robot 5 re-planned the path to cope with the trash can and the moving person on the way to the targets.

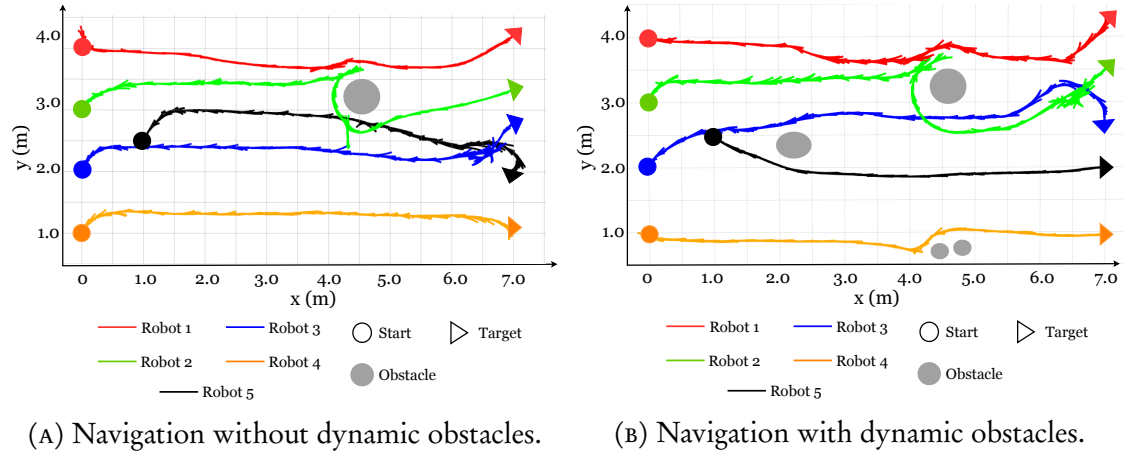


FIGURE 5.6: Trajectory generated from the team of 5 robots.

Interestingly, even though not meant to be tested in this mission, Figure 5.6 also shows the capability of each robot to avoid each other. This occurred since each robot is considered as a moving obstacle to other robots. In the testing mission, to arrive at the targets, Robot 1 and Robot 2 firstly navigated towards the static obstacle. As approaching the static obstacle, the space between the Robot 1 and the Robot 2 got tighter. To avoid colliding with Robot 1, Robot 2 re-planned another path around the static obstacle, then safely moved to the target. The ability of avoiding each other, in more details, are demonstrated in Figure 5.7. In this scenario, the team of robots was assigned a cooperative navigation task to a set of targets (denoted by the red arrows) such that the paths crossed each other. Figure 5.7(a) illustrates the planned paths to targets when no crossing situation detected. As proceeding to the targets, the robots navigated towards each other. When collisions between robots were detected, each robot tried to re-plan another path to the target as shown in Figure 5.7(b). After executing the new paths, Figure 5.7(c) shows that it was safe to navigate to the target

without any robot collisions. In Figure 5.7(d), the crossing situations had been resolved; thus, the team could arrive at the targets simultaneously without collisions with each other.

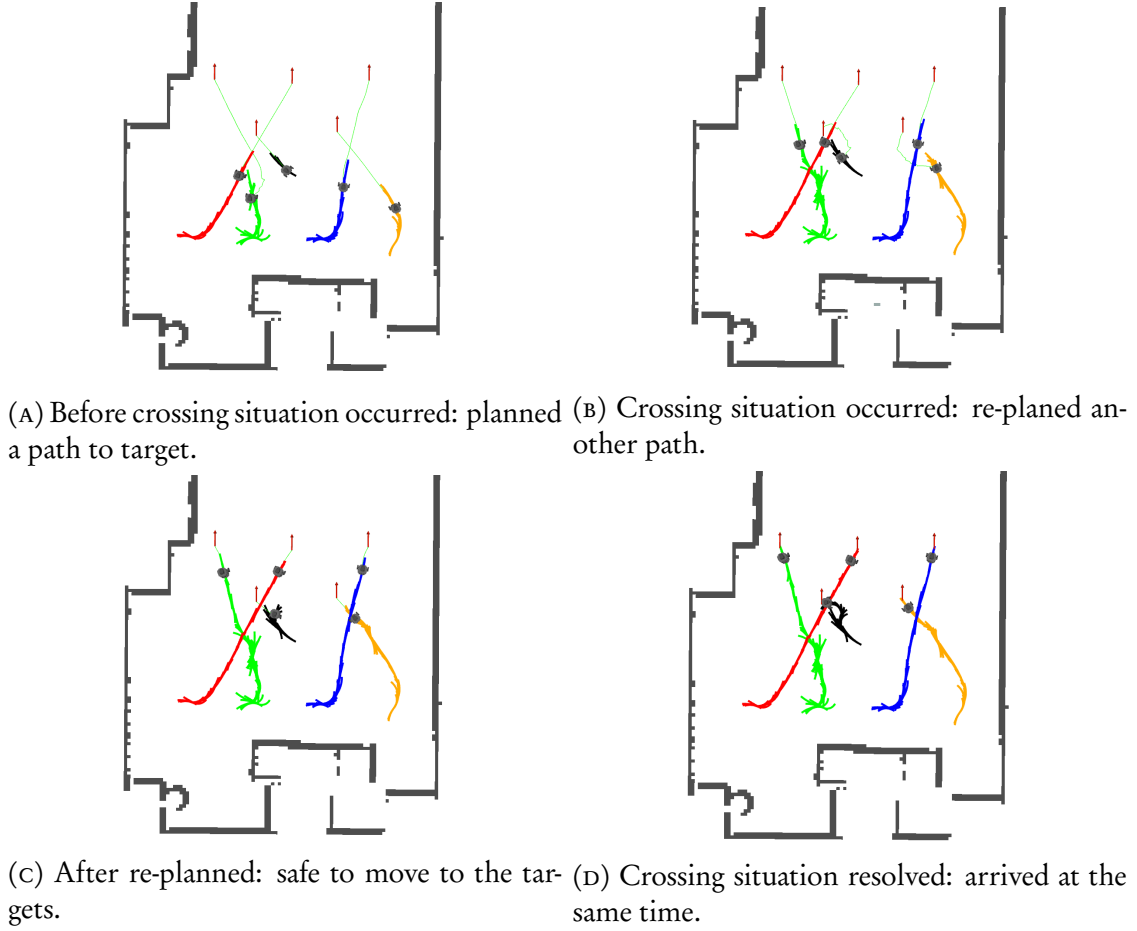


FIGURE 5.7: Handling a complex situation where crossing paths exists during navigation.

It is obvious that, a simultaneous-arrival mission is stated to be achieved if all robots in the team arrive at the targets at the same time. To verify the simultaneous arrival ability, Figure 5.8 shows the robots' ETAs over time in a navigation mission. A robot is considered as reaching the target when its ETA to target is 0 second. From the graph, the ETAs of all robots converged asymptotically to 0 second at the 78-th second since the starting time. In other words, all robots reached the targets simultaneously after 78 seconds of execution time. There were oscillations in the ETAs over the execution time in the navigation mission. The oscillations are due to (3.9) where the consensus time are decided based on negotiating the maximum value among the ETAs at every communication round. The oscillations in Figure 5.8 represents the process of consensus time negotiation where the peak is the negotiated consensus time ψ .

Another experiment was conducted to expose the robots to events of failure for the purpose of testing the member failure handling capability of SADWA (as discuss

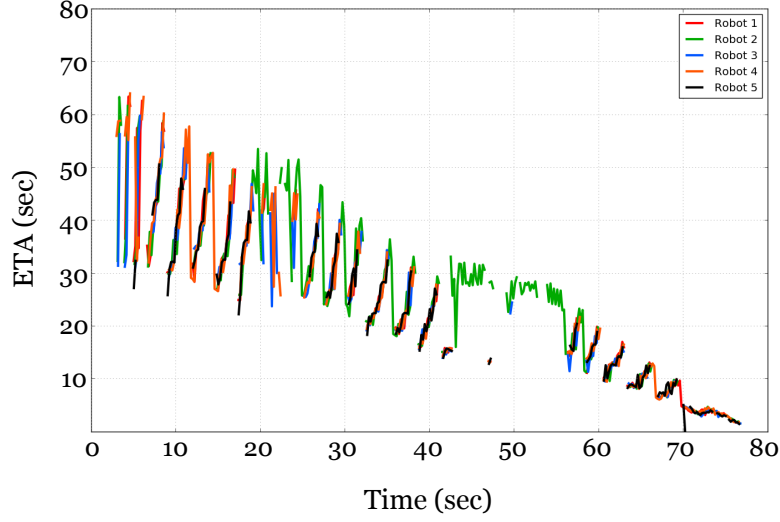
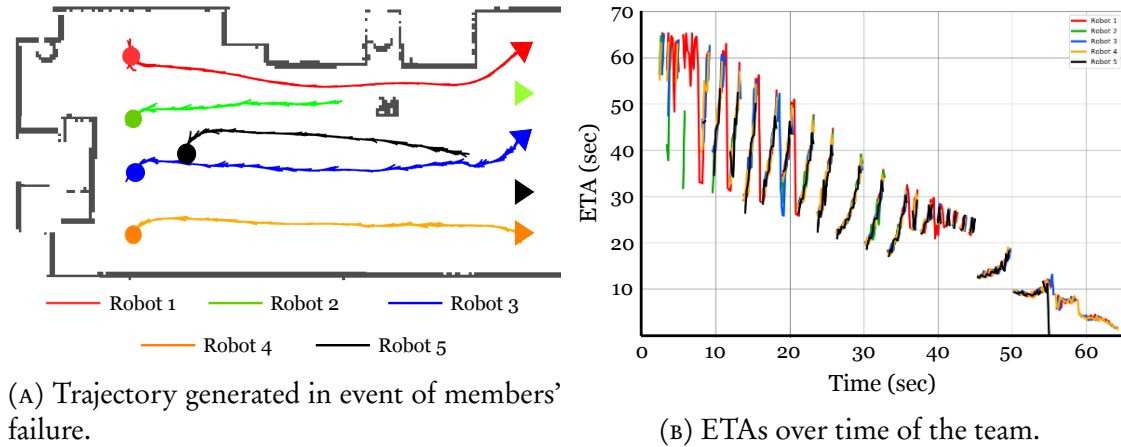


FIGURE 5.8: ETAs over time of 5 robots.

in Algorithm 2). Figure 5.9(a) illustrates that Robot 2 and Robot 5 were forced to stop before reaching their targets. This simulated the failures of the Robot 2 and Robot 5. From the graph, it can be seen that Robot 2 and Robot 5 failed at the 35-th and the 55-th second, respectively. The trajectory generated in Figure 5.9(b) shows that the rest of the team (Robot 1, Robot 3, and Robot 4) could complete the mission without the information from the failed robots. To be more specific, the active robots' ETA converged asymptotically to 0 second without the ETAs from the failed ones. It can be concluded that under the event of failure, the active robots in the team could still complete the mission.



(A) Trajectory generated in event of members' failure.

(B) ETAs over time of the team.

FIGURE 5.9: Simultaneous arrival in the event of team members' failure.

Figure 5.9 demonstrates an interesting effect of the consensus time under the events of member failure. As being discussed, the oscillations in the Figure 5.9(b) were the consensus time negotiation through the distributed algorithm's communication

rounds. Upon the event of member failures, the negotiation time to come to a consensus reduced significantly as can be seen from the 35-th second on. This is because under the events of member failure, after expiry removal occurs, less information is required to be exchanged across the robots.

To conclude that SADWA is working as proposed, 80 navigation missions had been tested with different sets of targets and disturbed by both static and dynamic obstacles. The experiments included several events of failure where navigation was aborted in the middle of the mission. Figure 5.10 shows the *time difference of arrival* (TDA) over all testing missions. The TDA of a simultaneous arrival task is defined as the time difference between the maximum time of arrival with the minimum time of arrival. Ideally, in a simultaneous arrival navigation task, the TDA is desired to be 0 second. As shown in Figure 5.10(a), for 80 navigation missions tested, the TDAs were always within 1 second. Figure 5.10(b) indicates that the mean TDA was 0.4 second. With the latency in message passing and processing time in SADWA, a value of TDA less than 1 second in a simultaneous arrival mission is acceptable.

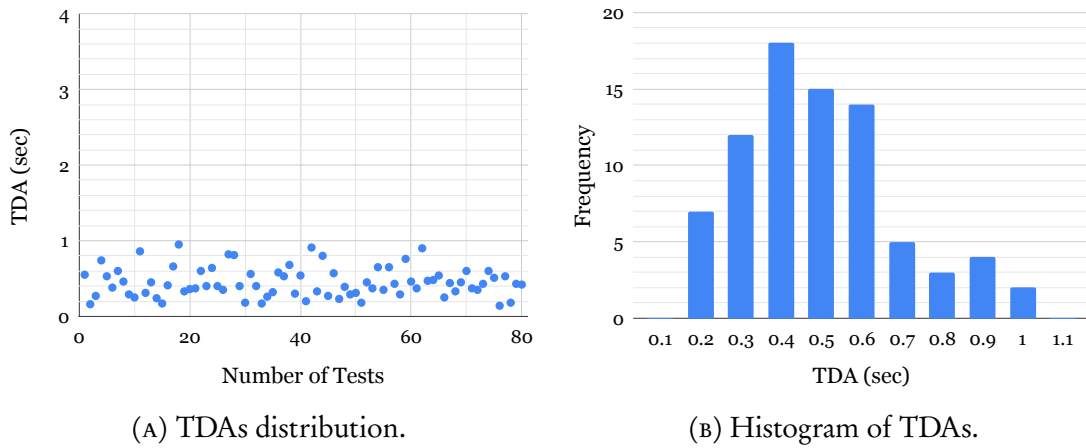


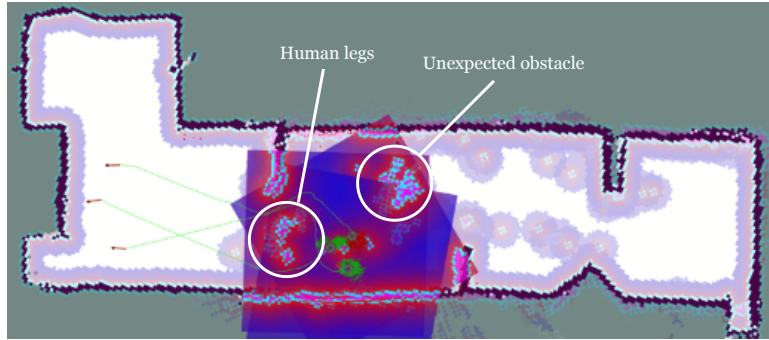
FIGURE 5.10: Simultaneous arrival's data points and histogram.

5.3.2 PHYSICAL ENVIRONMENT TESTING

To verify the behaviours in simulation environment, a team of 3 robots was tested in the physical environment introduced in Figure 5.2. The team was assigned a navigation mission to a set of targets 11 meters away. Along the way to the targets, a person slowly walked into the navigation path of the robots. A red box, as shown in Figure 5.11(a), is present as an unexpected obstacle. In terms of SADWA's performance, the behaviours in the physical environment were as expected as in simulation. Specifically, for the mission shown in Figure 5.11, the TDA was 0.4 second. The total navigation time to targets was 2 minutes and 30 seconds.



(A) A view from a physical mission.



(B) RViz view.

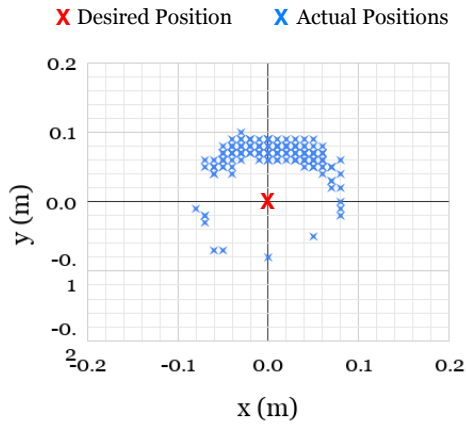
FIGURE 5.11: Operations in a complex environment with static, unexpected obstacles, and people.

5.3.3 DISCUSSION

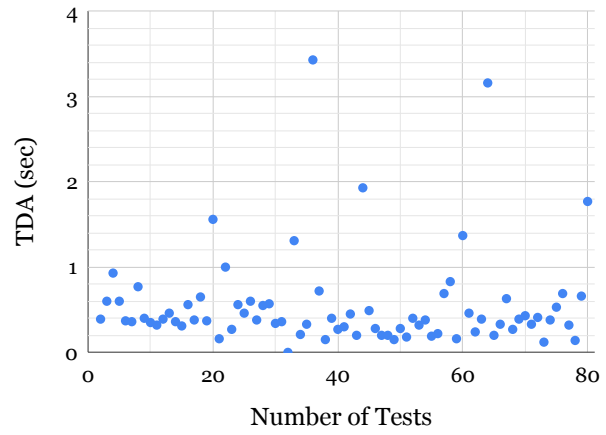
Even though simultaneous arrival appeared to be performing well, Section 5.3.1 did not discuss on the target arrival's accuracy. A robot is considered as reaching the target if its current position relative to the target's position is within a value of tolerance. This value of tolerance is called *position tolerance*. The less the position tolerance is, the more manoeuvre a robot is required to reach the target.

Figure 5.12 shows the comparison between the TDAs over 80 navigation missions with two values of position tolerance: 10 cm and 15 cm. From Figure 5.12(b), there were several outliers of the maximum TDA in navigation tasks with the position tolerance of 10 cm. In the worst case, the maximum TDA was around 3.6 seconds. A TDA of 3.6 seconds is definitely not desirable in a simultaneous arrival navigation task. It was not the case when increasing the position tolerance from 10 cm to 15 cm. With the position tolerance of 15 cm, all data points of the TDA were concentrated within 1 second.

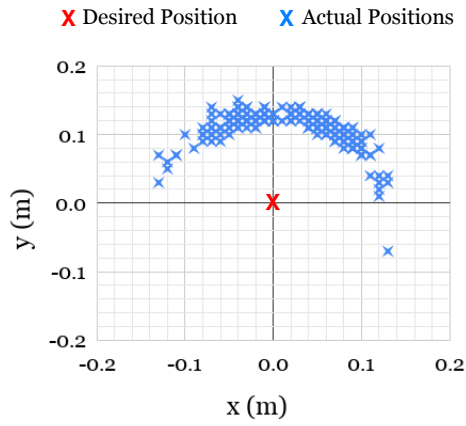
Figure 5.13 depicts the distributions of the maximum TDA in both values of position tolerance: 10 cm and 15 cm. Table 5.1 presents the numerical results over the mean, the median, and the standard deviation of the TDAs. The graph and the table show that, the TDAs with position tolerance of 10 cm were more spread out than the



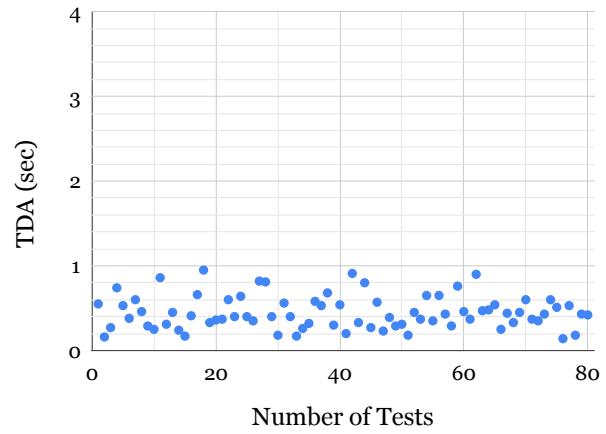
(A) Arrival positions with position tolerance of 10 cm.



(B) TDAs with position tolerance of 10 cm.



(C) Arrival positions with position tolerance of 15 cm.



(D) TDAs with position tolerance of 15 cm.

FIGURE 5.12: A trade-off between simultaneous arrival reliability and position tolerance.

TDAs with position tolerance of 15 cm. It can be concluded that, SADWA presents a trade-off between the simultaneous arrival reliability and the accuracy on target arrival.

It should be noted that this section does not present any results on the arrival orientations. As assumed in Subsection 3.1.2, SADWA does not search through the rotational velocity search space. As a result, SADWA only guarantees simultaneous arrival at the target's position only, not the orientation. In the future work, the orientation should be incorporated to better satisfy orientation-critical missions.

TABLE 5.1: Comparison between position tolerance of 10 cm and 15 cm.

	Position tolerance 10 cm	Position tolerance 15 cm
Mean (sec)	0.57	0.45
Median (sec)	0.39	0.41
Standard Deviation (sec)	0.60	0.19

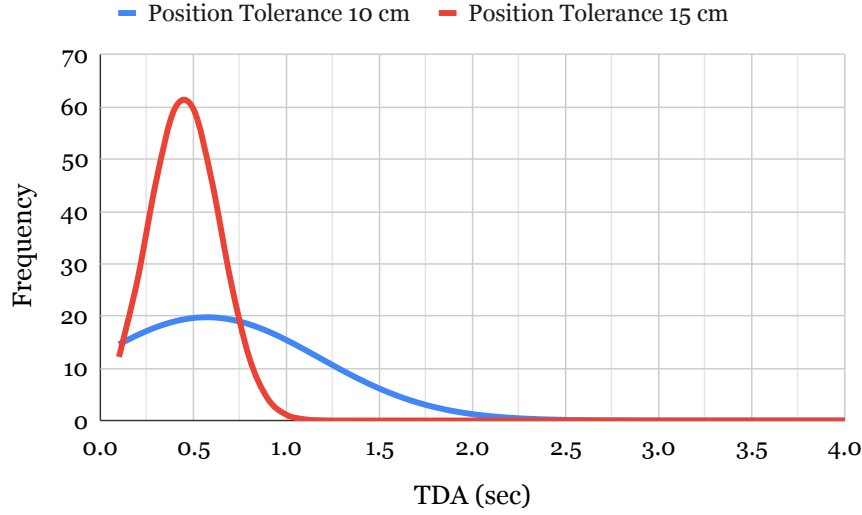


FIGURE 5.13: The distribution of the TDAs in simultaneous arrival tasks.

5.4 DISTRIBUTED MULTI-ROBOT COORDINATION FRAMEWORK

This section provides the results on the implementation of the DMRC framework proposed in Section 3.2. Results to showcase the ability of task synchronisation under normal operation and disturbances also provided.

5.4.1 TASK SYNCHRONISATION

Even though a new task is desired to be activated simultaneously across the robots, in a practical point of view, there are latencies coming from network communication and algorithm's complexity. Such latencies introduce a relatively small time difference between a new task activated in two robots which, in this project, is called *synchronisation latency*. As there are multiple robots present in the team, the maximum synchronisation latency across the robots is chosen as a performance measurement for task synchronisation. The less the maximum synchronisation latency, the better the performance of task synchronisation.

Figure 5.14 shows the maximum synchronisation latencies over 87 missions with 174 tasks carried out in both simulation and physical environment. As can be seen from the graph, the maximum synchronisation latencies were all within 0.8 second. Specifically, the average latency was 0.42 second. Figure 5.15 presents the maximum

synchronisation latency distribution. It can be seen that for most of the cases, the latencies were in the range of 0.4 second to 0.45 second.

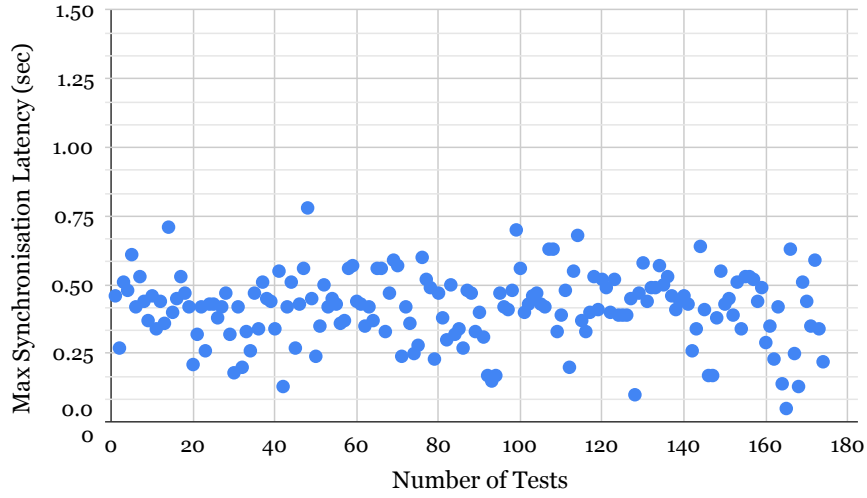


FIGURE 5.14: Maximum synchronisation latencies data points.

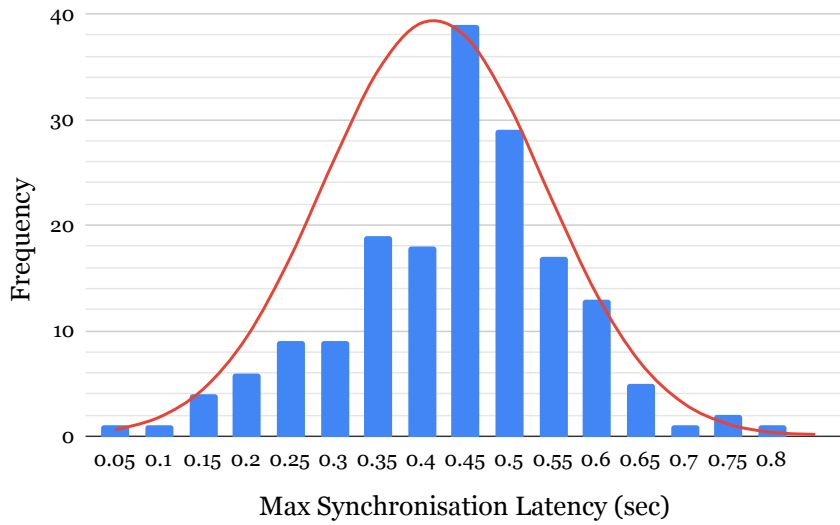
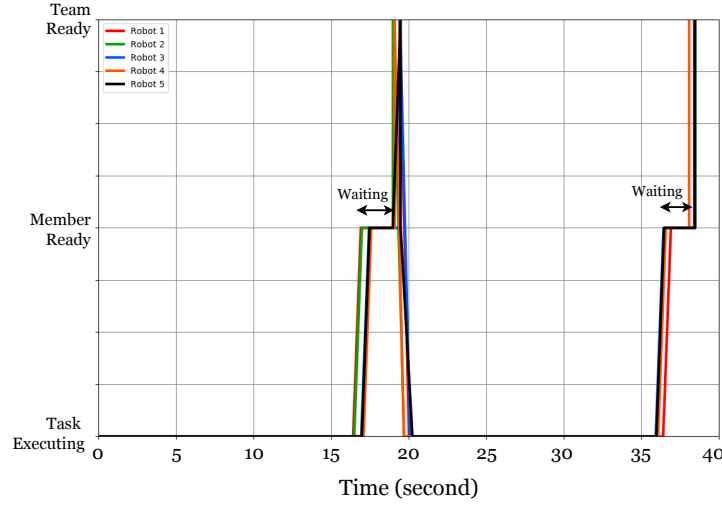


FIGURE 5.15: A histogram of synchronisation latencies.

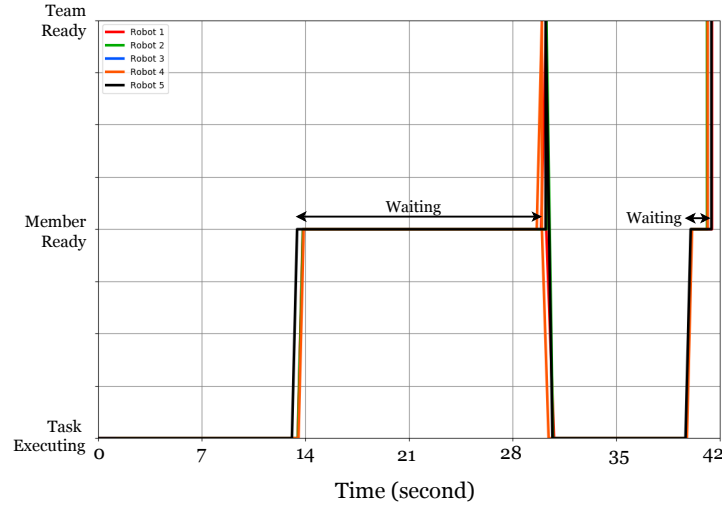
5.4.2 TEAM RESILIENCE

Team resilience is reflected by how reliable and robust the team is to achieve the mission under different disturbances.

There introduce three possible states in a cooperative task inheriting from the DMRC framework: (1) task executing - in which a task is being carried out, (2) member ready - in which a member has been completed the task and is waiting for other members, and (3) team ready - in which all members are completed the task and switching to execute another new task.



(A) Under normal operation.



(B) Under events of member failure.

FIGURE 5.16: Operation of expiry removal in a cooperative navigation task.

Figure 5.16 shows the states of a cooperative mission consisted of two tasks under normal condition (without disturbances) and under the failure of Robot 5. As the task being assigned was simultaneous arrival navigation, upon the event of target arrival, each robot should only wait others for a short period of time⁴ before the team starts executing another new task. These behaviours are indicated in Figure 5.16(a). However, under the events of member failure, it is expected that the waiting time at each robot should be longer⁵ to ensure failure did occur, not just temporary information loss. Figure 5.16(b) illustrates the state of a cooperative mission under the failure of Robot 5. As depicted, the active robots (Robot 1, Robot 2, Robot 3 and Robot 4) waited Robot 5 for 15 seconds. After this amount of time, expiry removal occurred; Robot 5 was removed out of the associated array in each active robots. The team could

⁴The waiting time under normal condition is the total time of the TDA in SADWA and the maximum synchronisation latency in DMRC framework.

⁵The DMRC framework is implemented such that the waiting time is a user-specified value.

still execute the new mission without Robot 5. It can be concluded that, the member failure handling's algorithm (Algorithm 4) enables the team to be resilient under the events of member failure.

The whole process described above is presented as a program log in Figure 5.17. It should be noted that ROS, so far, has not been supported human-readable timestamps. The timestamps provided in the log, instead, are the UNIX epoch time.⁶

```
[ INFO] [Timestamp: 1590061341.515679130]: Detected 4 neighbours currently in the network
(remove 0 agent(s) from the last mission).
[ INFO] [Timestamp: 1590061341.564274163]: Waiting for localization server...
[ INFO] [Timestamp: 1590061341.773649913]: Localization server started.
[ INFO] [Timestamp: 1590061341.787424712]: [New Mission] Localize with speed 0.70 rad/s,
covariance 0.030, timeout 40.0s]
[ INFO] [Timestamp: 1590061357.061123068]: [Status: SUCCEEDED] Localization succeeded.
[ INFO] [Timestamp: 1590061377.714447094]: Team ready.
[ INFO] [Timestamp: 1590061379.671807679]: Detected 4 neighbours currently in the network
(remove 0 agent(s) from the last mission).
[ INFO] [Timestamp: 1590061379.749104565]: Waiting for move_base server...
[ INFO] [Timestamp: 1590061380.076661737]: move_base server started.
[ INFO] [Timestamp: 1590061380.094694645]: [New Mission] Navigate to [-3.01, 0.60, 1.00]
[ INFO] [Timestamp: 1590061429.159472047]: [Status: SUCCEEDED] Navigation succeeded.
[ INFO] [Timestamp: 1590061474.370317023]: [Timeout] Remove ID 5. Last data: 40.054000s
ago.
[ INFO] [Timestamp: 1590061476.932700431]: Team ready.
[ INFO] [Timestamp: 1590061477.870270792]: Detected 3 neighbours currently in the network
(remove 1 agent(s) from the last mission).
[ INFO] [Timestamp: 1590061477.907358419]: Waiting for move_base server...
[ INFO] [Timestamp: 1590061478.235233495]: move_base server started.
[ INFO] [Timestamp: 1590061478.253160090]: [New Mission] Navigate to [-5.02, 1.00, 1.00]
[ INFO] [Timestamp: 1590061508.462994486]: [Status: SUCCEEDED] Navigation succeeded.
[ INFO] [Timestamp: 1590061512.249397097]: Team ready.
[ INFO] [Timestamp: 1590061513.927773863]: All task completed.
```

FIGURE 5.17: Program log of a robot under the event of another member's failure.

⁶The UNIX epoch time is defined as the number of seconds have elapsed since the UNIX epoch (00:00:00 UTC 1 January 1970); minus leap seconds. One could use a UNIX epoch time converter such as <https://www.epochconverter.com> to convert the timestamps to human-readable format.

CONCLUSION

6.1 RESEARCH SUMMARIES

This research aimed to study various techniques in the field of AI and information technology to allow a multi-robot system to carry out robustly and reliably cooperative missions under disturbances. Based on the analysis over various cooperative navigation missions enabled by our proposed techniques, it can be concluded that a distributed multi-robot system, with the aids from computer networks and basic AI techniques in navigation, can exhibit coordinated behaviours (simultaneous arrival on targets) under disturbances (dynamic obstacles, events of member failure). Several highlights on this research are summarised in this section.

A ROS Localisation Stack for autonomous localisation was implemented in this research. Localisation is done by performing a set of strategies on the KLD-Sampling Monte Carlo Localisation. Testing in simulation exhibited desired behaviours where the robots could localise correctly most of the time when using a relatively large number of particles. However, several limitations existed when deploying to physical environment. The Raspberry Pi 3 Model B+ in the Turtlebot3 Burger was not able to handle more than 2800 particles for localisation using KLD-Sampling Monte Carlo Localisation. This poses a huge constraint to the testbed whose operations are performed in a large environment. Further research is needed to optimally use the KLD-Sampling Monte Carlo Localisation for better performance.

In this research, a new technique called SADWA was proposed to incorporate simultaneous arrival to the Dynamic Window Approach - a renowned motion planning technique for obstacle avoidance. SADWA showed promising results in both simulation and physical environment, where static/dynamic obstacles were reliably avoided, events of member failure were handled, yet still ensure simultaneous arrival. The average time difference of arrival was 0.45 second. However, SADWA does not take into account the orientation at arrival. We suggest that, to ensure simultaneous arrival on the correct orientation, one can extend the SADWA's objective function such that it incorporates rotational velocities during the navigation mission.

To systematise coordination for a distributed multi-robot system, a framework named DMRC framework was implemented. It serves as an abstract interface for tasks to be inherited from. DMRC framework is able to synchronise tasks across the robots, and handle events of member failure, yet still ensure the mission can be completed. Testing in both simulation and physical environment showed expected results where the DMRC framework can handle various scenarios of member failure with the average synchronisation latency of 0.4 second.

With the aim of building a testbed to verify the proposed techniques, many different topics in computer networking were explored to ensure a robust communication medium was established to provide the testbed with reliable communications among team members. The testbed has deployed successfully an ad hoc network, along with a BATMAN-adv routing protocol. The proposed techniques were verified in the testbed with a team of 3 robots. The testbed was configured for mass-deployment, which means configuring a robot to add to the testbed requires only minor modifications. Due to the time constraint put on this research, ad hoc network along with BATMAN-adv routing protocol have not been fully tested. Further research is needed to provide a comparative study between different routing protocols' performance with an ad hoc network.

Overall, our work serves as a baseline for further ROS-based distributed multi-robot system in cooperative navigation missions.

6.2 RECOMMENDATIONS OF FUTURE WORK

Although the research showed optimistic results with the developed baseline testbed, there is plenty of room for improvement. This section recommends several potential techniques/frameworks as the future directions for the research.

In this research, we only proposed techniques to demonstrate simple coordinated behaviours. A more complex coordinated scheme, in the future, can be developed to leverage navigation missions. Robots in a team may share local information on their positions to help others with better localisation. Costmaps' information such as free area or obstacles may be shared to help a robot in case it gets stuck somewhere in the operating environment.

We attempted to use ROS2 in this research as it supports distributed network natively. Technique such as multi-master mechanism is no longer required in ROS2. Various state-of-the-art techniques in navigation and mapping such as Cartographer or DWB are integrated to ROS2 Navigation Stack. However, at the time of using, ROS2 was still under development and might be unstable even in the LTS version. Further work can utilise ROS2, if matured, to employ the advantages being stated.

CHAPTER A

SYSTEM USER MANUAL

A.1 PREREQUISITES

A.1.1 HARDWARE REQUIREMENTS

- **Turtlebot3 Burger:** is the main robot platform to be used in the testbed. The Turtlebot3 Burger will be referred to as *the robot*. Turtlebot3 Waffle can be used, but with no guarantee.
- **A laptop or a PC:** acts as a commander to specify the mission for the robots. The laptop or the PC will be referred to as *the commander*.
- **Raspberry Pi Model 3B+:** is to be configured as a network bridge, which will be referred to as *the bridge*.
- **Straight-through Ethernet cable:** is to connect the bridge with the commander.

A.1.2 SOFTWARE INSTALLATIONS

A.1.2.1 On the Commander

- (1) Install Ubuntu 16.04 LTS.
- (2) Install ROS Kinetic Desktop Full.

The installation instructions can be found at <http://wiki.ros.org/kinetic/Installation/Ubuntu>.

- (3) Install ROS Navigation Stack.

Although the Navigation Stack is supposed to be installed along with the ROS Kinetic Desktop Full, one still needs to ensure it is installed due to its criticality to the testbed's operations (discussed in Subsection 4.3.1). On the commander's terminal, type

```
sudo apt install ros-kinetic-navigation
```

(4) Install Turtlebot3 dependent packages.

These packages are critical for the robot's operations in both simulation and physical environment. One can install these packages by typing

```
$ sudo apt-get install ros-kinetic-joy
↪ ros-kinetic-teleop-twist-joy
↪ ros-kinetic-teleop-twist-keyboard ros-kinetic-laser-proc
↪ ros-kinetic-rgbd-launch ros-kinetic-depthimage-to-laserscan
↪ ros-kinetic-rosserial-arduino ros-kinetic-rosserial-python
↪ ros-kinetic-rosserial-server ros-kinetic-rosserial-client
↪ ros-kinetic-rosserial-msgs ros-kinetic-amcl
↪ ros-kinetic-map-server ros-kinetic-move-base
↪ ros-kinetic-urdf ros-kinetic-xacro
↪ ros-kinetic-compressed-image-transport
↪ ros-kinetic-rqt-image-view ros-kinetic-gmapping
↪ ros-kinetic-navigation ros-kinetic-interactive-markers
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src/
$ git clone
↪ https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git
$ git clone -b kinetic-devel
↪ https://github.com/ROBOTIS-GIT/turtlebot3.git
$ cd ~/catkin_ws && catkin_make
```

(5) Install multimaster_fkcie.

```
sudo apt install ros-kinetic-multimaster-fkie
```

It is the Multi-master Stack discussed in Subsection 4.2.2. More information on multimaster_fkcie can be found at http://wiki.ros.org/multimaster_fkcie.

(6) Install the software developed for the testbed

```
$ mkdir -p /fyp_ws/src
$ cd ~/fyp_ws/src
$ git clone
↪ https://github.cs.adelaide.edu.au/a1758405/fyp23101
$ cd ~/fyp_ws && catkin_make
```

One must be granted the permission to access the private Github repository above. To discuss about the permission, one can contact the author.

(7) Install chrony for time synchronisation

It should be noted that chrony can not be used along with ntp. To ensure proper functionality of chrony, it is recommended that one should completely remove ntp, if it is installed, by

```
sudo apt purge ntp
```

then install chrony

```
sudo apt install chrony
```

(8) Reboot the commander.

A.1.2.2 On the Robot

(1) Install Ubiquiti Raspberry Pi Image version 2019-06-19-ubiquiti-xenial-lxde

It is a ROS-customised image built upon Ubuntu 16.04. ROS Kinetic has been preinstalled in the image.¹

We strongly recommend one to allocate at least 2GB of swap space for the Ubiquiti Raspberry Pi Image after the installation is completed. We had experience system hang when installing ROS packages due to insufficient RAM in the Turtlebot3 Burger.²

(2) As the Ubiquiti Raspberry Pi Image is customised for Magni robot,³ one should disable Magni software run at start-up by typing

```
sudo systemctl disable magni-base
```

(3) Install Turtlebot3 dependent packages.

These packages are critical for controlling the robot. One can install these packages by the following commands

```
$ cd ~/catkin_ws/src
# Download packages from github
$ git clone
↳ https://github.com/ROBOTIS-GIT/hls_lfcd_lds_driver.git
$ git clone
↳ https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3.git
```

¹The image can be found at <https://downloads.ubiquityrobotics.com/pi.html>

²To add swap space, one can follow <https://www.digitalocean.com/community/tutorials/how-to-add-swap-space-on-ubuntu-16-04> to add swap space.

³More information can be found at https://learn.ubiquityrobotics.com/image_no_magni

```

# Delete some packages that are not needed for the robot
$ cd ~/catkin_ws/src/turtlebot3
$ sudo rm -r turtlebot3_description/ turtlebot3_teleop/
↪ turtlebot3_navigation/ turtlebot3_slam/ turtlebot3_example/
# Install Turtlebot3 dependent packages
$ sudo apt-get install ros-kinetic-rosserial-python
↪ ros-kinetic-tf
$ source /opt/ros/kinetic/setup.bash
$ cd ~/catkin_ws && catkin_make -j1

```

If the installation process returns no error, the Turtlebot3 Burger's preparation is done. One must run

```
roslaunch turtlebot3_bringup create_udev_rules
```

to use the USB port without acquiring root permission after the installation process is completed.

(4) Install the software developed for the testbed

```

$ mkdir -p /fyp_ws/src
$ cd ~/fyp_ws/src
$ git clone
↪ https://github.cs.adelaide.edu.au/a1758405/fyp23101
$ cd ~/fyp_ws && catkin_make

```

One must be granted the permission to access the private Github repository above. To discuss about the permission, one can contact the author.

(5) Install multimaster_fkcie.⁴

It is the Multi-master Stack discussed in Subsection 4.2.2. One can install multimaster_fkcie in the robot by typing

```
sudo apt install ros-kinetic-multimaster-fkie
```

(6) Reboot the robot.

A.2 SYSTEM CONFIGURATION

A.2.1 ON THE COMMANDER

(1) Append these line to ~/.bashrc:

⁴More information can be found at http://wiki.ros.org/multimaster_fkcie

```

source /opt/ros/kinetic/setup.bash
source ~/catkin_ws/devel/setup.bash
source ~/fyp_ws/devel/setup.bash

# Uncomment for simulation
# export ROS_MASTER_URI=http://localhost:11311
# export ROS_HOSTNAME=localhost
# export TURTLEBOT3_MODEL=burger

# Uncomment for physical robot deployment
# export ROS_HOSTNAME="10.42.0.x"
# export ROS_MASTER_URI=http://$ROS_HOSTNAME:11311
# export TURTLEBOT3_MODEL=burger
# export ROBOT_NAME=$HOSTNAME
# export ROBOT_ID='echo $ROS_HOSTNAME | cut -d . -f 4'

```

One can uncomment the corresponding chunk of code for running simulation or deploying physical robot. For the case of simulation, the commander runs only one ROS master with its localhost. For the case of physical robot deployment, 10.42.0.x is the commander's IP; the code above assigned the host name to the ROBOT_NAME and the last number of the IP to the ROBOT_ID. ROBOT_NAME and ROBOT_ID will be used for the software to be deployed on physical robot.

- (2) Source the ~/.bashrc by

```
$ source ~/.bashrc
```

then close all opening terminals so that the .bashrc takes effect.

- (3) For the robot to use multimaster_fkie, multicast must be enabled. To permanently enable the multicast, edit the file /etc/sysctl.conf and add the following line, or uncomment it, if it already exists, and change its default value.

```
net.ipv4.icmp_echo_ignore_broadcasts=0
```

then execute

```
sudo service procps restart
```

for multicast to take effect.

- (4) To check if multicast functions properly, run a ping check

```
ping 224.0.0.1
```

If it is working, the ping should return the IP addresses of all devices joining the multicast group.

- (5) Configure the commander as the time server for time synchronisation across the robots. One can replace the default `chrony.conf` under the directory `/etc/chrony` by the following

```

1 pool 2.debian.pool.ntp.org offline iburst minpoll 2 maxpoll 4
2 keyfile /etc/chrony/chrony.keys
3 commandkey 1
4 driftfile /var/lib/chrony/chrony.drift
5 log tracking measurements statistics
6 logdir /var/log/chrony
7 maxupdateskew 100.0
8 dumponexit
9 dumpdir /var/lib/chrony
10 local stratum 10
11 allow 10.42.0/24
12 logchange 0.5
13 hwclockfile /etc/adjtime
14 rtsync
15 makestep 1.0 3

```

In Line 1, the above configuration synchronises the commander's time to an NTP time server (`2.debian.pool.ntp.org`) with the minimum and maximum allowed synchronisation time are 2^{minpoll} sec and 2^{maxpoll} sec respectively. In the configuration above, the minimum synchronisation time is 4 seconds, and the maximum synchronisation time is 16 seconds. In Line 10 and Line 11, it sets up the commander as the NTP time server, which can receive time request from any devices with the subnet of `10.42.0/24`.

- (6) Reboot the commander.

A.2.2 ON THE ROBOT

- (1) Setup an ad hoc network with BATMAN-adv as its routing protocol. Open `/etc/network/interfaces` and type in the following

```

# interfaces(5) file used by ifup(8) and ifdown(8)
# Include files from /etc/network/interfaces.d:
source-directory /etc/network/interfaces.d

```

```
# The loopback network interface
auto lo
iface lo inet loopback

auto enx827eb610be8
iface enx827eb610be8 inet manual

auto wlan0
iface wlan0 inet manual
    wireless-channel 36
    wireless-essid meshnetwork
    wireless-mode ad-hoc

auto bat0
iface bat0 inet static
    pre-up /usr/local/sbin/batctl if add wlan0
    address 10.42.0.x
    netmask 255.255.255.0
    gateway 10.42.0.y
```

where 10.42.0.x is the robot's IP address, 10.42.0.y is the bridge's IP address.

Save the file and reboot the robot.

- (2) Replace the content of the file /etc/ubiquity/env.sh with the following

```
export ROS_IP="10.42.0.x"
export ROS_MASTER_URI="http://$ROS_IP:11311"
export TURTLEBOT3_MODEL=burger
export ROBOT_NAME=$HOSTNAME
export ROBOT_ID='echo $ROS_IP | cut -d . -f 4'
```

where 10.42.0.x is the robot's IP address as specified in the /etc/network/interfaces.

The code above automatically take the host name as the ROBOT_ID, and the last digits of the IP address as the ROBOT_ID.

- (3) Append these lines to the ~/.bashrc of each robot in the testbed

```
source /opt/ros/kinetic/setup.bash
source ~/catkin_ws/devel/setup.bash
source ~/fyp_ws/devel/setup.bash
source /etc/ubiquity/env.sh
```

After appending, source the `.bashrc` file by

```
source ~/.bashrc
```

- (4) Time synchronisation on the robot. One can replace the default `chrony.conf` under the directory `/etc/chrony` by the following

```
server 10.42.0.1 iburst prefer minpoll 2 maxpoll 3
keyfile /etc/chrony/chrony.keys
commandkey 1
driftfile /var/lib/chrony/chrony.drift
log tracking measurements statistics
logdir /var/log/chrony
maxupdateskew 100.0
dumponexit
dumpdir /var/lib/chrony
logchange 0.5
hwclockfile /etc/adjtime
rtcsync
makestep 1.0 3
```

The above configuration allows time synchronisation requests to be sent to 10.42.0.1 (NTP server). In the developed testbed, we configure the commander as the NTP server.

- (5) For the robot to use `multimaster_fk`, multicast must be enabled. To permanently enable the multicast, edit the file `/etc/sysctl.conf` and add the following line, or uncomment it, if it already exists, and change its default value.

```
net.ipv4.icmp_echo_ignore_broadcasts=0
```

then execute

```
sudo service procps restart
```

for multicast to take effect.

- (6) To check if multicast functions properly, run a ping check

```
ping 224.0.0.1
```

If it is working, the ping should return the IP addresses of all devices joining the multicast group.

- (7) Install the start-up services that will be run at robot's boot up. We provided 4 services: roscore, bringup, master-discovery and master-sync. These services can be installed by

```
$ sudo cp ~/fyp_ws/src/mission_bringup/scripts/roscore.service
↳ /etc/systemd/system
$ sudo cp ~/fyp_ws/src/mission_bringup/scripts/bringup.service
↳ /etc/systemd/system
$ sudo cp
↳ ~/fyp_ws/src/mission_bringup/scripts/master-discovery.service
↳ /etc/systemd/system
$ sudo cp
↳ ~/fyp_ws/src/mission_bringup/scripts/master-sync.service
↳ /etc/systemd/system
$ sudo cp ~/fyp_ws/src/mission_bringup/scripts/bringup
↳ /usr/sbin/
$ sudo cp ~/fyp_ws/src/mission_bringup/scripts/master-discovery
↳ /usr/sbin/
$ sudo cp ~/fyp_ws/src/mission_bringup/scripts/master-sync
↳ /usr/sbin/
$ sudo systemctl enable roscore
$ sudo systemctl start roscore
$ sudo systemctl enable bringup
$ sudo systemctl start bringup
$ sudo systemctl enable master-discovery
$ sudo systemctl start master-discovery
$ sudo systemctl enable master-sync
$ sudo systemctl start master-sync
```

- (8) Reboot the robot.

A.2.3 ON THE BRIDGE DEVICE

- (1) Setup an ad hoc network with BATMAN-adv as its routing protocol. Open /etc/network/interfaces and type in the following

```
# interfaces(5) file used by ifup(8) and ifdown(8)
# Include files from /etc/network/interfaces.d:
source-directory /etc/network/interfaces.d

# The loopback network interface
```

```

auto lo
iface lo inet loopback

auto enxb827eb5ffd5c
iface enxb827eb5ffd5c inet manual

auto wlan0
iface wlan0 inet manual
    wireless-channel 36
    wireless-essid meshnetwork
    wireless-mode ad-hoc

auto bat0
iface bat0 inet manual
    pre-up /usr/local/sbin/batctl if add wlan0

auto br0
iface br0 inet static
    bridge_ports enxb827eb5ffd5c bat0
    address 10.42.0.y
    netmask 255.255.255.0
    gateway 10.42.0.1

```

where 10.42.0.y is the bridge's IP address, 10.42.0.1 is the commander's IP address. It should be noted that, under any Linux distributions that are using NetworkManager, the Ethernet IP address will always be 10.42.0.1.

- (2) Save the below scripts under the name `enable_internet.sh`

```

#!/bin/bash
sudo batctl gw_mode server
sudo sysctl -w net.ipv4.ip_forward=1
sudo iptables -t nat -A POSTROUTING -o <ethernet_interface_name>
↳ -j MASQUERADE
sudo iptables -A FORWARD -i <ethernet_interface_name> -o bat0 -m
↳ conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
sudo iptables -A FORWARD -i bat0 -o <ethernet_interface_name> -j
↳ ACCEPT

```

Later on, one can run the script above to provide the Internet from the commander to the robots.

A.3 QUICK START

A.3.1 SIMULATION

The main package to interact with when using simulation is `mission_simulation`. In this subsection, we will guide you through the instructions to use the simulation of this testbed. These instructions below are executed in the commander.

- (1) Ensure that the chunk of code for physical robot deployment was commented out in the `~/.bashrc` file. The configuration should be

```
# Uncomment for simulation
export ROS_MASTER_URI=http://localhost:11311
export ROS_HOSTNAME=localhost
export TURTLEBOT3_MODEL=burger

# Uncomment for physical robot deployment
# export ROS_HOSTNAME="10.42.0.x"
# export ROS_MASTER_URI=http://$ROS_HOSTNAME:11311
# export TURTLEBOT3_MODEL=burger
# export ROBOT_NAME=$HOSTNAME
# export ROBOT_ID='echo $ROS_HOSTNAME | cut -d . -f 4'
```

then source the `~/.bashrc` file by typing `source ~/.bashrc`.

- (2) Firstly, one should open Gazebo simulation environment. At one terminal, type

```
roslaunch mission_simulation environment.launch
↵ world_name:=small_living_room.world open_gui:=True
```

The command above will open a Gazebo simulation environment as shown in Figure 5.1). We hard-coded the positions of 5 robots in the simulation. Five robots have the names of harry, chris, david, alice and zeping. It is likely that opening Gazebo's GUI can slow the commander down. One can specify `open_gui:=False` to run Gazebo environment in background.

We provided a program to support with simulation.

```
roslaunch mission_simulation control.py --name <robot_name>
↵ --id <robot_id> init --map_name <map_name>
```

In the command above, a map `map_name` is provided to the robot named `robot_name` with ID `robot_id` for navigation initialisation.

(3) In another terminal, run

```
roslaunch mission_simulation control.py --name harry --id 1
↪ init --map_name map_living_room
```

(4) In another terminal, run

```
roslaunch mission_simulation control.py --name chris --id 2
↪ init --map_name map_living_room
```

(5) In another terminal, run

```
roslaunch mission_simulation control.py --name david --id 3
↪ init --map_name map_living_room
```

(6) In another terminal, run

```
roslaunch mission_simulation control.py --name alice --id 4
↪ init --map_name map_living_room
```

(7) In another terminal, run

```
roslaunch mission_simulation control.py --name zeping --id 5
↪ init --map_name map_living_room
```

The above commands initialise navigation task with the map named `map_living_room` provided by default (shown in Figure 5.1). In case one designs his/her own Gazebo environment, a map of such an environment must be generated and located in `mission_simulation/map` with a different name.

The below commands are for mission initialisation.

```
roslaunch mission_simulation control.py --name <robot_name>
↪ --id <robot_id> navigate --log_name <log_path>
```

In the command above, the mission is initialised at a robot. The mission log will be saved in `log_path`.

(8) In another terminal, run

```
roslaunch mission_simulation control.py --name harry --id 1
↪ navigate --log_name log/harry_quick_start.log
```

(9) In another terminal, run

```
roslaunch mission_simulation control.py --name chris --id 2
↪ navigate --log_name log/chris_quick_start.log
```

(10) In another terminal, run

```
rosrun mission_simulation control.py --name david --id 3
↪ navigate --log_name log/david_quick_start.log
```

(11) In another terminal, run

```
rosrun mission_simulation control.py --name alice --id 4
↪ navigate --log_name log/alice_quick_start.log
```

(12) In another terminal, run

```
rosrun mission_simulation control.py --name zeping --id 5
↪ navigate --log_name log/zeping_quick_start.log
```

(13) In another terminal, run

```
roslaunch mission_simulation rviz.launch
```

Doing the steps above simulates 5 robots running in an environment. It should be noted that, one is not required to explicitly run `roscore`, as Gazebo already runs it in background.

It is worthwhile to mention the features we provided that one can use in RViz. Two options for localisation: *global localisation* and *area localisation* were developed. Global localisation is useful in case one does not know where the robot is in the environment; thus, the particles are spread out uniformly across the map. Area localisation is useful when one knows approximately where the robot is; the user can specify the area for which the particles concentrated. One can use RViz to specify targets for the team as well.

Figure A.1 shows the RViz's interface where the mission is specified. 2D Pose Estimate is to provide an initial area for the area localisation. 2D Nav Goal is to set the target for a robot. Figure A.1 also points out which button is to control which robot.

The steps to execute a navigation mission with area localisation are as follows

- (1) Click on the button 2D Pose Estimate corresponding with the robot to be localised. An arrow will appear.
- (2) Point the arrow at the area where the robot is estimated to be located. The arrow's orientation and the robot's actual orientation should not differ significantly.
- (3) Repeat for all other robots.
- (4) Click on the button 2D Nav Goal corresponding with the robot whose target to be specified. An arrow will appear.

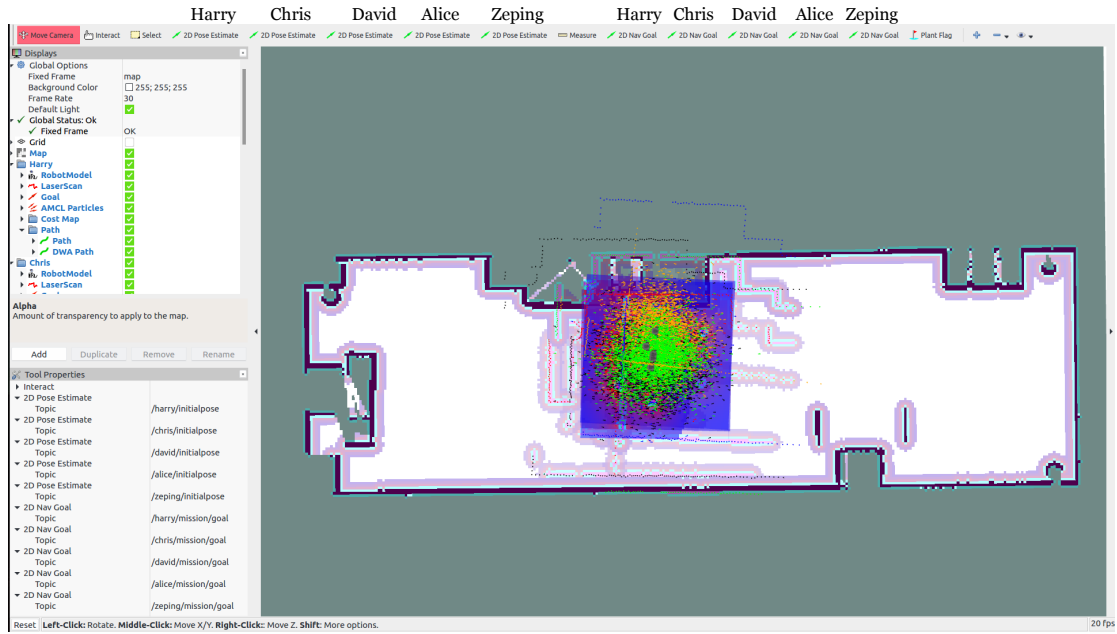


FIGURE A.1: RViz commander interface in simulation.

- (5) Click the arrow at the target to navigate to.
- (6) Repeat for all other robots.

In cases where the robots' initial areas are unknown, one can not just specify an arbitrary area as this would lead to incorrect localisation. Instead, global localisation is used in such case. The steps to execute a navigation mission with global localisation are as follows

- (1) Click on the button 2D Nav Goal corresponding with the robot whose target to be specified. An arrow will appear.
- (2) Click the arrow at the target to navigate to.
- (3) Repeat for all other robots.

In the steps above, one can specifies the targets for the team directly. The robots will localise themselves globally with the trade-off is the localisation accuracy.

For both area localisation and global localisation, the team can still navigate and arrive at the targets simultaneously.

A.3.2 PHYSICAL ENVIRONMENT

a. On the robots

When boot up the robot, several services will be run automatically. These services are for running local ROS master, robot bringup, master discovery, and master synchronisation. Before proceeding on to the instruction below, one

must ensure these services are running and return no errors. These services can be checked by

```
$ systemctl status roscore
$ systemctl status bringup
$ systemctl status master-discovery
$ systemctl status master-sync
```

The map provided to the testbed is assumed to be located at `mission_bringup/map`.

Below are the instructions on how to use the testbed on physical environment. These commands are to be run exactly in the same order at every robot.

(1) In a robot's terminal, run

```
$ ROS_NAMESPACE=$ROBOT_NAME roslaunch mission_bringup
↪ navigation.launch map_name:=<your_map_name>
```

This command will initialise the Cooperative Navigation Stack, and provide the robot with the map `your_map_name`.

(2) In another terminal, run

```
$ ROS_NAMESPACE=$ROBOT_NAME roslaunch mission_bringup
↪ mission.launch log_file_name:=<log_path>
```

This command will initialise the mission clients. The mission log will be saved to `log_path`.

b. On the commander

In the commander, no background services are running as in the case of the robot. Thus, one should manually start local ROS master, master discovery and master synchronisation.

c. In a commander's terminal, run

```
$ roscore
```

to start local ROS master.

d. In another terminal, run

```
$ roslaunch mission_bringup master_discovery
```

to start the discovery process.

e. In another terminal, run

```
$ rosrun master_sync_fkcie master_sync
```

to start the master synchronisation process. This command synchronises all topics and services from all discovered robots for the purpose of mission visualisation and controlling.

f. In another terminal, run

```
$ roslaunch mission_bringup rviz.launch
```

The robots will discover each other and establish communication links among the team. In the testbed we developed, there was 3 robots presented in the team

Two options for localisation: *global localisation* and *area localisation* were developed. Global localisation is useful in case one does not know where the robot is in the environment; thus, the particles are spread out uniformly across the map. Area localisation is useful when one knows approximately where the robot is; the user can specify the area for which the particles concentrated. One can use RViz to specify targets for the team as well.

Figure A.2 shows the RViz's interface in physical environment where the mission will be specified. 2D Pose Estimate is to provide an initial area for the area localisation. 2D Nav Goal is to set the target for a robot.

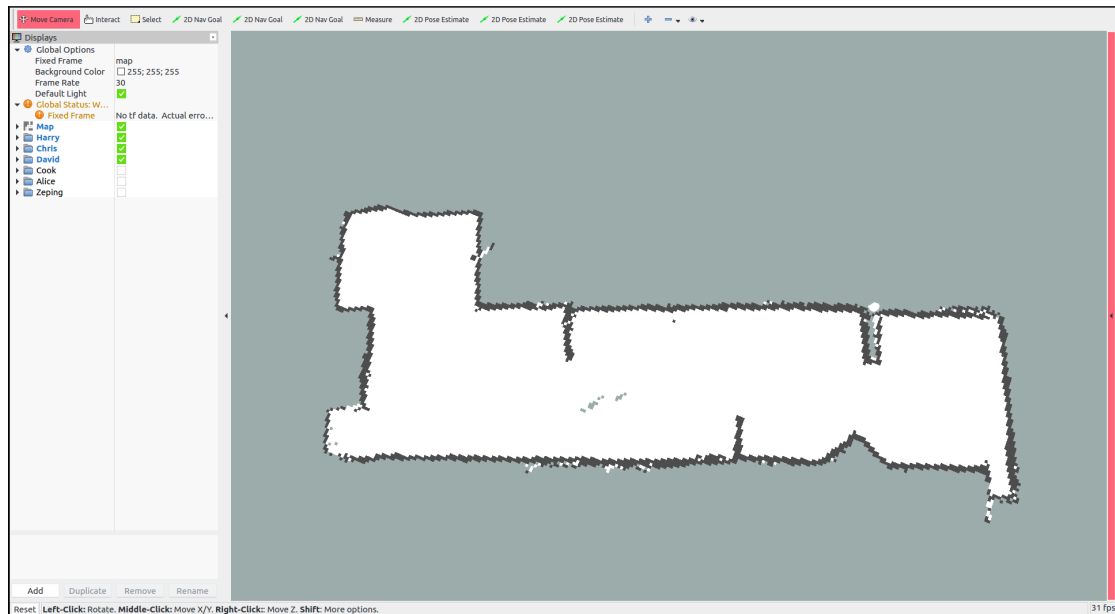


FIGURE A.2: RViz commander interface in physical environment's deployment.

The steps to execute a navigation mission with area localisation are as follows

- (1) Click on the button 2D Pose Estimate corresponding with the robot to be localised. An arrow will appear.
- (2) Point the arrow at the area where the robot is estimated to be located. The arrow's orientation and the robot's actual orientation should not differ significantly.

- (3) Repeat for all other robots.
- (4) Click on the button 2D Nav Goal corresponding with the robot whose target to be specified. An arrow will appear.
- (5) Click the arrow at the target to navigate to.
- (6) Repeat for all other robots.

In cases where the robots' initial areas are unknown, one can not just specify an arbitrary area as this would lead to incorrect localisation. Instead, global localisation is used in such case. The steps to execute a navigation mission with global localisation are as follows

- (1) Click on the button 2D Nav Goal corresponding with the robot whose target to be specified. An arrow will appear.
- (2) Click the arrow at the target to navigate to.
- (3) Repeat for all other robots.

In the steps above, one can specifies the targets for the team directly. The robots will localise themselves globally with the trade-off is the localisation accuracy.

For both area localisation and global localisation, the team can still navigate and arrive at the targets simultaneously.

A.4 TESTBED SOFTWARE PARAMETERS

The testbed software parameters are located under `mission_simulation/param` for simulation operations, and under `mission_bringup/param` for deployment in physical environment.

A.4.1 COOPERATIVE NAVIGATION STACK

This subsection presents only the parameters in the software that we developed for the testbed. Any other parameters of the original Navigation Stack can be tuned following the paper of Zheng [37].

A.4.1.1 *Simultaneous Arrival using DWA*

One can modify the parameters of Simultaneous Arrival using Dynamic Window Approach (SADWA) in the file `cooperation_local_planner_burger_params.yaml`. The two most important parameters for SADWA are

- `simultaneous_arrival_tolerance` (double, default: 0.5, unit: sec)

The TDA of the team will not be exactly 0 second due to network latency and algorithm's complexity. Instead, a value of TDA is specified as the tolerance. The team will accept any value of TDA within this amount of time.

- `communication_timeout` (double, default: 6.0, unit: sec)

This is the timeout in expiry removal implemented in Algorithm 2. The failed robot will be removed out of the robot list after this amount of time.

There is a parameter for robot's ID. One should not change this value as it is automatically assigned by the last number of the robot's IP address. However, if one wishes to change it, he/she can find the parameter under `launch/move_base.launch` with the name of `robot_id`.

A.4.1.2 Localisation Stack

One can modify the parameters of the Localisation Stack in the file `mission_params.yaml`. This file will also contain parameters for the Mission Clients, which will be described in Subsection A.4.3.

- `localization_covariance` (double, default: 0.01, unit: m²)

This is the localisation tolerance ε in Algorithm 5. The robot will stop the localisation process if σ_x^2 , σ_y^2 and σ_z^2 are less than this value.

- `localization_speed` (double, default: 1.0, unit: rad/sec)

Rotational speed to perform localisation strategy.

- `localization_timeout` (double, default: 30.0, unit: sec)

After this amount of time, if a robot can not localise, it will stop and cancel the localisation task.

A.4.2 DISTRIBUTED MULTI-ROBOT COORDINATION FRAMEWORK

One can modify the parameters of the Distributed Multi-Robot Coordination (DMRC) framework in the file `multi_agent_communication_params.yaml`.

- `simulation` (bool, default: false, unit: none)

Specify if one is running this framework under simulation or physical environment.

- `agent_list` (list, default: empty list, unit: none)

In case of running the DMRC framework under simulation, one needs to input a list of all host names in the testbed. This list will be used to establish communications as `multimaster_fkcie`. Example: `agent_list: [harry, david, chris, rasp.bridge, rasp.monitor]` specifies a testbed with the presense of robot harry, robot david, robot chris, a bridge `rasp.bridge` and a commander `rasp.monitor`.

In case of running this framework under physical environment, this parameter does not need to be specified. Example: `agent_list: []`.

- `ignore_pattern_list` (list, default: empty list, unit: none)

Any host names whose patterns are specified in this list will be ignored (not establish communication with). Example: With `ignore_pattern_list: [bridge, monitor]`, the team will just establish communications among the robot harry, david, and chris.

A.4.3 MISSION CLIENTS

- `update_progress_rate` (double, default: false, unit: Hz)

The rate in which each robot reports its own information to others.

- `team_localization_wait_timeout` (list, default: 30.0, unit: sec)

This value implements the timeout in Algorithm 4. Upon the failure of a robot during localisation, the rest of the team will wait the failed robot for an amount of `team_localization_wait_timeout`. After this period, the failed robot will be removed out of the robot's list. The active robots can keep executing the mission.

- `team_navigation_wait_timeout` (list, default: 7.0, unit: sec)

This value implements the timeout in Algorithm 4. Upon the failure of a robot during navigation, the rest of the team will wait the failed robot for an amount of `team_localization_wait_timeout`. After this period, the failed robot will be removed out of the robot's list. The active robots can keep executing the mission.

BIBLIOGRAPHY

- [1] A. S. Tanenbaum and M. v. Steen, *Distributed Systems: Principles and Paradigms (2nd Edition)*. USA: Prentice-Hall, Inc., 2006.
- [2] N. A. Lynch, *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.
- [3] A. Dhakal and X. Cui, “Blockchain and Smart Contracts for Internet of Things: A systematic literature review,” Apr. 2018.
- [4] S. K. Sarkar, *Ad Hoc Mobile Wireless Networks Principles, Protocols, and Applications*, 2nd ed. Boca Raton: CRC Press.
- [5] R. Siegwart and I. R. Nourbakhsh, *Introduction to Autonomous Mobile Robots*. USA: Bradford Company, 2004.
- [6] Y. Pyo, H. Cho, L. Jung, and D. Lim, *ROS Robot Programming*. ROBOTIS, Dec. 2017.
- [7] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.
- [8] R. E. Kalman, “A New Approach to Linear Filtering and Prediction Problems,” *Journal of Basic Engineering*, vol. 82, no. 1, pp. 35–45, Mar. 1960.
- [9] S. J. Julier and J. K. Uhlmann, “A New Extension of the Kalman Filter to Nonlinear Systems,” *Signal Processing, Sensor Fusion, and Target Recognition VI*, pp. 182–193, 1997.
- [10] F. Dellaert, D. Fox, W. Burgard, and S. Thrun, “Monte Carlo Localization for Mobile Robots,” in *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*, vol. 2, 1999, 1322–1328 vol.2.
- [11] D. Fox, “KLD-Sampling: Adaptive Particle Filters,” in *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, ser. NIPS’01, Vancouver, British Columbia, Canada: MIT Press, 2001, pp. 713–720.
- [12] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1959.
- [13] P. E. Hart, N. J. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, Jul. 1968.

BIBLIOGRAPHY

- [14] D. V. Lu, D. Hershberger, and W. D. Smart, "Layered costmaps for context-sensitive navigation," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014, pp. 709–715.
- [15] D. Fox, W. Burgard, and S. Thrun, "The dynamic window approach to collision avoidance," *IEEE Robotics Automation Magazine*, vol. 4, no. 1, pp. 23–33, Mar. 1997.
- [16] C. Rsmann, F. Hoffmann, and T. Bertram, "Integrated online trajectory planning and optimization in distinctive topologies," *Robot. Auton. Syst.*, vol. 88, no. C, pp. 142–153, Feb. 2017.
- [17] R. Li and Y. Shi, "Simultaneous Arrival of Multiple Agents with Different Targets," in *Proceedings of the 33rd Chinese Control Conference*, Jul. 2014, pp. 1674–1679.
- [18] T. McLain and R. Beard, "Coordination variables, coordination functions, and cooperative timing missions," *Journal of Guidance Control and Dynamics*, vol. 28, Nov. 2002.
- [19] Y. Dong and J. Huang, "Leader-Following Rendezvous with Connectivity Preservation of Single-Integrator Multi-Agent Systems," in *2012 12th International Conference on Control Automation Robotics Vision (ICARCV)*, 2012, pp. 1686–1690.
- [20] F. Xiao, L. Wang, and T. Chen, "Connectivity Rreservation for Multi-agent Rendezvous with Link Failure," *Automatica*, vol. 48, pp. 25–35, Jan. 2012.
- [21] Z. Kan, J. Klotz, E. L. P. Jr, J. M. Shea, and W. E. Dixon, "Decentralized Rendezvous of Nonholonomic Robots with Sensing and Connectivity Constraints," *Decentralized Rendezvous of Nonholonomic Robots With Sensing and Connectivity Constraints*, 2016.
- [22] M. B. Dias, R. Zlot, N. Kalra, and A. Stentz, "Market-based multirobot coordination: A survey and analysis," *Proceedings of the IEEE*, vol. 94, no. 7, pp. 1257–1270, Jul. 2006.
- [23] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach (6th Edition)*, 6th. Pearson, 2012.
- [24] P. Jacquet, P. Muhlethaler, T. Clausen, A. Laouiti, A. Qayyum, and L. Viennot, "Optimized Link State Routing Protocol for Ad Hoc Networks," in *Proceedings. IEEE International Multi Topic Conference, 2001. IEEE INMIC 2001. Technology for the 21st Century.*, 2001, pp. 62–68.

BIBLIOGRAPHY

- [25] D. Johnson, N. Ntlatlapa, and C. Aichele, “A Simple Pragmatic Approach to Mesh Routing using BATMAN,” in *In 2nd IFIP International Symposium on Wireless Communications and Information Technology in Developing Countries, Pretoria, South Africa*, 2008.
- [26] D. Murray, M. Dixon, and T. Koziniec, “An Experimental Comparison of Routing Protocols in Multi Hop Ad Hoc Networks,” in *2010 Australasian Telecommunication Networks and Applications Conference*, 2010, pp. 159–164.
- [27] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th. USA: Prentice Hall Press, 2014.
- [28] “IEEE Standard for Information Technology — Telecommunications and Information Exchange Between Systems Local and Metropolitan Area Networks—Specific Requirements - Part II: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications,” *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)*, pp. 1–3534, 2016.
- [29] C. Systems, “Traffic regulators: Network Interfaces, Hubs, Switches, Bridges, Routers, and Firewalls,” 1999.
- [30] D. L. Mills, “Internet time synchronization: The Network Time Protocol,” *IEEE Transactions on Communications*, vol. 39, no. 10, pp. 1482–1493, Oct. 1991.
- [31] A. Tiderko, F. Hoeller, and T. Röhling, “The ROS Multimaster Extension for Simplified Deployment of Multi-Robot Systems,” *Studies in Computational Intelligence Robot Operating System (ROS)*, Jan. 2016.
- [32] F. Sergi Hernandez Juan, “Multi-Master ROS systems,” 2015.
- [33] L. Harte, *Introduction to Data Multicasting, IP Multicast Streaming for Audio and Video Media Distribution*. Althos, 2008.
- [34] M. Ben-Ari and F. Mondada, “Elements of robotics,” *Elements of Robotics*, Jan. 2018.
- [35] E. Marder-Eppstein. (2019). Navigation stack, [Online]. Available: <http://wiki.ros.org/navigation> (visited on 05/26/2020).
- [36] D. Fox, “Adapting the Sample Size in Particle Filters Through KLD-Sampling,” *The International Journal of Robotics Research*, vol. 22, no. 12, pp. 985–1003, 2003.
- [37] K. Zheng, *ROS Navigation Tuning Guide*, 2017. arXiv: [1706.09068](https://arxiv.org/abs/1706.09068) [cs.R0].