CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering

Department of Computer Science

Bachelor's thesis

The Multi-Agent Path Finding Demonstrator

Tůma Ondřej

Open Informatics, Software

May 2022

Supervisor: RNDr. Miroslav Kulich, Ph.D.

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Tůma Ondřej**	Personal ID number:	**491867**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Software**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

The Multi-Agent Path Finding Demonstrator

Bachelor’s thesis title in Czech:

Demonstrátor systému plánování pro více agentů

Guidelines:

As part of the EU solution of the SafeLog project, a laboratory demonstrator with TurtleBot robots was created for trajectory planning for a group of robots in an automated warehouse. The aim of the thesis is to get acquainted with this environment and develop it further. The specific procedure is as follows:
1) Get acquainted with the current state of development of the demonstrator and the simulator for multi-agent planning (https://github.com/Kei18/mapf-IR).
2) Modify the simulator to serve as the basic user interface (GUI) of the demonstrator.
3) Display robot positions obtained from the Vicon system in the GUI.
4) Integrate the supplied components for planning and plan execution into the demonstrator.
5) Evaluate experimentally properties of the implemented system. Describe and discuss obtained results.

Bibliography / sources:

[1] K. Okumura, Y. Tamura and X. Défago, "Iterative Refinement for Real-Time Multi-Robot Path Planning," 2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2021, pp. 9690-9697, doi: 10.1109/IROS51168.2021.9636071.
[2] A. Andreychuk T. Rybecky M. Kulich, K. Yakovlev. On the application of prioritized safe-interval path planning with kinematic constraints to the single-shot pickup and delivery problem. 17th International Conference on Informatics in Control, Automation and Robotics, 2020.
[3] Tomáš Rybecký: Trajectory planning for a heterogeneous team in an automated warehouse, diploma thesis, FEL, CTU in Prague, 2020

Name and workplace of bachelor’s thesis supervisor:

RNDr. Miroslav Kulich, Ph.D. Intelligent and Mobile Robotics CIIRC

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **02.02.2022** Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

RNDr. Miroslav Kulich, Ph.D.	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature		Dean’s signature

III. Assignment receipt

.

Date of assignment receipt	Student’s signature

Acknowledgement

I would like to thank my supervisor RNDr. Miroslav Kulich, Ph.D., for excellent mentoring and guidance and for providing great advice when I needed it. I would also like to thank Ing. Tomáš Rybecký for providing the base software solution and for introducing me to the problem domain. Finally, I would like to acknowledge the co-authors of the original Fleet Management System and its components, mainly Ing. Lukáš Bertl and Ing. Jakub Hvězda Ph.D. The original mapfIR project also rightfully deserves appreciation.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date 18. May 2022

. . . .

Abstrakt / Abstract

Tato práce analyzuje a vylepšuje demonstrátor robotů pro automatizované sklady využívaný na pracovišti CIIRC. Hlavním úkolem této práce je vylepšit stávající software, přidat nové funkcionality a umožnit použítí formátu kompatibilního s *mapfIR* projektem. Výsledný software by měl zásadně zlepšit dojem z práce s demonstrátorem pro studenty, kteří budou v budoucnu vyvíjet algoritmy pro *mapfIR* solver. Nedílnou součástí práce je také vylepšení celkového dojmu při prezentaci externím subjektům.

Klíčová slova: software, vizualizace, robot

This work analyzes and improves upon the multi-agent planning system demonstrator currently used at the Czech Institute of Informatics, Robotics and Cybernetics. The main objective is to improve existing functionality, add new features and enable the usage of *mapfIR* format. The final software should significantly improve the developer experience for future students developing algorithms for the *mapfIR* solver, enhance the impression when presenting to 3^{rd} parties, and provide a better user experience overall.

Keywords: software, visualization, robot, pathfinding

Contents /

/ Figures

Abbreviations

- *Cmake* - C++ build system with a homepage at `https://cmake.org/`
- *Makefile* - Originally Unix C build system with a homepage at `https://www.gnu.org/software/make/`
- *mapfIR* - mapfIR project with source code available at `https://github.com/Kei18/mapf-IR`
- *OF* - openFrameworks C++ framework with source code available at `https://openframeworks.cc/`
- *original project* - Original FleetControl project with source code located at `https://gitlab.ciirc.cvut.cz/rybectom/SIPPDemonstrator`
- *XML-based format* - the IO format used in the *original project*
- *SafeLog project* - The SafeLog project `http://safelog-project.eu/`
- *Vicon* - Vicon motion capture system `https://www.vicon.com/`
- ROS - Robot Operating System

Chapter 1
Introduction

With the increased demand in the logistics industry in recent years, new solutions enabling higher efficiency and accuracy are necessary. Autonomous robot fleet systems are at the frontier of used solutions, enabling logistics providers to operate at a much larger scale. New algorithms are constantly being developed for this purpose. With that comes the necessity to test them and evaluate their properties. The real-world properties of such algorithms might not be obvious when they are developed in the academic environment. It is, therefore, necessary to create conditions similar to the ones in real warehouses in order to design the best algorithms that will be able to avoid situations that might arise only due to otherwise unexpected causes.

The goal of this thesis is to improve the software for autonomous warehouse demonstrations currently used by the Intelligent and Mobile Robotics Group (IMR) at Czech Institute of Informatics, Robotics and Cybernetics (CIIRC).

1.1 Current state

Currently, the IMR is operating a demonstrator of autonomous warehouse robots. Its main purpose is the execution of a solution to a pathfinding problem. The problem usually consists of several agents with assigned starting points and a set of goals that the agents have to arrive to. The solution is then created, assigning each agent a path to a specific goal. The demonstrator is part of the *original project*. The version currently used is hard to set up, supports only a proprietary format, and creates additional friction in the workflow. Moreover, the demonstrator's output is text-only, making it generally unappealing to work with. It enables only basic functionality for the simulations, which mainly include:

1. Loading generated simulation data (in the *XML-based format*)
2. Run against physical robots in a lab with commands in a predefined order

It does the basics necessary to run the simulation, but it does not enable the user to adjust or view the result. The information flow is limited, as the only information relevant to the execution is the feedback from robots after command execution. It has no continuous data about real-world positions. If a collision happens between two robots, the executor has no way of detecting and adjusting the robot's paths.

The IMR also experiments with the *mapfIR* project [1], a software-only solution for the generation and visualization of similar plans. The current data representation format used by the demonstrator is XML-based. It is not compatible with *mapfIR*'s input nor output format, which makes the software at its current state useless for students developing algorithms for the *mapfIR* solver.

Another disadvantage is the substantial amount of work and knowledge required to set up a basic demonstration run. This substantially limits possible use-cases for this project, as the setup takes significantly more time than the actual demonstration run. By making the setup process more straightforward, the project could be used more often, increasing its perceived usefulness and likely increasing its chances of getting future updates.

1.2 End goal

The goal is to create a version of the demonstrator that would simplify the setup process, improve the experience for developers working on an algorithmic solution with the *mapfIR* format, allowing them to switch between the original visualizer and this demonstrator seamlessly, and enable the possibility of presenting it to external visitors for representational purposes (i.e., possible applicants) by enhancing the user interface.

The Graphical User Interface (GUI) should enable the user to set up demonstration settings and interact with the system during the demonstration run.

Among other goals is the incorporation of the *Vicon* system, allowing for real-time tracking of robots. Another option available for getting real-time positions that will be used is the feedback from the robots themselves.

The department will benefit both internally and externally from this work. The internal benefits are mainly increased productivity of future developers working on this project and lesser time spent on the demonstration setup. The external benefits include the additional possibility of presenting to future applicants. The complete list of features should include:

1. Simplified demonstration setup
2. Purely virtual run
3. GUI with information about the current run and the ability to modify the run, such as disabling individual agents.
4. Standardized input and output formats with *mapfIR*
5. *Vicon* system integration
6. Ability to display real-time robot positions (*Vicon*/robots)
7. Easy integration of future changes to the demonstration execution

The rest of the text is as follows. *Chapter 2* analyzes the *mapfIR* project to help us understand the formats it uses and simplify further development. *Chapter 3* analyzes the current state of the software used as a basic prerequisite for its enhancement. *Chapter 4* goes over implementation decisions for the creation of the final software, while *Chapter 5* describes the actual implementation details. *Chapter 6* sums up the workflow and user experience and compares the original and new software. *Chapter 7* serves as the summary of this thesis.

Chapter 2
Analysis of the mapf-IR project

In this chapter, I will take a detailed look at the inner workings of the *mapfIR* project. This chapter also aims to point out the differences in formats and inner workings that might play an important role during implementation.

2.1 Introduction

mapfIR is a simulator and visualizer of Multi-Agent Path Finding (MAPF) [1]. Given a graph of connected nodes, a list of agents with their initial locations, and a set of target destinations, a solution of MAPF is a set of collision-free paths connecting the initial locations with target destinations. *mapfIR* is written in C++(17). Its main appeal for us is the ease of use and the usage of human-readable input and output formats that are generally easy to understand.

The project consists of 2 parts.

- **The solver** is a standalone *Cmake* application. It generates a simple CLI binary executable that allows the user to generate an execution plan based on a pathfinding problem definition. It supports multiple planning algorithms that can be switched using the command-line arguments.
- **The visualizer** is based on openFrameworks (*OF*) [2] and is built using *Makefile*. It is used to visualize the output of the solver once it has been generated. The user is able to change the visualization speed and highlight particular robots.

This split is necessary, as described in Section 4.2. The general workflow is to:

1. Run the solver on a problem set as described in Section 2.2.1, creating output in the format described in Section 2.2.2
2. Visualize the solution using the visualizer described in Section 2.3

Although the project description says it does support only macOS, there seems to be no problem compiling and running under Linux. This appears to be coming from the lack of documentation and documentation in an outdated state whenever it actually exists. This seems to be a common pattern for the *OF* applications and the project components as well.

2.2 The solver

This section aims to describe the solver and analyze the input and output formats used by the solver. An understanding of the formatting used will be necessary when migrating our application from the *XML-based format*.

The solver is used to generate an execution plan in discreet time steps. It focuses solely on the generation of static plans. It currently supports the following algorithms. Hierarchical Cooperative A* and Windowed Hierarchical Cooperative A* [3], Priority Inheritance with Backtracking (PIBT) [4], Conflict-based search (CBS) [5], Iterative Conflict-based search (ICBS) [6], A Bounded-Suboptimal Search variant of CBS (ECBS) [7], Revisit Prioritized Planning [8], Push and Swap [9], Extended Prioritized Algorithm (winPIBT) [10], and Iterative Refinement (IR).

2.2.1 Input format

The input is a description of the pathfinding problem that is given to the solver, which in turn, tries to create a solution for a given problem set.

```
map_file=arena.map
agents=300
seed=0
random_problem=0
max_timestep=500
max_comp_time=30000
36,33,5,23
3,17,23,33
30,15,9,27
19,35,42,30
8,21,2,18
```

The input format is illustrated above. The file starts with the name of a map file on the first line, continuing with the number of agents on the second line, followed by the seed of the problem, `max_timestep`, and `max_comp_time`. The seed is used to preserve the information about how to problem was generated and enable its regeneration. The `max_timestep` gives us the information about the total number of time steps in the simulation. The `max_comp_time` is used for debugging and performance monitoring and will not be important for the purpose of this thesis. The next „number of agents“ lines are in the format s_x s_y g_x g_y where s_x and s_y are the x and y coordinates of the starting point, respectively, and g_x, g_y are the x and y coordinates of the goal, respectively.

2.2.2 Output format

The output file is a solution to a given pathfinding problem on a given map, with additional metadata included.

```
instance=instances/arena_300agents_1.txt
agents=300
map_file=arena.map
solver=IR_HYBRID
solved=1
soc=12085
lb_soc=9722
makespan=80
lb_makespan=80
comp_time=414
```

```
11  iter=0,comp_time=20,soc=13284,makespan=13284
12  starts=(36,33),(3,17),(30,15),(19,35),...
13  goals=(5,23),(23,33),(9,27),(42,30),...
14  solution=
15  0:(36,33),(3,17),(30,15),(19,35),...
16  ...
17  80:(5,23),(23,33),(9,27),(42,30),...
```

The output format is as illustrated above, with the vital information being the number of agents on line 2, followed by the coordinates of starting points on line 12 in the format $(x_1,\ y_1)$, $(x_2,\ y_2)$... $(x_m,\ y_m)$, where x and y are the respective coordinates for each agent identified by the index of its coordinates on the row, and m is the number of agents. Followed by goals in the same format. Following are „makespan“ lines, each representing a single discrete timestep in the simulation, where each line has the coordinates of each agent in the above-mentioned format. The description of the additional, not so important (in the scope of this thesis) options is the following. The `instance` line stores information about the problem set used to generate the solution. The `map_file` option informs us about the map file used for the generation of the output. The `solver` and `solved` fields provide us with information about the used solver algorithm and whether the search for a solution was successful, respectively. The `soc` and `lb_soc` store information about the sums of cost for the whole simulation. It is generally used when comparing different algorithms for the same problem set. The `comp_time` line informs us about the computation time required to generate the solution and is generally used for benchmarking.

2.2.3 Map format

Below is an example map file. This file format is used both by the solver and visualizer to represent the environment. On the first line is the map type, represented by a string. Following are two lines representing the dimensions. On the 5^{th} line starts the actual map represented by m rows and n columns, where m and n are the height and width of the grid dimensions, respectively. Each grid tile is represented by a single ASCII character. Currently, only two characters are used.

- `.` - Representing blank space
- `T` - Representing an obstacle

```
1   type octile
2   height 14
3   width 13
4   map
5   T....TTT....T
6   T...........T
7   .............
8   .............
9   .............
10  .............
11  .............
12  ......T......
13  .............
14  .............
```

```
.............
.............
....T.T.T....
T..TTTTT....T
```

2.2.4 Coordinate system convention

All positional data assumes a grid representation of the simulated map. The representation of the data expects the coordinate (0, 0) to be the top-left corner of the simulation grid, while $(x_m - 1, y_m - 1)$ is expected to be the bottom-right corner of the grid, where x_m, y_m is the number of columns and rows in the simulation grid, respectively. This perception comes from the fact that in GUI applications, the coordinate system grows from the upper left corner.

2.3 The visualizer

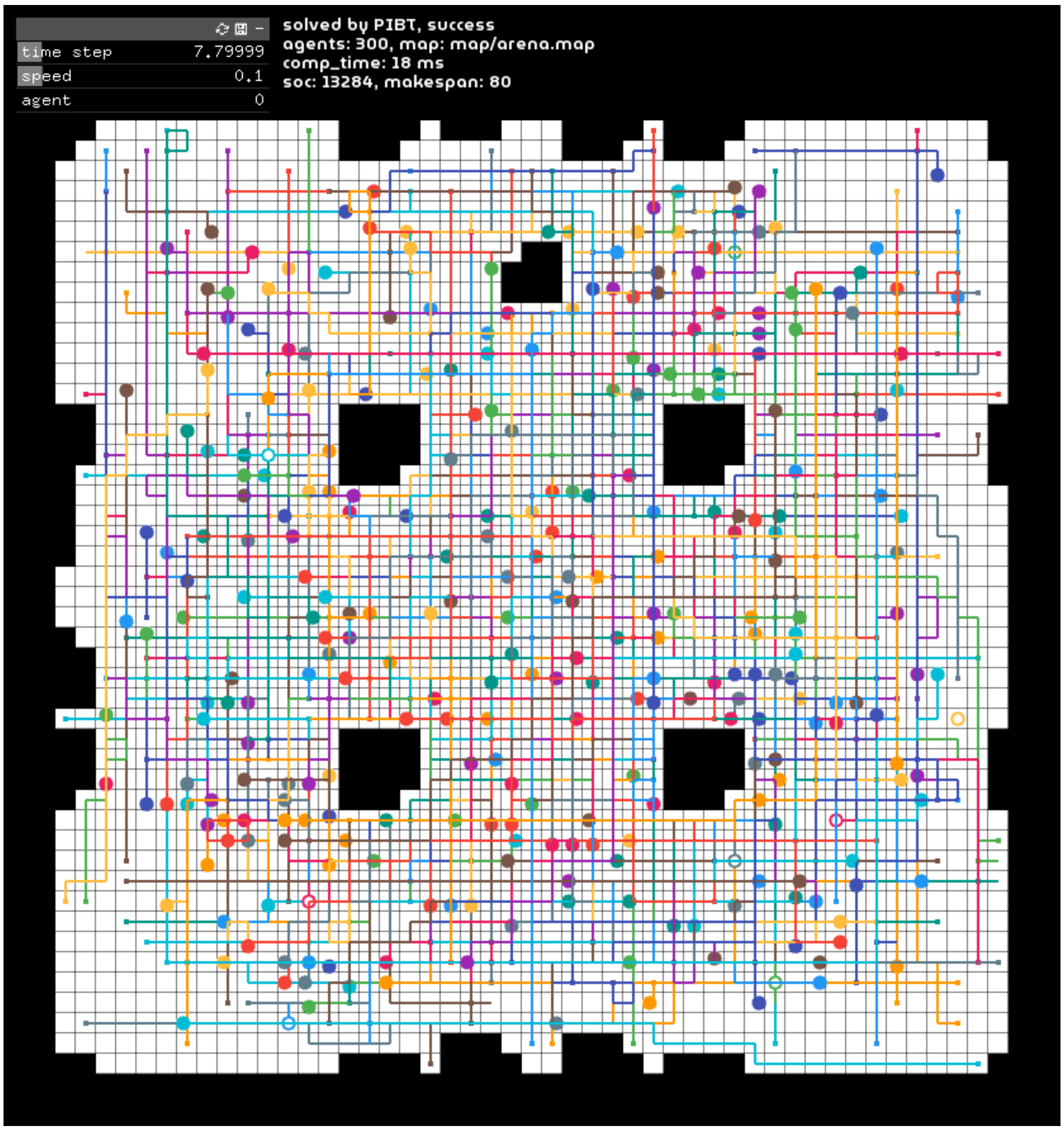
The visualizer uses input files conforming to the format described in Sections 2.2.2 and 2.2.3 to visualize the run of the generated simulation. It lets the user change the simulation speed and move through discrete time steps. Other features include selecting only a single agent to display, stopping the agent during the simulation, and visual features such as hiding/showing the lines to goals. The complete UI can be seen in Figure 2.1.

2.4 Benefits

The benefits of the format used by mapfIR are mainly:

- Human readability
- Simple parsing
- Ability to use community solvers

Additionally, I believe that switching to a standardized IO format will enable more students to use the demonstrator, and eventually introduce new use-cases that were not possible before due to the inaccessibility of the demonstrator. The ability to execute the generated execution plan on real robots might greatly increase the attractiveness of the field for students developing algorithms for pathfinding.



```
time step    7.79999
speed        0.1
agent        0
```

```
solved by PIBT, success
agents: 300, map: map/arena.map
comp_time: 18 ms
soc: 13284, makespan: 80
```

Figure 2.1. mapf-IR GUI

Chapter 3
Analysis of the FleetControl demonstrator

In this chapter, I will take a detailed look at the inner workings of the original Fleet-Control project. The purpose of this chapter is also to point out differences in formats and inner workings that we need to migrate in order to achieve full compatibility.

3.1 Project history

The original project was developed as part of the *SafeLog project*, which aimed to study and enable the cooperation of people and robots in a single robotized warehouse environment. Further work was done by Tomáš Rybecký as part of his thesis [11].

3.2 Input format

All data-related input and output files used in the original project are XML-based. The XML format has numerous advantages for advanced use-cases that need more metadata, but at the same time, it is hard to read by human operators and its parsing, as currently implemented, is overly complicated.

3.2.1 Map file

The map file `Projekt_mapa4.xml` in the root of the project was used as the source for the navigation system on robots. The file is in XML format and consists of a relatively simple definition of nodes that are directly referred to by id from the bar codes placed on the laboratory floor. The rest of the map file was unused but kept for possible backward compatibility. It includes edges definition, describing the possible routes.

Example node

```
<PickStationNode Id="464" X="3.5" Y="2.1">
```

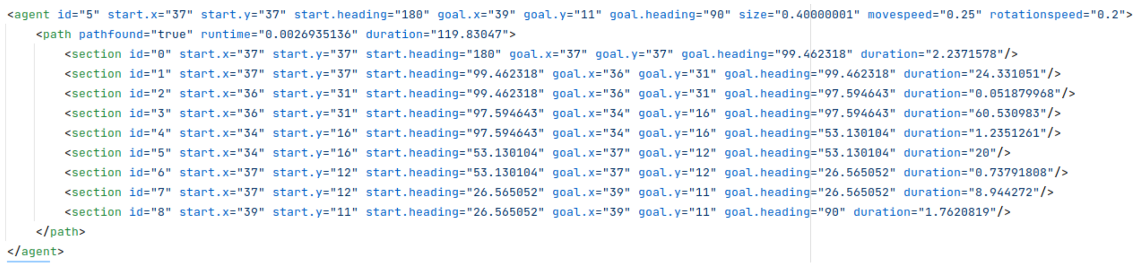
3.2.2 Coordinate system convention

The coordinate system representation in the file is independent of the rotation of the whole coordinate system. It is, by convention, expected to begin in the lower-left corner of the imaginary grid, growing linearly to the upper right corner of the grid. The rotation of the coordinate system plays a role only in the physical robots; therefore, keeping the convention standardized across both components simplifies the potential visualization of the plans and makes it simpler to imagine the final plan without the need to perform multi-dimensional rotations. This is in contrast to the coordinate system used by *mapfIR* described in Section 2.2.4, which begins in the upper left corner.

3.2.3 Execution plan

The execution plan was also supplied in the XML format, providing a list of n points relative to the grid coordinates for each robot, with duration for each step. The coordinates were scaled relative to the grid coordinate system, and it was necessary to provide this scaling information at runtime for successful coordinate transformation. This information is, however, not encoded in the file itself.

I'd also like to point out that, in contrast to the solution generated by *mapfIR*, this execution plan does not use discreet time steps to synchronize the whole simulation. It uses the timing information provided in each command definition to avoid collisions and ensure synchronization across commands. A single robot execution plan can be seen in Figure 3.1



```
<agent id="5" start.x="37" start.y="37" start.heading="180" goal.x="39" goal.y="11" goal.heading="90" size="0.40000001" movespeed="0.25" rotationspeed="0.2">
    <path pathfound="true" runtime="0.0026935136" duration="119.83047">
        <section id="0" start.x="37" start.y="37" start.heading="180" goal.x="37" goal.y="37" goal.heading="99.462318" duration="2.2371578"/>
        <section id="1" start.x="37" start.y="37" start.heading="99.462318" goal.x="36" goal.y="31" goal.heading="99.462318" duration="24.331051"/>
        <section id="2" start.x="36" start.y="31" start.heading="99.462318" goal.x="36" goal.y="31" goal.heading="97.594643" duration="0.051879968"/>
        <section id="3" start.x="36" start.y="31" start.heading="97.594643" goal.x="34" goal.y="16" goal.heading="97.594643" duration="60.530983"/>
        <section id="4" start.x="34" start.y="16" start.heading="97.594643" goal.x="34" goal.y="16" goal.heading="53.130104" duration="1.2351261"/>
        <section id="5" start.x="34" start.y="16" start.heading="53.130104" goal.x="37" goal.y="12" goal.heading="53.130104" duration="20"/>
        <section id="6" start.x="37" start.y="12" start.heading="53.130104" goal.x="37" goal.y="12" goal.heading="26.565052" duration="0.73791808"/>
        <section id="7" start.x="37" start.y="12" start.heading="26.565052" goal.x="39" goal.y="11" goal.heading="26.565052" duration="8.944272"/>
        <section id="8" start.x="39" start.y="11" start.heading="26.565052" goal.x="39" goal.y="11" goal.heading="90" duration="1.7620819"/>
    </path>
</agent>
```

Figure 3.1. Single robot execution plan

3.3 Components

The whole demonstration application consisted of two distinct parts communicating over TCP in the local network.

A custom layer built upon the nanomsg library serves as the common facilitator for all network communication across all components. It simplifies the interface provided by the library and provides unified access to the network layer. Nanomsg is a socket library that provides several common communication patterns. It aims to make the networking layer fast, scalable, and easy to use. It is implemented in C [12].

3.3.1 Server

The server component was expected to be run at static IP address `192.168.2.4`.This part of the application was responsible for reading, parsing, and converting the execution file into the grid coordinate system used by robots. Another of its responsibilities included handling the connections from individual robots. Once all connections have been established, the server would start sending individual commands to each robot, ensuring the order and timing of the executed commands.

3.3.2 Robots

The robot model used is TurtleBot 2. It is an open source, low-cost robot kit, consisting of a mobile base, single board computer, and the TurtleBot mounting hardware kit [13]. The robots are equipped with a Kinect sensor on the front, as can be seen in Figure 3.2. Vicon markers are glued to the top of the robot in a predefined constellation, as

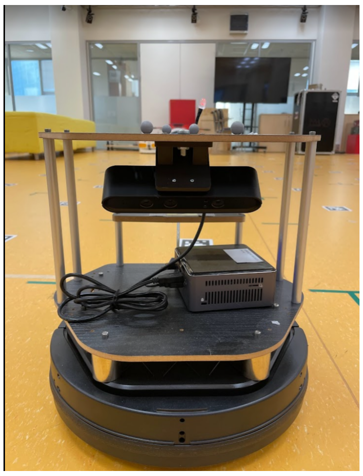


Figure 3.2. Robot front view

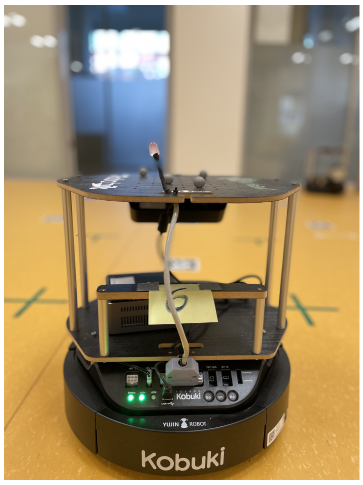


Figure 3.3. Robot rear view

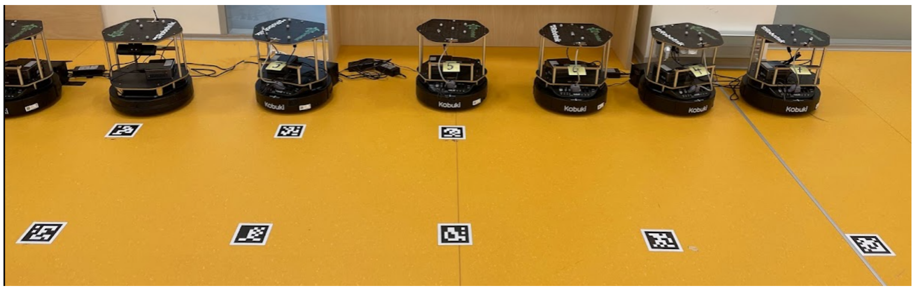


Figure 3.4. Robot fleet

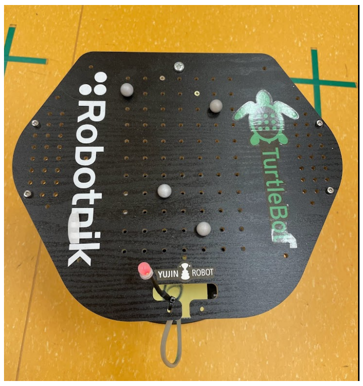


Figure 3.5. Robot top view

can be seen in Figure 3.5. Each robot is mapped to a name in the Vicon system based on the tag constellation. The complete fleet used during the demonstration can be seen in Figure 3.4, together with the bar codes used for their navigation.

The robot is running the Robot Operating System (ROS). ROS [14] is a set of software libraries and tools for building robot applications, providing advanced algorithms and complete framework for working with robots. The base of the software running on robots consists of a ROS package, a module responsible for navigation on the grid, and a simple communication layer executing the commands received from the server. This layer will be referred to as `runner` in the rest of this thesis. The robots were only responsible for executing the individual commands and had no context of the whole simulation or other robots.

3.4 Environment

The laboratory is in a room with the barcode grid layed on the ground, covering roughly $100m^2$ area as shown in Figure 3.6, with the Vicon cameras mounted on the ceiling (Figure 3.7).

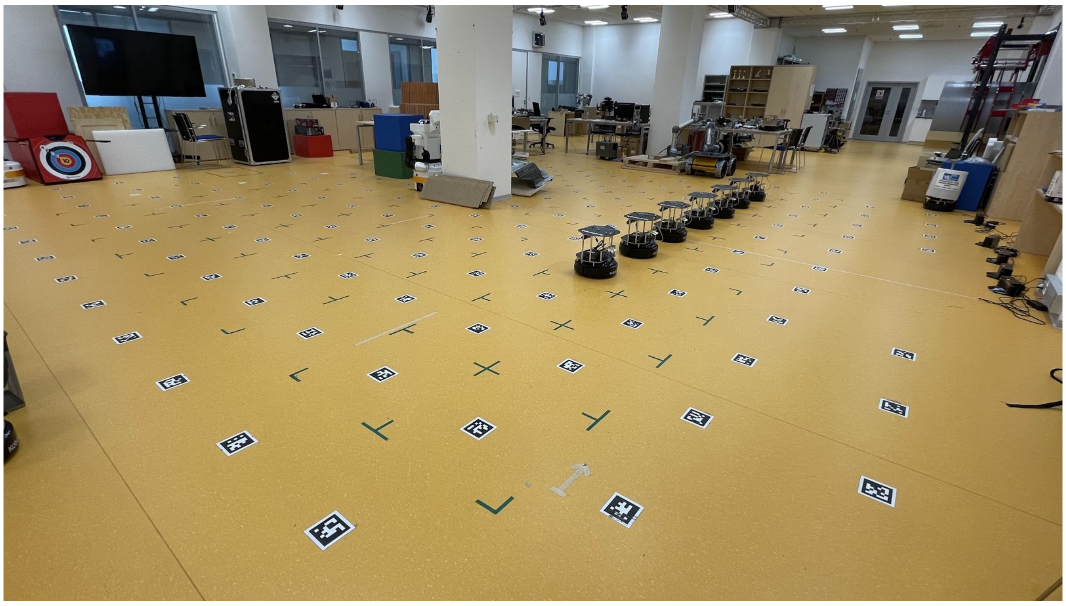


Figure 3.6. Complete area

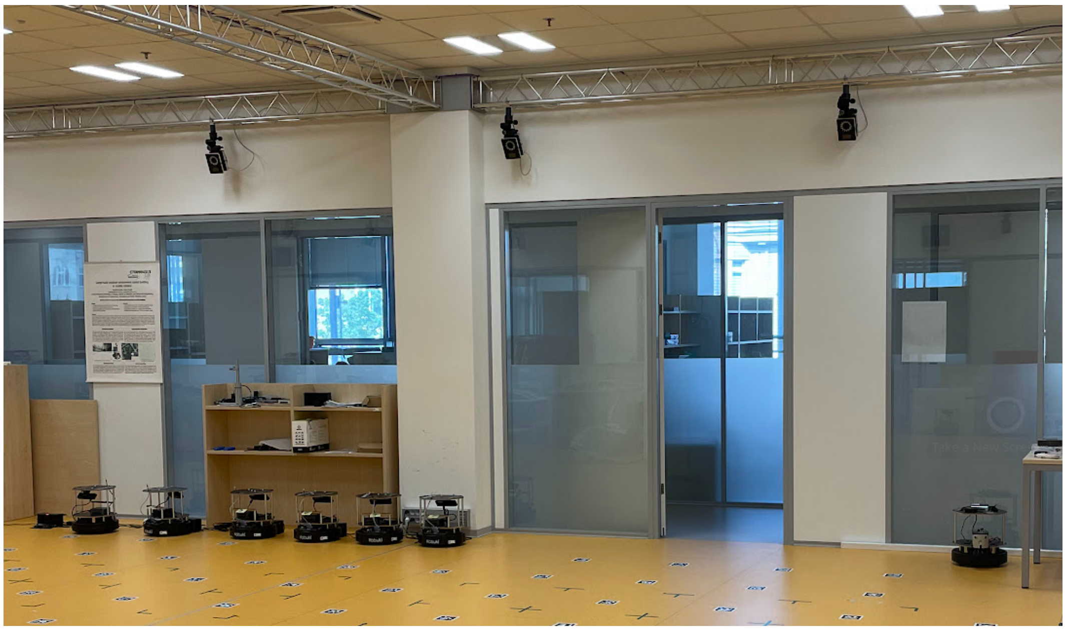


Figure 3.7. Vicon cameras

3.5 Server software analysis

The software structure was split into the server runner thread and serializable classes representing the individual messages. Other utility classes for time synchronization were also used, although their impact was negligible. The communication with robots happened in a loop in the server thread.

During a later stage, there were found some rather critical bugs in the nanomsg abstraction used by both the server and client that resulted in an array out of bounds write and corruption of the stack when the server's IP address length increased. This did not, however, affect the original project, as the server IP was static and relatively short.

3.6 Robot software analysis

3.6.1 Navigation

The navigation part proved to be well written and abstracted away the navigation problem completely. From the developer's perspective, the whole abstraction usage can be simplified to two synchronous method calls

- `int driveToPosition(double x, double y, double time);`
- `int turn (double angle);`

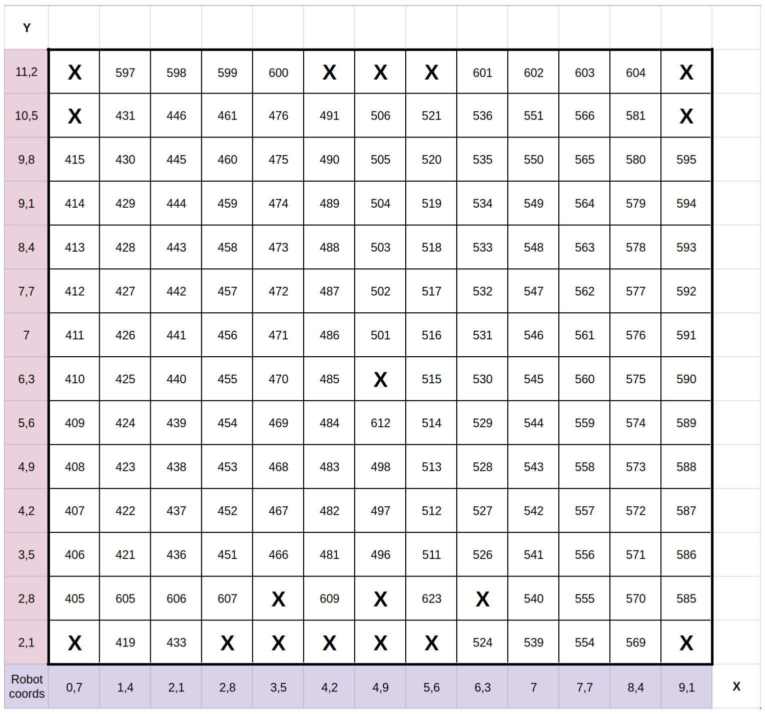
The underlying solution for the navigation system is based on the AprilTag project. It „is a visual fiducial system, useful for a wide variety of tasks including augmented reality, robotics, and camera calibration. Targets can be created from an ordinary printer, and the AprilTag detection software computes the precise 3D position, orientation, and identity of the tags relative to the camera. The AprilTag library is implemented in C with no external dependencies. It is designed to be easily included in other applications, as well as be portable to embedded devices."[15]

The internal representation of the grid can be seen below in Figure 3.8. All robot movement is calculated and realized relatively to this grid representation. The numbers on the individual grid tiles are embedded in the bar codes on the laboratory floor and are the ids of the nodes in the `Projekt_mapa4.xml` file. X represents a missing node/obstacle.

As an additional safety measure, the underlying software abstracts away direct collision handling, during which the robot stops all further execution to prevent any potential damage.

3.6.2 Runner

The runner consisted of a simple communication handler, calling the above-mentioned navigation API, performing the desired tasks, and returning the results. In case of not finishing the command, the robot would simply not send back the response, and the server would then reinitiate the command execution.



Y														
11,2	X	597	598	599	600	X	X	X	601	602	603	604	X	
10,5	X	431	446	461	476	491	506	521	536	551	566	581	X	
9,8	415	430	445	460	475	490	505	520	535	550	565	580	595	
9,1	414	429	444	459	474	489	504	519	534	549	564	579	594	
8,4	413	428	443	458	473	488	503	518	533	548	563	578	593	
7,7	412	427	442	457	472	487	502	517	532	547	562	577	592	
7	411	426	441	456	471	486	501	516	531	546	561	576	591	
6,3	410	425	440	455	470	485	X	515	530	545	560	575	590	
5,6	409	424	439	454	469	484	612	514	529	544	559	574	589	
4,9	408	423	438	453	468	483	498	513	528	543	558	573	588	
4,2	407	422	437	452	467	482	497	512	527	542	557	572	587	
3,5	406	421	436	451	466	481	496	511	526	541	556	571	586	
2,8	405	605	606	607	X	609	X	623	X	540	555	570	585	
2,1	X	419	433	X	X	X	X	X	524	539	554	569	X	
Robot coords	0,7	1,4	2,1	2,8	3,5	4,2	4,9	5,6	6,3	7	7,7	8,4	9,1	X

Figure 3.8. Robot positioning grid

3.7 Runtime behavior

The runtime behavior is consistent and reliable in its main functionality, which is inherently limited by the purpose of the software.

Due to the multithreaded nature, the server was sometimes logging to stdout multiple logs at a time, resulting in message corruption, but that was a minor inconvenience, that did not affect the executed demonstration in any way.

Chapter 4
Implementation decisions

In this chapter, I will briefly describe the implementation requirements provided and decisions that were necessary before the actual implementation.

4.1 Compatibility considerations

The *original project* uses *Cmake* build system. It provides great benefits compared to *Makefile*, mainly portability across multiple build systems. The primary decision was that we were to keep this build system for the server part of the application. This will be important later when choosing GUI in Section 4.2.

4.2 GUI library considerations

The primary candidate for the GUI library was *OF*. It provides an extensive amount of add-ons ranging greatly in functionality, with the pros being mainly:

1. Event loop synchronized with framerate
2. Interacting with the canvas using framework APIs
3. Rich set of GUI widgets (buttons, text fields, etc.)

with the only downside being its dependence on the *Makefile*. Therefore we are not able to use the official version with our build system.

The developers are rather defensive about migrating build systems. The recommended way of building *OF* app is to clone the official repository, create an app using their template, and extend the makefiles provided by the original repository. In order to be able to use this project with *Cmake*, it would be necessary to fork the original repository and create a header map for each component of the framework, potentially breaking support for unported add-ons. This would break the main benefit of the framework, extensibility, while the time investment necessary would certainly overrun the budget for this thesis.

There appear to be a few attempts to do this. Unfortunately, the vast majority of them are out of date and unusable without heavy modifications. During the preparation for this thesis, I tried to set up all of them without success, as they are out of date. Below are a few example approaches to this problem.

The first one being ofxCMake[1]. It attempts to provide the package as *OF* add-on. Unfortunately, the latest version is from Aug 13, 2017, and there seems to be no active

[1] `https://github.com/BildPeter/ofxCMake`

development keeping it in sync with current Linux distributions and versions of libraries used in them.

Another one being ofnode[1], taking a different approach. It has its own fork of the whole *OF* repository, with mappings for *Cmake*. The fork approach introduces inevitable split from the main branch, implicitly keeping it out of sync with no support for new features. The latest version is from Nov 7, 2018. I ran into several issues with system libraries with this version and was not able to compile even the bare repository.

From the forums, it seems that both approaches had been working for at least a few users for some time. The main issue with both approaches seems to be the development outside of the main repository, which led to their deprecation, as the developers had to maintain it themselves, which both of them stopped doing. It was then decided that it would be most beneficial to split the GUI into a separate application and keep using the framework as intended by the authors, as the approach to port it to *Cmake* had been tried before, and as we can see, it did not work in the long term.

The implementation will be, however, done with the possible switch to a single application in mind, trying to abstract most of the communication away from the developer, making the switch relatively easy, were the developers ever to introduce *Cmake* support.

4.3 GUI implementation considerations

The first aspect was speed and network bandwidth. The simulation might run hundreds of robots, depending on the simulation. So on each robot move, it is necessary to send the minimal amount of information required in order to keep the latency to a minimum. It was then decided that the plan would be synchronized between both applications on simulation start or plan change. Any messages regarding the robots' positions will strictly refer to the synchronized plan.

The second aspect was the ease of development. It is necessary to create enough abstraction for the developer not to have to worry about the network. A set of abstractions allowing to share messages and state in an asynchronous manner will be necessary. Ideally, the developer should not care whether the apps are running in different threads or processes, the API should remain the same.

4.4 Virtual run implementation considerations

In the original demonstrator, there was no support for a virtual run. It is, however, truly inconvenient having to have physical robots in order to evaluate an execution plan. The virtual run will also be useful for testing the server and frontend. For those cases, the server and frontend should not be able to distinguish between the real robots and the virtual run, as we should not introduce any code related to testing to the tested parts of the application. Given these constraints, the virtual run will be implemented using `virtualized robots` that will be listening and responding on the same ports as real robots would. This approach should make both use-cases as efficient as a real run while keeping test-related code to a minimum.

[1] `https://github.com/ofnode/of`

4.5 Coordinate system convention

The coordinate system of the *mapfIR* project is perceived to grow from the upper left corner, as described in Section 2.2.4, while the coordinate system of the original system grows from the lower-left corner, as described in Section 3.2.2. For the convenience of the user, it is best to keep the orientations of all coordinate systems used within the new demonstrator in the same direction. Since the robots and the Vicon coordinate systems are tied to the physical equipment in the laboratory, the new system will comply with the convention used by the original system. In the *mapfIR* project, the only place where the rotation of the coordinate system should matter is during the visualization, as the solver and visualizer use the same convention. This means that the perception of the grid coordinate system will be rotated when visualized on the *mapfIR* visualizer and the new demonstrator. This might cause minor inconvenience for the developers working with both projects, but it is necessary to keep all the components in the new system consistent without the need to perform rotations and confuse the developer while working with the software.

Chapter 5
Implementation

In this chapter, I will describe my implementation and the particular solutions used to comply with decisions made in *Chapter 4*.

5.1 Application architecture

The application is split between a server, a frontend, and a robot part. The server is a central part of the project, communicating with the frontend and the robots. The robots and the frontend communicate only with the server and are completely independent of each other.

5.1.1 Communication

All communication in the built system is based on an abstract interface built upon the same layer on top of nanomsg that has been used in the original project. This abstraction greatly simplifies the introduction of new message types and the general process of sending/receiving messages and synchronizing a global state.

The main classes used for communication are Messenger (asynchronous handler for inbound/outbound messages) and Message (serializable container), providing an easy-to-use interface and abstracting the network layer away from the programmer. The basic hierarchy of used communication classes is illustrated below.

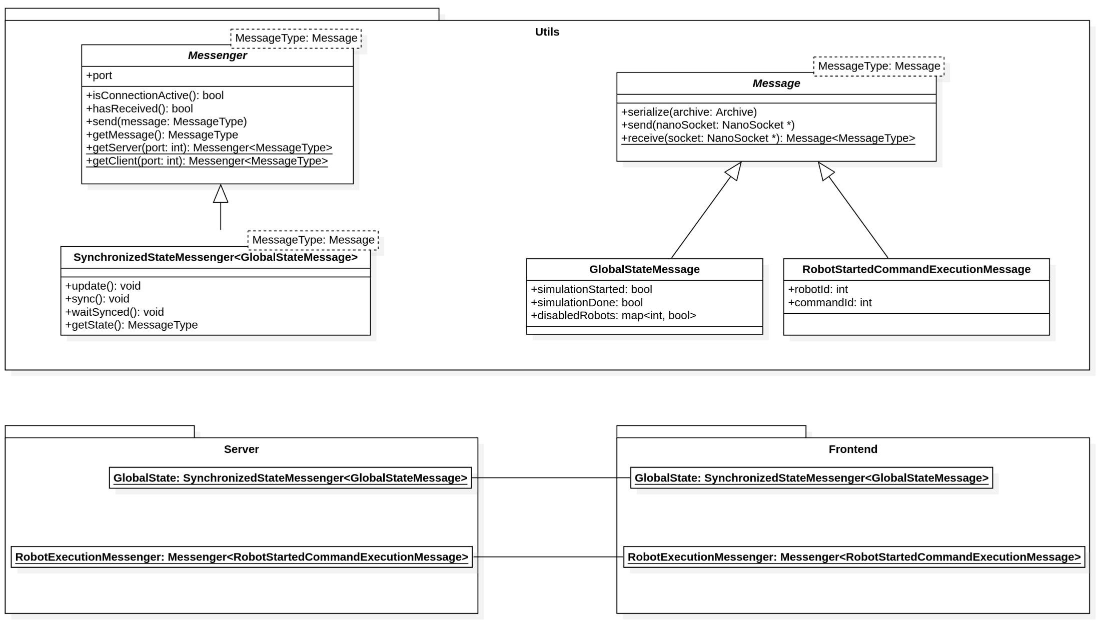

Utils
MessageType: Message
Messenger
+port
+isConnectionActive(): bool
+hasReceived(): bool
+send(message: MessageType)
+getMessage(): MessageType
+getServer(port: int): Messenger<MessageType>
+getClient(port: int): Messenger<MessageType>
MessageType: Message
SynchronizedStateMessenger<GlobalStateMessage>
+update(): void
+sync(): void
+waitSynced(): void
+getState(): MessageType
MessageType: Message
Message
+serialize(archive: Archive)
+send(nanoSocket: NanoSocket *)
+receive(socket: NanoSocket *): Message<MessageType>
GlobalStateMessage
+simulationStarted: bool
+simulationDone: bool
+disabledRobots: map<int, bool>
RobotStartedCommandExecutionMessage
+robotId: int
+commandId: int
Server
GlobalState: SynchronizedStateMessenger<GlobalStateMessage>
RobotExecutionMessenger: Messenger<RobotStartedCommandExecutionMessage>
Frontend
GlobalState: SynchronizedStateMessenger<GlobalStateMessage>
RobotExecutionMessenger: Messenger<RobotStartedCommandExecutionMessage>

Figure 5.1. Communication UML

The server and frontend each have their own instance of Messenger associated with a given Message type, communicating through API defined by a given messenger. In the case of SynchronizedStateMessenger, only the latest version of the synchronized state is publicly available outside of the class.

The asynchronous nature of the communication can be seen in the sequence diagram in Figure 5.2, where it can be clearly seen how the SynchronizedStateMessenger performs multiple actions and independently synchronizes its state, while the Frontend is free to do other actions.

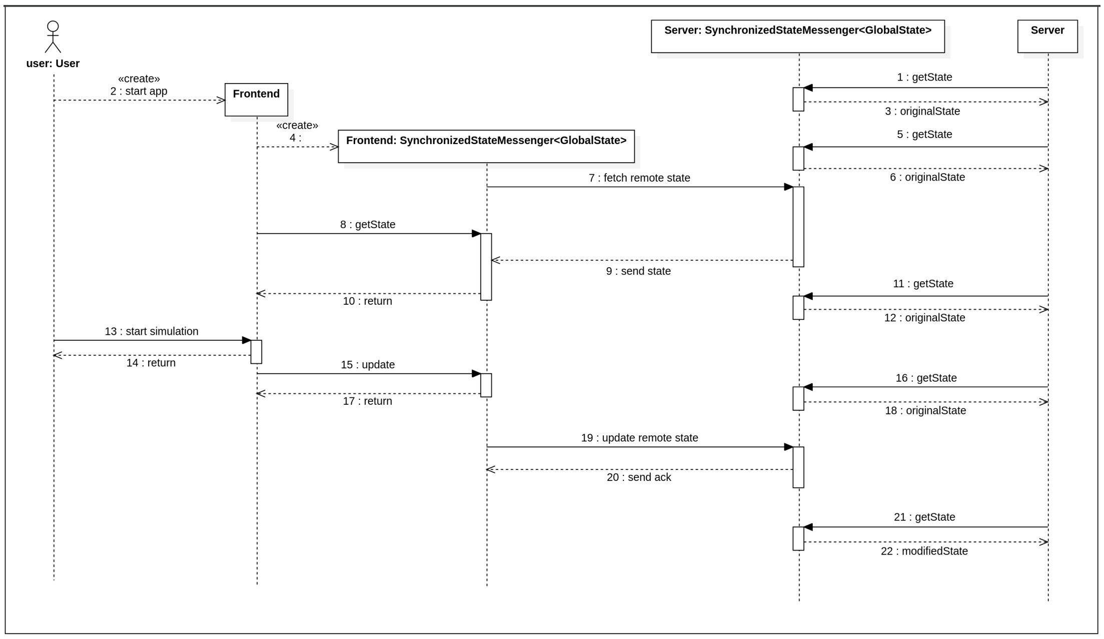


Figure 5.2. Communication sequence diagram

The asynchronous handlers allow both applications to run in the main thread performing latency-sensitive tasks. The current implementation uses a thread per messenger, mainly to eliminate application-wide locking and shared state, but another more optimal approach could be a single static event loop for all the messengers. It would be more efficient, but it could also introduce hard to debug bugs where one Messenger instance could affect every other messenger. It was decided to go with the threaded implementation. The Messenger architecture was, however, created with the possible switch to a different implementation in mind, and the possible switch should be only a matter of changing minor parts of the implementation.

During the implementation of the above-mentioned mechanisms, great measures were taken to avoid race conditions. The Messenger has an implementation of Lamport's logical clock [16], assuring the order of messages for each Message type. This is mainly to prevent the user from updating a state on the server based on a non-actual version.

5.1.2 Shared resources

Since one part of the application is built using *Makefile* and the other part using *Cmake*, it made sharing code across parts of the application more difficult. The configuration of *OF* only allows one user-defined include directory, so all the shared files have a flat structure in the `{projectRoot}/FleetControl/modules/utils/Comm` directory.

5.2 TurtleBot

The software running directly on TurtleBot is based on ROS. Its main purpose is to receive commands from the server and broadcast its position. It acts on the instructions by using the position API provided by the underlying layer described in Section 3.6.1. It also performs translations to and from its grid coordinate system. The robot is the only component that should be aware of its shifted grid relative to the standard cartesian coordinate system beginning in (0, 0).

TurtleBot software is built using CMake. The sources for the TurtleBot are in the `/src` directory, where the actual code resides in the `/src/agv_package/TurtleBot/` directory.

All relevant application information about robots has been moved to a single place. It is in a file named `robot_definitions.txt` in the root of the project. Description of each robot consists of its ID, IP address, port used for command reception, ssh user used for direct connections, name registered in the Vicon system and color displayed in the user interface. This file is used by the command-line utilities, the server, and the robots themselves, providing a single source of truth.

At runtime, the TurtleBot uses the following ports.

- single port defined in `/robot_definitions.txt` - server commands handling
- 9980 - position feedback

5.3 Server

The general idea behind the server architecture is that it should mainly handle the execution of the demonstration while aggregating data from *Vicon* and robots, and synchronizing them to the frontend.

The execution of the demonstration should be guided by the synchronization strategy (Section 5.6) currently selected.

The server code resides in the `{projectRoot}/FleetControl/modules/manager` directory. The only other code it touches should be the Shared resources further described in Section 5.1.2.

5.3.1 Vicon

The laboratory is equipped with a *Vicon* motion capture system, monitoring the area spanned by robots completely. The purpose of the system is to provide positions of defined known objects.

The system consists of a central server and high refresh rate motion cameras emplaced on the ceiling of the laboratory around the monitored area. The Vicon system is usually used in the movie industry but is by no means limited to it.

The tracking works by placing special tracking markers, as can be seen in Figure 5.3, and mapping each marker constellation to an identifier in the Vicon software. The system then automatically tracks the position, rotation, and speed of each marker constellation.

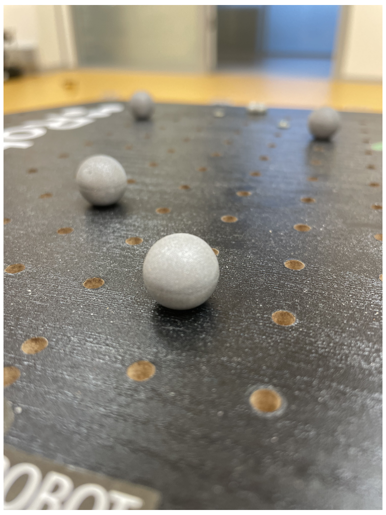


Figure 5.3. Vicon markers

Additional tooling is used to transform the source data from the *Vicon* system into a network stream that can be consumed by other applications running on the local network.

The actual usage in the server is then relatively straightforward. Each robot is assigned a name in the *Vicon* system that is referenced in the file with robot definitions in the root of the project. Afterward, when the *Vicon* system is running, we are able to transform the output data into a continuous stream of data, send it over the network to the Server, filter out only wanted objects, and save the current position provided by *Vicon* for each robot. The final position for each robot is periodically transferred to the frontend primarily to save network bandwidth. Given the high refresh rate of the positioning system and the relatively low speed of physical robots, the updating period was set to 100ms without any noticeable delay in the user interface.

5.3.2 Network communication

The server is the only component of the application that is aware of more than one other component. It serves as a central point for all other components. It communicates with the frontend, with each one of the robots and *Vicon*.

The following ports should be used at runtime.

- single port per robot for command execution
- single port per robot for position feedback
- single port for Vicon
- multiple ports for frontend connections (1 for each message type)

The message types used are associated with the following use-cases:

- sharing current Vicon positions
- sharing current robot's positions
- information about robot's command execution start
- information about robot's command execution blockage
- information about robot's command execution end
- sharing of simulation settings
- sharing of map and command list

The usage of the word „sharing“ in this context implies the usage of Synchronized-StateMessenger.

5.3.3 Virtual run

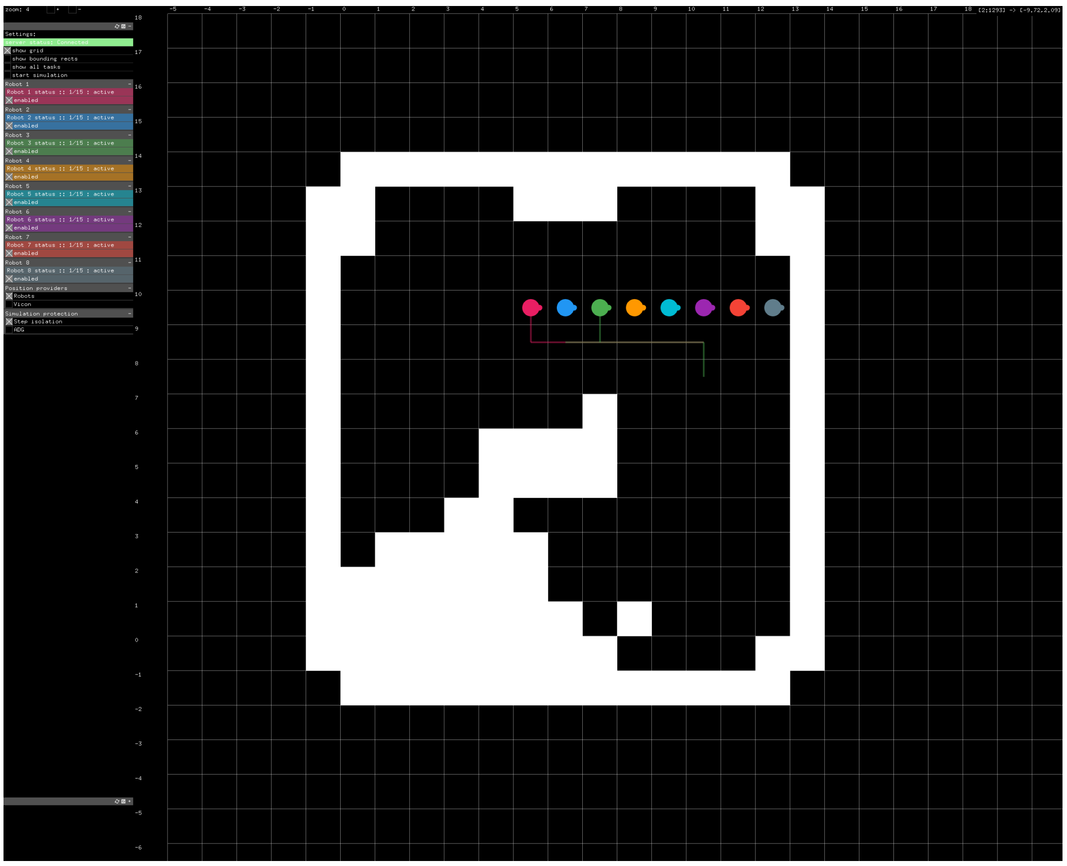
The virtual run is implemented through a class called RobotConsumer. It is started by providing the CLI option `--robot-consumers`. It runs in its own thread and, from the server's perspective, should be indistinguishable from real robots. It automatically creates sockets for each robot, allowing the server to initiate communication as it normally would. The RobotConsumer also broadcasts the robot's positions, mimicking the real behavior. The position used is the position of the last command executed if there is no command currently being executed.

The execution of a command is set up in a way that it takes a random amount of time from 1 to 5 seconds and is performed linearly, meaning that the perceived speed of the robot is constant. During the command execution, the consumer periodically broadcasts its position every 100ms. The position broadcasted during execution changes with the progress of the command linearly from the start to the end location. It is mainly used for developer convenience and for testing, as it does not require the setup of physical equipment and can be performed on a single machine.

5.4 Graphical user interface

The GUI is implemented using *OF*. It runs a single thread for main logic and event handling and another one for the GUI application. Communication with the server is directed through the Messenger abstraction, allowing uninterrupted handling of all events that might arise and providing smooth rendering for the application. All synchronous events and rendering happen in the main event loop of *OF*, providing a centralized state for the whole component. The code resides in `{projectRoot}/FleetControl/modules/app` directory

The GUI, as visualized in Figure 5.4, displays robot's positions based on position data from robots, or alternatively from a connection to the *Vicon* system, allowing the user to browse all time steps and show available information about them. The individual steps are shown after clicking on each robot, as can be seen in Figure 5.7. They can also be displayed for all the robots at once by selecting the option in the main admin panel described later. This is not used by default, as it adds a lot of unnecessary information to the GUI. They are displayed in the form of dots on the target grid tiles. When clicked, they populate the panel shown in Figure 5.6 with information about its coordinates, the order of the step in the robot's plan, and additional information about the robot, such as current connection status and IP address. The GUI also provides



```
zoom: 4   + -
Settings:
server status: Connected
[x] show grid
[ ] show bounding rects
[ ] show all tasks
[ ] start simulation
Robot 1
Robot 1 status :: 1/15 : active
[x] enabled
Robot 2
Robot 2 status :: 1/15 : active
[x] enabled
Robot 3
Robot 3 status :: 1/15 : active
[x] enabled
Robot 4
Robot 4 status :: 1/15 : active
[x] enabled
Robot 5
Robot 5 status :: 1/15 : active
[x] enabled
Robot 6
Robot 6 status :: 1/15 : active
[x] enabled
Robot 7
Robot 7 status :: 1/15 : active
[x] enabled
Robot 8
Robot 8 status :: 1/15 : active
[x] enabled
Position providers
[x] Robots
[ ] Vicon
Simulation protection
[x] Step isolation
[ ] ADG
```

```
[2:1293] -> [-9.72,2.09]
```

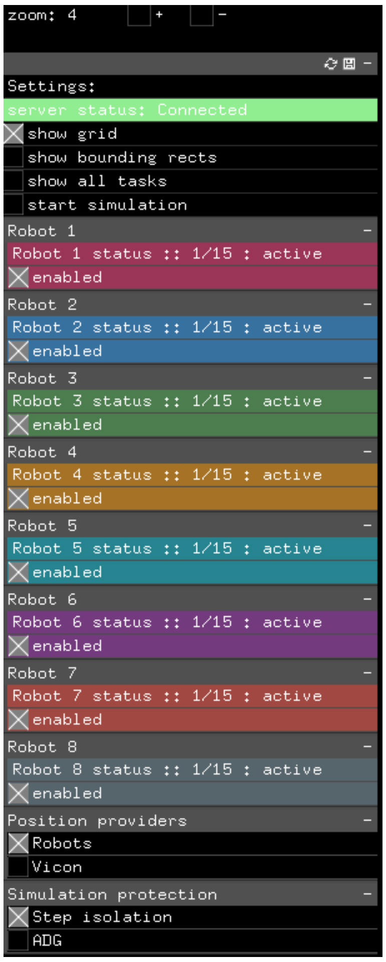
Figure 5.4. GUI

the user with the ability to disable individual agents for a given simulation run and to completely start and stop the simulation run.

Most of the settings can be changed in the admin panel (Firgure 5.5) on the left side of the application window.

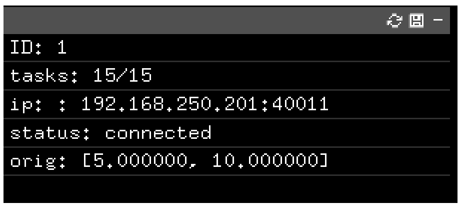
The admin panel allows the user to disable individual robots, which is useful if the battery of the robot dies, but the remaining robots are still good enough for demonstration purposes. The system then does not send new commands to the disabled robots. The options may be reset even during a demonstration run, so this option might be used to consistently test the synchronization algorithms by making a robot blocked while still being part of the execution plan. Below is the option to change the position provider for showing the real-time robot's positions in the user interface. This option does not override the robot's internal positioning system. The default option is Robot, as the *Vicon* startup time is not insignificant, the cameras have a limited lifespan, and for the majority of demonstration runs, the feedback from robots gives the same information within an acceptable margin of error. The simulation can be started using the `start simulation` button. This also allows the user to disable the simulation again after the start of the execution, preventing the server from sending any new commands to any of the active robots.

5.5 Coordinate system



```
zoom: 4    + -
Settings:
server status: Connected
[✓ show grid]
show bounding rects
show all tasks
start simulation
Robot 1
Robot 1 status :: 1/15 : active
[✓ enabled]
Robot 2
Robot 2 status :: 1/15 : active
[✓ enabled]
Robot 3
Robot 3 status :: 1/15 : active
[✓ enabled]
Robot 4
Robot 4 status :: 1/15 : active
[✓ enabled]
Robot 5
Robot 5 status :: 1/15 : active
[✓ enabled]
Robot 6
Robot 6 status :: 1/15 : active
[✓ enabled]
Robot 7
Robot 7 status :: 1/15 : active
[✓ enabled]
Robot 8
Robot 8 status :: 1/15 : active
[✓ enabled]
Position providers
[✓ Robots]
Vicon
Simulation protection
[✓ Step isolation]
ADG
```

Figure 5.5. Admin panel



```
ID: 1
tasks: 15/15
ip: : 192.168.250.201:40011
status: connected
orig: [5.000000, 10.000000]
```

Figure 5.6. Robot panel

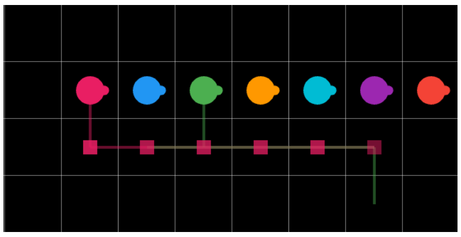
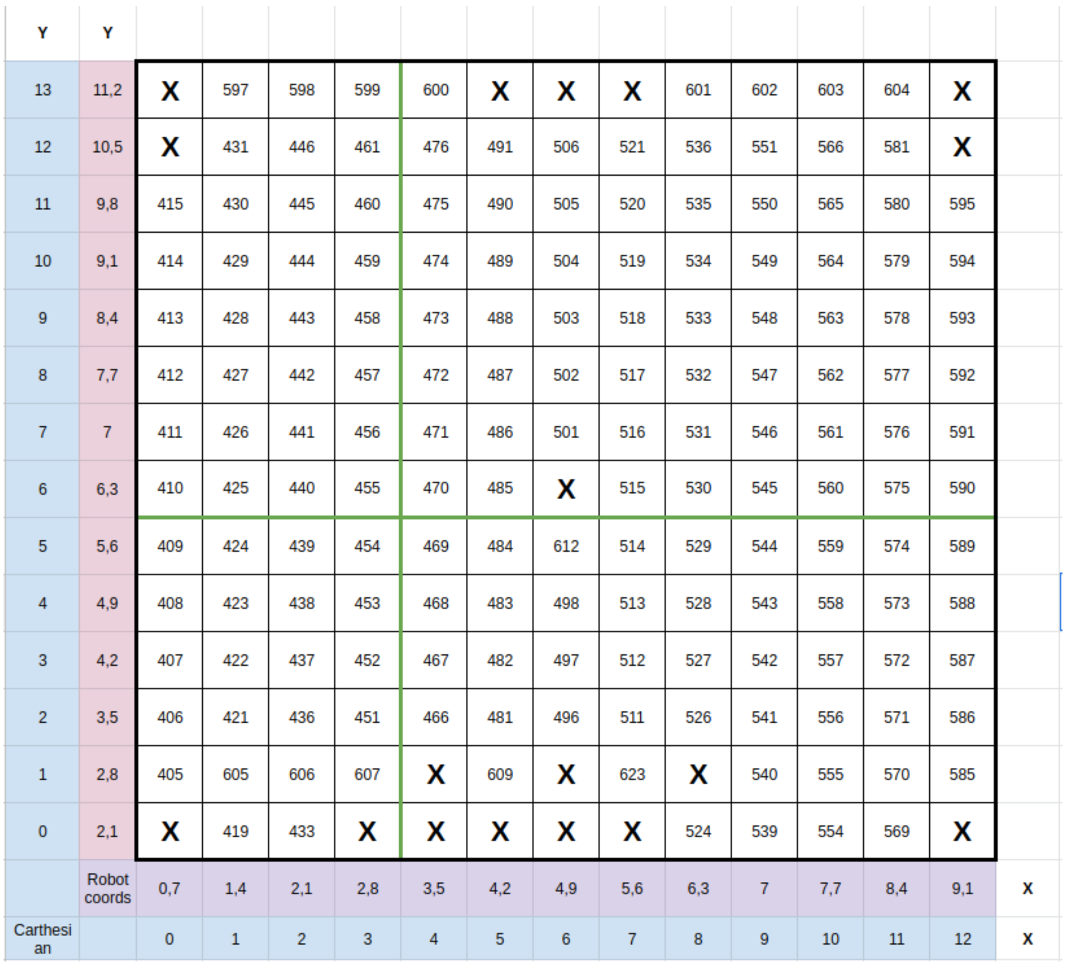


Figure 5.7. Plan highlight

The reference coordinate system used across the system is relative to the output of the solver. All reference, robot, and vicon coordinate systems can be seen in Figure 5.8 highlighted in blue, purple, and green, respectively. The reference coordinate system starts at the point (0, 0) in the lower-left corner of the demonstration grid, just as a convention. The robots have the coordinate system shifted and scaled. The beginning of the Vicon coordinate system can be seen highlighted in green. The robot coordinate system scale is roughly equivalent to the scale of Vicon, meaning that the distance between two neighboring nodes is roughly 0.7m.

The translation from Vicon coordinate system to the base one happens on the server right after the reception of Vicon data and no other system component is aware of the different coordinate system used by Vicon. The translation from base to robot



Y	Y														
13	11,2	X	597	598	599	600	X	X	X	601	602	603	604	X	
12	10,5	X	431	446	461	476	491	506	521	536	551	566	581	X	
11	9,8	415	430	445	460	475	490	505	520	535	550	565	580	595	
10	9,1	414	429	444	459	474	489	504	519	534	549	564	579	594	
9	8,4	413	428	443	458	473	488	503	518	533	548	563	578	593	
8	7,7	412	427	442	457	472	487	502	517	532	547	562	577	592	
7	7	411	426	441	456	471	486	501	516	531	546	561	576	591	
6	6,3	410	425	440	455	470	485	X	515	530	545	560	575	590	
5	5,6	409	424	439	454	469	484	612	514	529	544	559	574	589	
4	4,9	408	423	438	453	468	483	498	513	528	543	558	573	588	
3	4,2	407	422	437	452	467	482	497	512	527	542	557	572	587	
2	3,5	406	421	436	451	466	481	496	511	526	541	556	571	586	
1	2,8	405	605	606	607	X	609	X	623	X	540	555	570	585	
0	2,1	X	419	433	X	X	X	X	X	524	539	554	569	X	
	Robot coords	0,7	1,4	2,1	2,8	3,5	4,2	4,9	5,6	6,3	7	7,7	8,4	9,1	X
Carthesian		0	1	2	3	4	5	6	7	8	9	10	11	12	X

Figure 5.8. Overlayed coordinate systems

coordinate system and in reverse happens on the robot and no other system component is aware of the different coordinate system used by the robots.

5.6 Robot synchronization strategies

The solver solves the problem set in discreet time steps. The solution should guarantee that the simulation will be run without collisions. However, that can be quite inefficient, as, in the real world, the speeds of individual robots may differ from the expected ones, and making all robots wait for a single slow robot may lead to a bottleneck in the whole system. It is, therefore, desirable to be able to break the distinct step barrier. That may, however, break the consistency of the plan and introduce potential collisions. Once we disable this implicit safeguard, it is necessary to introduce an additional, possibly more efficient solution that will prevent collisions and enable the simulation to run without inconsistencies.

By eliminating the implicit barrier, we are able to dynamically enable and disable multiple synchronization strategies, making the system more modular and allowing the development of new algorithms and the demonstrator without any modification to existing strategies. Each strategy should act independently on the state of other strategies.

Currently, the Discreet time step strategy, which is a reimplementation of the original implicit barrier and Action Dependency Graph strategy are available.

5.6.1 Discreet time step

This synchronization strategy reintroduces the constraints given by the solver's output, keeping all robots in the same time step until the last one finishes execution. The reintroduction of the implicit barrier as a modular strategy makes it great for testing new planning algorithms, where the original solution of the solver can be inspected without real-world inconsistencies.

5.6.2 Action Dependency Graph

The second currently implemented approach is to construct an action dependency graph (ADG) [17] of the nodes and ensure the correct execution order for actions on a given node. An edge in the ADG represents a move between two nodes at a certain time step in the originally planned simulation. It also contains information about its dependencies (the edges that lead to the same end node but should happen before this one). Edge execution is only allowed when all of its dependencies have been resolved. When an edge is executed, it unblocks all the edges that it might have been blocking, further progressing the simulation.

The ADG greatly increases the efficiency, allowing robots to move until they break the dependency graph. In the original paper, a replanning algorithm is also introduced, but it was not implemented here for demonstration purposes, as changing the plan during execution would be quite contrary to the idea of the demonstrator, where we are demonstrating the execution plan itself. It might, however, be quite an interesting demonstration on its own.

5.6.3 Demonstration

In Figures 5.9- 5.23, you can see part of the execution using ADG. The Figures are not spaced evenly relative to the simulation time. Instead, they were selected to represent significant events during the execution.

Before Figure 5.11, the execution is running normally. However, somewhere between Figures 5.10 and 5.11, the green robot has been delayed. In order to continue execution, the pink robot has to wait. This is represented by having the currently waiting robot's grid block highlighted in red, as can be seen in Figure 5.11. The pink robot continues to wait until the next node in its desired path is free (from the ADG perspective), as displayed in Figure 5.15. This means that every robot that planned to travel through the node before this one already did so.

The inverse situation, where the green robot is blocked by the pink robot, can be seen in Figures 5.20 and 5.21. The rest of the demonstration does not need to avoid any collisions; it is therefore not included in this example. The robots will just return to their initial positions without any unexpected behavior.

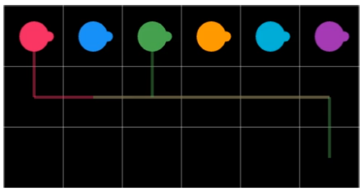


Figure 5.9. Step 1

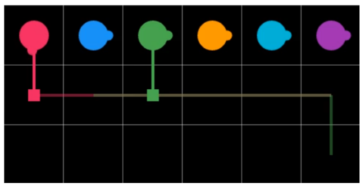


Figure 5.10. Step 2

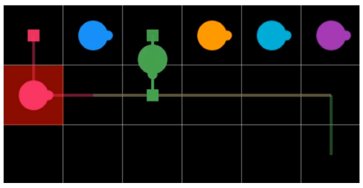


Figure 5.11. Step 3

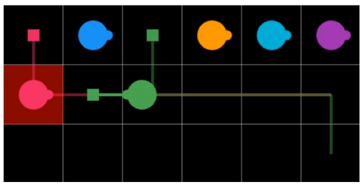


Figure 5.12. Step 4

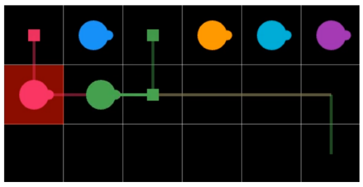


Figure 5.13. Step 5

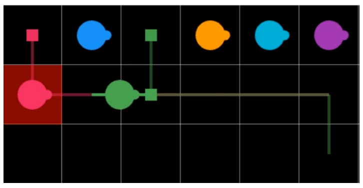


Figure 5.14. Step 6

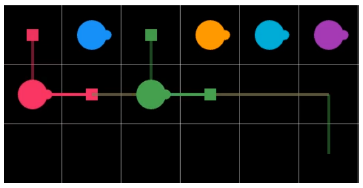


Figure 5.15. Step 7

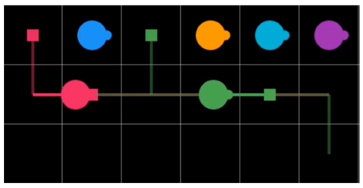


Figure 5.16. Step 8

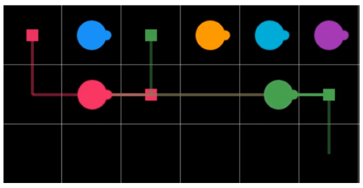


Figure 5.17. Step 9

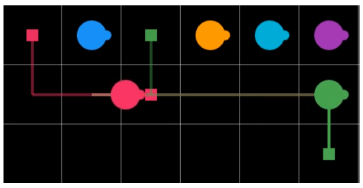


Figure 5.18. Step 10

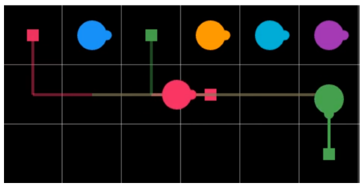


Figure 5.19. Step 11

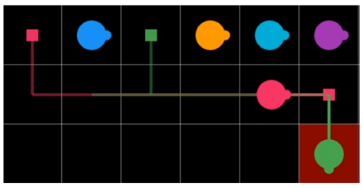


Figure 5.20. Step 12

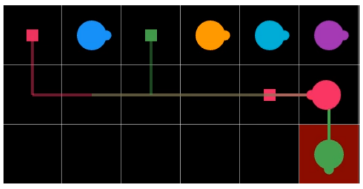


Figure 5.21. Step 13

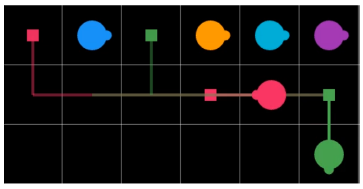


Figure 5.22. Step 15

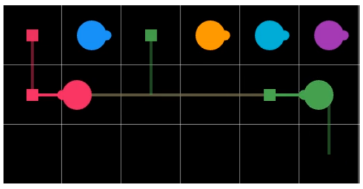


Figure 5.23. Step 16

Chapter 6
Implemented system properties

The main goal of this chapter is to describe and evaluate the properties of the implemented system and compare them with the properties of the previously used system, while pointing out the benefits and discussing potential improvements.

6.1 Demonstration setup

This section describes the process before the demonstration can be run. It includes updates from source code, build of the robots, server, set up of the connections and synchronization of all the elements of the demonstration.

The setup has to be done before each demonstration run, which in turn limits the development speed when performed inefficiently. Therefore, I attribute a large impact on the perception of the project as a whole to the tooling provided for the developers working with the demonstrator.

6.1.1 Original system

In the original system, the developer was required to manually connect to each robot while typing the robot's IP address and local account password for each one. Then login to the VCS, pull the newly updated source code, and ensure it builds correctly on the system. This task was time-consuming and did greatly affect the developer experience.

6.1.2 New system

In the new system, the preferred way to run the demonstration is through the shell script `ciirc-exec.sh` in the project root. Commands interacting with the server are:

- `server:init` - performs initial cloning of the repository
- `server:update` - performs a hard reset of the tracked branch and pulls the latest changes from the VCS on the server, and fixes permissions for executable files.
- `server:build` - builds the software on the server machine.
- `server:start` - starts the server software
- `server:stop` - stops the server software
- `server:ssh` - opens an interactive shell in the root of the project on the server

Commands interacting with the robots are:

- `robots:ssh <robotId>` - opens an interactive shell in the root of the project on the robot
- `robots:init <robotId>` - performs initial cloning of the repository

- `robots:update <robotId>` - performs a hard reset of the tracked branch and pulls the latest changes from the version control system (VCS) on the robot, and fixes permissions for executable files.
- `robots:update:all` - executes `robots:update <robotId>` for all available robots.
- `robots:inspect <robotId>` - connects to a running tmux session on the robot
- `robots:start <robotId>` - starts the robot software
- `robots:start:all` - performs `robots:start <robotId>` for all available robots.
- `robots:stop` - stops the software running on all robots.

The script is intended to reduce the time necessary for the setup and automates most of the initial setup tasks that had to be done manually before. Commands that are connecting directly to robots use the mapping defined in `/robot_definitions.txt` and perform actions using the defined credentials. The `:update` script first copies the ssh keys used for the git repository to the target machine, removing the need to ever type a password during script execution, while keeping the access restricted.

There are additional scripts for verifying the network connectivity, which proved to be very helpful when eliminating possible bug sources that, in many cases, led to the network connectivity.

This greatly decreases the developer time needed to review any changes done to the source code. Great developer experience is also ensured by defining the source repository, branch, and local ssh credentials to server and robots in a single place at the start of the script, allowing them to deploy the complete system from different repository/branch easily.

6.1.3 Comparison

The original user experience (UX) required the user to have extensive knowledge of the system, know the IP addresses of each one of the robots, and it had a significant time impact. The new UX enables anybody with access to the repository and local network to deploy all the robots in under 3 minutes. Single robot testing can be set up in under a minute. That is a huge time saving when compared to the original workflow, where the update would take approximately 10 minutes after getting to know the system and required an additional 2 hours of getting to know the system.

6.2 Runtime behavior

6.2.1 Original system

In the original system, the only way for the user to determine what is the current execution state was to look at logs in the server console or connect manually through ssh to the tmux sessions open on each robot. It required the developer to type the password repetitively, and during my time testing, proved to be really discouraging from getting actual work done.

6.2.2 New system

Since the new system has GUI, it provides a great experience to the user. The user can easily change the demonstration parameters, enable/disable particular robots, and change demonstration execution algorithms at runtime. The GUI also provides information about the connection status for the server, and individual robots, further improving the setup process visibility.

Improved is also the developer experience, as it is possible to stop the demonstration after each step, which makes debugging the whole application a lot simpler, should it ever be needed.

6.2.3 Comparison

In the new system, the user can change the parameters of robots, see their individual positions, enable/disable them and see the avoided potential collisions. Since the original system did not have a GUI, there is no point in directly comparing the UX.

6.3 Runtime demonstration impression

When presenting to 3^{rd} party, the speed and consistency of setup are even more crucial, as it can have a negative effect on the presentee should anything go wrong. The visualization becomes more important than the runtime properties, as the 3^{rd} party is mostly unaware of them and should be hidden when briefly presenting the result of a planner/execution algorithm. When presenting the new system, the continuous feedback from robots, *Vicon*, and visualization of plans provide a great impression and are largely able to mitigate insignificant inconsistencies should they happen during runtime.

6.4 Runtime demonstration problems

The whole setup is mainly dependent on the condition of the local network. The network consists of multiple subnets with relatively low-end endpoints, limiting the network throughput and introducing hard-to-debug problems for the system administrators. During testing, there have been found inconsistencies when routing in local subnets. This has been mitigated to a degree by reversion of the connection flow, where each robot now acts as a server from a networking point of view, waiting for connection from the centralized runtime server, avoiding the dependency on a static IP address of the server. This makes it possible to run the demonstration from any suitable device connected to the local network.

Chapter 7
Summary

The aim of this thesis was to improve upon the previous version of the demonstrator and achieve compatibility with the format used by the *mapfIR* project. The work was structured as follows.

At first, the *mapfIR* project was analyzed in order to identify key features and decide which ones are desirable in the final software. Afterward, the *original project* was reviewed to determine the current state of development and the actual usage. Then, the software design and decisions for implementation were discussed. The final stage of the project was the implementation itself, where the core features were implemented first, followed by the non-crucial ones.

The evaluation metrics for the project are the factual criteria given in the assignment. Particularly

1. Get acquainted with the current state of development of the demonstrator and the simulator for multi-agent planning (https://github.com/Kei18/mapf-IR).
2. Modify the simulator to serve as the basic user interface (GUI) of the demonstrator.
3. Display robot positions obtained from the *Vicon* system in the GUI.
4. Integrate the supplied components for planning and plan execution into the demonstrator.
5. Evaluate experimentally properties of the implemented system. Describe and discuss obtained results.

Given that the system performs the tasks as expected, I am glad to conclude that the criteria mentioned above were successfully fulfilled.

Furthermore, I'd like to point out one aspect of the final software that has not been introduced as evaluation criteria but proved to be very important during my encounter with the original software. It is the actual user experience of the software and relevant processes. This topic has been discussed in *Chapter 6*, and it has been greatly enhanced when compared to the original project. It plays a significant role for new developers and might greatly increase the motivation to continue upon the outcome of my work.

Future steps will probably include the addition of new synchronization algorithms (Section 5.6) and the introduction of in-app rerouting, incorporating selected algorithms supported by *mapfIR* and enabling easy addition and development of custom algorithms. This project, as designed, expects additional modifications to happen in the future and should make them as easy as possible.

Appendix A
Contents of the attached CD

```
├── demonstrator ............................ Complete demonstrator sources
│   ├── src ............................................ Sources for the robots
│   └── FleetControl
│       └── modules
│           ├── app .................................. Sources for the frontend
│           ├── manager .................................. Sources for the server
│           └── utils ....................... Sources for the shared resources
└── thesis ............................................... Sources for the thesis
```

References

[1] Keisuke Okumura. *Kei18/MAPF-IR: Iterative refinement for real-time multi-robot path planning (IROS-21).*
`https://github.com/Kei18/mapf-IR`.

[2] Openframeworks. *Openframeworks/openframeworks: OpenFrameworks is a community-developed Cross Platform Toolkit for creative coding in C++.*
`https://github.com/openframeworks/openFrameworks`.

[3] David Silver. Cooperative Pathfinding. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment.* 2021, 1 (1), 117-122.

[4] Keisuke Okumura, Manao Machida, Xavier Défago, and Yasumasa Tamura. *Priority Inheritance with Backtracking for Iterative Multi-agent Path Finding.* In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19.* International Joint Conferences on Artificial Intelligence Organization, 2019. 535–542.
`https://doi.org/10.24963/ijcai.2019/76`.

[5] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence.* 2015, 219 40-66. DOI https://doi.org/10.1016/j.artint.2014.11.006.

[6] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, Oded Betzalel, David Tolpin, and S. E. Shimony. *ICBS: The Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding.* In: *SOCS.* 2015.

[7] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. *Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem.* In: *SOCS.* 2014.

[8] Michal Čáp, Peter Novák, Alexander Kleiner, and Martin Selecký. Prioritized Planning Algorithms for Trajectory Coordination of Multiple Mobile Robots. *IEEE Transactions on Automation Science and Engineering.* 2015, 12 (3), 835-849. DOI 10.1109/TASE.2015.2445780.

[9] Ryan Luna, and Kostas E. Bekris. *Push and Swap: Fast Cooperative Path-Finding with Completeness Guarantees.* In: *IJCAI.* 2011.

[10] Keisuke Okumura, Yasumasa Tamura, and X. Défago. winPIBT: Expanded Prioritized Algorithm for Iterative Multi-agent Path Finding. *ArXiv.* 2019, abs/1905.10149

[11] Tomáš Rybecký. *Trajectory planning for a heterogeneous team in an automated warehouse.* 2020.

[12]
`https://nanomsg.org/`.

[13] *What is a TurtleBot?*
`https://www.turtlebot.com/about`.

[14] Inc. Open Source Robotics Foundation. *Robot operating system.* `https://www.ros.org/`.

[15] *Apriltag.* `https://april.eecs.umich.edu/software/apriltag`.

[16]

[17] Wolfgang Hönig, Scott Kiesel, Andrew Tinka, Joseph W. Durham, and Nora Ayanian. *Persistent and Robust Execution of MAPF Schedules in Warehouses.* 2019.