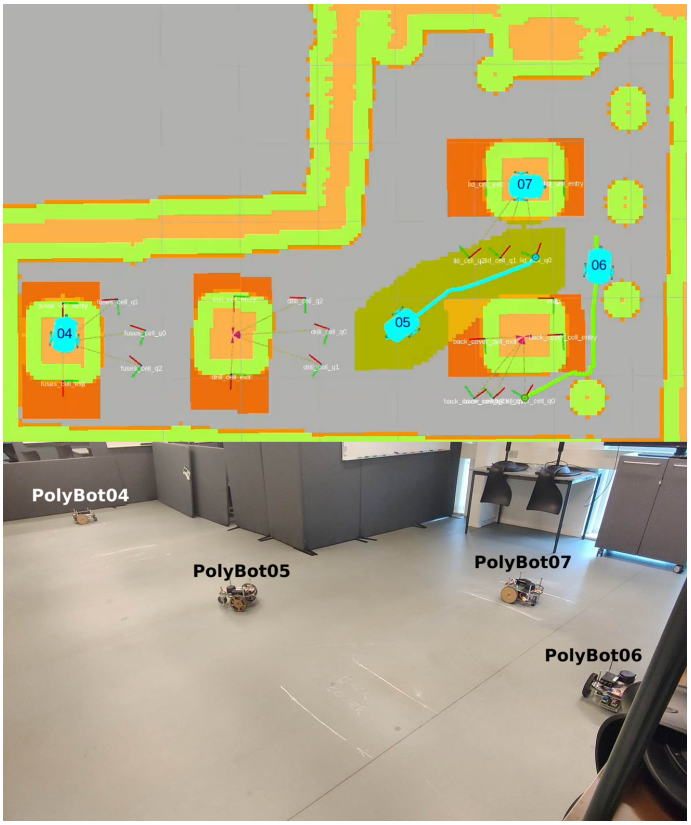
SPICE: Swarm Production Implemented in a re-Configurable Environment

Project Report

Group 1050 ROB10



Aalborg University
Electronics and IT

Electronics and IT
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
SPICE: Swarm Production Implemented in a re-Configurable Environment

Theme:
Swarm Production

Project Period:
Spring Semester 2023

Project Group:
Group 1050

Participant(s):
Anders Clement
Jonas Andersen
Oliver Beyer Lauritsen

Supervisor(s):
Casper Schou

Page Numbers: 80

Date of Completion:
June 2, 2023

Abstract:

In this work, the initial steps for implementing the novel production paradigm of swarm production have been made. The system consists of 5 PolyBot mobile robots acting as carrier robots for material handling, 4 virtual workstations for part processing, and a central swarm manager. Three different traffic management strategies have been implemented and tested, one of these being the novel concept of dynamic queue positions. Testing of the traffic management strategies shows that those considering the temporal dimension are more suited for swarm production, similar to solutions found for the multi-agent-path-finding/multi-agent-pickup-and-delivery problem. Furthermore, dynamic queue points were found to potentially improve the scalability of a swarm production system. Future work includes scaling up the number of robots, and further iteration of the system.

Contents

List of Figures

Acronyms

ROS2	Robot Operating System 2
ERP	Enterprise Resource Planning
MES	Manufacturing Execution System
MLT	Manufacturing Lead Time
WIP	Work-In-Progress
MAPF	Multi-Agent Path Finding
EoA*	Extenstions of A*
ICTS	Increasing Cost Tree Search
CBS	Conflict-Based Search
CP	Constraints Programming
MAPD	Multi-Agent Pickup and Delivery
RPi4	Raspberry Pi 4
FPSP	Fixed Priority Spacial Planner
STAP	Space-Time A* Planner
FPSP-SQ	Fixed Priority Spacial Planner with Static Queues
FPSP-DQ	Fixed Priority Spacial Planner with Dynamic Queues
BT	Behavior Tree

Preface

Aalborg University, June 2, 2023

Anders Clement
<acleme18@student.aau.dk>

Jonas Andersen
<jan18@student.aau.dk>

Oliver Beyer Lauritsen
<olauri18@student.aau.dk>

Chapter 1

Introduction

In recent years, production paradigms have been changing to accommodate higher demand for flexibility in production, enabling mass customization, and enabling more efficient production.
Such an example is reconfigurable manufacturing systems, which enable rapid topology changes of the production floor, in order to alter the production flow, making line production reconfigurable. [1]
Another approach is matrix production, in which workstations are placed in a grid, and workpieces and required materials are delivered to workstations by mobile robots. This enables flexible routing and redundancy in production. [2]
Based on these two production paradigms, Schou et al. [3] envision swarm production as the next production paradigm to arise, combining the reconfigurability of reconfigurable manufacturing systems with the flexible routing of matrix production, by allowing mobile robots to transport workpieces and materials, with workstations being mobile as well. This would allow for unprecedented flexibility but poses several challenges in designing and implementing such a system.
These challenges include: How to determine an appropriate topology, how to coordinate movement between both the mobile robots carrying workpieces and materials and the likely larger workstations, and how to schedule tasks for both the mobile robots and workstations.
This project will investigate traffic management in a swarm production setting, by implementing a swarm production testbed in order to gain a practical understanding of the issues at hand.
As an overview of this report, Chapter 2 will investigate related works, and further describe the focus of this project. The designed system is described in Chapter 3, where it is explained how the workstations and mobile robots of the swarm production are set up, and how three different traffic management strategies are designed. Based on this design, Chapter 4 describes the implemented system and the hardware on which it is built. Thereafter, Chapter 5 describes how the imple-

mented system was tested, in order to understand the efficacy of the implemented traffic management systems, which shows the interdependency of traffic management to the rest of the swarm production system. The results from these tests are then discussed. Lastly, Chapter 6 will discuss the overall project and outline possible future work for improving the performance of the implemented system, before concluding on the key findings in this project.

Chapter 2

Related Works

This chapter will look into related works regarding swarm production and what aspects define swarm production. Then a closer look into existing methods for how to manage a swarm, and lastly, the delimitation for this project.

2.1 Swarm Production

An emerging field of research is that of swarm production, as described by Schou et al. [3]. In short, swarm production is a method of achieving the highest degree of product routing flexibility and a changing topology to optimize for the current production.
Schou et al. define swarm production as a production system consisting of multiple free-moving swarm agents with different purposes. They propose an example of swarm production with two types of mobile agents: Part transport robots, and process unit robots. The transport agents are simple and cheap mobile robots, that constitute the majority of the swarm, carrying one product per robot to be processed. The process units are bigger and more advanced mobile robots, that allow for the processing of the product carried by the transport agents.
Although swarm production is inspired by the field of swarm robotics, the fields have some differences as discussed by Hamann:
"Sometimes multi-agent or multi-robot systems are difficult to separate conceptually from swarm robotics systems. Multi-agent systems, however, often rely on the distribution of or access to global information, broadcasts, sophisticated communication protocols (e.g., negotiations) that require reliable robot–robot communication, explicit assignments of roles that require robots to identify individual robots or to know the total swarm size. These systems typically do not scale also because of applied, non-scalable technologies such as Bluetooth or WLAN (wireless local area network, aka WiFi) [4].
Unlike swarm robotics in general, in swarm production, there is a clear global goal of the swarm: Produce a set of products as demanded by e.g. the Manufacturing

Execution System (MES). Additionally, it is preferred to use all available central information as long as it improves the production outcome or the value proposition of the production.
New communication technology like 5G allows for considerably better connectivity, reducing common communication problems in dense swarms of robots. Several works investigate improved communication, thus enabling larger robot swarms [5, 6].
Even though swarm production in its current state of research typically utilizes central information and global knowledge to optimize topology and plan routes for robots, it is not hard to envision a future in which true swarm robotics with no centralized control is able to produce the wanted products in an efficient self-organized manner. However, considering the novelty of swarm production approaches, utilizing central approaches will be mostly considered, as a decentralized approach is not yet practical.
A final consideration for swarm production is how traffic management, the physical size and shape of the factory floor, task allocation, and the workstation topology are all interdependent. If workstations are moved closer in order to reduce pathing distance, but the current production tasks have very short cycle times, the high number of transport agents may lead to inefficient traffic management, in which case longer pathing distances can be more efficient by spreading out traffic for the overall production rate. Therefore, ideally, a combined algorithm could be developed to solve these issues simultaneously. However, due to the high complexity of the individual problems, the complexity of the combined problem scales even worse. Therefore, it is instead preferred to solve the problems individually, yet consider the interdependency of these systems during development.

The following sections, in this chapter, will go through related works, regarding the necessary technologies that facilitate the creation of a swarm production system.

2.2 Manufacturing Systems

Swarm production can be categorized as a fully automated flexible production system, due to the high flexibility of product processing and its capacity to operate without human intervention for multiple cycles. This case of manufacturing system excels at producing a variety of parts or products, with little to no changeover times, and the system is able to facilitate the production of various schedules of products with high customizability, without the need for them to be a part of a single batch. However, the high flexibility does set some requirements for the products: The differences between parts processed by the system are not significant [7]. In other words, the produced products should to some extent share processes and workstations, as otherwise workstations would be badly utilized.

A manufacturing system consists of many different sub-systems that have to work together. Groover et al. [7] has categorized five levels in a manufacturing system that each control different aspects of the system. These levels are, starting from the lowest level: The device level, machine level, cell or system level, plant level, and enterprise level.
The device level and machine level control hardware functions. The device level is concerned with low-level hardware such as sensors, and actuators, whereas the machine level is concerned with controlling, in the case of swarm production, workstations, and carrier robots.
The cell or system level represents a group of machines from the machine level, and its focus is to manage the coordination between the different machines in the system, machine loading, and material handling.
The plant level manages order processing, quality control, and process planning. The MES often works on this level.
The highest level is the enterprise level which controls functions necessary to manage the company, such as marketing and sales, accounting, design, and research. Enterprise Resource Planning (ERP) works on this level [7].
Where based on the layers of swarm production defined by Avhad et al. [8], the machine level controls individual robots in the fleet, and the cell or system level contains the swarm manager, and topology manager.
The swarm manager's job is to allocate tasks to different carrier robots in the system, allocate workstations for the carrier robots to move to, and for a centralized planning approach, plan a collision-free path to the workstations as well. In the case of a decentralized planning approach, the planning is left to the machine level. The purpose of the topology manager is to manage the layout of the production floor and is explained in more detail in Section 2.3.

Evaluation of Manufacturing Systems

In order to track the performance of a manufacturing system, several metrics can be used. Commonly used metrics are cycle time, production capacity, utilization, Manufacturing Lead Time (MLT), and Work-In-Progress (WIP).
Cycle time describes the time of a single processing step, where the manufacturing of a product often involves several processing steps. The cycle time includes the actual processing time, tool handling time, and work part handling time. In other words, it is defined as the time it takes for a part to start being processed until the next processing step begins [7].
Production capacity is then defined as the maximum production rate given some assumed operating conditions e.g. the number of workstations available, and operating hours. The utilization then describes the proportion of time that a workstation is used in relation to the available time [7].

MLT describes the total amount of time a product spends going through the system, including delays and time spent in buffers/queues, whereas the WIP metric describes the number of products that are currently being processed or are between processing in the manufacturing system. A high amount of WIP will often lead to a longer MLT. It is often a concern to minimize the MLT [7].
In order to optimize these metrics, all layers of the manufacturing system have to work together, therefore it is important to consider how these interact in a complete system.

2.3 Workstation Topology

Unlike other manufacturing paradigms, workstations are intended to be either continuously relocated, or more likely in the close future, relocated daily or per batch. However, the complexity of calculating an optimal topology quickly reaches unmanageable proportions. Instead, near-optimal solutions are sought after.
To the best of the authors' knowledge, little work address finding the topology of a swarm production. However, Avhad et al. [8] introduce a framework for swarm production, in which MES and ERP systems interface with a planning and scheduling layer. In this layer, a topology manager is intended to find topologies for upcoming batches.
In order to select a topology, the total distance traveled by robots is considered, using the Euclidean distance between workstations. New topologies are found using spring topology, and if for a new batch, the temporal cost of reconfiguring is less than what can be saved with the new topology, the factory floor is reconfigured to the new topology.

2.4 Swarm Management

Managing a fleet of robots is not trivial and consists of multiple routing and planning problems of high computational complexity. These problems include figuring out which product task to assign to what carrier robot, determining which workstations to go to for processing, creating a route visiting all necessary workstations for the task at hand, and creating a plan or trajectory without collisions or deadlocks with other robots in the system.

2.4.1 Task Allocation

Bullo et al. [9] have investigated the problem of dynamical vehicle routing, that is, the problem of finding optimal multi-vehicle routes to perform tasks that are generated over time. They review multiple strategies and create their own algorithms

for allocating tasks and creating routes in a dynamic environment for both centralized and decentralized systems with considerations for task time constraints and task priorities, taking basis in queuing theory. Their solution promises performance guarantees for several important problems, however, they mention that work is still needed in order to achieve more general performance guarantees.
Hasgül et al. [10] has implemented a project-oriented task scheduling approach of a simulated multi-robot system. In their approach, they have a centralized "administrator agent" whose purpose is to distribute tasks to the available robots in the system. They use critical path methods, and resource leveling in order to allocate tasks to the agents in the system. This approach takes into account the uncertainties of a real-world scenario and will dynamically re-plan task allocations if tasks are delayed by a threshold value. Tests in a simulated environment showed that the developed method is viable for task scheduling in a multi-robot system.
Managing the planning of individual robots paths will be explored in the next sections.

2.4.2 Traffic Management

In order for robots to traverse a factory or warehouse, they must be able to move around without colliding with other agents or the environment. A commonly used method for avoiding collisions in factories and warehouses is by utilizing traffic rules [11]. These ensure that every individual knows how to act when driving past others or when crossing intersections. However, a drawback of traffic rules is that they require the agents to stop and wait depending on other agents and to take longer routes to their goal in order to adhere to the traffic rules in the case of one-way streets. Additionally, traffic rules may need to be reconsidered if the topology of the shopfloor changes, which is expected to happen often in swarm production.
Digani et al. [12, 13] have created an algorithm that automatically can create traffic rules for any given industrial warehouse layout by creating several one-way corridors and intersections. They also created an algorithm for how the robots can traverse the intersections collision-free. They have not tested their algorithm in a factory environment, however, having the functionality to automatically create traffic rules in a factory environment would be necessary in order to use traffic rules for swarm production.
Another method for dynamically creating traffic rules has been made by Mai et al. [14] who created an algorithm that plans collision-free trajectories for multiple agents. It utilizes a conflict-based search approach to create a traffic rule at the location of the conflict in order to resolve it. The traffic rule is a simple right of way given to the direction of one of the robots, which is chosen by minimizing the additional path distance that is added to all robots currently planning to go through the location where the conflict is. The traffic rule is kept and used for all

robots passing that location afterward. They do this in discretized space assuming holonomic robots.
Lastly, Chung et al. [15] present a method to spread out the traffic and avoid congested areas. Their method adds a layer to centralized control that adds cost for moving through highly trafficked areas. This ensures that more robots choose a longer route in order to avoid trafficked areas. Their method does not ensure collision avoidance as it is expected that the robots can handle collision avoidance and deadlocks in a decentralized manner. This method was made to improve performance on already functioning systems.
Utilizing traffic rules is a simple and effective method for avoiding collisions without using a lot of computational power. However, if more computational power is available then better solutions exist such as Multi-Agent Path Finding (MAPF) algorithms.

2.4.3 Mulit-Agent Path Finding

MAPF algorithms aim to solve the problem of finding paths for multiple agents e.g. robots, in an environment, where every agent reaches its goal without colliding with any of the other agents.
MAPF expands on the single agent shortest path problem, which is the problem of finding a path connecting the start vertex to a goal vertex in an undirected graph.
Optimal solutions for solving single-agent path planning have been made, these include Dijkstra's algorithm and later A* [16, Ch. H.2]. Several extensions of A* have been made in order to accommodate different agent constraints and behaviors. Using one of these algorithms a path for a single agent can be calculated in real-time.
However, when finding a path for multiple agents the approach of using A* is not computationally viable as the size of both the search space and branching factor grows exponentially with the number of agents. One approach for solving the MAPF problem is to decouple the problem into multiple single-agent path planning problems and then use A* to solve each problem independently with minimal interaction between the agents.
Prioritized planning takes such an approach to solve the MAPF problem. Prioritized planning works by assigning a priority to each agent in the system, and then a viable path is a path that avoids conflicts with paths created by other agents of higher priorities. Researchers have experimented with different design choices when implementing the prioritized planner [17].
These include setting priorities according to the distance between the start and goal of an agent and switching priorities if planning fails with the current agent priorities [18].
Others have limited how far into the future each agent considers higher priority

agent paths. Here the agents only plan to avoid conflicts for a specified number of steps and when the agent has moved these steps it re-plans the next steps in the same way [19].
Prioritized planners are neither complete nor optimal, meaning that there is no guarantee that they will find a solution, even though the problem is solvable, and the returned plan is not necessarily the shortest. Still, it remains to be one of the most popular approaches to solving the MAPF problem due to its simplicity and computational efficiency [17].
Prioritized planners have been proven to be complete in the special case of well-formed MAPF problems, that being MAPF problems where there exist viable plans for each agent that do not traverse any start or goal location of other agents [17].
Simpler complete algorithms exist, like the Push-and-Swap algorithm which is complete as long as the graph has at least two unoccupied vertices. The Push-and-Swap algorithm work by using two action primitives i.e. the push primitive and the swap primitive. The push primitive moves an agent towards its goal while trying to push other agents out of its way. If pushing fails it uses the swap primitive to swap the positions of agents [20].
Even though a complete algorithm will return a solution if there is one, given the requirements for completeness are fulfilled, there is no guarantee for the quality of the solution. This is where optimal solvers come into the picture.
Stern [17] categorizes the optimal MAPF solvers into four high-level categories: Extenstions of A* (EoA*), Increasing Cost Tree Search (ICTS), Conflict-Based Search (CBS), and Constraints Programming (CP).
EoA* use versions of the A* algorithm designed to minimize the branching factor in order to make solutions computationally feasible.
ICTS creates a tree of agent plan sizes, where the root node of the tree is a vector of the optimal path sizes for each agent not considering the other agents' paths. The children of a node are then the result of adding one to the size of one of the plans in the parent node. The algorithm then searches through the tree breadth-first in order to find the node with the least cost which is also conflict-free.
CBS works by solving a sequence of single-agent path planning problems. It then looks for conflicts in the agent plans and solves these. One method of solving these is with so-called meta agents. Meta agents merge conflicting agents into a single meta agent in order to find a joint solution for these agents [17].
CP models the MAPF problem as a constraint optimization problem [21] and then uses a general-purpose constraint solver in order to find the solution.
There exist many different solutions for solving the MAPF problem each with strengths and weaknesses, with no clear guidelines as to which one to choose for a given problem. As in most of these cases, the choice depends on finding a balance between the quality of the solution and computational effort [17].

2.4.4 Multi Agent Pickup and Delivery

A specialization of the MAPF problem is the Multi-Agent Pickup and Delivery (MAPD) problem. In the MAPD problem, multiple agents need to reach first a pickup location, followed by a delivery location, both of which are unique to the agent. Some MAPD algorithms also consider the possible set of pickup and delivery locations, choosing the next task for the robot based on some heuristic in order to minimize travel distance.
Early work for MAPD algorithms were developed for warehouses, an increasingly large industry [22]. Following this, work has focused on making the problem lifelong [23], that is, how can an algorithm be developed to run for a large set of tasks, and not just be a one-shot planner. The developed method is called Token Passing, and works by having a central token of all tasks and current plans. This token is then passed to individual robots, which can then select a task, and create a collision-free plan, based on the knowledge in the token. Kinematic constraints of robots have also been considered, in order to create plans for robots that are non-holonomic [24]. Work has also been done in combining the path planning and task allocation problem, in a manner to further optimize the overall efficiency. [25]. Optimizing for idle time in sortation centers has also been done [26], in which the priority is to keep a high utilization of the sorting machines. However, as outlined by Salzman et al. [27], little research has focused on the layout of the warehouse and its impact on overall system performance. Some work has also been done using reinforcement learning for the MAPD problem [28].
An important distinction between MAPD and swarm production is the fact that several workstations can perform the same action, thus the delivery location can be one of multiple locations. Additionally, a product will often require several processing steps, by several workstations, instead of just a single pickup and delivery. However, the algorithms are readily applicable for traffic management in swarm production, as all paths can be considered a path from one workstation to the next, which is a special case of the MAPD problem where the pickup and delivery location is the same.
For swarm production, it is likely possible to consider the entire path-planning problem of a workpiece along with allocating the workpiece both to a carrier robot and appropriate workstations. However, several of these problems have a high computational complexity when looking for the optimal solution, which outgrows computational power at even small scales.
However, if the tasks are known beforehand, more efficient solutions be can found to the MAPD problem by using offline data [25].
Also, instead of planning the path for an entire workpiece, it can be split into independent paths between each workstation. In that case, the complexity is reduced but guarantees for optimality are also lost for the problem as a whole. In practice, these considerations must be balanced.

2.5 Delimitation

Based on the related works, a swarm production system consists of the following components:

1. Carrier robots moving the workpieces and additional materials between workstations
2. Mobile workstations which can easily be moved in order to change the topology of the swarm production
3. Swarm management, which handles task assignment, topology optimization of workstations, traffic management of all robots in the swarm, and interfaces with MES and ERP systems

In order to develop a system within the practical limits of this project, a minimum viable system is outlined.
A minimum viable swarm production system would, by definition, need some amount of free-roaming agents of different types.
The number of different types of agents can be as low as two, as exemplified by Schou et al. [3], and the quantity of each agent is not well defined and could, in theory, be as low as two agents of each type. However, more agents are required to experience the issues of traffic management, task allocation, and topology optimization. Even though there are no rules for the number of agents in the swarm, the system should be flexible requiring a minimum of system downtime to introduce new agents to the system.
The swarm production system needs some form of ERP and MES stand-in, however, for the system to work, only a flow of production tasks is needed, accompanied by some data collection of production metrics for evaluation.
The production tasks should be mixed batches with flexible processing flow, in order to utilize the advantages of swarm production.
A minimum version of a swarm manager has to be able to allocate tasks to agents in the swarm and allocate workstations to the processing steps of the tasks. Strictly speaking, the topology manager is not required as in the short term the topology of the shop floor is static, however, the swarm production system should be designed in a way such that it requires minimum effort to change the topology.
Robust low-latency communication is required in order to facilitate the required communication between the swarm manager and the agents of the swarm if built on a larger scale. However, for smaller systems, more common technologies can be used.
Lastly, a minimum viable traffic management system cannot necessarily be easily built, as with many mobile robots, deadlocks, and collisions can easily occur. However, some simple rule-based systems may be able to handle small scales.

In short, a minimum viable system is defined as:

- Free-roaming agents of different types.
- At least two different types of agents.
- At least two agents of each type.
- Minimum downtime when adding an agent to the swarm.
- Must be able to allocate production tasks.
- Must be able to allocate workstations.
- The system should support mixed batch production.
- Utilise some traffic management strategy to avoid deadlocks and collisions.

This minimum viable system is used as a basis for creating a proof of concept of swarm production.
As will be described in more detail later in Section 4.1, five mobile robots are available for implementing and testing a swarm production system.
Therefore, it is decided to focus on traffic management, and build a swarm production testbed in which the mobile robots will move between workstations, but using simple task allocation and workstation allocation algorithms with a static workstation topology. In other words, it is investigated how related works in traffic management of mobile robots can be transferred to swarm production, in order to better understand both practical and theoretical issues in doing so. Additionally, with only five robots available, only the material handling will be considered in this system and the restocking of workstations will be omitted for future iterations.
Based on the discussion throughout Subsection 2.4.2, Subsection 2.4.3, and Subsection 2.4.4, two types of traffic management strategies are identified: Planning in which agents with individual planning and control are nudged e.g. by a higher cost in trafficked areas by a centralized controller and a centralized approach in which planning of all agents are considered simultaneously.
Therefore, both options will be investigated, in order to better understand how previous experience in this regard translates to swarm production.

Chapter 3

System Design

This chapter will go through the design of the created swarm production system. First, a specific swarm production case will be described, based on this an overview of the system is presented followed by a description of the design of each component of this system.

3.1 Use Case

In order to design a swarm production system a specific case has to be chosen. Swarm production has aspects similar to matrix production and flexible manufacturing systems in terms of flexible routing and re-configurable layout, respectively. Therefore, a task requiring flexibility has been chosen, to showcase the flexible aspects of swarm production. Taking inspiration from the swarm production example explored by Schou et al.[3] a case is defined. The case is based on the AAU Smart Factory, which is a flexible manufacturing system, that produces mass-customizable dummy phones. The production system is based on FESTO CP Factory modules. Though the system is flexible it has to be configured manually and the routing between workstations is static [29].
A phone may require four different processes: Adding the lid with a PCB, inserting fuses to the PCB, drilling holes in the PCB, and lastly, closing the phone by adding a back cover. A phone will always require the lid and back cover, but whether it needs fuses and drilled holes, depend on the specific product.
Four different tasks have been defined requiring different processing steps, these are:

- Add lid, and add back cover.
- Add lid, insert fuses, and add back cover.
- Add lid, drill hole, and add back cover.

- Add lid, insert fuses, drill holes, and add back cover.

The task is built as a tree of nodes with a root node. Each node is a process the product has to undergo. Once the product reaches a leaf node in the process, the product is finished. A node can point to multiple other nodes in the next layer of the tree, this means there are multiple possible processes to advance to. An example of a specific product tree can be seen in Figure 3.1.

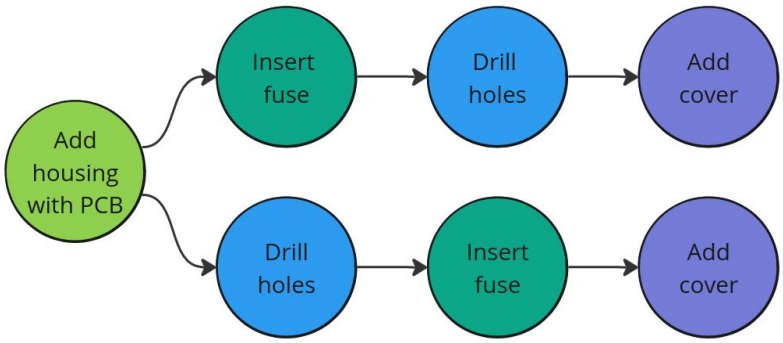


Figure 3.1: Example of a specific product tree with multiple possible routes, showing what processes the product needs to undergo. This phone requires drilling and fuses on the PCB in any order.

The different processes ensure that products will differ in the required routing as they require different workstations. Additionally, adding the fuses to the PCB and drilling the holes in the PCB does not have a required order. This allows identical products to take different routes in the product tree, potentially reducing the completion time by avoiding trafficked areas or workstations that are not currently free.
In the AAU Smart Factory, the phones are routed on small pallets on a conveyer belt running between workstations, with one phone per pallet. Each pallet has all the task information of the specific dummy phone they are carrying. In order to move this production system to a swarm production context the pallets have to be free moving, not constrained to the conveyer belt, and the workstations have to be able to be moved automatically. With Schou et al's. example, in mind, the pallets are exchanged with small and cheap carrier robots able to move freely on the production floor. The workstations can also be deployed as mobile robots, designed as gantry systems allowing carrier robots to pass under them, where products are processed in the center, and when the processing is done, the carrier robot moves out of the opposite side from the entrance.
A centralized system then hosts the swarm manager and topology manager, which enables interfacing with the swarm, MES, and ERP systems. [3]

3.2 System Overview

In order to create a swarm production environment, a system was designed that can manage the entire fleet of robots and control these robots to their relevant locations.
An overview of the designed swarm production system can be seen in Figure 3.2.

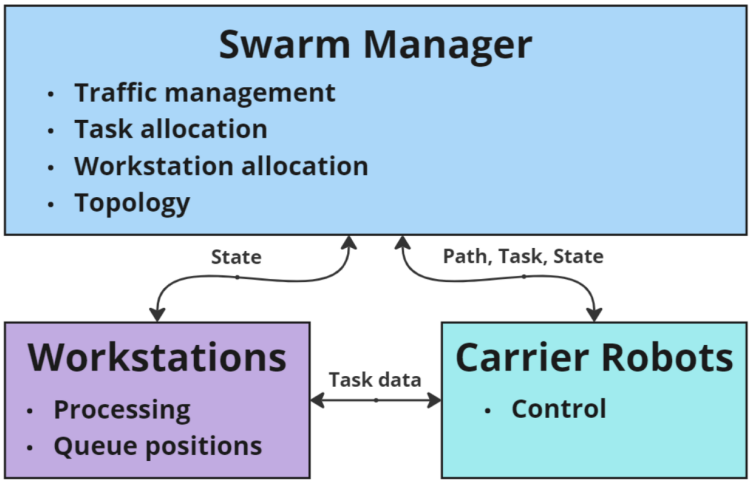


Figure 3.2: Overview of the designed swarm production system. Each arrow indicates the direction of communication between the components and the text indicates the communicated information.

The system is split up into three layers: A global swarm manager layer, a carrier robot layer, and a workstation layer.
The swarm manager layer controls the swarm on a high level, this includes managing what tasks the carrier robots have to perform, and which specific workstations to go to in order to complete the multiple processes required to complete the specific task. The swarm manager keeps track of what robots are part of the swarm, what type the robots are, and what their current state is. It allocates tasks to the carrier robots when they are available for a new task. When allocated a task, each individual carrier robot takes care of knowing what process is required next, for their workpiece. However, what specific workstation they should go to is decided by the swarm manager layer. Workstation allocation is done by a greedy approach where the nearest appropriate workstation for the required task is chosen. It is also the swarm manager layer that calculates a path for each robot such that it can take other robots' paths into account when planning.
The carrier robot layer is local to each carrier robot so that each carrier robot is managed individually. It takes care of the low-level control of the carrier robot in regards to getting sensor input in order to localize itself and control the actuators. This handles how the robot follows the path given by the swarm manager layer.
The workstation layer handles how the workstation communicates with the swarm

manager layer and the individual carrier robots that have been assigned to the workstation for processing. It also manages the processing and location of its queue positions, which is where the carrier robots will wait until there is room in the workstation for the robot. Queueing will be discussed further in Section 3.5.

3.3 Agents of the Swarm

Adhering to the use case outlined in Section 3.1, a swarm was designed. The Swarm consists of two types of agents: Real carrier robots and virtual workstations. This section will describe the design of these agents in order to enable the swarm production system.

3.3.1 Carrier Robot

The carrier robots are designed to be free-roaming agents, each being able to carry a single product, associated with a tree of processing steps as described in Section 3.1.
The carrier robots are designed to be independent agents of the swarm. This allows for the system to have a flexible amount of carrier robots, with minimal setup time. Each carrier robot is defined by a unique ID. The carrier robots are designed to handle their own sensor data, localize themselves in the environment, and follow the plan given to them by the swarm manager.

Carrier Robot State Machine

In order to implement functionality for the carrier robots, it was decided to use the classical approach of finite state machines. Looking at Figure 3.3, an overview of the designed states can be seen. The following will briefly describe each of these states.

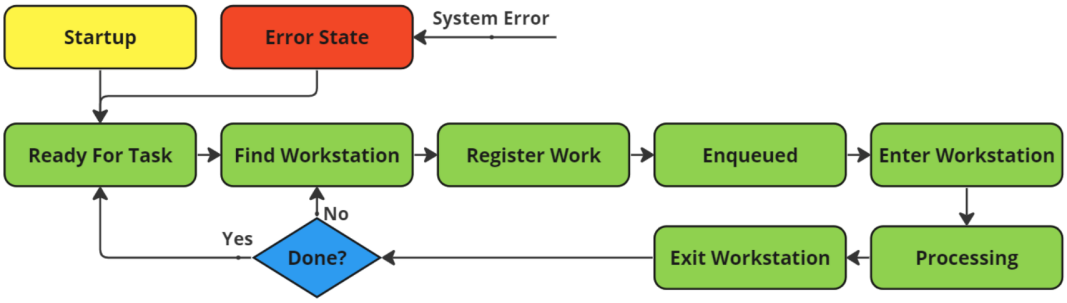


Figure 3.3: Overview of the carrier robot state machine

Startup: Once booted, the robot in this state ensures that its stack is started correctly, and once the robot's internal state is fully started, it registers itself with the

swarm manager. Once registered at the swarm manager, the robot changes state to ready for task.
Ready for task: In this state, the robot is ready to get a task. Once the task allocator allocates a task to the robot, the robot changes its state to find workstation.
Find workstation: With the task, the robot needs to proceed with the first step of the task. Since multiple workstations can be available with the same process, the robot asks the workstation allocator for a workstation based on the task's current workstation options. Once a workstation is allocated, the robot changes its state to register work.
Register work: In this state, the robot establishes communication with the allocated workstation. Here the robot is given an initial queue pose, as well as the entry, processing, and exit poses for the particular workstation. At this point, the robot starts a heartbeat with the workstation, in order to ensure that if either the workstation or the robot goes offline, the other can handle it appropriately after a certain timeout. Then, the robot changes state to enqueued.
Enqueued: Once registered at the workstation, the robot navigates toward its queue pose. Here the workstation can publish new queue poses to the robot, which it will navigate towards. Once within 1 meter of the queue pose, the robot transmits a message to the workstation that it is ready to be called into the workstation. Once called by the workstation, the robot changes state to enter workstation.
Enter workstation: In this state the robot first navigates to the entry point of the workstation and then proceeds towards the processing pose in the center of the workstation in a straight line. Once there, the robot changes state to processing.
Processing: In this state the robot awaits the workstation transmitting a message that the work is done. Then the robot changes state to exit the workstation.
Exit workstation: In this state the robot navigates in a straight line to the exit pose of the workstation. The heartbeat with the workstation is stopped once this point is reached. Once the robot arrives at the exit pose, the robot checks if there is more work to be done on the current task. If not, the robot changes its state to ready for task, otherwise, it changes its state to find workstation and proceeds with the next part of the task.
Error state: If an error occurs in the above states, the robot will enter an error state. In the current design, the robot clears the task and all other workstation data. Then the robot changes its state to ready for task. Thus the designed functionality here is simplistic. However, functionality to handle restarting the current task, or otherwise clearing the robot of the current task pallet could be implemented in this state.

3.3.2 Workstation Design

For this system, the workstations are created so that the carrier robots can drive through them when a product needs to be processed by the workstation. Having workstations where the exit is differently located than the entry allows for the next carrier robot to begin entering while the first carrier is still exiting.
The workstation units are designed to be, similar to the carrier robots, self-contained agents in the swarm, in order to achieve the high flexibility required by swarm production. This lets any number of workstations be placed in the system with minimal setup time, the only required setup is a unique ID, type of workstation, and initial position. Four types of workstations are created in order to abide by the process requirements for the selected task. The types are lid cell, fuses cell, drill cell, and back cover cell.
Each workstation consists of three points: An entry point, the core point where the processing of the product occurs, and an exiting point.

Workstation State Machine

Similarly to the carrier robots, a finite state machine was implemented to manage the workstation behavior. Looking at Figure 3.4, an overview of the states of the workstations can be seen. The following will briefly describe these states.

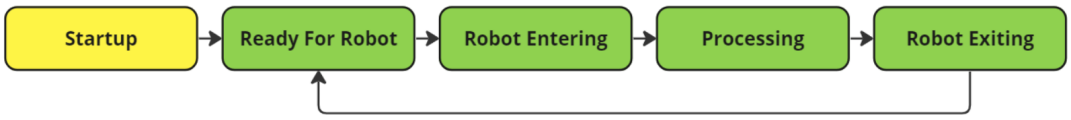


Figure 3.4: Overview of the workstation state machine

Startup: Similarly to the carrier robots, the workstations start in a startup state. Once it is fully started it then registers itself with the swarm manager before changing state to ready for robot.
Ready for robot: After startup, the workstation allocator can allocate robots to the workstation. When a carrier robot is ready for processing at the workstation, the workstation calls it in, and changes state to robot entering.
Robot entering: In this state the workstation waits for the carrier robot to enter. Once the carrier robot transmits a message saying it is in position, the workstation changes state to processing.
Processing: In this state the relevant processing in the workstation would happen. In the current design, the workstation simply awaits a fixed amount of time (5 seconds) before transmitting a message to the robot that it is done processing. Then it changes state to robot exiting.
Robot exiting: The workstation awaits the carrier robot exiting the workstation. Once it has exited, the workstation is ready for another robot and changes state to

ready for robot.

Unlike the carrier robots, the workstation does not have an error state, as the only failure is loss of communication or loss of heartbeat with the carrier robot. Therefore, it simply proceeds to the ready for robot state in case of an error.
In the background of these states, the workstation manages the registration of new carrier robots, as well as handling queue positions and publishing these to the carrier robots as will be described in Section 3.5.
In this system iteration there is missing a workstation relocation state, that should be entered when a topology manager decides to change the topology of the shop floor. This state would handle workstation movement and how to interact with the rest of the swarm while moving.

3.4 Planners

As discussed in the delimitation in Section 2.5, two different approaches to traffic management will be investigated.
The traffic management system must be able to handle the fact that several carrier robots will have to move in close proximity around the workstations.
Therefore, the first chosen approach is to create prioritized plans such that certain robots are allowed to move freely, while other lower priority robots must wait or path around higher priority robots. This approach is inspired by the methods discussed in Subsection 2.4.2. This planner will be referred to as Fixed Priority Spacial Planner (FPSP).
The other approach will instead consider the planning problem using temporal information. This allows for waiting in time but also requires that the execution of the plans is synchronized in time with the plan for all robots. This planner will be referred to as Space-Time A* Planner (STAP).

Fixed Priority Spacial Planner

The FPSP uses a costmap to create paths.
The costmap includes both static obstacles such as walls, and dynamic obstacles such as other robots. The costmap, therefore, needs to be updated frequently such that the costs in areas near other robots are correct when a path is calculated.
In the costmap, some areas are given a lethal cost, meaning the planner is not allowed to plan through that area. This cost is given to areas occupied with obstacles to avoid collisions or used to define areas where the planner should not create a path through.
With the costmap, FPSP uses A* to create a plan for the robot.

As the workstations will be present in the costmap, a straight-line planner is used to generate straight-line plans from a starting position to the goal position, by linearly interpolating the positions. This is used for planning from the entry of the workstation, to the processing pose, and to the exit. In this manner, a plan can be made to traverse the workstation which is otherwise considered of lethal cost.
As mentioned in Section 2.4.3, there exist different design possibilities regarding prioritized planners, how the priority order is set, and how it should take their path into account when planning. The systems using FPSP were made such that they could be switched between a fixed priority order and two different dynamic priority orders.
It can occur that a robot cannot plan a path due to it being blocked by a higher-priority robot's path. To avoid the robot going to the error state unnecessarily, the robots are designed to have the option to wait in their current position if they cannot plan a path due to other robots.

Space-Time A* Planner

For STAP, as the name suggests, space-time A* will be used to plan a path considering also the temporal dimension. This is inspired by the token passing algorithm [23]. It is similar to FPSP in the sense that plans are still prioritized, but the prioritization is first-come, first-serve. That is, all future plans must respect the already planned paths, and not collide with them. Therefore, it is possible for an agent that currently would not be able to move using FPSP to create a plan to wait, already knowing when the other robots will have moved away. To create a map of the factory floor for STAP, the map used for navigation is discretized into cells that can fit the carrier robots. Unlike FPSP, no cost is assigned to the cells, only whether a cell is free or blocked e.g. by a wall.

Handling Occupied Workstations

However, besides being able to create a plan, it is also necessary to consider where the robot should be when there is currently no free workstation for the robot to enter.
Two methods can readily be thought of to handle this. Knowledge about the state of other robots and workstations can be used to know when a robot will move into the workstation, and thus make a plan for another robot to the same entry position at the workstation while accounting for the fact that the next robot must be somewhere until the workstation entry is clear. Alternatively, a queue can be used, in which each robot will get a unique goal location, such that no such information about when other robots will move on is required.
Based on these two options, it was decided to use queueing of robots, instead of making the planners more complex by also having to consider when other robots

will move on. This also means that all robots will have unique start and goal positions, which is necessary for most MAPF planners in order to have a well-formed problem.

3.5 Queuing

In a swarm production system, the utilization of workstations is assumed to be significantly more valuable than the utilization of carrier robots, meaning that it is desirable to have carrier robots ready at the workstations for processing, as soon as the previous is done in order to maintain high workstation utilization.
It can be envisioned that a combined task allocation and MAPF algorithm can schedule carrier robots to arrive at workstations at the exact time necessary, in order to uphold the utilization of the workstations. However, besides the fact that the complexity of this problem is high, it must also account for the fact that the movement of robots is noisy. Both in regards to local control and external effects like a suddenly blocked path.
A solution to make the system able to handle these disturbances is by taking inspiration from traditional manufacturing systems that use buffers, and have queuing locations for the carrier robots such that they can move to a queuing location close to the workstation, rather than all robots trying to path to the entry of the workstation, in order to be ready to enter the workstation as fast as possible. As discussed in Section 3.4, this will also allow for using well-known MAPF planners.
The placement of these queue positions is important, as placing them far away from the workstation will lead to more travel time from the queue position to the workstation entry point, however placing them too close to the workstations entry and exit points can lead to deadlocks and possible collisions due to the close proximity of robots. However, this depends on the used planner.
In a manufacturing system with static topology, queuing positions can remain statically placed in relation to the given workstation, however, due to the dynamic topology in a swarm production setting, it is believed that these static queue point locations are not ensured to be optimal or even possible without a large increase in the workstations' footprint, due to the dynamically changing layout of the production floor. The challenge is to find a position where every queue position is reachable while reducing the distance to the workstation.

Static Queue Points

Two methods of placing queue points in the swarm production system are designed, where the first of these uses static queue positions. Static queue positions are placed with a static transform in relation to the workstation. The queue points are placed at a distance of 0.75 m to the side of the workstation center, spaced by

0.6 m from entry towards the exit of the workstation. These values were chosen empirically. Furthermore, all queue points are rotated such that the direction of the robot when queuing at the point, faces away from the workstation, and is parallel to other queuing robots. This is done so that when the carrier robot needs to enter the workstation it can drive straight out of the queue position.
Figure 3.5 shows a workstation setup, with four workstations, each with three queue points occupied by a carrier robot.

Figure 3.5: Static queue positions for a four-workstation setup, with a green carrier robot at each queue point.

Dynamic Queue Points

The problem with statically placed queue positions is that in a swarm production system, the topology of the production floor is unknown, and the queue positions for the last configured topology might not be optimal or even valid for a new configuration. An alternative to statically placed queue points is to place the queue points dynamically in the environment with consideration towards the other agents and obstacles in the environment. Positioning of dynamic queue points in swarm production has, to the best of the authors' knowledge, not been explored in research so far.
The belief is that it would prove beneficial to have the queuing carrier robots move out of the way of the moving carrier robots in order to avoid possible deadlocks occurring because of a queuing carrier robot blocking the path of a moving carrier robot.

In order to have the queue points position themselves according to their environment, a method taking inspiration from the potential field approach is used.
The potential field is designed to have an attractive potential for each queue point, towards the entry of the parent workstation, and repulsion from static obstacles in the environment, carrier robots not headed to the queue point, robot paths, and all workstations including itself in order to keep exit and entry points of these free.
The attractive potential is calculated using equations by Choset [16, Ch. 4]:

$$U_{att}(q) = \begin{cases} \frac{1}{2}\zeta d^2(q, q_{goal}), & d(q, q_{goal}) \leq d^*_{goal}) \\ d^*_{goal}\zeta d(q, q_{goal}) - \frac{1}{2}\zeta (d^*_{goal})^2, & d(q, q_{goal}) > d^*_{goal}) \end{cases}$$

where ζ is a parameter scaling the attractive potential, d is the distance from the current point q and the goal point q_{goal}, and d^* is a threshold distance where the potential function switches from growing quadratically to linearly.
The repulsive potential is then calculated using [16, Ch. 4]:

$$U_{rep}(q) = \begin{cases} \frac{1}{2}\eta(\frac{1}{D(q)} - \frac{1}{Q^*})^2, & D(q) \leq Q^* \\ 0, & D(q) > Q^* \end{cases}$$

where η is the gain on the repulsive potential, D(q) is the distance from point q to the obstacle, and Q^* is the threshold for how far away the obstacles affect the queue position.
Using these potential functions and specific gains for different repulsive obstacles in the environment and attraction gain towards the parent workstation, the queue points can be dynamically placed, and move around in order to free up space for moving carrier robots. In the current design, carrier robots enqueued at a queue point will continue following the queue point as it moves around in order to always stay out of the way of moving carrier robots.
The queue points are oriented so that the enqueued robots are always facing towards the entry of the workstation.
Figure 3.6 shows a workstation setup, with four workstations, each with three queue points occupied by a carrier robot, using dynamic queue points.

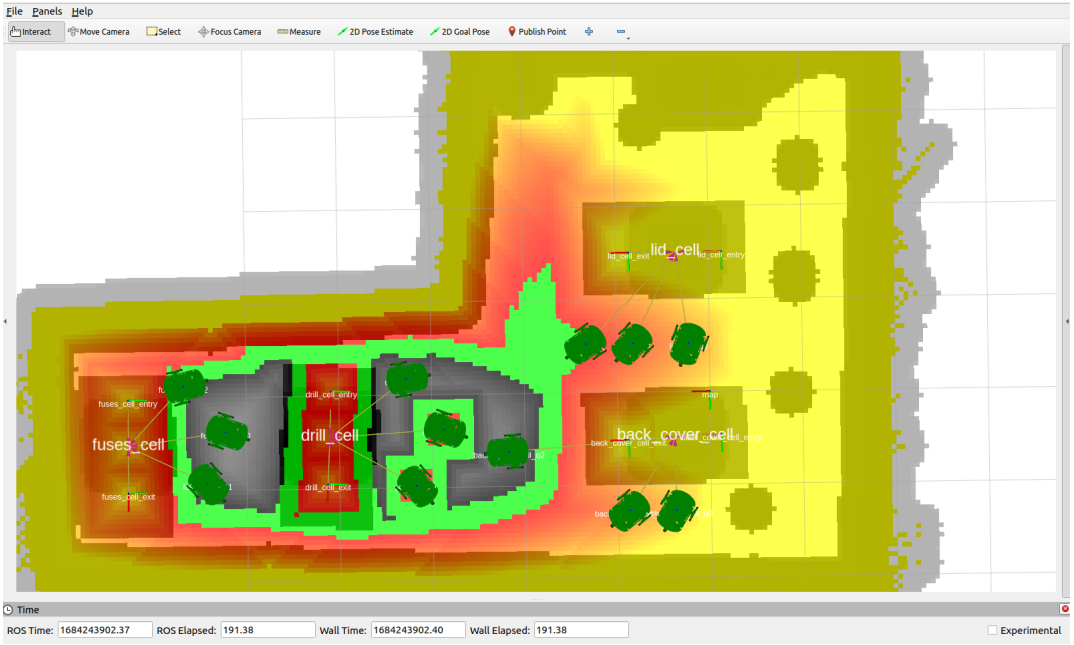


Figure 3.6: Dynamic queue positions for a four workstation setup, with a green carrier robot at each queue point. The potential field for the drill-cell queue points can be seen where yellow represents a high cost and grey is a low cost. Note that there is no repulsion from the carrier robots in the figure as they are purely visualizations.

Chapter 4

Implementation

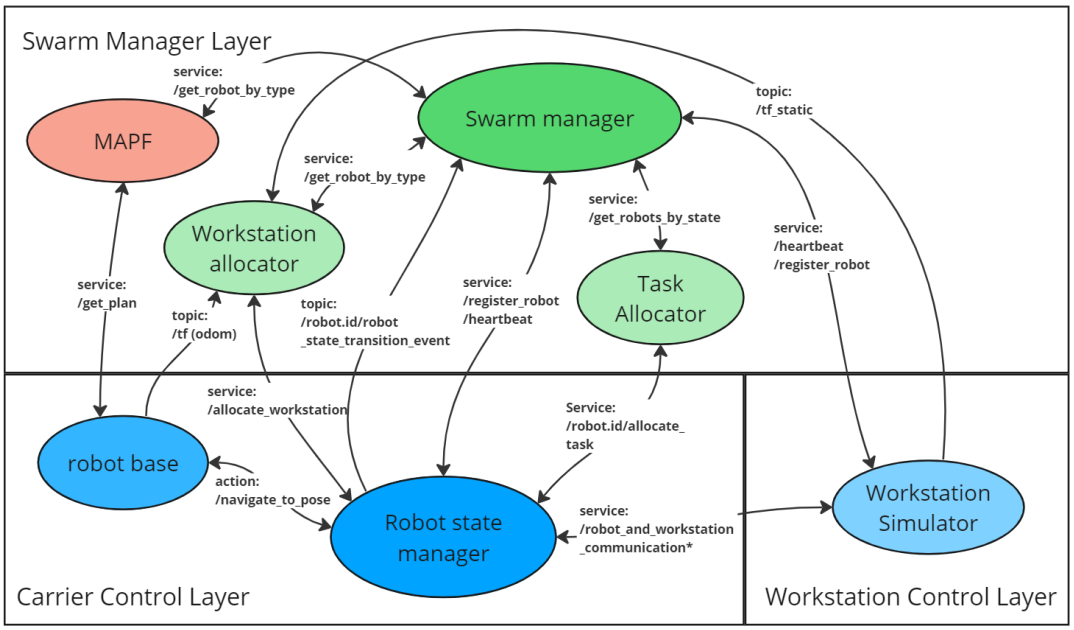


Figure 4.1: Overview of the implemented swarm production system. Each arrow indicates the direction of communication between nodes and the text indicates the communication type. The nodes depicted are not equal to ROS2 nodes, as some contain multiple ROS2 nodes, which are merged here for simplicity.

This chapter describes how this system has been implemented. This chapter will go through the hardware of the carrier robots, the Robot Operating System 2 (ROS2) framework used, and how the design described in Section 3.2 has been implemented, where each layer has been split into ROS2 nodes as can be seen in Figure 4.1. This figure shows an overview of the implemented swarm production system.

As discussed in Chapter 3, the swarm manager layer consists of a swarm manager which holds the state and information of all robots in the system, a task allocator which assigns tasks to robots, a workstation allocator which the robots ask to get a specific workstation based on their next processing step, and finally, the MAPF planner, which calculates plans for the robots. The Carrier control layer has the robot base which is a set of nodes with drivers and interfaces to hardware and sensors on the robot, as well as a software stack for navigation. Additionally, there is the robot state manager, which encapsulates the swarm-level functionality of the carrier robots. Finally, a workstation simulator node is implemented which simulates all workstations in the systems, each of which manages its assigned robots and queue positions.
Lastly, in Section 4.7 some networking problems that have been encountered are described along with the used solution.

4.1 Hardware - Polybots

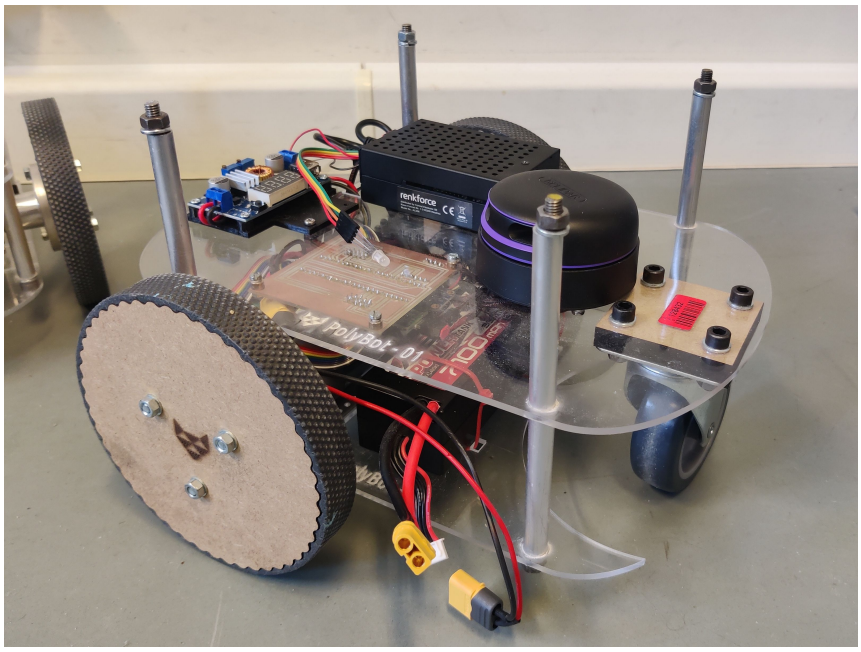


Figure 4.2: A PolyBot robot used in the project as one of the carrier robots with the top layer removed

The carrier robots used in this project are based on the PolyBot mobile platform, as seen in Figure 4.2. The robots are differential drive robots, with two motorized wheels in the front, and a caster wheel at the back. It is built up of three layers, where the bottom layer contains two DC motors, a Teensy microcontroller, an IMU, and either a 7100 or 5500 mAh LIPO battery as the power supply for the robot. The

IMU is directly connected to the Teensy using I2C. The DC motors are controlled using a pair of H-bridges. The second layer is used for the Raspberry Pi 4 (RPi4), an RPLidar A3, and a buck converter for 5V power. The RPi4 is used to run the ROS2 stack and drivers for the RPLidar. The RPLidar is a small 360 deg Lidar with an operating range of 0.2 m to 10 m, with an angular resolution of 0.225 deg to 0.450 deg. [30]
The top layer is empty and can be used for placing objects that have to be transported by the robot.

4.2 ROS2

This project was created using ROS2 Humble running on Ubuntu 22.04. ROS2 is a collection of libraries and tools designed for building robot systems. ROS2 is the second generation of ROS1. Both frameworks enable the use of open-source drivers and state-of-the-art robot algorithms.
A ROS2 system consists of multiple programs, called nodes, that interface with each other using topics, services, and actions. Topics are named and can only have one message type. The message type is user-defined and can contain any relevant data. Anyone can publish and subscribe to a topic in order to send and receive data. Services allow for sending a request and getting a response, best used for synchronous communication between nodes. Finally, actions are a combination of an initial request, which is either accepted or rejected. If accepted, the action starts some long-running process, which can provide feedback and finally a result once the action has finished.
In order for nodes to discover each other, by default, a simple discovery process is run which publishes a multicast message from each node containing all necessary information for other nodes to discover its topics, services, and actions. This also happens periodically, and when a node shuts down, it publishes a multicast message that it is closing.
In order to implement this functionality, ROS2 is built on different middleware implementations for Data Distribution Service (DDS) / Real-Time Publish Subscribe (RTPS).
This means that serialization, transportation, and quality-of-service are handled in the middleware layer. [31]
Besides facilitating networking, ROS2 has convenient tools for visualizing data, such as Rviz2. Rviz2 can be used to visualize a wide variety of data from a robot system, such as costmaps, sensor data, transforms as well as data from custom topics [32]. Rviz2 has been used to a great extent in this project for debugging purposes and to record data.
Another data collection tool available in ROS2 is ROS2 bag. ROS2 bag is a tool for collecting and storing large amounts of data coming from topics, in order for later

playback and data processing. [33]

4.3 Carrier Robot Control Layer

The carrier robot control layer, as can be seen in Figure 4.3, has two elements: The robot base which handles the data from the hardware and the navigation stack, and the robot state manager, which handles the carrier robots' states and communication with the workstation and swarm manager layer.

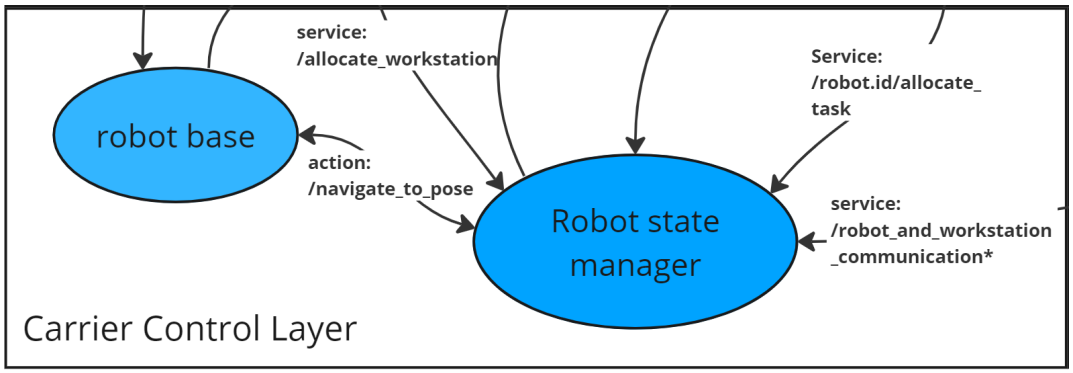


Figure 4.3: Overview of the designed carrier control layer. Each arrow indicates the direction of communication between nodes and the text indicates the communication type. The full overview can be seen in Figure 4.1.

4.3.1 Robot Base

In order to use the PolyBots, a software stack for interfacing with sensors and controlling the actuators is necessary. Additionally, advanced behaviors such as navigation and collision avoidance must also be implemented.
Instead of building a new software stack from scratch, the robot software will use the Linorobot project [34], which is a ready-to-go ROS2 software stack for simple mobile robots.

Hardware Interface

As described in Section 4.1, the PolyBot robots have Lidars, IMUs, and DC motors. For the IMU and DC motors, the Linorobot2 hardware project is used [35]. A fork with a configuration for the PolyBot is already made by the AAU Smart Production Lab [36].
Some small changes were necessary to add support for multiple ROS2 domain IDs and easier namespacing of robots. The motors are controlled using velocity control using a PID loop. Therefore, these gains were tuned to minimize the effects of the

non-linear velocity response of the robots, by targeting a fast response with strong integrative action. These changes were done in a separate fork which is available at Github [37].
With this, a micro-ros-agent can be run on the RPi4, which publishes odometry and IMU measurements from the robot, as well as subscribing to and sending commanded velocities to the motor controllers on the microcontroller.

Lidar and Robot Boot

The last sensor on the PolyBot is the Lidar. Drivers for the RPLidar A3 are freely available both from source and to install from apt on Ubuntu. This driver is simply run along with the micro-ros agent.
With the lidar driver and the micro-ros agent, the complete sensor and hardware interface is available. Therefore it was decided to compose these in a ROS2 launch file, which was added as a system service to start when the RPi4 is booted using systemd.

Navigation Stack

In order for the robots to move around and have more advanced behaviors, Nav2 [38] was used. As a part of the Linorobot2 project, a configuration for Nav2 is readily available. Similar to the hardware drivers, a fork was made from the AAU Smart Production Lab, in order to allow for changes. This fork is available at Github [39].
The Nav2 stack is split primarily into navigation and localization. Localization is achieved using the Lidar, IMU, and wheel odometry, and navigation is done by default using A* planners in costmaps, which creates plans for controllers to follow.
With this setup, the software stack is now capable of running Simultaneous Localization And Mapping (SLAM) in order to create a map of its environment, and then navigate in it afterward. However, more work is required in order to run multiple robots simultaneously.

Separation of Robots: Namespacing

In order to use multiple robots, some additional work must be done in order to separate the robots in the ROS2 network.
As can be seen in Figure 4.4, several nodes are run, each also publishing and subscribing to different topics. Notable, no effort has been made here to separate multiple robots. Thus, if another robot is started on the same network and ROS2 domain ID, the topics will interfere.
Therefore, work was done in order to namespace the nodes and their topics to be completely distinct from each other. For most of the stack, this can be done by

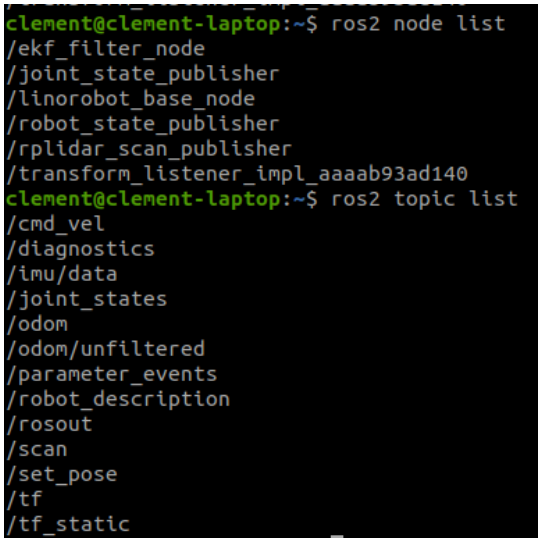


Figure 4.4: The ROS2 topics and nodes using the Linorobot2 stack. The navigation stack is left out for simplicity

using a launch file, by making a GroupAction and PushRosNameSpace substitutions [40].
For the navigation stack, there are configurable parameters that can be used for namespacing. However, special care must be taken to properly remap topics, as some nodes in the navigation stack expect global topics like /tf, and /scan. Here, remappings had to be manually added for such topics, in order to use the same configuration file for all robots, but different namespaces. In practice, this is done by remapping global topics, e.g. '/scan' to 'scan', which is a relative name, and will get the namespace prefix. Therefore, each robot will get a separate topic for scan data.
Finally, tf2 which is used to handle transforms for the robots in ROS2, assumes a single topic with all tf2 frames having unique names. However, this would require changing the robot description of each robot during launch or having additional unique configurations for each robot.
Instead of doing this, the /tf and /tf_static topics were also namespaced. This means that all robot data is completely isolated, as can be seen in Figure 4.5.

Accessing Relevant Robot Data

In order to gain access to relevant data about the robot, a ROS2 node outside of the robot's namespace must then either subscribe to topics on individual robot namespaces, or it could be implemented that the robots publish select information on global topics. The latter approach was taken for tf2 data, with each robot publishing its position to a global /tf topic at 10 Hz. This allows all robots to know the

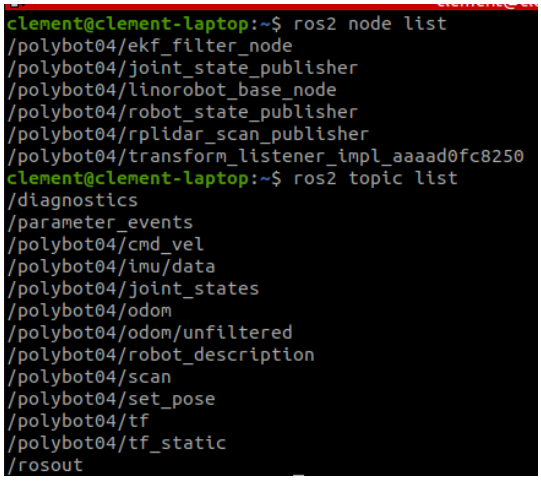
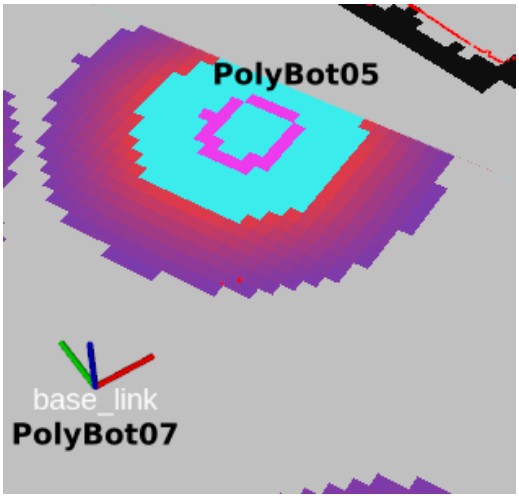
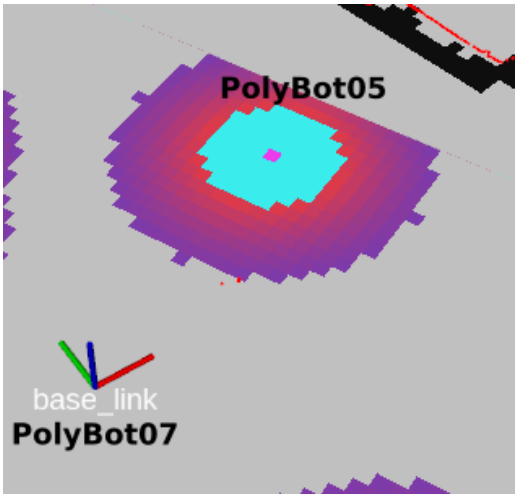


Figure 4.5: The ROS2 topics and nodes using namespacing. The navigation stack is left out for simplicity

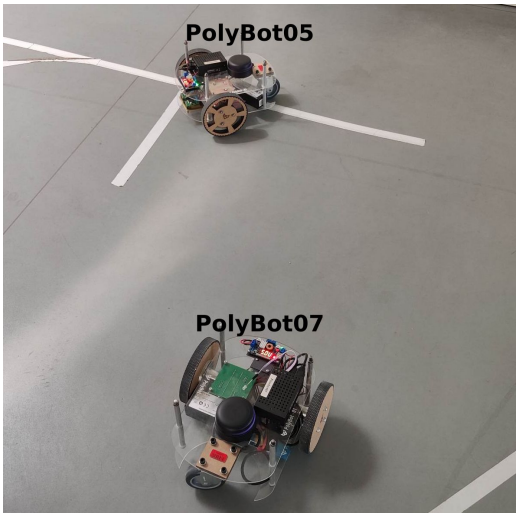
position of other robots, as well as simplifying the tf2 tree by only publishing the transform from the base_link frame of the robot to the map, instead of the full tf2 tree.
Utilizing the global positional data of the robots, a Nav2 costmap plugin was written.
This dynamic obstacle plugin transforms the poses of all other robots in the swarm into each robot's local costmap, in order for their controllers to be aware of other robots and avoid collisions. This was done as the LiDAR of the robots rarely was able to detect the other robots due to only the support rods, and the LiDAR itself being visible from the perspective of other robots. The resulting local costmap for PolyBot07, using the dynamic obstacle plugin, can be seen in Figure 4.6a, and be compared to the local costmap in Figure 4.6b not using it. Comparing them, it can be seen that with the dynamic obstacle plugin a circle of obstacle points is placed in the costmap at the location of PolyBot05, where without using the plugin the robots rely solely on laserscans to detect each other, leading to only one pixel in the local costmap being placed for PolyBot05, in this case. Using this plugin, the robots are consistently aware of the position of each other.
In order for this to work, and for global tf data to be reliable, it is assumed that each robot is localizing itself in a map which must be common for all robots.
It was tried to use a global map server instead of running a map server on each robot, but it was found not straightforward, as the Nav2 stack uses composition. For this, it was found possible how to remap topics, but not how to remap services. This would have allowed for a global map server, instead of running an additional namespaced map server on each robot.
Composition works by combining several ROS2 nodes in a single process, which



(a) The local costmap of PolyBot07 can be seen while using the dynamic obstacle plugin. The pink squares in the costmap represent obstacles, where the dynamic obstacles plugin creates a circle of obstacle points at the location of PolyBot05.



(b) The local costmap of PolyBot07 without the dynamic obstacle plugin can be seen. The pink squares represent obstacles in the costmap, the dynamic obstacle plugin is not used here, resulting in only one cell in the costmap representing the location of PolyBot05.



(c) The relative physical location of the two PolyBots, as shown in the costmaps.

Figure 4.6: Comparison between using the dynamic obstacle plugin and not. PolyBot07 and PolyBot05 are placed near each other. The dynamic obstacle plugin can be seen that when using it, a circle of obstacle points is created in the local costmap, whereas if it is not used only a single point in the costmap is marked as an obstacle

among other things, allows for that process to run topic/service discovery for all contained nodes, contrary to each node running its own discovery process.
Using composition for the Nav2 stack allowed the CPU usage of the RPi4 to go from ∼60 % with the stack running, to ∼50 % CPU usage. More importantly, it was observed that when other robots launch their stack, a high CPU load would happen on all other robots on the same network. This was found to in part be due to the discovery process of the ROS2 nodes and will be discussed in further detail in Section 4.7.

4.3.2 Robot State Manager

The robot state manager contains the swarm-level functionality of the carrier robots, using the robot base for moving to workstations, in order to execute tasks as given by the task allocator. It has been implemented following the design as laid out in Section 3.3.1.
In order to do this, the relevant clients for planning services, and navigation is set up, and the robot state manager ensures that the swarm manager is aware of the robot.
Additionally, the robot state manager publishes information with regard to the current status of the task, how much time it has spent on it, and which type of task it is. This is used for gathering overall system performance data during testing.

4.4 Swarm Manager Layer

The swarm manager layer, as can be seen in Figure 4.7, consists of all the central intelligence which, in this system, has been split into four elements: The swarm manager node, the task allocator, the workstation allocator, and the MAPF. Where two versions of the MAPF component have been made (FPSP and STAP).

4.4.1 Swarm Manager Node

The swarm manager node, not to be confused with the swarm manager layer, is a ROS2 node, which has the main task of keeping track of all agents in the swarm and their states. This information can then be retrieved by other nodes.
During the startup state of both carrier robots and workstations, as described in Subsection 3.3.1 and in Subsection 3.3.2, respectively, they register at the swarm manager. In order to register, each robot sends its unique ID, which is then stored at the swarm manager. This ID consists of a unique name and the type of the robot. The type can be a carrier robot or each type of workstation. A heartbeat is then started from the robot to the swarm manager. Given that the heartbeat is sent regularly, the robot is considered connected and in a well-formed state. If the

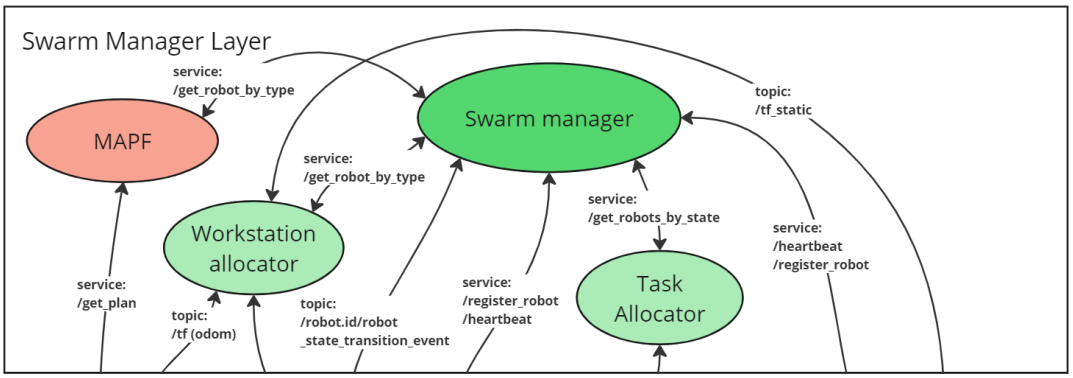


Figure 4.7: Overview of the designed swarm manager layer. Each arrow indicates the direction of communication between nodes and the text indicates the communication type. The full overview can be seen in Figure 4.1.

heartbeat timeouts (after 10 seconds), the robot is considered unknown. If another robot tries to register with an already registered ID, it will be rejected. In the case of a carrier robot disconnecting or restarting, this simply means that it will wait the 10 seconds before it can register itself again.
Additionally, both the carrier robots and workstations publish their current state, such that it can both be monitored centrally and be made available to other systems.
This is done through services, through which e.g. the task allocator can ask the swarm manager node for carrier robots that are in the ready for task state, and proceed to allocate tasks to them. Likewise, the workstation allocator polls the swarm manager for available workstations by asking for robots of a certain type, e.g. drilling cells.

4.4.2 Task Allocator

As described in Section 3.1, a task consists of several processes, some of which must be done in a specific order, and some do not.
The task allocator was made to construct and cycle through the four designed tasks. Then, the task allocator simply polls the swarm manager for carrier robots in the ready for task state and then creates a task for each of those.
Each task is stored as a tree of options, in which the carrier robot then can proceed down its branches. The carrier robot can then use these individual processes for the workstation allocator.
As currently implemented, the task allocator tries to allocate tasks on a timer every 10 seconds.

4.4.3 Workstation Allocator

Once a carrier robot has a task, it will ask the workstation allocator for a workstation given its current options. As discussed in Section 3.1, an example task can have added a housing with PCB, but can then either go for fuses or get holes drilled. In this case, the possible next workstation types are sent from the carrier robot to the workstation allocator.
Based on this information a workstation is then selected and sent to the carrier robot which then proceeds with its task.
In order to inform this decision, the workstation allocator polls the swarm manager for available workstations and their positions. Based on this, once a carrier robot requests a workstation, a greedy approach is chosen, returning the workstation closest to the robot.

4.4.4 MAPF Fixed Priority Spacial Planner

In order to consolidate information from multiple agents and be able to plan collision-free paths for all of them, two different centralized MAPF planners have been made: The FPSP, described in this section and the STAP described in the following section.
As mentioned in Section 3.4 the FPSP is used for path planning between workstations and then a straight-line planner is used for entering and exiting workstations without considering obstacles. It was, therefore, decided to let the carrier robot ask for a plan specifying a certain planner. Thus, the MAPF planner hosts a service that plans a path from start to goal using the wanted planner type and returns it to the robot. In addition, all current plans for all robots are published as markers to be visualized using Rviz2.
By storing previous plans for all robots, it is straightforward to consider other robots' plans when planning for a new robot.

Fixed Priority Spacial A* Planner

The FPSP is an implementation of a space-only prioritized A* planner. The planner uses Nav2's implementation of the A* algorithm. It takes a costmap and then calculates the cheapest path from cell A to cell B. In order to consider the path of other carrier robots and their priorities, the location and current paths for all higher-priority robots are added to the costmap, when calculating a path. These higher priority costs are set as lethal obstacles and are the same cost as given to walls and are thereby defined as not traversable when planning. In Figure 4.8 the resulting costmap can be seen. The costmap shown in this case is the one used when planning for PolyBot06. The blue line is the current path of PolyBot05. It can be seen that this plan is encircled, by cells with lethal cost, shown in the costmap

as the yellow cells. The green line is the resulting path of PolyBot06. The obstacle points of a higher priority robot are calculated using the poses of the current plan of this robot and expanding cost around all these by 0.35 m. Furthermore, a circle of obstacle points is added around all robots no matter the priority of these, in order to block a plan from going through other robots even if they are of lower priority. The circle of lethal points is a filled-in circle with a radius of 0.25 m, stopping a plan that otherwise would cause collisions from being created.

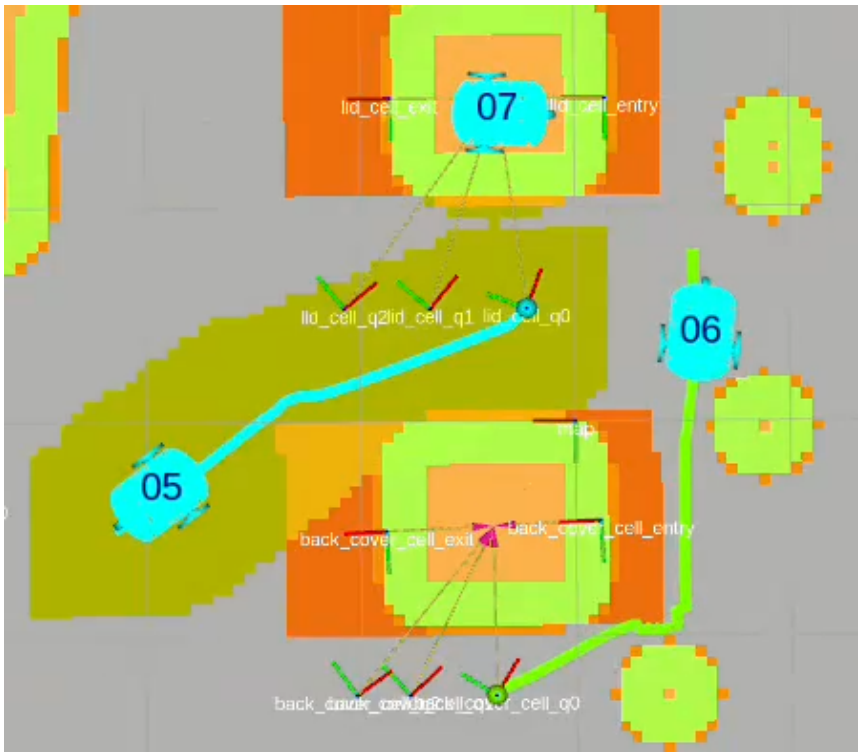


Figure 4.8: The resulting prioritized costmap is used when planning with FPSP. The figure shows the costmap used when creating a plan for PolyBot06. The blue line represents PolyBot05's current plan, and the green line is the resulting plan of PolyBot06. It can be seen that the current plan for PolyBot05 is placed in the costmap as lethal for PolyBot06, shown as yellow cells around the blue path.

The priorities of the robots are statically set according to the ID in ascending order, meaning PolyBot01 has the highest priority and e.g. PolyBot05 has a lower priority. Based on the methods explored in Subsection 2.4.3, two different dynamic priority schemes have been implemented. The first scheme was to base the priorities on the shortest path so that carrier robots that had a short distance from their current point to their goal would get a higher priority.
The other scheme served as a possible remedy to the non-completeness of the planner by changing priorities on failed planning so that if a valid path could not

be found between a carrier robot's position and goal, it would receive the highest priority, in order for a valid plan to be created.
Though promising in theory, initial testing showed that these priority schemes led to oscillations in the priority hierarchy which could cause deadlocks or collisions as their planned paths oscillated too quickly for the controller to operate properly, making the dynamic priority schemes non-viable without further work. Therefore only the fixed priority scheme is used with this planner.

Implementing a Wait Condition

Instead of having dynamic priorities, a wait action was implemented, letting a carrier robot wait if planning failed. The wait action is performed if no path can be found using the prioritized planner, but a path exists using a standard A* planner which ignores other robots and their paths. This lets carrier robots wait in place until there is free space for them to move.
To have the robots wait at their current position and not follow their given path, it was chosen to make changes to a default Nav2 Behavior Tree (BT) and create a custom condition node.
Nav2 utilizes BTs which is a tree structure of tasks to be completed. In general, BTs are built from flow control nodes and execution nodes. In the tree, all leaf nodes are execution nodes, and the rest are flow control nodes [41, Ch. 1]. BTs are executed depth-first from left to right, however, flow control nodes can change the flow. Specifically for Nav2, action, and condition nodes are used as execution nodes, and decorator nodes, control nodes, and recovery nodes are used as flow control nodes [42]. Action nodes contain the actual execution of tasks, condition nodes allow for selecting actions nodes based on the state of the system, decorators allow for e.g. running a task at a certain frequency, or inverting a condition node's result, and control nodes can change the flow of the BT. Recovery nodes allow for trying another set of actions given that the previous branch failed.
Nav2 uses BTs as they are a more scaleable framework as opposed to a finite state machine and are also easier to understand for a human as they make the many transitions between nodes and states easier to understand due to the structured and methodical flow through the tree. [43, 41]
As mentioned, Nav2 has a library of existing BTs that are readily available. These BTs uses both standard BT nodes [44] and their own Nav2 specific nodes [42].
The changes made to the default Nav2 BT and the creation of a custom condition node in order to implement the wait functionality, were necessary as existing nodes only allow for waiting for a pre-specified duration but the amount of time the robot must wait to have a clear path is not known before-hand.
The custom node created is the wait-condition node, as can be seen in Figure 4.9, which shows the changes made to the Nav2 BT. The wait-condition node was implemented to hold the answer from the planner in regard to whether or not it

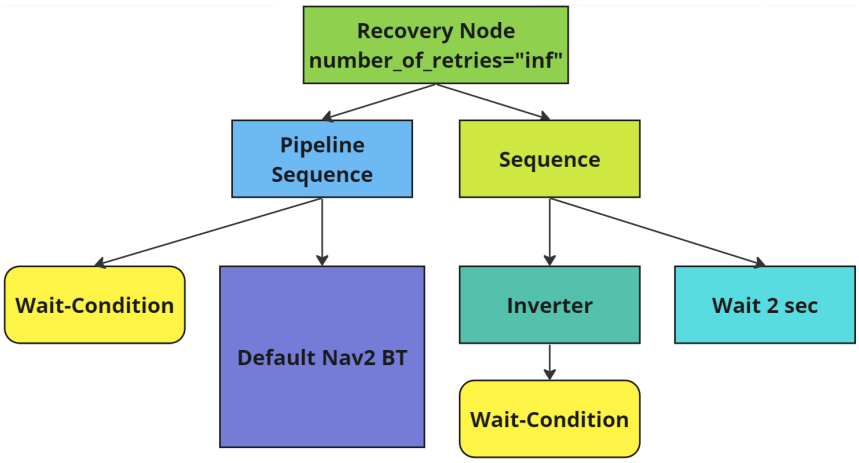


Figure 4.9: The custom behavior tree used for FPSP. The necessary conditions and sequences have been added to enable waiting, and otherwise continue regular navigation. The purple box is the existing Nav2 BT and the other boxes are then the nodes added in order to enable waiting for an unspecified duration to be possible.

should wait. In order to have the wanted flow in the BT, the wait-condition node was added along with some flow control nodes to the top of the existing Nav2 BT. This results in the BT entering a loop when the planner returns that the robot should wait.

The controller used along with FPSP is DWB [45], which is the default Nav2 controller. The DWB controller generates a set of trajectories that is then evaluated by several critics. The critics score each generated trajectory and the sum of their scores determines the chosen command velocity passed to the actuators. Adjusting these critics affects how the controller follows the given plan e.g. whether it strictly follows the path or more loosely in order to smooth out the trajectory.

Straight-Line Planner

As discussed in Section 3.4, a straight-line planner is implemented in order to enter the workstations for processing. This functionality is implemented as a straight-line planner, which then means that the robot state manager can navigate through the workstations using the same functionality as all other navigation. Although the straight-line planner does not consider any obstacles, the controller on the robot still avoids dynamic obstacles so, if the lidar detects an obstacle in front of the robot, the controller will still stop and not follow the given path.

4.4.5 MAPF Space-Time A* Planner

As discussed in Section 3.4, it was chosen to implement a MAPF algorithm utilizing space-time A* for swarm production.
In order to be able to fully implement an algorithm with the rest of the system, it was decided to implement a variant of the token passing algorithm [23]. The token is a centralized shared piece of memory in which the current location, path, and goal are stored for all agents. Agents then plan one after another. The first agent is free to plan, but the following agents must only create plans that do not collide with existing plans. This is done by storing the first agent's plan in the token, which the following agent then can access.
The planning problem is considered to be to navigate in a graph consisting of connected vertices. In this project, 4-way connectivity is used, that is, agents can only move left/right and forwards/backward. This assumes holonomic agents that can move in all these four directions. Each path then consists of a set of vertex-timestep pairs. A new plan must then not include any pair of vertice-timestep pairs that coincide with other plans. Additionally, two paths cannot share an edge between two vertices at the same timestep. This is in order to ensure that two agents can not swap places, which would result in a collision, even though the same vertice-timestep pairs are not occupied.
Finally, agents cannot share a goal, and therefore static queues are used at the workstations. This allows agents to path towards the workstations even though it is occupied.
Unlike the token passing algorithm, task allocation is done by the task allocator, and the agents thus cannot choose their next task. The token passing algorithm also includes a step to ensure that an agent without a task and plan, which is waiting at a certain position, will move if another agent needs to move to that position. This was not implemented due to time constraints but will be necessary for larger systems in order to avoid deadlocks.

Space-Time A* Planner

In order to implement the STAP algorithm, it was decided to build on a project by Sven Koening [46]. Here a visualization and some groundwork for loading different scenarios were already made, making it easier to implement and test the planner before applying it to the real robots. An example scenario can be seen on Figure 4.10.
The STAP algorithm was implemented to mimic the token passing algorithm [23]. Therefore, each agent is planned consecutively, and their path and goal are stored centrally. Other agents without a plan are considered a static obstacle that cannot be pathed through.
In order to plan a path for an agent, the heuristic value for the planner is calculated

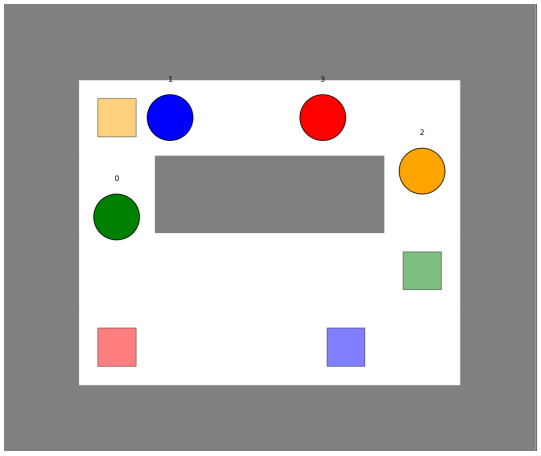


Figure 4.10: An example of a planning problem, using the visualization from Koenig [46]. The circles are the agents which must path to their respectively colored square

using Dijkstra's algorithm. Besides finding the heuristic values for the planner, this also serves as a check that a valid plan exists without considering other robots. If this is not the case, Dijkstra's algorithm will be unable to connect the starting and goal vertex of the agent.
If the goal is reachable from the starting location of the agent, space-time A* is then used to find the path of the agent. In order to consider obstacles, all vertices and edges that are either constrained by static obstacles, other agents' paths, or waiting agents with no path are removed from the graph. No kinematic constraints are considered for the agents, and thus the resulting path is a set of vertice-timestep pairs the agent must path through.
Importantly, it is necessary to implement a limit to how far forward in time an agent is allowed to plan, as otherwise space-time A* will run indefinitely when a goal is blocked by other agents. The maximum time was set to the total number of vertices in the graph multiplied by four. This limit is arbitrary and is simply in place to stop the planner in case no plan can be found within a reasonable time. In case no plan can be found within the maximum time, the agent is asked to wait until the next timestep, at which point it tries to plan again. This allows other agents with no plan to obtain one, and thus loosen the constraints of the planning problem. An example of this can be considered based on Figure 4.10. If none of the agents currently have a plan, and the first agent to plan is the orange one, its goal is currently unreachable due to the blue and green agent which has no plan. However, if it waits for a timestep, the blue and green could have made a plan, and thus the orange agent can now also reach its goal.
Since no movement of waiting agents is implemented, the current planner can deadlock in case an agent is waiting at the goal of another agent. However, in the case of swarm production, the agents should not be stopped, but continu-

ously receive new tasks. In case an agent is without a task, and has just exited a workstation, if no task can be given, it can be implemented to navigate the agent towards a holding area, in which it will not block other agents' goals. However, since in this project, agents continuously receive new tasks, no such functionality is implemented.
Besides taking care to stop the planner if it runs for too long, it is also important to ensure that the planner runs in time until the end of all other paths. This is necessary, as other agents may path through the goal vertex of the new plan after the time at which the agent can reach it. Thus terminating early would fail to take into consideration that another agent will collide at a later point in time. An example of this can be seen on Figure 4.11. Here, the green agent plans first and thus has no constraints besides the walls. Therefore, it creates a plan to go toward its goal (green square). Next, in the same timestep, the blue agent plans to move to its goal just beside itself. In the next timestep, the green agent will only move up one cell, and thus the blue agent's goal is free in the next timestep. Therefore this plan is valid up until this point. If planning is stopped at this point, the green agent will collide with the blue agent in the second timestep. However, if the blue agent plans until the time at which the green agent reaches its goal, no collisions will occur, as the blue agent will correctly see that the green agent will pass through its goal.

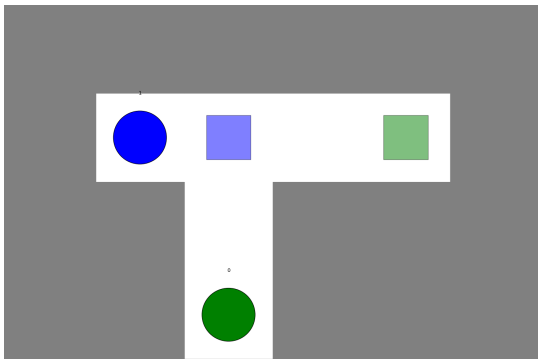


Figure 4.11: An example scenario showing why it is necessary to plan as a minimum to the timestep of the currently longest plan, as otherwise, the blue agent will interfere with the green agent's path

Once the plan is made, any additional timesteps at the end of the path in which the blue agent will not be moving can be trimmed away.
With this implemented, the final planner is capable of planning for multiple agents with overlapping paths spacially but separated in time, as seen in Figure 4.12.
As implemented in Python, the planner can create a plan to random goals for 100 agents in an open 64x64 cell environment with an average planning time per agent of 50 ms. Since a plan must only be made once, and the agents are not replanned during the execution of their path, this means that the planner is sufficiently fast

for this project.

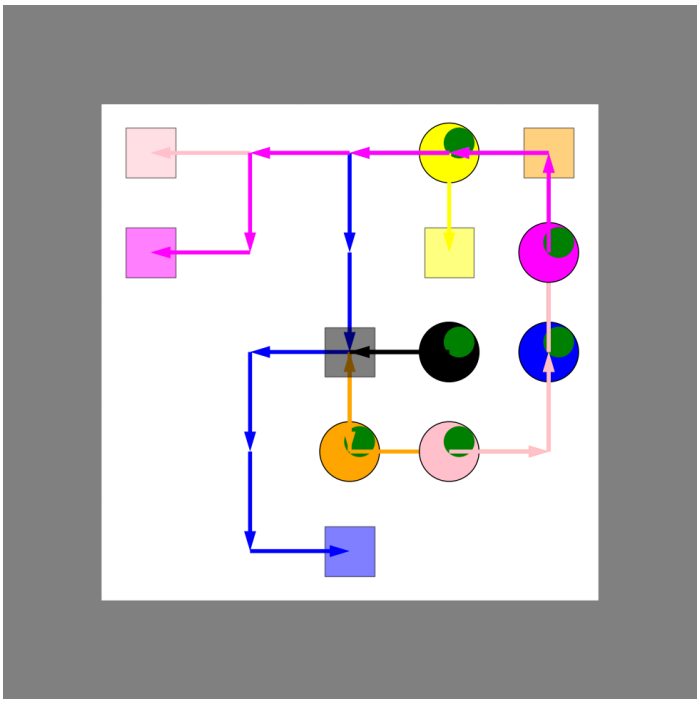


Figure 4.12: An example environment with 5x5 cells, and 6 agents shown with circles moving between random goals shown with squares, with their paths shown.

Executing the Plans

In order to execute the plans of the agents, the position of each agent is tracked and synchronized. Once all agents are at the location corresponding to their current point in their path, all agents are allowed to move to the next point simultaneously. In order to allow for some variability in both the control and the localization of the robots, the carrier robots are considered at their points when they are within a threshold of 0.25 meters of the intended position. The cells are chosen to have a size of 0.5 x 0.5 m based on the PolyBot radius of 0.22 m, measured from the wheel axle to the back of the robot. The same map used for the navigation stack is then used for planning, although it is discretized to the larger cell size. In a simulated environment, this works efficiently, with agents being able to move right behind each other. When applied to the carrier robots, synchronized movement cannot be guaranteed with the current planner, as it does not take into account the rotations of the robots. Therefore, some robots may need to first rotate toward the next vertex before they can move there. Meanwhile, other robots cannot yet enter the

start vertex of that robot. Therefore an additional constraint was added to the planner, that no agent is allowed to move to a vertex that was occupied by another robot in the previous timestep. This constraint spaces out the robots, at the cost of longer plans in time, but gives a buffer in which agents can freely move to the next vertex at their individual speeds.
With this additional constraint, the planner was found to create satisfactory plans for the carrier robots.
A video showing STAP without additional spacing [47] has been made available on Youtube, along with videos of the added spacing [48], and of 100 agents in a 64x64 cell environment [49].

STAP Navigator

In order to allow carrier robots to use STAP, a navigator was implemented. This navigator keeps track of the carrier robot's current position and calculates control input to the carrier robot's actuators. In order to get plans from the planner, the navigator first requests to join the planner at its current position. This allows the planner to verify that the robot is within the planner's map and that no other agents are currently planning to that position, or are already there. This ensures that the planner's state is not invalidated.
Once the navigator has joined the planner, all other agents will thereafter consider the new agent in their plans.
Then, the navigator implements an action server to request and follow a plan from the STAP, which can be used by the robot state manager to move to workstations and queue points.
Once a plan is requested by the robot state manager node, the navigator requests a plan from the planner. If a plan is given, it will now control the carrier robot's position to follow the plan published by the planner, providing feedback on the current distance to the goal to the robot state manager node. Once at the goal, the action server returns the result of the navigation goal to the robot state manager. Otherwise, the navigator will notify the robot state manager that its goal is invalid.

Robot Controller

In order to track the trajectory of the STAP, it is most important to ensure that the robots closely follow their assigned vertex/cell, instead of prioritizing e.g. smooth curves. A readily available controller for this was not found, and therefore a simple control strategy was developed to better fit this use case. The control strategy is inspired by a pure pursuit controller with an additional consideration that the robot must never move too far away from its path.
The controller works as follows.

First, calculate the Euclidean distance, d_g, from the robot to the current navigation goal and the heading difference, ϕ, between the heading of the robot and the heading towards the navigation goal.
Based on this, an appropriate linear velocity, V, and angular velocity, ω should be calculated.
Using a distance threshold $d_{max} = 0.1\ m$, and a heading threshold $\phi_{max} = 0.25\ rad = 14$ deg, three cases can arise:

A: $d_g > d_{max}\ and\ |\phi| > \phi_{max}$
The robot is not at the navigation goal, and the robot heading is not aligned with the navigation goal. Apply a proportional control law for rotational velocity: $\omega = \phi$, $V = 0$.

B: $d_g > d_{max}\ and\ |\phi| < \phi_{max}$
The robot is not at the navigation goal, but the robot heading is aligned with the navigation goal. Apply a proportional control law for linear and angular velocity: $\omega = \phi$ and $V = d_g$

C: $d_g < d_{max}$
The robot is at the navigation goal and must turn towards the goal's orientation. Apply a proportional control law for rotation velocity: $\omega = yaw_{error}$, $V = 0$ where yaw_{error} is the shortest distance to the correct orientation.

For all three cases, the following velocity constraints are applied when the respective velocity is not 0:

1. $\omega_{min} < |\omega| < \omega_{max}$ using $\omega_{min} = 0.1\ rad/s$, $\omega_{max} = 1.56\ rad/s$
2. $V_{min} < |V| < V_{max}$ using $V_{min} = 0.1\ m/s$, $V_{max} = 0.4\ m/s$

Once applied, it was observed that the robots track the lines more accurately than with the controller used for FPSP.
With this controller, and the navigator, the Nav2 stack is no longer necessary. Therefore, it was decided for STAP to reduce the Nav2 stack to only use the Nav2 localization stack. This significantly reduced CPU usage and network load. However, using this simple controller also means that the robot controller is unaware of its surroundings, and will not avoid obstacles or other robots, but only try to follow its given plan.

4.5 Workstation Control Layer

In this section, the implementation of the workstation control layer as can be seen in Figure 4.13 is described. It is only simulated in this project meaning no physical workstations have been used.

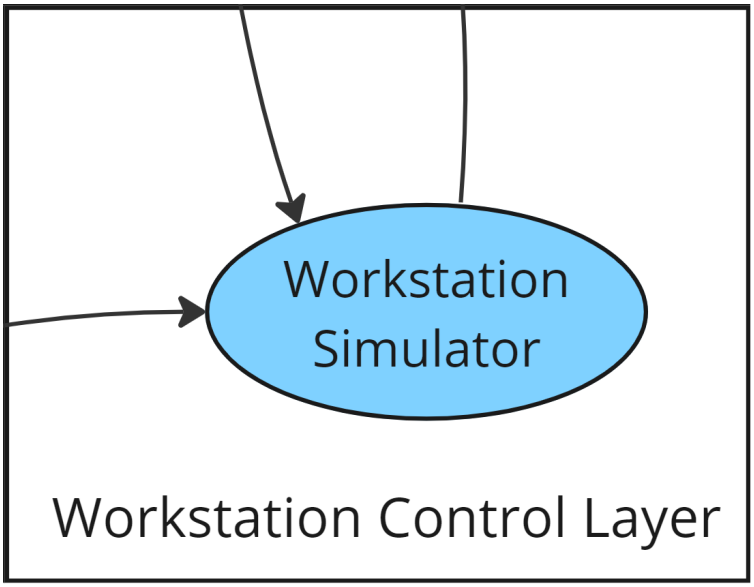


Figure 4.13: The workstation simulator implements the workstation control layer as a part of the whole system as seen in Figure 4.1.

4.5.1 Workstation Simulator

As there was no time to implement physical workstations in the system simulated workstations, as described in Subsection 3.3.2, were implemented in a workstation simulator node. The job of the simulator node is to initialize several virtual workstations, at different positions in the map. Simply, the workstation simulator node constructs a given number of workstations, of different types, and unique IDs so that these workstations can be registered at the swarm manager and be used for system testing purposes. The behavior of the workstations when created by the workstation simulator, is identical to the expected behavior of a physical workstation.
In order to be able to plan paths around the workstations a Nav2 plugin was written that creates an obstacle in the global map around the placement of the virtual workstations. The workstations are modeled as a square obstacle with lethal cost, with a width of 1 meter, and a rectangle that extends an additional 0.25 m along the length of the workstation from the entry and 0.5 m from the exit with a high cost, to avoid robots driving close to entry and exit points of the workstation, unless necessary.

4.5.2 Workstations

The workstations in the system consist of three layers that each add some functionalities to the workstations. These three layers are the workstation state machine

layer, the queue manager layer, and the queue position manager layer.

Workstation State Machine Layer

The workstation state machine layer, manages the state transition of the workstation, as described in Section 3.3.2, it determines the behavior of the workstation along with the flow of information to and from the workstation. It manages the initialization and deinitialization of states, so only the functions relevant to the current state are available.

Workstation Queue Manager Layer

The queue manager layer's job is to handle the queue points of the workstation, beginning with initializing a number of queue points at set transforms. After this, the queue manager manages which queue points are occupied by robots, allocates free queue points to newly enqueued carrier robots, and frees queue points previously occupied by a carrier robot once they exit the workstation. The queue points are assigned in a first come first serve manner, meaning the first carrier robot to be registered at the workstation and reach the queue point, will be the first robot in the queue to be processed by the workstation.

Workstation Queue Position Manager Layer

The workstation queue position manager layer manages the movement of queue point positions in the case of dynamic queue points. When static queue points are used this layer is turned off. The position of the queue points is calculated using potential fields as described in Section 3.5. The movement of these points is then determined by a maximum velocity variable and a minimum area of interest variable.
The maximum velocity determines how fast the queue point can change its location, set to be similar to the carrier robots' max velocity, and the least area of interest variable determines the minimum area the queue point checks before updating its location to the cheapest point. This is done in order to make sure that queue points move at speeds, the carrier robots can follow along with.
The potential fields of the queue points are calculated and stored using a Nav2 costmap data structure. A flow chart of the process for calculating new queue positions can be seen in Figure 4.14.
Initially when the costmap from the global map server is received the repulsive potentials are calculated using a recursive grass fire algorithm with 4-point connectivity, only expanding to the new cell if the new cost is higher than the previous cost and if the cell is a valid point for the queue position to be at.

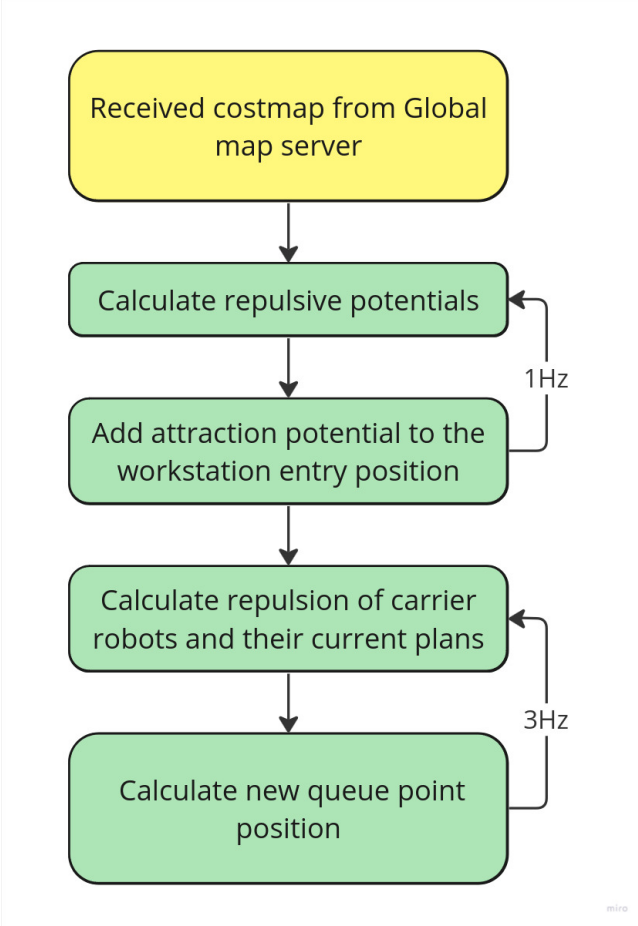


Figure 4.14: Flow chart showing how dynamic queue point positions are updated using potential fields.

A valid point is defined to be a point lying within a polygon described by the vertices of the manufacturing floor. The validity of a given point can then be determined by counting the number of intersections between the polygon and a ray coming from the point of interest. If the number of intersections is odd then the point lies within the polygon and is valid, if it is even the point lies outside of the polygon and is not a valid point. The implementation of this algorithm was created by Randolph Franklin [50]
After the initial step the repulsive potentials are updated at 1 Hz for static obstacles such as walls and workstations and more dynamic repulsive potentials such as carrier robots and their paths are updated at 3 Hz.
Then the position of the queue points is updated by checking for the min cost of valid points in a square grid with size $move\_dist \times move\_dist$, where $move\_dist$ is defined as $move\_dist = maximum\_velocity \times \Delta time$, $\Delta time$ being time since last

queue position update, and *maximum_velocity* being set to 0.5 m/s.

4.6 System Vizualisation

In order to visualize system data during testing and tuning of the swarm production system a GUI and a mesh publisher were created.

Graphical User Interface

The GUI was created for debugging, tuning, and testing purposes, using the python module tkinter, and can be seen in Figure 4.15.
The GUI consists of three tabs labeled: "Robot state", "Queue costmap params" and "System Monitor".
The robot state tab shows the current state of all available robots in the system, both workstations and carrier robots.
The "Queue costmap params" tab was created in order to aid the tuning of the dynamic queue points and consists of a slider for each tuneable variable for the positioning of the dynamic queue points.
Lastly, the "system monitor" display information regarding the performance of the system, this tab shows CPU usage, download and upload rate, total packets sent and received, and packet loss, all at the level of the operating system.

Robot Mesh Publisher

A robot mesh publisher was created in order to improve the spatial visualization of the carrier robots and robot states.
Figure 4.16 shows three carrier robots visualized with a robot mesh. The number and state are displayed on top of the carrier robots, the color of the mesh also represents the state of the carrier robot, where a green color represents the ready and waiting to get a task state, and blue means the robot currently has a task and red means the robot is in the error state.
The workstations also have their ID and current state displayed on top of them.

4.7 Practical Implementation Issues

During the implementation of the system, some networking issues and CPU usage issues were encountered when using multiple robots, these are described in full in Appendix B. However, in this section, only the issues relevant to the final version of the system, along with the chosen solutions are described.
It was found that when starting a navigation stack for a robot while other robots were operating, using the same ROS2 domain ID, would cause both their CPU

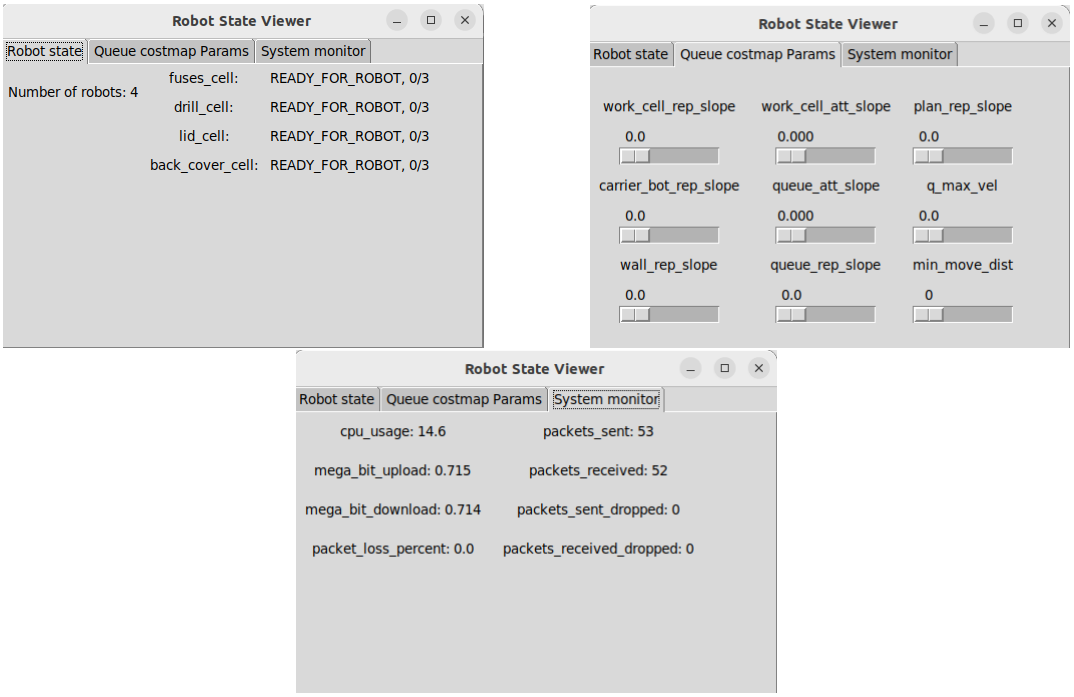


Figure 4.15: The three tabs of the GUI display robot states, queue costmap parameters, and system network performance and CPU usage.

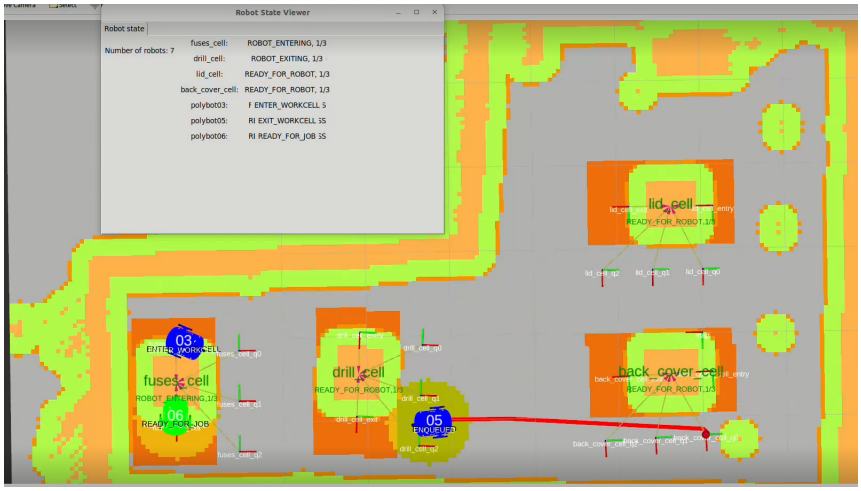


Figure 4.16: Visualization of three robot meshes, their ID, and state, and the ID of workstations can also be seen. In the top left the robot state tab of the GUI can be seen.

usages to spike to 100 %. Therefore, the navigation stack was chosen to be run from a central PC with more processing power, rather than from each individual carrier robot's RPi4, such that the robots would only be running the robot base.

This would avoid overloading the onboard RPi4.
Furthermore, It was chosen to use CycloneDDS as the ROS2 middleware rather than the ROS2 Humble default middleware, which is FastDDS. This is due to FastDDS occasionally terminating a node running a service server if the service was not able to discover the response topic of a client within 100 ms. CycloneDDS simply ignores the client if the service is unable to find the response topic within this time limit.
Lastly, it occurred that the internal time for the robots was not synchronized, causing them to not be operating properly. To ensure time synchronization between all components of the system, Chrony [51] was added to all robots and central computers.

Chapter 5

Testing

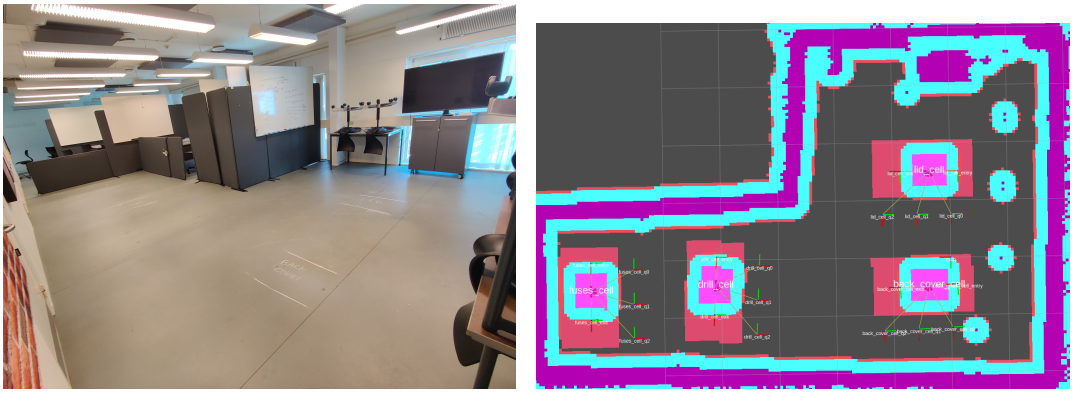
In order to evaluate the system, a test was carried out comparing Fixed Priority Spacial Planner with Static Queues (FPSP-SQ), Fixed Priority Spacial Planner with Dynamic Queues (FPSP-DQ), and STAP. In this chapter, the test setup, the test protocol, and how they are evaluated, will be described. Then the results will be analyzed.

5.1 Test Setup

The tests are based on the use case as described in Section 3.1. The test setup consisted of an "L" shaped room acting as the production floor where 4 simulated workstations were placed, as can be seen in Figure 5.1. These 4 workstations were of the types: Lid cell, drill cell, fuses cell, and back cover cell. The workstations were each assigned 3 possible queue locations. The room was cleared of obstacles such as tables and chairs, however, it was not possible to remove all tables from the room, so the rightmost wall was lined with tables, this was kept consistent throughout the entire duration of testing. For the testing, the task allocator was set to allocate the four tasks, describe in Section 3.1, in an ordered manner, such that every fourth task allocated would be the same. This was done to ensure that the four tasks are equally distributed and that the order is consistent throughout the tests.

5.1.1 Test Protocol

The same test protocol was used for all tests.
For the FPSP planners, the robots were always started in a prioritized order, with the robots with the highest priority being started first. For STAP the order does not matter, as priorities are dynamic. The robots were started one at a time, due to previously mentioned problems, in Section 4.7, regarding starting several robots



(a) Physical test setup with workstations positions outlined with chalk

(b) Map of the test setup, showing four labeled workstations with 3 queue points each.

Figure 5.1: Test setup with workstation positions outlined with chalk in the physical setup and labeled positions in the map.

at the same time. This also has the added benefit of only having one shared initial position for all carrier robots, instead of an initial starting position for each robot. Each test was run for 10 minutes, 10 times, with three, four, and five carrier robots, for FPSP-SQ, FPSP-DQ, and STAP. When the 10 minute timer ran out, the test was stopped and reset.
To summarize the test protocol: For each planner, with 3, 4, and 5 robots, repeat 10 times:

1. Start the central stack, initialize the workstations, the swarm manager, the task allocator, the workstation allocator, and the planner (FPSP/STAP)
2. For each carrier robot, start the navigation stack, thereby initializing the localization, navigation, and controller plugins of the robots.
3. Start the robot state manager node of one carrier robot, making it available to the swarm manager, such that it will be allocated tasks.
4. When the last started robot state manager of a carrier robot has a plan to enter the first workstation, the next in line is started until all carrier robots have been started.
5. When all carrier robots have been successfully started, recordings are started, and a timer for 10 min is started.
6. Once 10 min timer runs out, stop the recordings.

The mentioned recordings refer to ROS2 bags that collected data published on these topics:

1. /robot_state_transition_event, data regarding the states of both carrier robots and workstations.
2. /tf, data regarding the transforms of each robot and workstation.
3. /planned_paths, visualization of positional data of planned carrier robot paths.
4. /prioritized_costmap, the costmap used by the prioritized planner in order to create prioritized plans.
5. /system_performance, data regarding the system performance of the central pc running the central stack, and all carrier robot navigation stacks, including network performance.
6. /task_data, data regarding the tasks, such as completion time, type of task, and which carrier robot it has been assigned to.

Besides the data collected in the ROS2 bag, collisions, deadlocks, bugs, and other types of errors were noted and a screencast of Rviz2 showing the robot positions, planned paths, appropriate costmaps, and the GUI for robot state monitoring was recorded while performing the tests. Videos showcasing FPSP-SQ [52], FPSP-DQ [53], and STAP [54] during testing with five carrier robots, have been made available on YouTube.

5.2 Results

This section will first describe the gathered data with regard to the number of tasks completed and workstation occupancy, followed by the notes with regard to system errors and collisions.

5.2.1 Task and Workstation Data

Using the test setup and method described in the previous section, the results from each of the tests have been gathered in ROS2 bags and the relevant data has been converted to a CSV file and processed using R. Normality and homogeneity of variance are tested on the sampled data in order to determine the appropriate statistical test for comparing the samples. Normality is tested using the Shapiro-Wilk test and homogeneity of variance is tested using Bartlett's test. Then for comparing two samples, a t-test was used if the samples are normally distributed and the variance is homogeneous, otherwise, a Mann-Whitney-Wilcoxon test with continuity correction is used. For comparing more than two samples the ANOVA test is used for normally distributed data with homogeneous variances otherwise a

Kruskal-Wallis test was used. A significance level of $p<0.05$ is used for all statistical comparisons. All statistical data can be seen in Appendix A.
Additionally, for the box plots, a data point is determined to be an outlier using the interquartile range criterion: A data point is an outlier if it is either 1.5 times the interquartile range smaller than the first quartile or 1.5 times the interquartile range greater than the third quartile.

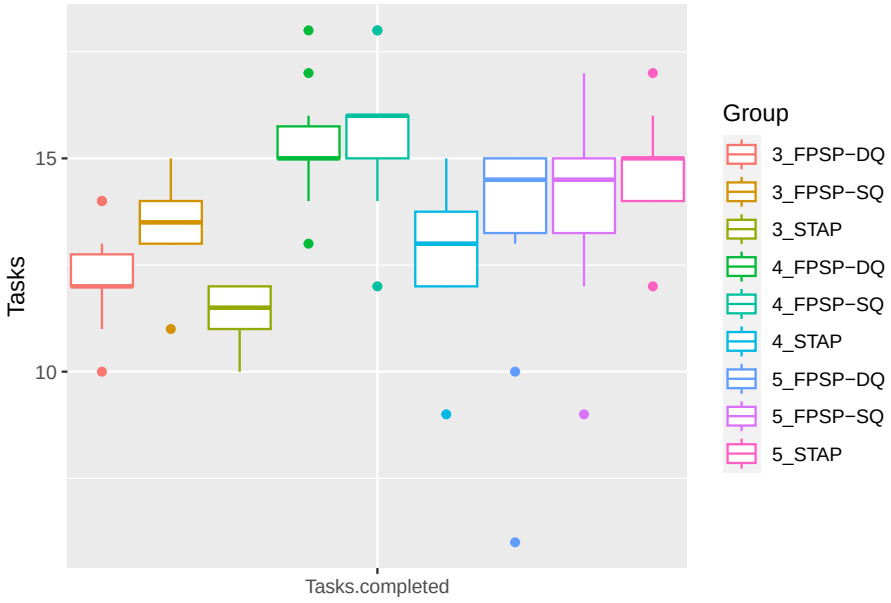


Figure 5.2: Boxplot for the number of completed tasks in 10 minutes for each of the three systems with 3, 4, and 5 robots operating. The number in each group represents the number of operating robots.

In Figure 5.2 it can be seen that for both the FPSP-SQ and the FPSP-DQ, four robots completed the most tasks in 10 minutes. Whereas, the STAP completes more tasks the more robots are operating. These differences were all found to be significant with $p=0.035$, $p<0.001$, and $p<0.001$ for FPSP-SQ, FPSP-DQ, and STAP, respectively. Additionally, comparing the systems against each other, FPSP-SQ completes more tasks at three robots with a significant difference compared to FPSP-DQ ($p=0.03$) and STAP ($p=0.001$). With four carrier robots FPSP-SQ and FPSP-DQ have similar performance ($p=0.68$) and both, are significantly better than STAP ($p=0.001$).
When 5 robots are operating no significant differences in completed tasks are observed when comparing the methods.
Figure 5.3 shows that for all three systems, the average time it takes to complete a task significantly increases with the number of operating robots (max $p<0.001$). Comparing the three systems it can be seen that the system utilizing STAP uses more time to complete tasks for both 3, 4, and 5 robots, all being significant differences with maximum $p<0.014$. The FPSP-SQ has the lowest average time per

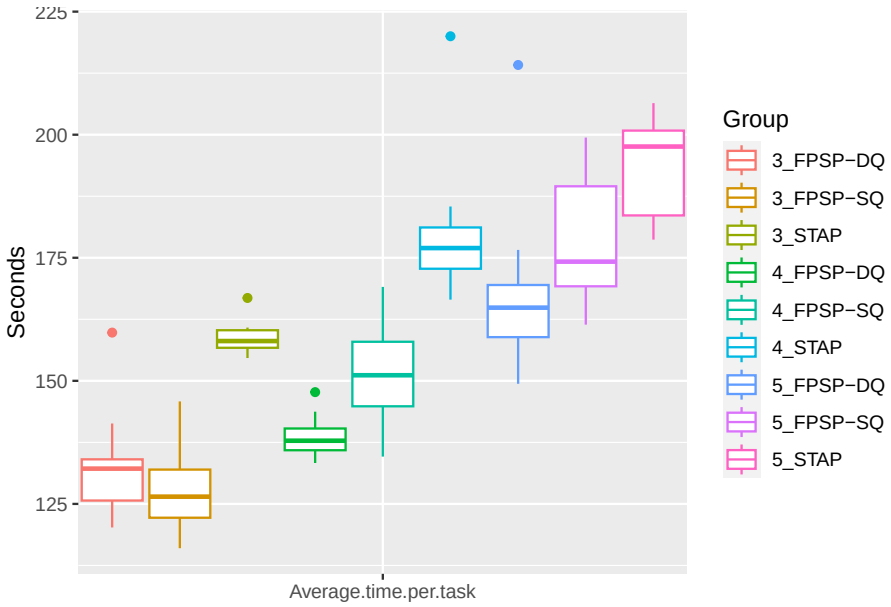


Figure 5.3: Boxplot for the average time used to complete a task, for each of the three systems with 3, 4, and 5 robots operating. The number in each group represents the number of operating robots.

task, when using 3 robots, though not significant (p=0.295) compared to FPSP-DQ. When using 4 or 5 carrier robots, FPSP-DQ has the lowest average time per task, significant at 4 robots (p=0.014), and close to significant at 5 robots (p=0.054), compared to FPSP-SQ.

Looking at the average cycle time, which can be seen in Figure 5.4, for both the STAP and the FPSP-SQ the average cycle time changes with the number of operating robots (max p=0.02). STAP has a significantly faster cycle time using three carrier robots compared with FPSP-SQ (p=0.02).

No significant difference between the number of carrier robots and cycle time was found for FPSP-DQ (p=0.342). Looking at the data for 5 robots, the median is close to that of when using 3 or 4 robots but with a large variance. Looking at the median, the FPSP-DQ has the highest cycle time when using 3 robots, but has the lowest when using 4 or 5 robots, though only a significant difference between the planners was found at 3 robots (p=0.034 compared to FPSP-SQ and p<0.001 for STAP).

The average occupancy of workstations can be seen in Figure 5.5. Workstation occupancy is the percentage of time the workstation is not in 'ready for robot' or 'startup' state. That is, the proportion of time a robot is occupied by a carrier robot. It shows that the STAP occupies the workstations less when running with 3 or 4 robots than the FPSP-SQ and the FPSP-DQ all significant with p<0.001. Using 5 robots only a significant difference was found between STAP and FPSP-SQ (p=0.004).

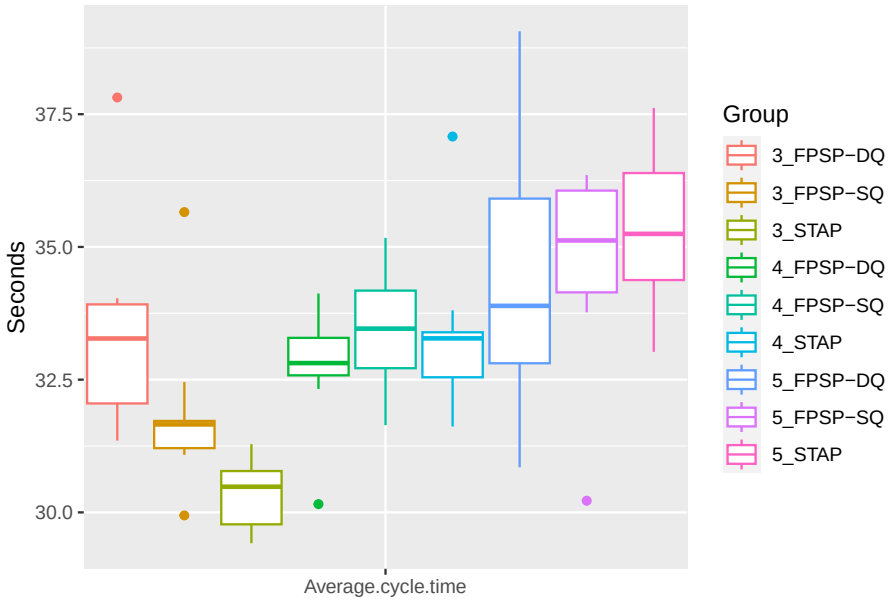


Figure 5.4: Boxplot for the average time in seconds used to complete a cycle, which is defined as the time from when the robot starts moving from its queue position to the workstation until it has been through the workstation and has reached the exit. The number in each group represents the number of operating robots.

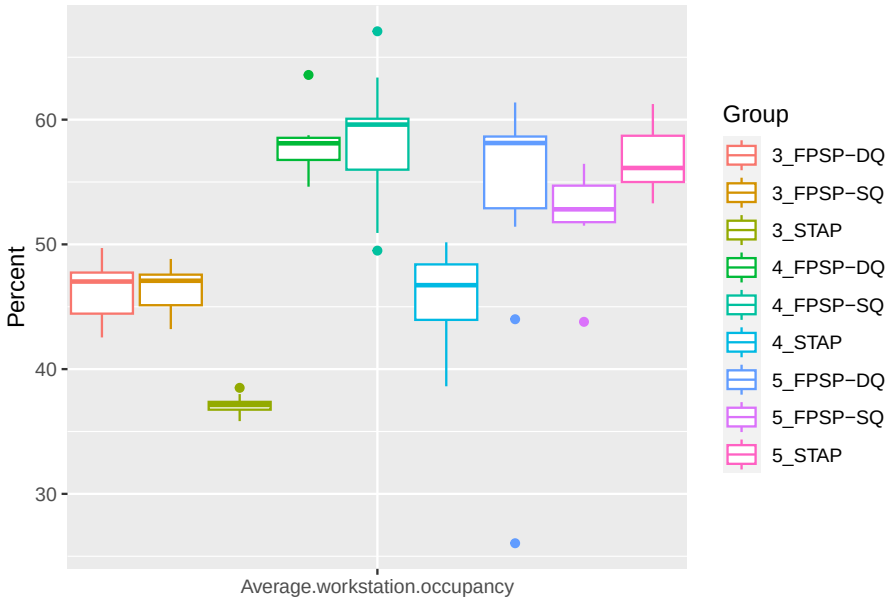


Figure 5.5: Boxplot for the average occupancy of the workstations in percentage meaning how much of the total 10 minutes was the workstations occupied by a carrier robot. The number in each group represents the number of operating robots.

The average occupancy time of workstations changes for all methods, depending on the number of robots used ($p<0.001$, $p=0.002$, and $p<0.001$ for FPSP-SQ, FPSP-DQ and STAP respectively). For STAP the number of carrier robots seems to increase occupancy time, whereas, for FPSP-SQ and FPSP-DQ 4 robots have a higher occupancy time than when using 5 robots.
When comparing the average occupancy of workstations and the total tasks completed, there is a similar trend where STAP has a lower overall rate of task completion and workstation occupancy, compared to the others, when running with 3 and 4 robots but are similar when running with 5 robots.

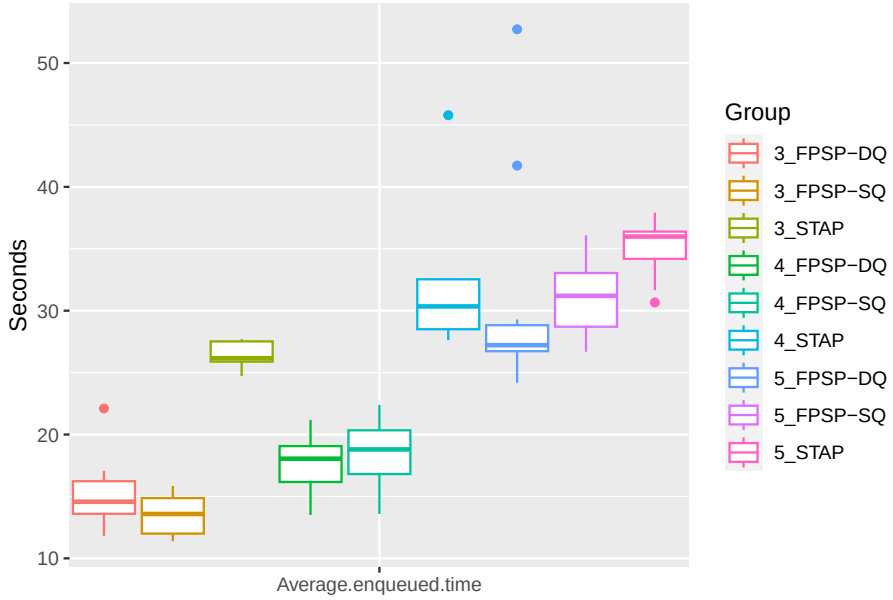


Figure 5.6: Boxplot for the average duration in seconds of how long the carrier robots were in the enqueued state. They enter the enqueued state whenever they are given a queue position by a workstation. This means that this time duration accounts for both the travel time to the queue position and the time waiting at the queue position.The number in each group represents the number of operating robots.

The average enqueued time for the carrier robots can be seen in Figure 5.6. It shows similar trends as the average time per task, where the STAP has a higher average of the three systems with 3, 4, and 5 operating robots, all being significant (maximum $p=0.026$).
However, looking at the graph STAP seems to scale better with the number of carrier robots in the system, compared to FPSP-SQ and FPSP-DQ. STAP has a higher starting enqueued time, but the enqueued time scales less with the number of robots compared to the FPSP-DQ and the FPSP-SQ.

5.2.2 Collisions, Deadlocks and System Errors

As mentioned, it was noted when two carrier robots would collide or deadlock, as well as any observed bugs in the system.
These observations have been categorized into seven categories:

1. 'Straight-line planner' - Using FPSP-SQ/FPSP-DQ in some cases, the carrier robot would omit the navigation to the entry point of the workstation, before applying the straight-line planner to move to the center of the workstation. In case this would cause a deadlock, because the straight-line planner does not consider other robots, the robot was manually moved to release the deadlock.

2. 'Stuck in workstation' - Using FPSP-SQ/FPSP-DQ, since there is no physical workstation, the robot controller may move close to the workstation in the costmap. Since localization inaccuracies can cause jumps in the robot's perceived position, this can move its current position to the edge of the workstation's lethal cost. In that case, the planner is unable to plan, and thus the carrier robot will be stuck. Once observed, the robot was manually moved out of the lethal cost.

3. 'Deadlock (no collision)' - Using FPSP-SQ/FPSP-DQ, two robots could come sufficiently close that the controller was unable to move the robot due to the risk of collision, but without actually colliding. In this case, one of the two robots would be manually moved to release the deadlock.

4. 'Fatal collision' - A collision in which two robots collide in such a manner that the robots are unable to move further, e.g. facing each other and both driving forwards. In this case, one robot would manually be moved away.

5. 'Small collision' - For STAP, sometimes two robots would scrape each other but would be able to continue moving.

6. 'System error' - Any bug or crash leading to a restart of a robot state manager of a carrier robot

7. 'Unable to move' - Using FPSP-SQ/FPSP-DQ, robots stop executing plan due to unknown reasons. In this case, the robot was moved slightly, after which the robot would resume normally.

Table 5.1, Table 5.2, and Table 5.3 show the amount of these errors for 3, 4, and 5 robots, respectively.

Table 5.1: Summarized errors noted during testing with a three-carrier robot system for FPSP-SQ, FPSP-DQ, and STAP, respectively. A '-' indicates a non-applicable category of error.

Errors for three robots			
Error categories	FPSP-SQ	FPSP-DQ	STAP
Straight-line planner	0	9	-
Stuck in workstation	1	7	-
Deadlock (no collision)	0	2	-
Fatal collision	2	2	2
Small collision	-	-	1
System error	1	0	0
Unable to move	1	2	-

Table 5.2: Summarized errors noted during testing with a four-carrier robot system for FPSP-SQ, FPSP-DQ, and STAP, respectively. A '-' indicates a non-applicable category of error.

Errors for four robots			
Error categories	FPSP-SQ	FPSP-DQ	STAP
Straight-line planner	0	18	-
Stuck in workstation	0	8	-
Deadlock (no collision)	2	6	-
Fatal collision	4	2	2
Small collision	-	-	0
System error	2	0	0
Unable to move	0	0	-

Table 5.3: Summarized errors noted during testing with a five-carrier robot system for FPSP-SQ, FPSP-DQ, and STAP, respectively. A '-' indicates a non-applicable category of error.

Errors for five robots			
Error categories	FPSP-SQ	FPSP-DQ	STAP
Straight-line planner	1	7	-
Stuck in workstation	8	28	-
Deadlock (no collision)	23	12	-
Fatal collision	0	3	4
Small collision	-	-	6
System error	4	1	0
Unable to move	3	2	-

5.3 Test Discussion

In order to gain a better understanding of the test results. The observations will, in the following, be discussed and possible explanations for the observations will be laid out.

Scalability

Overall, a general trend is seen that STAP scales more linearly for all metrics as a function of the number of robots. However, as seen in the required average time per task, STAP has a lower throughput per robot. This is likely explained by the fact that STAP executes its plans in ticks, and thus the average speed of the robots is lower, as well as the paths being square instead of smooth curves making the distance traveled longer.
FPSP-SQ and FPSP-DQ seem to scale worse, in some cases peaking with four robots as seen with tasks completed and workstation occupancy. This is likely due to the inherent difference between the planners, that STAP considers robots in space-time, and thus has a larger space in which to search for solutions. This is unlike FPSP-SQ/FPSP-DQ, which must spread the robots out spatially instead. Often leading to the robots taking a much longer path instead of waiting for another carrier robot to pass them.

Queueing Strategies

Looking at the average cycle time, it can be seen that it rises for STAP with more robots. Since the part of the cycle time that is spent within the workstation is the same regardless of the number of robots in the system, this rise is caused by a rise in time spent moving from the queue point to the entry of the workstation. This happens as several robots may be enqueued but not necessarily placed in the order in which they are entering the workstation. Thus, it can occur that a robot has to path around the other robots in order to enter the workstation, which happens more frequently with more robots in the system.
Looking at the cycle times for FPSP-SQ and FPSP-DQ. FPSP-SQ has a lower cycle time with 3 robots than FPSP-DQ which can also be seen by it completing more tasks. It is possible that these differences are due to the queue points being placed further away from the workstation using FPSP-DQ, increasing the travel time from the queue point to the entrance. The cycle time for static queue positions increases with the number of carrier robots, where there was not found a significant difference with the number of robots when using dynamic queue positions, which indicates that dynamically placed queue points scale better, however, there is a bigger variance with dynamic queue points which is undesirable.

Observed Errors

Looking at the number of collisions for STAP, it was seen to rise with the number of robots. This is likely due to the fact that robots will more often drive closer to other robots. Considering the robots' radius of 0.22 m (from the wheel axle to the back of the robot), the minimum cell size for STAP is $2 * robot\_radius = 0.44\ m$. On top of this, localization inaccuracy, and controller inaccuracy should be added to the cell size. With this in mind, the chosen cell size of 0.5 m is likely too small for the robot to stay within. This is believed to be the main cause of collisions, and therefore it is reasonable to see an increase of both 'fatal collisions' and 'small collisions' for STAP with more robots.
Since STAP ticks when all robots are within 0.25 m of their intended position, this further adds to the collision risk, since it allows robots to cut off corners. This can be alleviated in several ways. First, currently, the controller will stop moving towards its current given position by STAP as soon as the next position is received. The cutting of corners can be avoided by allowing STAP to tick, but redesigning the controller to first go within the 0.1 m as outlined in Subsection 4.4.5 before going to the next position. Second, the goal tolerance of 0.25 m with STAP can be decreased at the cost of robots slowing more down before receiving their next position. Last, the cell size can be increased to better accommodate these inaccuracies, at the cost of a larger floor footprint.
As can be seen in Table 5.1, Table 5.2, and Table 5.3, the number of errors and collisions grew much more rapidly for FPSP-SQ/FPSP-DQ from 3 to 5 robots. The 'straight-line planner' error is an implementation mistake that is solvable. It happened most for FPSP-DQ, and only once for FPSP-SQ. However, 'Stuck in workstation' and 'Deadlock (no collision)' also rise rapidly with more robots for FPSP-SQ/FPSP-DQ, compared to errors with STAP.
The required spatial spread of carrier robots is likely one of the causes of the 'Stuck in workstation' errors seen for FPSP-SQ/FPSP-DQ, which happens when a robot is moving too close to a workstation. This happens because robots are more likely to be pushed closer to workstations due to the paths of higher-priority robots.
Especially FPSP-DQ suffers from this problem, in this case, the increased amount of errors is thought to be due to the dynamic queue points being placed too close to the workstation area with lethal cost and always facing the entrance. This, in combination with a controller that prioritizes a smooth curve when following the path, leads to the robot cutting corners when moving, driving it into the lethal area of the workstations. This is only a problem with simulated workstations as having a physical workstation could enable the robots to better keep a distance using the lidar and the local costmap.
If another controller was used along with FPSP that more strictly followed the plan, these errors would likely be reduced.
Additionally, a bigger inflation cost of the workstations could have compelled the

robots to keep a greater distance from the workstations, reducing the likelihood of them getting stuck in the workstations.
FPSP-DQ performs similarly to FPSP-SQ in most cases, the only significant differences were observed with the number of completed tasks using 3 robots, the time per task at 4 robots, and the average cycle time using 3 robots.
Curiously FPSP-DQ has a significantly lower average time per task compared to FPSP-SQ with 4 robots, and close to significant with 5, one would then expect that FPSP-DQ, would then be able to complete more tasks in the same time, however, the number of tasks completed in these scenarios are very similar. One explanation for this observation can be found by looking at the errors.
As can be seen in Table 5.2 and Table 5.3, FPSP-DQ get stuck in the workstations more often than FPSP-SQ. This can cause the carrier robot to abandon its current task as it enters the error state, which does not count toward the number of completed tasks and time per task. Thus it would appear that the FPSP-DQ must have abandoned its task more often than FPSP-SQ, thus giving a lower average time per task, but a similar number of tasks completed.

Test Error Sources

Multiple sources of errors for the test methods have been identified, these include that the test setup was not kept exactly identical for all tests. Specifically, the positions of the workstations were not kept identical for STAP and the FPSP planners. Due to the discretization of the map by STAP, the position of the workstations had to be changed for all, entry, exit, and queue positions, to be valid. That meant that the fuse and drill cells were rotated 90 deg and they were both moved 0.5 m to the right when looking at Figure 5.1.These changes could potentially have slightly changed the outcome of the tests.
During testing a bug was observed causing the carrier robots to not go through all four process steps in the fourth task mentioned in Section 3.1. This caused the carrier robots to report that they had finished their task after the third step and therefore not going to the back cover cell as intended. However, as this was consistent through all tests, it does not skew the results when comparing the systems.

Chapter 6

Discussion and Conclusion

This chapter will discuss the results of the project, how to improve the implemented system, and outline its relevance for swarm production. Future work will be discussed before the final conclusion of the project is made.

6.1 Discussion

Two types of planners were implemented in this project, FPSP/STAP, with an additional difference in queueing strategy leading to FPSP-SQ/FPSP-DQ. Based on the test results and the following discussion, it is clear that a planner considering the temporal position of robots leads to a production system that scales better with more robots.
Although some changes can be made to FPSP such that it would perform better, and make fewer errors like getting stuck in a workstation, it is unlikely to be able to improve sufficiently in its current format to scale better than STAP.
With regard to queueing strategy, it has been shown that it is possible to dynamically position queue points, which could be used to design a better queueing strategy for STAP, in order to reduce the observed increase of cycle time with more robots.

The Importance of Scaleability in Practice

Several problems have been discovered while creating these systems in regard to having a ROS2 network where information is required to be sent between the agents in the swarm and the central PC. These problems have been mentioned in Section 4.7, and although the problems have been minimized in the implemented system such that a test could be carried out, the problems are still present and must be solved for a larger-scale system.

A possible solution that has not been tried but could reduce the problem is to have each agent on its own ROS2 domain ID and then implement a communication bridge between the relevant data of each agent and the central PC, rather than relying on the ROS2 network for all the communication.
Considering the collisions, network problems, and other system errors, it has been made clear that it is necessary to build swarm production testbeds in order to discover these issues, which a simulated environment would not have experienced. Likewise, this project was limited to 5 PolyBots, and therefore the scale of the experiment was limited. Using more robots would allow for a better understanding of the system's scalability.

Workstation Topology

As implemented and tested, all workstations are statically positioned. However, the implemented traffic management algorithms are not based on experience as e.g. a reinforcement learning agent is, but only use current information. Therefore the implemented algorithms should handle changing topology. However, since no tests were carried out with a changing topology, it is not possible to conclude the effect of topology with regard to overall production efficiency.

Prospects for Swarm Production

Considering the implemented system and the overall vision of swarm production, the scale of the challenges has been made clear. First, implementing algorithms for traffic management can be readily inspired by related works in MAPD. However, the effects of queueing at workstations, task allocation, and workstation topology all affect each other. Given a task allocation strategy of assigning a robot a task when it is free, necessarily leads to the requirement of queueing and therefore it affects the possible topologies of the workstations.
It has also been seen that the more simple traffic management strategy of FPSP was sufficient for small non-dense traffic scenarios. Considering the ease of implementation and execution, this could be a valid choice if task allocation is done based on workstation utilization instead of immediately assigning a task to all idle carrier robots.
Considering the fact that the value proposition of swarm production is flexible routing allowing mass customization using flexible topology to optimize workstation utilization, it is important to continue work on ensuring a more consistent cycle time for robots. This could be done with a better queueing strategy, which would allow for a more consistent production system.

6.2 Future Work

In order to further develop the system, some key points should be improved.

Scaling Up

First, the number of robots should be increased to better understand the scalability of the system. In order to keep a more consistent cycle time at the workstations, this requires implementing a better queueing strategy at the workstations which allows for faster access to the entry of the workstation. This can be done by changing the queue to function as a line, with robots moving closer to the entry point as the line shortens. Using such a method, the time to reach the entry point from the queue will be more deterministic.
Additionally, the number of queue positions for each workstation should be adapted to the number of operating robots, in order to ensure scaleability of the system. Alternatively, the robots could drive to a waiting area if all workstations have full queues. Either of these should be implemented in order to ensure a carrier robot is not waiting at an arbitrary position, possibly blocking other robots.
Based on the experience with dynamic queue points from this project, it is suggested to investigate dynamic queue sizes using dynamically found queue positions to allow for a changing topology in a small footprint on the factory floor.

Improving Plan Execution

For the STAP planning method, an improvement for the execution of plans has been created by Honig et al. [55] using a so-called action-dependency graph for MAPD in warehouses. Robot paths are broken down into actions and dependencies. Actions represent movements such as going from A to B in the path, where dependencies restrict the execution of an action, such that robot2 is not allowed to perform action a2, letting it go from A to B, until robot1 has completed its action a1 of moving from e.g. B to C. This system could improve the performance of STAP, by letting carrier robots move asynchronously, and not wait for other robots' turns or other delays caused by unforeseen circumstances. It can also be investigated if considering the rotation of the robots with the STAP is possible without increasing the computational complexity prohibitively.

Understanding Topology, and Task and Workstation Allocation

Once a larger system is run, it can be better understood how task allocation and topology affect the production rate of the system. Therefore, such a system would be an interesting testbed for further developing topology managers and task allocators. For further testing of a workstation allocator, it would be required to utilize

more than one of each workstation and have a workstation allocator which allocates e.g. based on current queue times rather than Euclidean distance as has been implemented but not explicitly tested in this project. Currently, the workstations have been statically placed during the testing, however, as it is important how well a system can adapt to changes in the topology, this will have to be investigated further.

6.3 Conclusion

This project set out to investigate swarm production and how to implement it using physical robots. This has been accomplished by implementing a system that complies with the requirements for a swarm production system, as described in Section 2.5. Based on related works and the constraints of the project, it was chosen to design and implement two distinct traffic management algorithms FPSP/STAP, with an additional traffic management strategy for FPSP, which is to apply dynamic queue positions.
Testing shows that for FPSP, dynamic queueing performs equal to using static queues when operating with three to five carrier robots. This shows potential for dynamic queuing as it allows for a smaller footprint of the workstations when configuring the system's topology. Additionally, it can allow for easier adjustments for the number of queue positions at a workstation. It is, therefore, believed that dynamic queuing has the potential to perform better than static queues in larger systems with more agents, however, larger-scale testing is required to support this.
Experiments show that considering the temporal dimension while planning paths for robots in situations with high robot density, as was done with STAP, allows for a more robust and safe execution of the navigation of the robots.
Additionally, it has been found that further development of STAP is likely to be a good candidate for traffic management in larger systems. Experience can be drawn from warehousing robots solving the MAPD problem.
In conclusion, traffic management strategies used in warehousing robots can be transferred to swarm production, but further work must be done to better understand scalability, and how task allocation and workstation topology are interdependent on traffic management strategies.

Appendix A

Statistical Tests

The three tables showing the p-values from comparing the three systems can be found in this appendix. Additionally, a fourth table comparing each of the three systems against themselves in regard to the number of operating robots can be seen. The p-values for comparing FPSP-DQ and FPSP-SQ can be seen in Table A.1. The p-values for comparing FPSP-SQ and STAP can be seen in Table A.2. The p-values for comparing FPSP-DQ and STAP can be seen in Table A.3. Since some data did not fulfill the prerequisites completely for using a t-test, a Mann-Whitney-Wilcoxon test with continuity correction has also been used to support the results. Likewise, in Table A.4, some data did not fulfill the prerequisites completely for using a one-way ANOVA test, therefore, a one-way Kruskal–Wallis test has also been used to support the results. Both the t-test and the ANOVA test require that the data is normally distributed and has homogeneity of variance. In order to check if the data has homogeneity of variance Barlett's test was used and to check if the data is normally distributed the Shapiro-Wilk test was used.

Table A.1: p-values from the t-test and the Mann-Whitney-Wilcoxon test with continuity correction for comparing the system using FPSP-DQ and FPSP-SQ. p-values based on data that did not fulfill the prerequisites for a t-test have been denoted with a * symbol.

	t-test			Mann-Whitney-Wilcoxon		
System metric	3 robots	4 robots	5 robots	3 robots	4 robots	5 robots
Completed tasks	0.032	0.682	0.465*	0.036	0.483	0.666
Average time per task	0.295	0.008*	0.153*	0.307	0.014	0.054
Average cycle time	0.066*	0.194	0.687*	0.034	0.345	0.521
Average workstation occupancy	0.958	0.840*	0.809*	1.000	0.473	0.104
Average enqueued time	0.147	0.490	0.924*	0.212	0.427	0.162

Table A.2: p-values from the t-test and the Mann-Whitney-Wilcoxon test with continuity correction for comparing the system using FPSP-SQ with STAP. p-values based on data that did not fulfill the prerequisites for a t-test have been denoted with a * symbol.

	t-test			Mann-Whitney-Wilcoxon		
System metric	3 robots	4 robots	5 robots	3 robots	4 robots	5 robots
Completed tasks	<0.001*	0.001	0.498	0.001	0.003	0.640
Average time per task	<0.001*	<0.001*	0.014	<0.001	<0.001	0.014
Average cycle time	0.019*	0.811*	0.398*	0.002	0.473	0.571
Average workstation occupancy	<0.001*	<0.001	0.007*	<0.001	<0.001	0.004
Average enqueued time	<0.001	<0.001*	0.006	<0.001	<0.001	0.009

Table A.3: p-values from the t-test and the Mann-Whitney-Wilcoxon test with continuity correction for comparing the system using FPSP-DQ with STAP. p-values based on data that did not fulfill the prerequisites for a t-test have been denoted with a * symbol.

	t-test			Mann-Whitney-Wilcoxon		
System metric	3 robots	4 robots	5 robots	3 robots	4 robots	5 robots
Completed tasks	0.095*	0.001	0.170*	0.095	0.002	0.281
Average time per task	<0.001*	<0.001*	0.001*	0.001	<0.001	0.003
Average cycle time	<0.001*	0.386*	0.277	<0.001	0.623	0.212
Average workstation occupancy	<0.001*	<0.001	0.363*	<0.001	<0.001	0.850
Average enqueued time	<0.001*	<0.001*	0.174*	<0.001	<0.001	0.026

Table A.4: p-values for 3, 4, and 5 robots operating within the three systems. p-values based on data that did not fulfill the prerequisites for an ANOVA test have been denoted with a * symbol.

	ANOVA			Kruskal–Wallisn		
System metric	FPSP-DQ	FPSP-SQ	STAP	FPSP-DQ	FPSP-SQ	STAP
Completed tasks	0.007*	0.035	<0.001*	<0.001	0.017	<0.001
Average time per task	<0.001*	<0.001	<0.001*	<0.001	<0.001	<0.001
Average cycle time	0.198*	<0.001*	<0.001*	0.342	0.002	<0.001
Average workstation occupancy	0.002*	<0.001*	<0.001*	0.002	<0.001	<0.001
Average enqueued time	<0.001*	<0.001*	<0.001*	<0.001	<0.001	<0.001

Appendix B

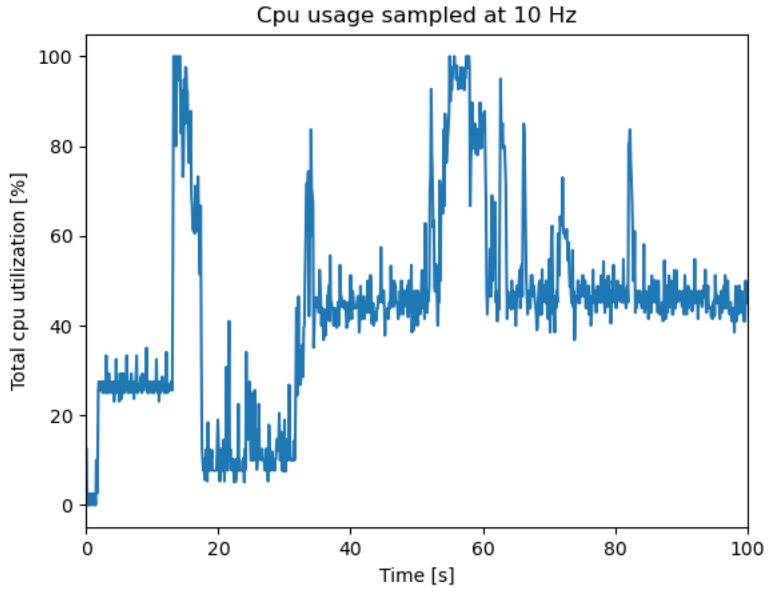
Networking and CPU usage Problems

During the implementation of the system, some networking issues and CPU usage issues were encountered. Therefore, the following will describe these issues, along with the chosen solutions.

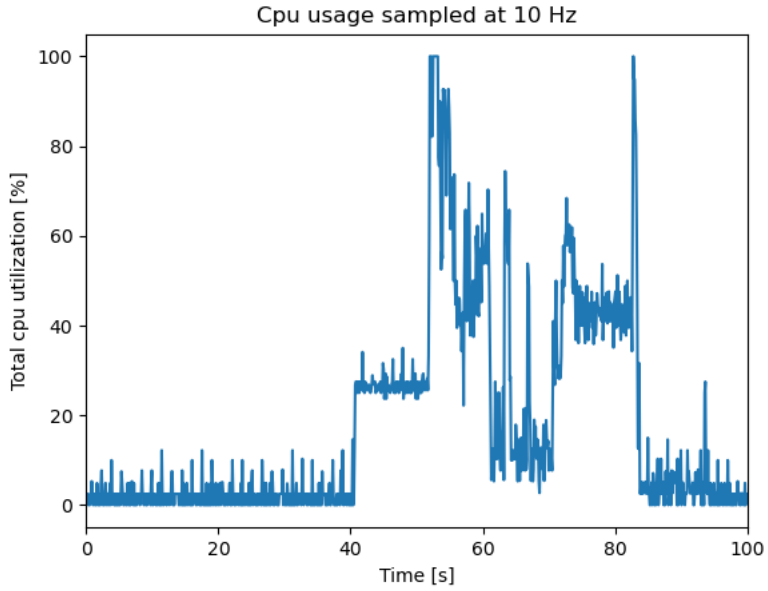
Navigation Stack

In general, for ROS2 it is possible to separate robots using ROS2 domain IDs. Robots can only discover other robots using the same domain ID. By default, each ROS2 node runs a simple discovery process, in which a multicast message is broadcasted to reach other nodes with the same ROS2 domain ID. This process is adequate for relatively simple systems. However, with several robots, this process can incur a high network load.
It was found that when activating the navigation stack on a robot, if any robot were already currently running on the same ROS2 domain ID both of the robots would then have their CPU usage spike. This could cause an error state that could not be recovered from where the robot had its CPU at maximum usage continuously and not be able to receive new ROS2 messages. If the spike did not cause the error state with the CPU usage staying at maximum capacity, the new steady state of the CPU usage was approximately double what it was before the second robot was activated.
It was initially assumed that this spike which was only seen during a robot's startup could be caused by the discovery process. This theory was enhanced during some tests described later in this section, as the spike always occurred regardless of which nodes were activated. Additionally, the size of the spike appeared to be depending on the number of nodes activated. The only prerequisite that could prevent the spike was to start the second robot on a different ROS domain ID.

However, this is not a viable option as communication between robots is necessary. To prevent this spike in CPU usage, the Nav2 stack was changed to use composition, to reduce the amount of discovery that had to be run. This reduced the spike slightly but not enough to prevent the robots from entering the error state requiring a restart of their navigation stack.
Another method for reducing the discovery is using one or multiple centralized discovery servers. They can be set up, to avoid the high amount of multicast network traffic. By default in ROS2, besides ROS2 Galactic, FastDDS is used as the middleware layer for networking. Thus a FastDDS discovery server was set up using the provided documentation [56].
The discovery server was set to run on a separate RPi4 on the network. For simplicity, the discovery server was added as a systemd service to automatically start when the RPi4 boots.
For each robot, the following variable must then be exported to the environment in which the ROS2 nodes are run:
`$ export ROS_DISCOVERY_SERVER="discoveryserver.local:11811"`
With the variable being the IP address and port of the discovery server. This was done using host names, to not set up static IP addresses. This requires the 'avahi-daemon' to be installed (`$ sudo apt install avahi-daemon`) and activated (`$ systemctl enable avahi-daemon`).
Using a centralized discovery server reduced the CPU usage spike such that the error state only occurred sometimes rather than every time. However, as every robot that is being activated has a high chance of causing any running robot to go into the error state, this is essentially still unusable as simply having three robots running simultaneously was a rare occurrence.
Using the discovery server several small tests were performed to determine and visualize the extent of the CPU spikes. In Figure B.1 the CPU usage can be seen of two robots when the navigation stack is started on both, sequentially. PolyBot07's initial spike in the first 40 seconds is its navigation stack starting and afterward, it runs at about 40 % to 50 % CPU usage. After 40 seconds the navigation stack is started on PolyBot05. During this process at about 50 seconds PolyBot07 spikes to 100 % CPU usage and then returns to about 40 % to 50 % CPU usage. However, when the navigation stack is terminated on PolyBot05 at 85 seconds, PolyBot07 has a small spike reaching about 80 % CPU usage. With the discovery server in use, there should only be very little effect on PolyBot07, because the two robots share no direct information besides the global tf topic. Nevertheless, its CPU usage spikes to 100 %. Additionally, in Figure B.2 only the robot base is run on PolyBot07 and PolyBot05 so there should be no effect on each other at all but interactions are still seen, as a spike occurs at 20 seconds when the robot base is started on PolyBot05. The robot base is stopped on the polybo05 at 50 seconds, at which point a small spike is observed. It can also be seen that CPU usage of PolyBot07



(a) CPU usage of PolyBot07



(b) CPU usage of PolyBot05

Figure B.1: The CPU usage of PolyBot07 and PolyBot05, respectively. The navigation stack is started on PolyBot07 and left running while on PolyBot05 the navigation stack is started and then terminated once the startup process has been completed.

stays increased after the robot base on PolyBot05 is stopped. However, this only happened occasionally.

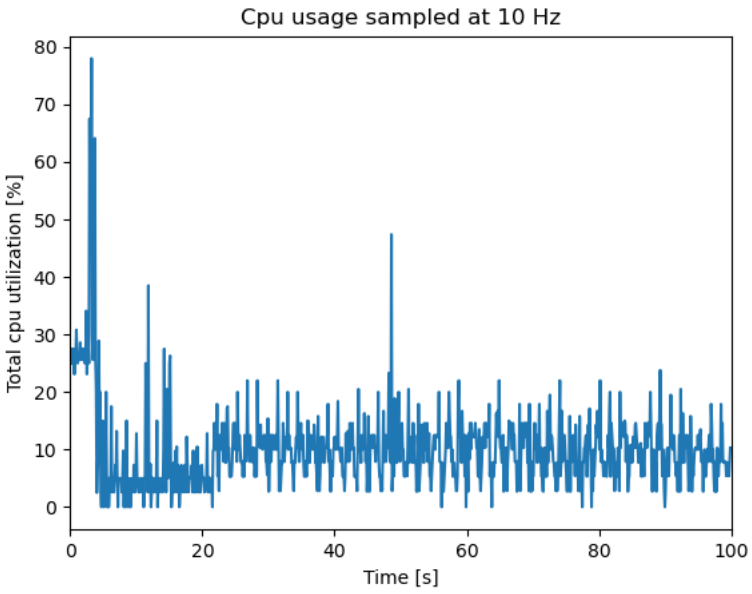


Figure B.2: CPU usage of PolyBot07. Namespaced bring-up is started on both PolyBot07 and PolyBot05 simultaneously. After 20 seconds it is terminated on PolyBot05 while PolyBot07 continues to run.

With the tests, it could thereby be concluded that the spikes in CPU usage occurred no matter which and how many nodes were run but the magnitude of the spike depended on the number of nodes starting.
It was, therefore, decided to reduce the robots' CPU usage by only having the robot base running. This meant that the navigation stacks would instead be running on a central device with more CPU power. This is not ideal, however, it should not make a difference as long as both the robots and the central device running the navigation stacks have a sufficiently fast connection. Only having the robot base running on the robots decreased the required CPU such that they no longer entered the error state, as even with the spikes, their CPU usage never hit 100 %.
With the navigation stack running from a central PC, the CPU spikes can still be seen when a carrier robot's navigation stack is started. It causes all robots currently operating to lose localization for up to several seconds causing the operating agents to collide with walls or each other, as they continue their current path but without being able to update their local position. The CPU spike is very small on the central PC compared to its limits and it is therefore not the CPU spikes that cause the localization errors but rather an overload on the ROS2 network.

Crashing Service Servers due to Networking

At a later stage, another error started to occur frequently. The swarm manager node terminated with an exception that one of its services failed to send a response to a client. This was found to happen if the service cannot find the client's return topic within 100 ms of the service call. It was found that for FastDDS specifically, when a service and a client communicate via WiFi [57], this error could cause the entire node to terminate, although termination seems out of proportion to the error. This is a well-known error that FastDDS has, but a solution has not yet been found. However, when using CycloneDDS as the ROS2 middleware, this error does not cause the node to terminate, rather, the service ignores the client and proceeds from there, making it more robust. With the navigation stack moved to a central device there was no longer a need for having a discovery server and therefore no longer a need to use FastDDS as middleware. Therefore, the system was changed to use CycloneDDS instead of FastDDS to avoid nodes with services from terminating untimely.

Time Synchronization

Occasionally, the robots would enter an erroneous state when running only the robot base on the robots. In this state, the localization would drop the messages from the LiDAR with the reason being that the message buffer is full or that getting transform data would require extrapolation into the past. It was found that this could be due to the internal time for the robots not being synchronised [58]. To ensure synchronization, Chrony [51] was added to all robots and central computers, which solved the problem.

Bibliography

[1] Yoram Koren, Uwe Heisel, Francesco Jovane, Toshimichi Moriwaki, Gumter Pritschow, et al. "Reconfigurable manufacturing systems". In: *CIRP annals* 48.2 (1999), pp. 527–540.

[2] Michael Trierweiler, Petra Foith-Förster, and Thomas Bauernhansl. "Changeability of Matrix Assembly Systems". In: *Procedia CIRP* 93 (Jan. 2020), pp. 1127–1132. ISSN: 2212-8271. DOI: 10.1016/J.PROCIR.2020.04.029.

[3] Casper Schou, Akshay Avhad, Simon Bøgh, and Ole Madsen. "Towards the Swarm Production Paradigm". In: *Lecture Notes in Mechanical Engineering* (2022), pp. 105–112. ISSN: 21954364. DOI: 10.1007/978-3-030-90700-6_11/FIGURES/2. URL: https://link.springer.com/chapter/10.1007/978-3-030-90700-6_11.

[4] Heiko Hamann. *Swarm Robotics: A Formal Approach*. Springer, 2018. ISBN: 9783319745268. DOI: 10.1007/978-3-319-74528-2. URL: https://doi.org/10.1007/978-3-319-74528-2.

[5] Ignacio Rodriguez, Rasmus S. Mogensen, Allan Schjorring, Mohammad Razzaghpour, Roberto Maldonado, et al. "5G swarm production: Advanced industrial manufacturing concepts enabled by wireless automation". In: *IEEE Communications Magazine* 59.1 (Jan. 2021), pp. 48–54. ISSN: 15581896. DOI: 10.1109/MCOM.001.2000560.

[6] C. Santiago Morejon Garcia, Rasmus Liborius Bruun, Filipa S.S. Fernandes, Troels B. Sorensen, Nuno K. Pratas, et al. "Robust Decentralized Cooperative Resource Allocation for High-Dense Robotic Swarms by Reducing Control Signaling Impact". In: *IEEE Access* 10 (2022), pp. 111477–111492. ISSN: 21693536. DOI: 10.1109/ACCESS.2022.3213307.

[7] Mikell Groover. *Automation, production systems, and computer-integrated manufacturing*. eng. Fifth edition, Gl... Always Learning. Boston: Pearson, 2015. ISBN: 1-292-07611-9.

[8] Akshay Avhad, Casper Schou, and Ole Madsen. *Topology Planning in Swarm Production System: Framework and Optimization*. EasyChair, 2022.

[9] Francesco Bullo, Emilio Frazzoli, Marco Pavone, Ketan Savla, and Stephen L. Smith. "Dynamic Vehicle Routing for Robotic Systems". In: *Proceedings of the IEEE* 99.9 (2011), pp. 1482–1504. DOI: 10.1109/JPROC.2011.2158181.

[10] Servet Hasgül, Inci Saricicek, Metin Ozkan, Osman Parlaktuna, S Hasgül, et al. "Project-oriented task scheduling for mobile robot team". In: *Journal of Intelligent Manufacturing* 20 (2009), pp. 151–158. DOI: 10.1007/s10845-008-0228-8.

[11] Absolute Traffic Management. *Warehouse Traffic Management for Safety*. 2021. URL: https://absolutetraffic.com.au/warehouse-traffic-management/ (visited on 04/18/2023).

[12] Valerio Digani, Lorenzo Sabattini, Cristian Secchi, and Cesare Fantuzzi. "Ensemble Coordination Approach in Multi-AGV Systems Applied to Industrial Warehouses". In: *IEEE Transactions on Automation Science and Engineering* 12.3 (July 2015), pp. 922–934. ISSN: 15455955. DOI: 10.1109/TASE.2015.2446614. URL: https://ieeexplore.ieee.org/document/7151854.

[13] Valerio Digani, Lorenzo Sabattini, and Cristian Secchi. "A probabilistic Eulerian traffic model for the coordination of multiple AGVs in automatic warehouses". In: *IEEE Robotics and Automation Letters* 1.1 (Jan. 2016), pp. 26–32. ISSN: 23773766. DOI: 10.1109/LRA.2015.2505646. URL: https://ieeexplore.ieee.org/document/7347373.

[14] Sebastian Mai and Sanaz Mostaghim. "Collective Decision-Making for Conflict Resolution in Multi-Agent Pathfinding". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 13491 LNCS (2022), pp. 79–90. ISSN: 16113349. URL: https://link.springer.com/chapter/10.1007/978-3-031-20176-9_7.

[15] Jen Jen Chung, Carrie Rebhuhn, · Connor Yates, Geoffrey A Hollinger, and · Kagan Tumer. "A multiagent framework for learning dynamic traffic management strategies". In: *Autonomous Robots* 43 (2019), pp. 1375–1391. DOI: 10.1007/s10514-018-9800-z. URL: https://doi.org/10.1007/s10514-018-9800-z.

[16] Howie Choset, Kevin M. Lynch, Seth Hutchinson, George A. Kantor, Wolfram Burgard, et al. *Principles of robot motion theory, algorithms, and implementation*. eng. Intelligent robotics and autonomous agents. Cambridge, Mass: MIT Press, 2005. ISBN: 0-262-30395-7.

[17] Roni Stern. "Multi-agent path finding – an overview". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 11866 LNAI (2019), pp. 96–115. ISSN: 16113349. URL: https://link.springer.com/chapter/10.1007/978-3-030-33274-7_6.

[18] Anton Andreychuk and Konstantin Yakovlev. "Two Techniques That Enhance the Performance of Multi-robot Prioritized Path Planning". In: *arXiv* (2018). arXiv: 1805.01270v1.

[19] David Silver. "Cooperative Pathfinding". In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 1.1 (2021), pp. 117–122. ISSN: 2334-0924. DOI: 10.1609/AIIDE.V1I1.18726. URL: https://ojs.aaai.org/index.php/AIIDE/article/view/18726.

[20] Ryan Luna and Kostas E. Bekris. "Efficient and complete centralized multi-robot path planning". In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Sept. 2011), pp. 3268–3275. DOI: 10.1109/IROS.2011.6095085. URL: http://ieeexplore.ieee.org/document/6095085/.

[21] Hennie de Harder. *Constraint Programming Explained. The core of a constraint programming... | by Hennie de Harder | Towards Data Science*. URL: https://towardsdatascience.com/constraint-programming-explained-2882dc3ad9df (visited on 04/18/2023).

[22] Peter R. Wurman, Raffaello D'Andrea, and Mick Mountz. "Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses". In: *AI Magazine* 29.1 (Mar. 2008), pp. 9–9. ISSN: 2371-9621. DOI: 10.1609/AIMAG.V29I1.2082. URL: https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/2082.

[23] Hang Ma, Jiaoyang Li, T K Satish Kumar, and Sven Koenig. "Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks *". In: *Arxiv* (2017). arXiv: 1705.10868v1.

[24] Hang Ma, Wolfgang Hönig, T. K. Satish Kumar, Nora Ayanian, and Sven Koenig. "Lifelong Path Planning with Kinematic Constraints for Multi-Agent Pickup and Delivery". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01 (July 2019), pp. 7651–7658. ISSN: 2374-3468. DOI: 10.1609/AAAI.V33I01.33017651.

[25] Minghua Liu, Hang Ma, Jiaoyang Li, and Sven Koenig. "Task and Path Planning for Multi-Agent Pickup and Delivery". In: *AAMAS '19: Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems* (2019), pp. 1152–1160. URL: https://dl.acm.org/doi/10.5555/3306127.3331816.

[26] Ngai Meng Kou, Cheng Peng, Hang Ma, T. K. Satish Kumar, and Sven Koenig. "Idle Time Optimization for Target Assignment and Path Finding in Sortation Centers". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.06 (Apr. 2020), pp. 9925–9932. ISSN: 2374-3468. DOI: 10.1609/AAAI.V34I06.6547. URL: https://ojs.aaai.org/index.php/AAAI/article/view/6547.

[27] Oren Salzman and Roni Stern. "Research Challenges and Opportunities in Multi-Agent Path Finding and Multi-Agent Pickup and Delivery Problems". In: *AAMAS '20: Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems* (2020), pp. 1711–1715. URL: https://dl.acm.org/doi/10.5555/3398761.3398959.

[28] Tsz-Kit Lau Lau and Biswa Sengupta. "The Multi-Agent Pickup and Delivery Problem: MAPF, MARL and Its Warehouse Applications". In: *Arxiv* (2022). arXiv: 2203.07092v1.

[29] Ole Madsen and Charles Møller. "The AAU Smart Production Laboratory for Teaching and Research in Emerging Digital Manufacturing Technologies". In: *Procedia Manufacturing* 9 (Jan. 2017), pp. 106–112. ISSN: 2351-9789. DOI: 10.1016/J.PROMFG.2017.04.036.

[30] *RPlidar A3 – SLAMTEC Global Network*. URL: https://www.slamtec.ai/home/rplidar_a3/ (visited on 05/16/2023).

[31] *About different ROS 2 DDS/RTPS vendors — ROS 2 Documentation: Humble documentation*. URL: https://docs.ros.org/en/humble/Concepts/About-Different-Middleware-Vendors.html (visited on 06/01/2023).

[32] ROS. *rviz/UserGuide - ROS Wiki*. URL: https://wiki.ros.org/rviz/UserGuide (visited on 05/26/2023).

[33] ROS. *rosbag - ROS Wiki*. URL: http://wiki.ros.org/rosbag (visited on 05/26/2023).

[34] *linorobot/linorobot2: Autonomous mobile robots (2WD, 4WD, Mecanum Drive)*. URL: https://github.com/linorobot/linorobot2 (visited on 05/26/2023).

[35] *linorobot/linorobot2_hardware*. URL: https://github.com/linorobot/linorobot2_hardware (visited on 05/26/2023).

[36] *AAUSmartProductionLab/linorobot2_hardware*. URL: https://github.com/AAUSmartProductionLab/linorobot2_hardware (visited on 05/26/2023).

[37] *Anders-Clement/linorobot2_hardware*. URL: https://github.com/Anders-Clement/linorobot2_hardware (visited on 05/26/2023).

[38] *Nav2 — Navigation 2 1.0.0 documentation*. URL: https://navigation.ros.org/ (visited on 05/26/2023).

[39] *Anders-Clement/linorobot2: Autonomous mobile robots (2WD, 4WD, Mecanum Drive)*. URL: https://github.com/Anders-Clement/linorobot2/ (visited on 05/26/2023).

[40] *launch/architecture.rst at humble · ros2/launch · GitHub*. URL: https://github.com/ros2/launch/blob/humble/launch/doc/source/architecture.rst (visited on 05/26/2023).

[41] Michele Colledanchise and Petter Ögren. *Behavior Trees in Robotics and AI*. Boca Raton: CRC Press, July 2018. DOI: 10.1201/9780429489105.

[42] *Introduction To Nav2 Specific Nodes — Navigation 2 1.0.0 documentation*. URL: https://navigation.ros.org/behavior_trees/overview/nav2_specific_nodes.html (visited on 06/01/2023).

[43] *Navigation Concepts — Navigation 2 1.0.0 documentation*. URL: https://navigation.ros.org/concepts/index.html#behavior-trees (visited on 06/01/2023).

[44] *Nodes Library | BehaviorTree.CPP*. URL: https://www.behaviortree.dev/docs/category/nodes-library/ (visited on 06/01/2023).

[45] *DWB Controller · GitHub*. URL: https://github.com/ros-planning/navigation2/blob/main/nav2_dwb_controller/README.md (visited on 06/01/2023).

[46] Sven Koenig. *Home Page of Sven Koenig*. URL: http://idm-lab.org/index.html (visited on 04/18/2023).

[47] *Multi-Agent-Path-Finding with no additional agent spacing - YouTube*. URL: https://www.youtube.com/watch?v=GjrJAW0br5s (visited on 05/30/2023).

[48] *Multi-Agent-Path-Finding with additional agent spacing - YouTube*. URL: https://www.youtube.com/watch?v=m_IJ6QFfRGo (visited on 05/30/2023).

[49] *Multi-agent-path-finding for 100 agents in a 64x64 cell environment - YouTube*. URL: https://www.youtube.com/watch?v=grxHwsuWfRc (visited on 05/30/2023).

[50] Randolph Franklin. *Determining if a point lies on the interior of a polygon*. URL: https://www.eecs.umich.edu/courses/eecs380/HANDOUTS/PROJ2/InsidePoly.html (visited on 05/26/2023).

[51] *chrony – Introduction*. URL: https://chrony.tuxfamily.org/ (visited on 05/26/2023).

[52] *FPSP with static queue positions - YouTube*. URL: https://www.youtube.com/watch?v=lx9IaCllenk (visited on 06/01/2023).

[53] *FPSP with dynamic queue positions - YouTube*. URL: https://www.youtube.com/watch?v=ilW_ymQ8CQo (visited on 06/01/2023).

[54] *STAP running with 5 robots - YouTube*. URL: https://www.youtube.com/watch?v=P4YGv7rGAnw (visited on 06/01/2023).

[55] Wolfgang Honig, Scott Kiesel, Andrew Tinka, Joseph W. Durham, and Nora Ayanian. "Persistent and Robust Execution of MAPF Schedules in Warehouses". In: *IEEE Robotics and Automation Letters* 4.2 (Apr. 2019), pp. 1125–1131. ISSN: 23773766. DOI: 10.1109/LRA.2019.2894217.

[56] *Use ROS 2 with Fast-DDS Discovery Server — Fast DDS 2.11.0 documentation*. URL: https://fast-dds.docs.eprosima.com/en/latest/fastdds/ros2/discovery_server/ros2_discovery_server.html (visited on 05/29/2023).

[57] Connor Anderson. *Unhandled Terminate w/ Services Over Wifi Network*. 2022. URL: https://github.com/ros2/ros2/issues/1253 (visited on 04/27/2023).

[58] Leonti. *Time synchronization approaches*. 2021. URL: https://answers.ros.org/question/379750/time-synchronization-approaches/ (visited on 04/27/2023).