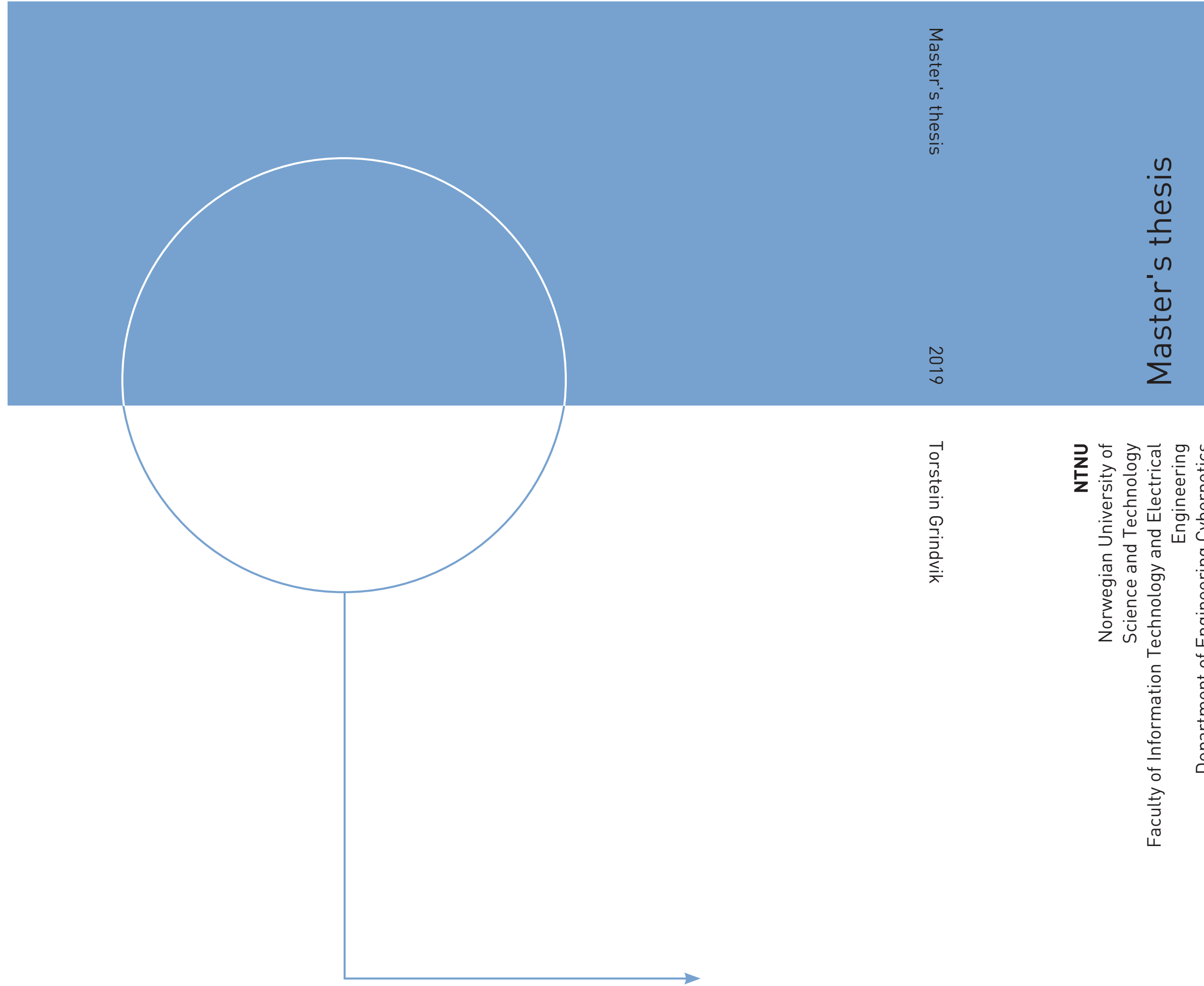


Torstein Grindvik

Creating the foundations of a graphical SLAM application in Modern C++

June 2019

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Norwegian University of
Science and Technology

Creating the foundations of a graphical SLAM application in Modern C++

Cybernetics and Robotics
Submission date: June 2019
Supervisor: Tor Onshus

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Foreword

The base of this master's thesis has been made possible due to the access of several valuable resources.

A shared workspace with Endre Leithe and Kristian Fjelde Pedersen has been valuable in order to interact with- and have access to their experience and knowledge about the robots used in the project. The powerful desktop machine provided by NTNU eased the development process. Similarly, the testing of the project would not be possible without the access to the hardware given by Nordic Semiconductor, and the hardware available for purchase through Omega Workshop.

The base of the graphical representation of this thesis is made possible by the work of Laurent Gomila, the author of the *Simple and Fast Multimedia Library*. Similarly, the user interface is entirely dependent on the work of Omar Cornut, the author of the user interface library *dear imgui*. Pathfinding was made possible by the authors of the *Boost Graph Library*: Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. Lastly, the many authors of the C++ standard library are greatly appreciated.

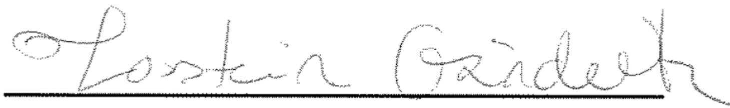
The Java application created over several years by several authors proved a great source of inspiration for the new C++ application. It helped clarify both strong points of the current implementation such as a graphical representation of the mapping process and friendly user interfaces, and weak points such as the complex communication stack. The new C++ application has a flexible graphical representation, a clear user interface, and a solid new communication stack based on the latest technologies in the world of IoT.

A collaboration effort with Endre Leithe proved useful. I created a *legacy layer* to allow his robot with legacy hardware to connect to the new server application—which uses the Thread network protocol stack. The legacy layer bridged the gap between Thread and a protocol available on his robot: I2C. He then modified the robot software to understand the translated messages coming to and from his I2C bus, proving a successful endeavour.

The work on this thesis has given me a very valuable experience as a developing programmer. I am happy that I sought to use modern C++ features and proven design patterns, which allowed me to reflect on my choices through-

out the implementation of the application—and grow as a programmer in the process.

I wish to thank my supervisor Tor Onshus for his guidance during the course of the thesis.



Torstein Grindvik
Trondheim, June 2019

Problem Statement

A Java server has been written for NTNU's SLAM (Simultaneous Localisation and Mapping) project, which has been ongoing since 2004. Reportedly, students have expressed the wish to port this to another language, as Java is not the most familiar language to a majority of the students having participated in previous work. Therefore, an initiative to rewrite the server in the more familiar language C++ has been sought.

Today, several robots are using Bluetooth to communicate with a Java server running on a desktop computer. The server accepts incoming connections, reads and maps data, and replies with commands. The map is the currently sensed and interpreted representation of the physical area by the robots. The commands tell robots where to go next according to a path-finding algorithm, intended to expose the surrounding area as efficiently as possible.

As much functionality as possible is to be ported onto the new C++ architecture, and the transition from Java to C++ should be as painless as possible for the future students taking on the project. The new server should be structured, documented, and tested.

The rewrite from Java to C++ presents an opportunity: Characteristics which have worked well for the Java application should be implemented and used in the C++ version—problematic areas should be identified and avoided.

In the following chapters of this thesis, the student will:

- Inspect and analyse the server written in Java.
- Decide which components should be translated.
- Document the new server design.
- Implement the new C++ server.
- Use means of testing and profiling the server.
- Create guides and examples on using and extending the new server.

Summary

This thesis was tasked with rewriting a complete SLAM server application in the C++ programming language, as the Java server application previously in use was to be discarded. The task included creating a graphical application for drawing measurements sent by robots, creating a user interface in order to let users issue control over various tasks, and implementing a future-proof communication method with robots.

The communication stack used by the Java server application built upon Bluetooth Low Energy and user-implemented protocols. This implementation was entirely discarded, as it was found to greatly hinder development progress and provided several limitations. A case study was conducted in order to consider the benefits of an alternative communication stack. The new stack used Thread networking technology, and solves the limitations of the previous stack. Robots did not have support for the new stack as newer hardware was required. This has been mitigated to some degree by creating a legacy layer consisting of hardware with support for the new technology and the software to control it, although this solution is meant for the short term. It is advised to upgrade robots to new hardware.

Experiments using robots proved the new server application to work as intended by the scope of the thesis: A graphical C++ SLAM server application built in modern C++ with easily extensible user interfaces and features successfully plotted robot measurements via a more sustainable software stack, with documented examples on how to further extend the application. A simulated software robot has been used to test the server and can be activated through the user interface. This has allowed for testing the robustness of the server, verifying robot control panel features, and profiling long-running tests without the need of physical robot interaction. Tests using real robots were successful as the new application plotted robot measurements and issued commands for robots to move towards. Robots responded accordingly.

Compared to the Java server application, the new application is more flexible in development and has more useful basic features, but does not yet have higher level SLAM features. Several technological limitations are no longer present due to the new application: An arbitrary number of robots can be connected,

communication is less brittle and greatly simplified, and the necessary hardware and software setup is simplified for next generation robots.

Oppsummering

Denne tesa hadde som oppgåve å omskrive ein komplett SLAM serverapplikasjon i C++ programmeringsspråket, då Java serverapplikasjonen tidlegere i bruk skulle bli forkasta. Oppgåva innbefatta å lage ein grafisk applikasjon for teikning av målingar sendt frå robotar, å lage eit brukergrensesnitt for å la brukarar yte kontroll over forskjellege oppgåver, og å implementere ein framtidsretta kommunikasjonsmetode med robotar.

Kommunikasjonsoppsettet brukt av Java serverapplikasjonen bygde på Bluetooth Low Energy og brukerimplementerte protokollar. Denne implementasjonen vart forkasta, då den forhindra framgang i utvikling og medfulgte fleire begrensningar. Eit eksempelstudie vart utført for å vurdere fordelar ved bruk av eit alternativ kommunikasjonsoppsett. Det nye oppsettet brukte Thread nettverksteknologi og løyste begrensningane ved forrige oppsett. Robotane hadde ikkje støtte for det nye oppsettet då nyare maskinvare var påkrevd. Dette vart løyst til ein viss grad ved å lage eit overgangslag som bestod av maskinvare med støtte for den nye teknologien og programvara for å styre den, men denne løysinga er kun meint som ei midlertidig løysing. Det er råda å oppgrade robotar til ny maskinvare.

Eksperimenter med bruk av robotar viste at den nye serverapplikasjonen virka som tenkt gitt omfanget til tesa: Ein grafisk C++ SLAM serverapplikasjon bygd i moderne C++ med enkelt utvidbart brukergrensesnitt og eigenskapar plotta målingar frå robotar vellukka via eit meir berekrafig programvaregrunnlag, med dokumenterte eksempler på videreutvikling av applikasjonen. Ein simulert programvarerobot har vore brukt til å teste serveren og kan bli aktivert via brukergrensesnittet. Dette har tillatt testing av robustheita til serveren, verifikasjon av eigenskapar til robot-kontrollpanel, og profilering av langtidstestar utan behov for fysisk robotinteraksjon. Testar med ekte robotar gikk bra, då den nye applikasjonen plotta robotmålingar og ga kommandoar for robotar til å bevege seg mot. Robotar reagerte i samsvar med dette.

Samanlikna med Java serverapplikasjonen er den nye applikasjonen meir fleksibel i utvikling og har fleire nyttige grunnleggjande eigenskapar, men har enno ikkje SLAM eigenskapar på høgt nivå. Fleire teknologiske begrensningar er no ikkje lenger til stades som følgje av den nye applikasjonen: Eit vilkårleg

antal robotar kan no bli tilkopla, kommunikasjonen er mindre skøyr og svært forenkla, og den nødvendige maskinvara og programvareoppsettet er forenkla for neste generasjons robotar.

Conclusion

The new SLAM server written in modern C++ successfully builds a solid foundation for solving higher level SLAM tasks, and has been created with extension in mind.

A modern C++ approach is taken throughout the source code, which aids in readability, maintainability, and is safer in terms of avoiding resource leaks. The design has been clearly documented and reasoned about. The implementation has also been documented and reasoned about on a file-by-file basis, expressing the intent of every source file.

The current application has a number of features. The graphical window supports drawing all basic graphical primitives. The window also support moving the user-view around the map, rotating, zooming in and out, and flipping the vertical axis. These controls are available through the user interface. Support for MQTT is also built into the user interface, with manual publish and subscribe controls, and an overview of incoming and outgoing messages. Various other useful panels are available, such as panels which are dynamically created when new robots connect, and control over robot pathfinding. Through this control panel, robots can be issued to move towards any point on the application grid. The application grid is an abstract representation of the physical world the robots move in. Robots can be asked to move directly towards a point, or to be sent towards waypoints which avoid obstacles using pathfinding algorithms. Perhaps the most important change is the move to use the Thread protocol for communication, and not reimplementing Bluetooth support. This move has made mesh networking automatic, communication is encrypted, large messages are handled automatically, checksums and retries are automatic, and there are no limitations on the number of connected entities. The cost of this move has been the fact that it requires new hardware. Robots are currently using an older generation Nordic Semiconductor chip. This means a legacy layer was needed in order to bridge the technology gap. This was successfully created, but it has limitations which are not ideal. Therefore, a clear step towards meaningful progress is to upgrade the robots by porting applications to the new generation Nordic Semiconductor chips (or similar) which support the Thread protocol. The additional benefit of this upgrade is the fact that robots

then do not need a dongle, as the required hardware support is built into the robot's CPU which also runs the robot application. Similarly, the computer running the server application no longer requires a dongle; the Java application required a Bluetooth dongle. This means the number of devices to maintain is substantially reduced, which greatly reduces overall project complexity.

The Java application has been superseded and is not needed for the tasks which can be handled by the new application. Some newer versions of the Java application have experimented with higher level SLAM tasks such as algorithms for traversing the map efficiently, and using techniques for mitigation of pose estimation errors which are inherently present in SLAM applications. These techniques are not present in the new application, thus they need implementation. Other tasks such as connecting robots, mapping measurements, and controlling robots manually or by pathfinding is implemented and correctly working in a more flexible manner and should now be the default way of approaching the SLAM project.

Contents

Foreword i

Problem Statement iii

Summary v

Oppsummering vii

Conclusion ix

1 Introduction 1

1.1 Motivation 1
1.2 Previous Work 2
1.3 Scope 3
1.4 Roadmap 4

2 Background: C++ Programming and Environment 7

2.1 Integrated Desktop Environment 7
2.1.1 Visual Studio Community 7
2.2 Often Used Language Features 10
2.2.1 Namespaces 10
2.2.2 Templates 11
2.2.3 Inheritance 12
2.2.4 Polymorphism 13
2.2.5 Operator Overloading 15
2.2.6 Smart Pointers 16
2.2.7 Lambdas 17
2.2.8 Keywords 17
2.3 External Tools 19
2.3.1 Vcpkg 19
2.4 External Libraries 20
2.4.1 Boost 20
2.4.2 SFML 23
2.4.3 imgui 24

3 Background of Project: SLAM **27**

4 Java Server Application **29**
4.1 Codebase Summary 29
4.1.1 Namespace: `no.ntnu.et` 30
4.1.2 Namespace: `no.ntnu.hkm` 33
4.1.3 Namespace: `no.ntnu.tem` 34
4.2 High-Level Overview of Features 42
4.2.1 Graphics, User Interface 42
4.2.2 Robot Handling 43
4.2.3 SLAM 43
4.2.4 Simulation 43
4.2.5 Communication 43
4.2.6 Miscellaneous 44
4.3 Discussion 45

5 Case Study: Thread **47**
5.1 Thread 47
5.1.1 IEEE 802.15.4 49
5.1.2 IPv6 50
5.1.3 6LoWPAN 52
5.1.4 Nodes and Devices Types 52
5.1.5 Addressing 54
5.1.6 Network Creation and Joining 56
5.2 OpenThread 58
5.3 MQTT 59
5.3.1 Quality-of-Service 59
5.3.2 MQTT-SN 60
5.4 In Practice: Nordic Semiconductor SoC and Raspberry Pi 61

6 Design of New Server **65**
6.1 Major Goals 65
6.2 Design Patterns 66
6.2.1 Use Callbacks Generated By Events 66
6.2.2 Use Inheritance Where Appropriate 66
6.2.3 Single Responsibility 66
6.2.4 Modern C++ 67
6.2.5 Asynchronous Single-Threaded Until Avoidable 68

6.3 Main Ties Functionality Together 69

7 Results **71**

7.1 Overview 71
7.2 Codebase Structure 73
7.2.1 Namespace Explanations 73
7.3 Codebase Walk-through 74
7.3.1 Namespace: `TG::gui` 74
7.3.2 Namespace: `TG::graph` 77
7.3.3 Namespace: `TG::networking` 78
7.3.4 Namespace: `TG::utility` 78
7.3.5 Namespace: `TG::application` 78
7.4 Callbacks and Events 80
7.4.1 Enabling Class 80
7.5 Various Design Goals 81
7.5.1 Inheritance 81
7.5.2 Single Responsibility 82
7.5.3 Modern Codebase 82
7.6 Asynchronous And Single-Threaded 83
7.7 Graphical Implementation 84
7.8 User Interface 88
7.9 Communication Stack 94
7.10 Finding Paths 95

8 Testing and Profiling **97**

8.1 Testing 97
8.2 CPU Usage 98
8.3 Memory Usage 100

9 Extending and Using The New Server **103**

9.1 Running the Server 103
9.2 Extending the Server 104
9.2.1 Example: Adding Drawables 104
9.2.2 Example: Adding a Panel 106
9.2.3 Example: Adding Callbacks to a Class 108
9.2.4 Example: Running a Task 108

10 Using a Legacy Robot **111**
10.1 Connecting a Legacy Robot 111
10.1.1 Overview 111
10.1.2 Implementation 112
10.1.3 Discussion 114
10.2 Communication Protocol 116
10.2.1 I2C 117
10.2.2 MQTT 118
10.3 Real World Use: Detecting a Circular Track 118

11 Discussion and Future Work **121**
11.1 Discussion 121
11.1.1 C++ Application Versus Java Application 121
11.2 Future Work 125

Bibliography **127**

A Files Overview **131**

Chapter 1

Introduction

Aim of This Chapter

- Explain motivation for the thesis.
- Outline related previous work.
- Outline the scope of the thesis.
- Present a roadmap of the remaining content of the thesis.

1.1 Motivation

A graphical Java server application has been developed and can perform some tasks with a few robots as of the start of this thesis.

Students have expressed a wish to use a more familiar language more suited to the background of the students which tend to work on the SLAM project. C++ is the language of choice for this task. There is an opportunity to use modern features of C++—stemming from the advances in the language which were introduced by the C++11, C++14, and C++17 standards. Using modern techniques in C++ inspires code which performs well—perhaps the most defining characteristic of C++—but is also readable, maintainable, safe, and extensible.

The robots currently rely on Bluetooth technology for communications. There is an opportunity to assess the level of success this has had as well, as there are other potential wireless technologies available if desired.

The overarching design goal of the server application is to create a modularised implementation where extending the graphical part, the user interface, and robot-interactions is easy for next generations of developers. This can be achieved by creating logically separated classes, by having a high sense of cohe-

sion in classes, and by the use of design patterns which inspire decoupling.

1.2 Previous Work

This section will briefly outline previous work on the server side of the project. For an overview of previous work related to the project robots, see [15] and [19].

In 2016, the server application in use was a MATLAB [18] application [28]. That year, the Java application was created. The authors ported most functionality from the MATLAB application, and created a graphical application via a modularised system architecture, which also featured a user interface [28]. At the same time, communication went from using Bluetooth to Bluetooth Smart. The desktop application was given a Nordic Semiconductor [30] USB dongle with a specialised firmware on-board in order to communicate with robots. [39] added simulation, navigation, and mapping to the application. [17] added the use of automatic repeat request (ARQ) in 2017. The intention was to allow sending larger messages, avoid loss of messages, and to make robots addressable. The communication layer was more or less completely overhauled at this point. The interaction with dongles before this point was based around sending commands between the server and the dongle (which was physically placed in the USB port of the server machine). This command set was custom made and performed tasks such as scanning for other dongles and connecting to them. This system was discarded, and command sets are no longer in use. The dongles now continuously scan for other dongles and try to connect to any in range. Sending messages at this point relies on a specific format. This format is seen in Figure 1.1. Another required step at this point is to encode all outgoing messages via Consistent Overhead Byte Stuffing (COBS) [17] and similarly decode all incoming messages via COBS, as well as to validate checksums.

[19] experimented with improving the mapping experience of robots server-side. Instead of requiring each robot to specify an initial position in the global coordinate system, separate reference frames were allowed. This then required a map-merging algorithm. A particle filter was applied in an effort to improve robot poses.

In 2018, [9] changed the Java application and the robots in use to use Cartesian coordinates. Before this, the server and robots had communicated via relative polar coordinates. For example, a robot could be instructed to turn 45 degrees clockwise, and then travel a distance of 20 centimetres. An algorithm to create lines from clouds of points was experimented with, for use on the

mapping side of the server.

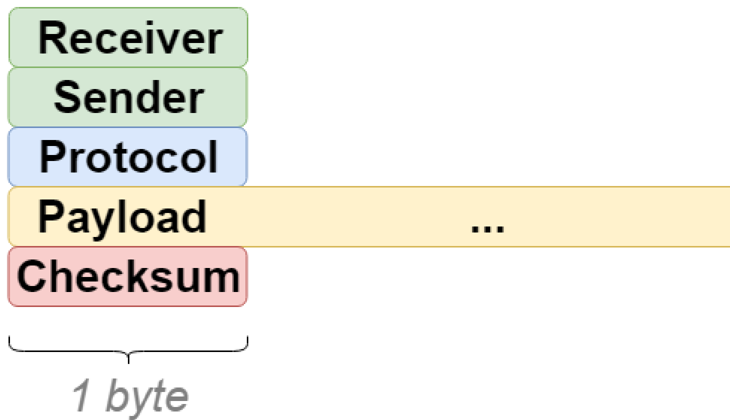


Figure 1.1: Network frame as of [17].

1.3 Scope

The scope of the server application is large. Focus will be entirely on creating a solid foundation for solving SLAM problems for future developers. Major goals are:

- Java application inspection: Get a sense of how the system currently works, and decide what should be rewritten in the new application.
- Graphics: Find a suitable library for drawing graphics on a screen. Support for drawing various primitives (triangles, circles, lines, and similar) should be supported. Support for drawing higher-level entities such as images could also be valuable.
- User interface: A library for graphics does not necessarily support user interface elements such as menus, panels, nodes, buttons, and similar. If support is not built in, a suitable user interface library should be found.
- Robot communications: The Java application communicates not directly via Bluetooth, but via a serial interface to a USB dongle which in turn communicates via Bluetooth. An evaluation of this roundabout setup should be done.

- Robot interaction: When communication is solved, some manner of interacting with the robots should be in place. For example, receiving data about robot positions, obstacles, and instructing robots to move to a location.
- Trial and error: Given a working new application, experiments with robots should take place.
- Profiling: The working application should be profiled—its memory- and CPU footprint should be inspected for potential issues.
- Document and instruct future developers: Weight should be placed on instructing future generations of the project. This means providing examples and explanations of how to best approach development on the new server application.

1.4 Roadmap

A roadmap is presented since the structure of the thesis is somewhat non-traditional. After the design is presented, the results are given in the form of a presentation of the implementation. Still, a few (content-)chapters after the results are added—concerning the actual use of the application with robots, testing and profiling, and a chapter specifically targeting use and extension of the application.

The contents of the thesis is laid forth as follows:

- Chapter 2 and Chapter 3 provide background information. Chapter 2 explains various parts of the C++ language which might be unfamiliar. Examples of modern C++ patterns often used in the project are given. External tools and libraries are presented. Chapter 3 briefly introduces the general context of the project—SLAM.
- Chapter 4 presents the Java application, and finds out which parts should be rewritten in the new application and which parts should be left out (or implemented later).
- Chapter 5 presents a case study performed in the early stages of the thesis concerning an alternative approach to robot communication.
- Chapter 6 concerns the design principles and choices made for the new application.

- Chapter 7 presents the new application in its final state, and compares it with the design principles made.
- Chapter 8 concerns testing the finished application in terms of performance.
- Chapter 9 is targeted at future users of the application, and future developers. Running the application is covered. Weight is placed on showing how to further develop the application.
- Chapter 10 presents the work done in order to connect a robot without the necessary hardware to the new server, and the results which followed.
- Chapter 11 discusses parts of the new application which were successful, and considers possible problem areas—and ties this together with guidelines on future work.

Chapter 2

Background: C++ Programming and Environment

Aim of This Chapter

- Introduce the integrated desktop environment (IDE) used, and project settings used within the IDE.
- Explain and show examples of C++ language features often used in the source code of the project.
- Introduce C++ language features which might be new to programmers not having used more recent features of C++ such as features introduced in the C++11, C++14, and C++17 standards.
- Familiarise the reader on external tools and libraries used.

2.1 Integrated Desktop Environment

2.1.1 Visual Studio Community

Visual Studio Community 2017 [10] was used as the integrated desktop environment during development. Some important features are:

- Setting breakpoints for stopping code at certain points.
- Inspecting values and objects during paused run-time.

- Refactor symbols and function prototypes project-wide.
- Profile memory- and CPU usage.
- View memory and registers during paused run-time.
- Create definitions from declarations.
- Go to definitions from declarations, and vica versa.

Using Visual Studio allows for quickly setting up projects via the integrated *New project...* wizard, which creates project files describing the organisation of files, compiler flags, and all other settings chosen via the IDE. This allows developers to develop in an integrated manner as project settings can be shared via the project files, and allows for quickly sharing setups without needing to spend time on creating custom toolchains. This is also due to Visual Studio bundling its own Visual C++ compiler. The drawback of this approach is the tight integration; the development is now coupled with the IDE. This also implies developing on Windows only. It should be noted that it is only the build process that is Windows specific. The source files should be compilable for other platforms, however the build process for those platforms must then be set up.

Project Configuration

Some important configuration options are presented.

Figure 2.1 shows the setting for adding additional include paths to the compiler. This is used in order to let the compiler know where to look for include files. This allows including headers via paths relative to the paths added. If a user wanted to include the header `C:/Thirdparty/Include/Lib/Lib.h`, the path `C:/Thirdparty/Include` could be added to the path setting. This would allow including the header via the line `#include "Lib/Lib.h"` in source files.

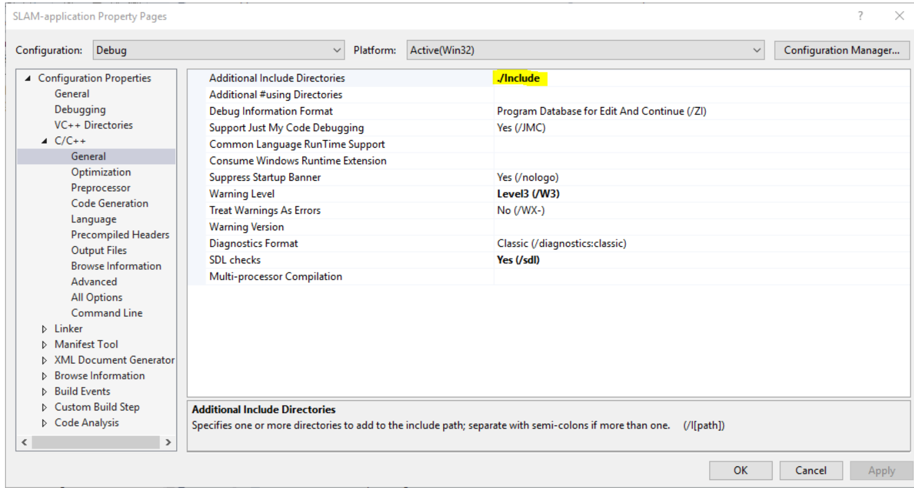


Figure 2.1: A relative path being used as an additional include directory.

Figure 2.2 shows the setting for adding additional preprocessor definitions. It is common for thirdparty libraries to rely on preprocessor flags in order to let the user choose between different behaviours of the library. Issues around preprocessor definitions can be debugged by increasing the compilation verbosity in the project settings, as the default verbosity is quite terse. When increased, more output is presented when compiling—including all preprocessor definitions and include paths.

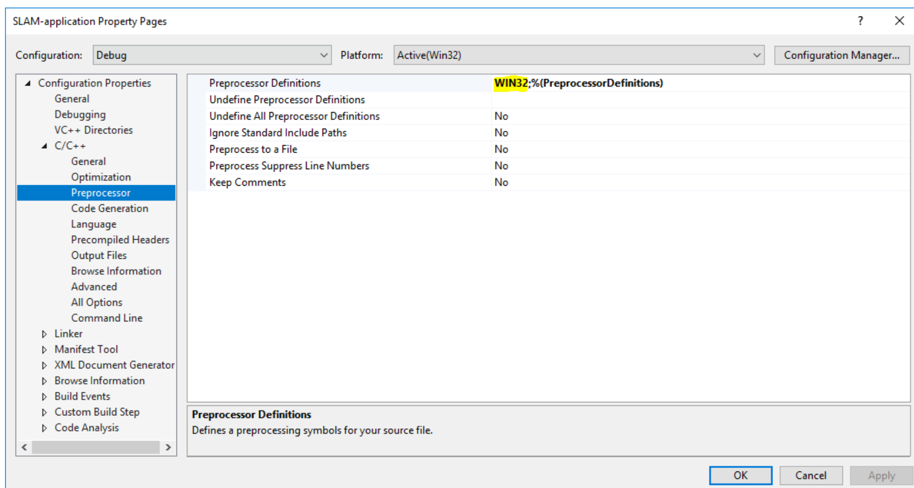


Figure 2.2: A custom preprocessor definition added to the project.

Figure 2.3 shows the setting for which language standard to compile for.

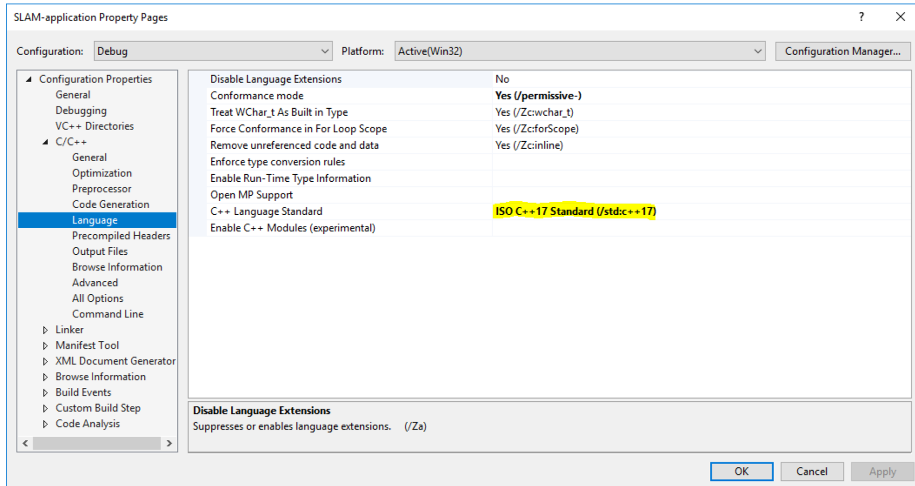


Figure 2.3: C++17 chosen as standard to compile for.

Notice no figures are included concerning the *Configuration Properties—Linker* options. This is explained further in the section concerning external tools, Section 2.3.

2.2 Often Used Language Features

This section provides small examples and descriptions of various C++ language features. The C++ programming language is fairly large and is very intricate when scrutinised. The aim of this section is to refresh concepts in a brief manner, in order to aid in reading the project source code efficiently. Concepts which will be encountered frequently or which are syntactically different from other languages are focused on. More details can be found in online references [5] or in book form [36].

2.2.1 Namespaces

Namespaces allows grouping code into logical categories, aiding in readability. They also allow for avoiding collisions in symbol names. A thirdparty math library using vectors could define a class `vector` which could collide with the standard library's `vector` class. However, the standard library uses the names-

pace `std`. This means the syntax `std::vector` is used to specify intent of which version to use.

Namespaces can be entered by the syntax shown in Listing 2.1. Usage of both single and nested namespaces are shown.

Listing 2.1: Using namespaces.

```
namespace vehicle {
    void drive() {
        // This function addressed by vehicle::drive
        ...
    }
}

namespace vehicle::air {
   void fly() {
        // This function addressed by vehicle::air::fly
        ...
   }
}
```

Declaring a namespace deeper than a single level in one go as is done for `namespace vehicle::air` is a new feature of C++17.

One way to group code into namespaces are via a top-level namespace (e.g. author, organisation), then module subject or topic, and then specialisation. Examples are `NTNU::pedagogy::presentation`, `NTNU::pedagogy::writing`, and `Equinor::math::complex`.

2.2.2 Templates

Templates are a key concept in generic programming, and widely used in C++ [36]. Templates allow several types to share the same business logic.

An example of a template function is given in Listing 2.2.

Listing 2.2: A template function.

```
template <class T>
bool is_strictly_larger(const T& t1, const T& t2) {
    return (t1 > t2);
}
```

The function `is_strictly_larger(...)` is declared to accept comparison of any given class, and returns a boolean indicating the result. Note that usage of this template function is checked at compile-time. This means that passing the wrong type of class to the function would cause an error during compilation. This would be the case for any class not supporting the operations performed in the template function body. In this case this would mean any class not supporting the comparison operator.

A class can accept template parameters. An example of a template class is given in Listing 2.3.

Listing 2.3: A template class.

```
template <class L, class R>
class Related {
public:
    Related(L l, R r);
    L left() const;
    R right() const;
private:
    L l_;
    R r_;
};

template <class L, class R>
Related<L, R>::Related (L l, R r) : l_(l), r_(r) { }

template <class L, class R>
L Related<L, R>::left() const {
    return l_;
}

template <class L, class R>
R Related<L, R>::right() const {
    return r_;
}
```

The example shows the use of several template parameters for a single class. This abstraction shows the early stages of implementing a base class where two generic classes have a left-right relationship.

This class could be used as seen in Listing 2.4. Notice the template class `Related` is invoked with angle brackets, where the types of the template parameters are specified.

Listing 2.4: Template class usage.

```
#include "Related.hpp"
#include "Hands.hpp"
#include "Feet.hpp"

int main() {
    LeftArm larm;
    RightArm rarm;

    LeftFoot lfoot;
    RightFoot rfoot;

    Related<LeftArm, RightArm> arms{larm, rarm};
    Related<LeftFoot, RightFoot> feet{lfoot, rfoot};

    ...

    return 0;
}
```

2.2.3 Inheritance

Inheritance is a core object-oriented programming concept and allows code reuse and grouping functionality for similar classes.

A class can inherit behaviour from a parent class (also called a super-class). The functions and members of the parent class are now also a part of the child class. The child class can add new functions and members. A given function can also be overridden, allowing specialisation of parts of an interface.

Inheritance then allows several specialised child classes to share a common set of functionality from a parent. An example is given in Listing 2.5.

Listing 2.5: Inheritance.

```
// File: Base.hpp
class Base {
public:
    Base(float a) : a_(a) {}
    float get_half() {return a_ / 2.0f;}
private:
    int a_;
};

class Extension : public Base {
public:
    Extension(float f) : Base(f), f_(f) {}
    float get() {return f_;}
private:
    int f_;
};

// File: Main.cpp
int main() {
    Extension e {5.0f};
    auto whole = e.get();       // whole is 5.0f
    auto half = e.get_half();   // half is 2.5f
    ...
}
```

The example shows a class `Extension` which inherits the functionality of its parent `Base`, and simultaneously extends the behaviour by adding a new function. An instance of an `Extension` is created and used.

2.2.4 Polymorphism

Polymorphism is further split into *run-time polymorphism* and *compile-time* polymorphism.

Run-time polymorphism concerns using different types via some commonality. Listing 2.6 shows run-time polymorphism.

Listing 2.6: Run-time polymorphism.

```
#include <iostream>

struct Tool {
    Tool() {};
    virtual void use() const {
        std::cout << "Using Tool" << std::endl;
    }
};

struct Wrench : public Tool {
```

```
11      Wrench() {};
12      void use() const override {
13          std::cout << "Using Wrench" << std::endl;
14      }
15  };
16
17  struct Screwdriver : public Tool {
18      Screwdriver() {};
19      void use() const override {
20          std::cout << "Using Screwdriver" << std::endl;
21      }
22  };
23
24  struct SomeTool : public Tool {
25      SomeTool() {};
26  };
27
28  void use_tool(const Tool& tool) {
29      tool.use();
30  }
31
32  int main() {
33      Wrench wrench;
34      Screwdriver screwdriver;
35      SomeTool sometool;
36
37      use_tool(wrench);        // Outputs "Using Wrench"
38      use_tool(screwdriver);   // Outputs "Using Screwdriver"
39      use_tool(sometool);      // Outputs "Using Tool"
40
41      ...
42  }
```

The function `use_tool(...)` shows that the three structures can all be treated as `Tool`s at run-time. Note that structures are simply classes with `public` as the default access modifier of members. The example also shows the useful pattern of treating specialised classes as their parent class.

Compile-time polymorphism refers to templates and overloading of functions. Function overloading can be seen in Listing 2.7.

Listing 2.7: Compile-time polymorphism.

```
1   #include <iostream>
2
3   struct Hat {};
4
5   void typehint(int) {
6       std::cout << "Int!" << std::endl;
7   }
8
9   void typehint(double) {
10      std::cout << "Double!" << std::endl;
11  }
12
13  void typehint(float) {
14      std::cout << "Float!" << std::endl;
15  }
16
17  void typehint(Hat) {
18      std::cout << "Hat!" << std::endl;
19  }
20
21  int main() {
22      typehint(1);        // Outputs "Int!"
23      typehint(1.0);      // Outputs "Double!"
24      typehint(1.0f);     // Outputs "Float!"
25      typehint(Hat{});    // Outputs "Hat!"
```

```
26
27      ...
28  }
```

As the function `typehint(...)` has the same base name for all its definitions, it is overloaded. Overloaded functions can also have a different number of arguments and different return types.

2.2.5 Operator Overloading

Overloading has been briefly visited through compile-time polymorphism. Still, the specific case of operator overloading is given an example as it is widely used and can be quite powerful. Listing 2.8 shows a class having two operators overloaded.

Listing 2.8: Operator overloading.

```
1  #include <iostream>
2
3  struct Coffee {
4      Coffee(float temperature, float amount) :
5          temp_{temperature}, ml_(amount) {}
6
7      float temperature() const { return temp_; }
8      float amount() const { return ml_; }
9
10     Coffee operator+(const Coffee& coffee) {
11         auto total_ml = (ml_ + coffee.amount());
12         auto final_temp = ((temp_ * ml_) + (coffee.temperature() ↩
               * coffee.amount())) / total_ml;
13
14         return Coffee{final_temp, total_ml};
15     }
16     friend std::ostream& operator<<(std::ostream& os, const ↩
           Coffee& coffee) {
17         return os << "Temperature: " << coffee.temp_ << " C, ↩
               amount: " << coffee.ml_ << " ml";
18     }
19
20 private:
21     float temp_;
22     float ml_;
23 };
24
25 int main() {
26     Coffee coffee1{80, 100};
27     Coffee coffee2{50, 50};
28
29     Coffee mixed_coffee = coffee1 + coffee2;
30
31     // Outputs "Temperature: 70 C, amount: 150 ml"
32     std::cout << mixed_coffee << std::endl;
33
34     ...
35 }
```

A simple class `Coffee` has a temperature and an associated amount. The addition operator is overloaded to allow adding two coffees together. The returned `Coffee` instance has the amounts added and the simplified approximate mixed

temperature. The insertion operator is overloaded in order to let streams such as `std::cout` accept a `Coffee`.

2.2.6 Smart Pointers

The most common smart pointers `unique_ptr` and `shared_ptr` will be discussed briefly. Smart pointers are the recommended way to manage resources since C++11. They should be used by default when resources are allocated [33][38].

Consider Listing 2.9.

Listing 2.9: Smart pointers.

```
1  void complex_thing()
2  {
3      auto pobj = new SomeObject();
4
5      ...
6
7      delete pobj;
8  }
9
10 void complex_thing_modern()
11 {
12     auto pobj = std::make_unique<SomeObject>();
13
14     ...
15     // Safe!
16 }
```

The first function allocates memory on the heap, and is responsible for manually deleting it. A missed delete results in a memory leak. This can happen due to a complex function having several return paths, or perhaps a rare exception happens which does not cause the program to crash, but forgets an appropriate deallocation of the memory via `delete`.

The second function uses the smart pointer of type `std::unique_ptr`. This is the most efficient smart pointer and does not cause additional overhead when compared to traditional raw pointers. It deallocates the resource it owns when going out of scope. Calls to `new` and `delete` are encapsulated away from the user, and usage of smart pointers is exception safe with no memory leaks.

A unique pointer implies ownership of the resource pointed to. Transferring ownership is possible via `std::move(...)`. Transferring ownership is needed when the intent is to keep the resource alive but the scope of the smart pointer is about to expire. The memory must be given to a new owner in order to not be invalidated. If shared ownership is required, a `shared_ptr` is recommended. A shared pointer has a slight cost increase in its use. It points not only to the resource itself, but to a control block as well. The control block contains a reference count. If another shared pointer is made (e.g. via copying the first

smart pointer), this shared pointer will point to the same resource and the same control block. The reference count within the control block increases by one. If the reference count reaches zero, the resource is deallocated.

2.2.7 Lambdas

Lambdas (also called *closures*) creates unnamed functions capable of capturing variables in the surrounding scope by reference or copy. They are available since C++11.

Listing 2.10 shows a simple lambda assigned to a variable. The lambda can later be invoked by syntax equal to calling a ”regular” function.

Listing 2.10: A simple lambda function assigned to a variable.

```
#include <iostream>

int main() {
    int value = 0;

    auto fun = [&]() {
        value += 10;
        std::cout << "Value is: " << value << '\n';
    };

    fun(); // Outputs "Value is: 10"
    fun(); // Outputs "Value is: 20"
    fun(); // Outputs "Value is: 30"

    return 0;
}
```

The lambda definition starts with a pair of hard brackets. These indicate which variables from the surrounding scope should be *captured* and made available for use in the lambda function body. Placing ampersand here means everything is available for use by reference—which means the changes are persistent (i.e. non-local to the lambda). Placing an equals sign (`[=]` instead of `[&]`) captures everything by value instead. Individual variables can be named in order to be more specific about what to capture. The parenthesis act the same as for defining regular functions. This allows the lambda to be called with arguments.

Note that lambdas need not be assigned to anything. They are very useful when constructed in place of an argument to a function call.

2.2.8 Keywords

Some commonly used keywords are presented briefly. The explanations and examples are not exhaustive, but aims to target the most frequent uses of the keywords.

const

The `const` keyword is an important tool in aiding readability and protecting against misuse. Consider Listing 2.11.

Listing 2.11: Using `const`.

```
struct Animal {
    int legs();
    int eyes() const;

private:
    int num_legs;
    int num_eyes;
};
```

The function-call `eyes()` of class `Animal` is expected to return the number of eyes of the given animal instance. The function-call has been marked `const`. The function `legs()` has not been marked as such. This means that the instance might have its members changed as a result of the call to the function. As a result, a user can not read this function and be sure of the intention of the API creator. The function might return the number of legs, but it might also add a leg, remove a leg, or change the instance in some other unpredictable manner. As such, `const` should be used on any function not changing its underlying instance.

Consider Listing 2.12.

Listing 2.12: Using `const`.

```
void InspectType(Type t) {
    ...
}

void InspectType2(const Type& t) {
    ...
}
```

The first function accepts some type for inspection. As the type is not accepted by pointer or reference, the type is copied into the function body. This is potentially an expensive operation. As such, the second function uses the frequently seen pattern of using a `const`-reference. This allows a function to accept a type without copying it. The function refers to the same memory via a *reference*, but the compiler protects the passed object from being modified via the `const` qualifier.

As a rule of thumb, if an object is larger than three to four sizes of the system's underlying pointer size it should be considered to be passed by reference instead of by value for performance reasons [36].

virtual

Consider Listing 2.6. The base class is `Tool`, and it has a `virtual` function `use()`. The function `use_tool(...)` accepts any `Tool`, and calls that tool's `use()` if appropriate. This captures the use of virtual functions. If non-virtual functions are used, any child class passed as its parent (e.g. a saw passed as a tool) will call its parent function instead of its own. Thus the removal of the `virtual` keyword (the `override` keywords would have to be removed as well in order to compile, as an error is generated when they no longer override anything) would lead to the output "Using Tool" three times in place of the output seen in the original listing.

override

`override` is used for readability and to ensure safer refactoring. If a function having the appended `override` keyword has no matching `virtual` function in a parent class, an error is shown during compilation. This also then guards against typographic errors.

auto

The `auto` keyword lets the compiler deduce the type of a variable when it is being defined since C++11. The type is inferred from the right hand side of the definition. This can save retyping type information and is the recommended default when the result should be apparent [36]. Note that since that since the type is deduced from the right hand side, the right hand side must not be ambiguous in its type. If `auto` is used in such cases, an error will be given during compilation. In other cases, implicit conversions might take place. This can lead to unexpected behaviour.

2.3 External Tools

This section provides overview and use of third party tools used in the implementation stages of this thesis.

2.3.1 Vcpkg

Vcpkg is a freely available tool made by Microsoft [20]. The goal is to simplify the distribution, installation, and use of C and C++ libraries. Installing new

libraries is done in a single command line through Powershell (or other shells) [22]. Git [11] is used to download Vcpkg locally. It is the recommended way to use Vcpkg as it provides a way to update the available packages. To clarify, Vcpkg frequently provides updates to the packaged libraries (and adds new libraries). In order to update Vcpkg itself, a simple `git pull` command is issued—this enables the access of new libraries and updates.

Example installation instructions and installing a package is shown in Listing 2.13.

Listing 2.13: Vcpkg installation example.

```
PS C:\Example> git clone https://github.com/Microsoft/vcpkg.
    git
PS C:\Example> cd vcpkg
PS C:\Example\vcpkg> .\bootstrap-vcpkg.bat
PS C:\Example\vcpkg> .\vcpkg integrate install
PS C:\Example\vcpkg> .\vcpkg install boost
```

Note that the bootstrapping is only done once. Integrating the install is also only done once. Thus, after the commands in the listing are issued once, additional `vcpkg install <library>` commands are the most common use case (along with updating Vcpkg). Also note that by default, 32-bit versions of libraries are install as shared libraries. Static libraries and/or 64-bit version can be specified if desired.

Integrating the install is optional but very useful. This command makes all libraries downloaded automatically available in Visual Studio projects. The user does not need to manually include folders containing headers or libraries, and does not have to explicitly name library files for the linker. This greatly simplifies package management.

2.4 External Libraries

Various third-party libraries were used in implementing the new application. Some context for the majority of these is given here.

2.4.1 Boost

Boost [2] is a collection of open-source and peer-reviewed libraries. The libraries are well renowned in the C++ community and tightly integrate with the C++ standard libraries. Boost libraries have previously been integrated into the standard libraries, and more will be integrated in the future [37]. Boost consists

of many libraries. Only the most relevant libraries to this thesis' implementation will be discussed.

Graph

Boost Graph [4] provides a generic interface for using graphs and provides some common algorithms often used with graphs. These algorithms include depth- and breadth-first search, Dijkstra's shortest paths, topological sort, and several others. The library offers ways to find vertices associated with an edge, all edges in/out of a vertex, to allow or disallow directed edges, to allow or disallow parallel edges, get vertex degrees, get all adjacent vertices, and so on. Iterators are also central. These allow iterating over all vertices, or over all edges, and similar. Dynamic properties allows putting and getting custom properties for all edges and vertices.

An example of a useful graph built into the library is the grid graph. A figure depicting a grid graph is shown in Figure 2.4. The figure indicates we can create a two-dimensional grid of any size. Higher dimensions are possible. It is also possible to make the edges wrap, such that the first column is adjacent to the last column, and the top row is adjacent to the bottom row. The main feature of a grid graph is that all vertices have edges in and out from their horizontal and vertical neighbours.

The graph library is suitable for path-finding when it is possible to represent the terrain to be traversed as a graph.

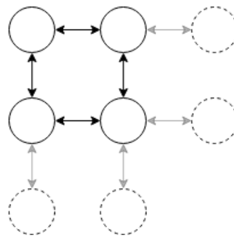


Figure 2.4: A grid graph.

Fiber

Boost Fiber [3] is a library that provides user-space threads. User-space stands in contrast to kernel-space, which is a more privileged and potentially more expensive mode of operation. Thus fiber-threads do not necessarily issue calls to

the operating system. The fibers are scheduled cooperatively. This means fibers are not preempted, and will have to explicitly yield. When yielding, a context switch occurs. The context is restored when the yielding fiber is resumed at a later point. Scheduling follows a round-robin scheme unless otherwise specified.

A fiber is bound to an underlying thread. Multiple fibers running on the same thread are lock-safe, as they can not run concurrently. A fiber will always run on the same thread, unless explicitly migrated. Note that fibers running on different threads will have to use synchronisation primitives. Boost Fiber primitives should be used, as using standard thread synchronisation would potentially yield a whole thread, which can possibly contain a large amount of fibers (we normally only wish to yield a single fiber at a time).

An example of usage is given in Listing 2.14. Note that some parts (e.g. headers) are left out for brevity.

Listing 2.14: Boost fiber usage.

```
1  void chprinter(std::string const& str)
2  {
3      for (auto c : str)
4      {
5          std::cout << "Character: " << c << std::endl;
6          boost::this_fiber::yield();
7      }
8  }
9
10 int main()
11 {
12     std::cout << "Begin!" << std::endl;
13
14     boost::fibers::fiber fiber_1(boost::bind(chprinter, "ABC"));
15     boost::fibers::fiber fiber_2(boost::bind(chprinter, "XYZ"));
16
17     fiber_1.join();
18     fiber_2.join();
19
20     std::cout << "End!" << std::endl;
21
22     ...
23 }
```

The associated output of running the program is given in Listing 2.15. This briefly shows the main point of using fibers—execution is interleaved via explicitly yielding execution at appropriate times. This enables multiple tasks to run more or less at the same time (assuming no task does any blocking which takes more than several milliseconds without yielding).

Listing 2.15: Boost fiber output.

```
Begin!
Character: A
Character: X
Character: B
Character: Y
Character: C
```

```
Character: Z
End!
```

2.4.2 SFML

SFML (Simple and Fast Multimedia Library) is a multi-platform multimedia library used for graphics, networking, audio, system-related tasks, and spawning windows (on which to e.g. draw graphics) [32].

SFML is a quick way to make a graphical application. The graphics module is the most relevant module for this thesis. A bare-bones SFML graphical application example is given in Listing 2.16.

Listing 2.16: SFML application.

```
1  #include "SFML/Graphics.hpp"
2  int main()
3  {
4    sf::RenderWindow window(sf::VideoMode(400, 300), "Example");
5
6    sf::CircleShape circle1(10); // radius is 10 px
7    sf::CircleShape circle2(20);
8    sf::CircleShape circle3(30);
9
10   circle1.setFillColor(sf::Color::Green);
11   circle2.setOutlineColor(sf::Color::Red);
12   circle2.setOutlineThickness(3.f);
13
14   circle2.setPosition({ 100, 100 });
15   circle3.setPosition({ 250, 200 });
16
17   while (window.isOpen())
18   {
19     sf::Event event;
20     while (window.pollEvent(event)) { ; }
21     window.clear();
22     window.draw(circle1);
23     window.draw(circle2);
24     window.draw(circle3);
25     window.display();
26   }
27 }
```

This example shows the core concepts of a graphical SFML application:

- Events: Each render loop, all spawned events are consumed. These can be key- and button presses, resize events, sensors, text, and so on.
- Clear: The previously buffered draws are cleared.
- Draw calls: Each point, line, triangle, circle and so on is drawn onto a buffer.
- Display: All buffered draws are now displayed onto the window being rendered.

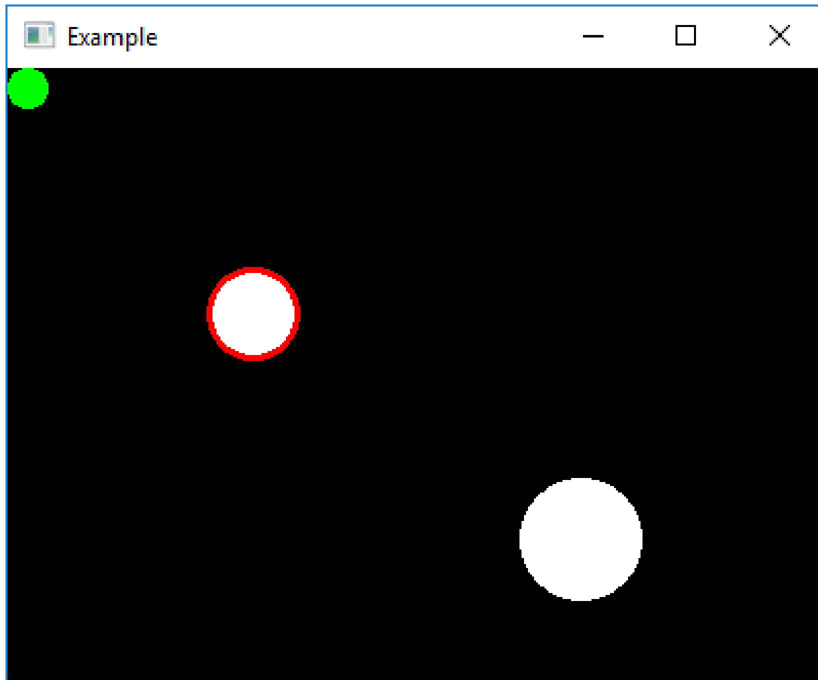


Figure 2.5: Example of creating a window and drawing some circles.

The produced graphical application can be seen in Figure 2.5.

`SFML/Graphics` has many other useful features for graphical presentations. Some are listed below.

- Scaling, rotating, and moving drawn elements.
- Displaying pictures (via sprites and textures)—not only primitives.
- Setting colours including alpha values of drawn elements.
- Setting frame rates, window sizes, and spawning additional windows.
- Drawing several thousands of elements with no considerable impact on frame rates.

2.4.3 imgui

Dear ImGui provides a user interface framework for graphical applications [6]. It has a small compatibility layer which allows it to be directly used with SFML [7]. Dear ImGui is self contained (no external dependencies), and can be used

with any renderer. It is well suited for creating user interfaces for tools, with menus, buttons, sliders, colour wheels, line plots, and many more built-in types.

An example of usage is given in Listing 2.17.

Listing 2.17: Imgui application.

```
// Create a window called "My First Tool", with a menu bar.
ImGui::Begin("My First Tool", &my_tool_active, ↩
    ImGuiWindowFlags_MenuBar);
if (ImGui::BeginMenuBar())
{
    if (ImGui::BeginMenu("File"))
    {
        if (ImGui::MenuItem("Open..", "Ctrl+O")) {/* Do stuff */}
        if (ImGui::MenuItem("Save", "Ctrl+S"))   {/* Do stuff */}
        if (ImGui::MenuItem("Close", "Ctrl+W"))  {my_tool_active ↩
            = false;}
        ImGui::EndMenu();
    }
    ImGui::EndMenuBar();
}

// Edit a color (stored as ~4 floats)
ImGui::ColorEdit4("Color", my_color);

// Plot some values
const float values[] = { 0.2f, 0.1f, 1.0f, 0.5f, 0.9f, 2.2f };
ImGui::PlotLines("Frame Times", values, IM_ARRAYSIZE(my_values));

// Display contents in a scrolling region
ImGui::TextColored(ImVec4(1,1,0,1), "Important Stuff");
ImGui::BeginChild("Scrolling");
for (int n = 0; n < 50; n++)
    ImGui::Text("%04d: Some text", n);
ImGui::EndChild();
ImGui::End();
```

The produced user interface can be seen in Figure 2.6.

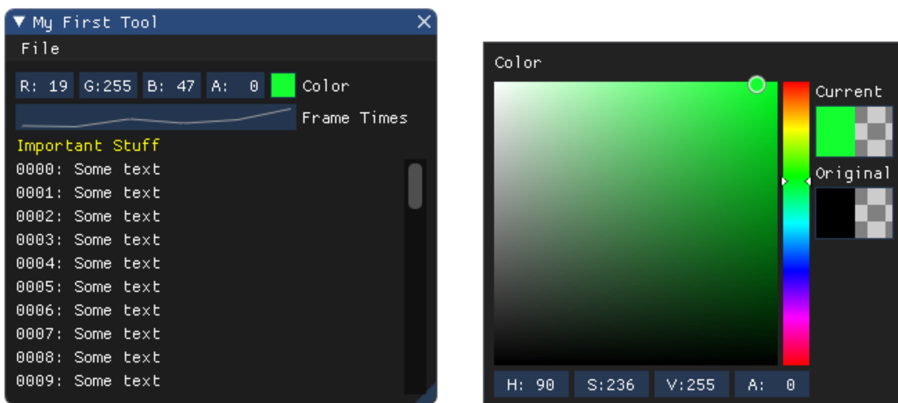


Figure 2.6: Example imgui usage producing a simple tool.

Chapter 3

Background of Project: SLAM

Aim of This Chapter

- Present the larger context of the thesis—SLAM.
- Contextualise this thesis' relation to the larger problem.
- Outline some of the characteristics of tackling a SLAM problem.

This thesis intends to build a solid foundation to solve the *simultaneous localisation and mapping* (SLAM) problem by creating a well-designed server application in C++. As the scope of building such a foundation is quite large, the SLAM problem will only be touched upon. However, SLAM is the core context of the project as a whole, and as such it will be introduced briefly in this chapter.

The SLAM problem involves placing a robot into an unknown area with no prior knowledge of the surrounding environment [8]. The robot has sensors on-board which are able to feed the robot with information about the environment, for example distances to obstacles found. This enables the robot to gradually estimate a map of the environment. As robots typically have a limited amount of on-board resources, one common scenario is to communicate measurements to an external device such as a desktop server. The robot's local coordinate system is known; it is defined by the robot itself. As errors in estimation gradually grow, it becomes harder and harder to decide the true location of the robot in global (*"true"*) coordinates. The SLAM problem is often divided into two main categories: online and offline. The offline problem typically collects all odometry results and all sensor results, and generates a probabilistic map and path from the given data. The online problem typically estimates the robot

position incrementally for each piece of sensor data and odometry measurement. Note that uncertainties for SLAM robots lie both in estimating the movement of the robot itself as encoders (or other types of sensors) have uncertainty, and measuring distances to obstacles also carries uncertainties. During the SLAM problem's lifetime, only the observation of previously known landmarks leads to a reduction of the aggregate uncertainty.

Theoretically, SLAM is considered a solved problem [8]. This holds for areas which are small, closed and static. If the area of interest is open, changing, and relatively large, the SLAM problem is still open for research [41]. There are many features that distinguishes a particular SLAM problem, some of which are [41]:

- Should the number of samples from sensors be high enough to be able to generate a photo-realistic representation of the environment, or should a more data-efficient approach be pursued—for example only collect data of surrounding *features*?
- Should we attempt a topological, qualitative representation of the environment where features are labelled, and try to relate features to each other?
- Should we prepare some information about the environment to be mapped beforehand, and try to identify key locations of this map while the problem is being tackled?
- Should we assume the environment is non-changing, or do we allow a time variant environment?
- How much uncertainty will we allow the robot to accumulate during the lifetime of the mapping? This affects our ability to recognise a *closed loop*—meaning we have arrived at some previous location we have already visited. Do we allow the use of external data—for example GPS data—to help us reduce uncertainty?
- Will we use a single robot to perform the SLAM problem, or will we use multiple cooperative robots at the same time?

Solutions to the SLAM problem can be quite involved, and is out of scope for this thesis.

Chapter 4

Java Server Application

Aim of This Chapter

- Get an overview of the scope of the current Java server application by inspection of the entire codebase.
- Summarise the findings of looking into the codebase into a higher-level overview.
- Locate positive and negative aspects of the Java application.

Based on availability of source files and the wish to inspect the most recent version of the Java application possible, the Java source used is from [25]. This source is also bundled with this thesis for completeness—see Appendix A.

The intent of looking at all the source files of the Java application is to make sure no implemented features are missed, and as such the scope of the implementation should be clear. This sets the stage for what the new application should hope to achieve, and which parts should be left behind and which should be kept.

4.1 Codebase Summary

This section provides a brief insight into the entire codebase of the Java application as interpreted by manual inspection. Note that the previous authors of the Java application have separated their contributions by prefixing their classes and functions by short keywords—initials of names. This essentially creates namespaces (see Section 2.2.1). This allows for a natural structuring of the summary, and thus the overview of the individual three namespaces `no.ntnu.et` (Eirik Thon [39]), `no.ntnu.hkm` (Henrik Kaald Melbø [19]), and `no.ntnu.tem` (Thor Eivind Andersen and Mats Rødseth [28]) follows.

4.1.1 Namespace: `no.ntnu.et`

This namespace is further divided into `general`, `map`, `mapping`, `navigation`, and `simulator`.

`no.ntnu.et.general`

`Angle.java` is a simple representation of an angle in degrees. It is also able to draw a line with a given angle onto graphics.

`Line.java` represents lines, and has constructors for creating lines given a pair of coordinates.

`Position.java` represents an *(x, y)* coordinate in the form of two `double`s. It can do common operations like summing positions, copying positions, getting distances between positions, but also graphical operations like drawing a circle or a cross at the given position.

`Pose.java` represents a robot's position and orientation (or angle). It can be moved, rotated, transformed, and more. It can also be drawn graphically.

`Utilities.java` is a collection of various functions with no particular commonalities. These include polar to Cartesian conversion, getting colours via numbers, getting intersection points between lines and circles, and more.

`Vertex.java` is a short abstraction of a graph vertex.

`Observation.java` collects four coordinates corresponding to robots which use four infrared sensors simultaneously.

`Navigation.java` contains a function for getting the angle towards a point.

`no.ntnu.et.map`

`Cell.java` represents cells in a grid map. It is used to give meaning to positions in a given grid map, such as making a position *free* or *occupied* (for traversal).

`GridMap.java` represents the map in the form of a grid of rows and columns. This map is able to accept locations which then might get occupied, or might get free. There is also functionality for higher-level compound tasks such as a function which "Finds and returns the MapLocation of all cells that are free, not restricted and has an unobserved neighbour". The map is also dynamic in size—it can change during runtime.

`MapLocation.java` represents a location within the map, as a *(row, col)* pair.

`no.ntnu.et.mapping`

`MappingController.java` runs the mapping process in a thread. It accepts robots which are then mapped during the lifetime of the program (or until removed). It also has general matrix operations, a line algorithm, and can initiate docking (for robots with docking capabilities).

`Sensor.java` simply has a position (as an *(x, y)* pair), and a flag member named `isMeasurement`.

`MeasurementHandler.java` accepts a robot with an initial pose. The class has an array of four sensors. The main functionality of this class is updating measurements from robots. The measurements seem to be received not as coordinate points, but as raw measurements from robot sensors. This means measurements must be calculated by taking physical robot parameters into account, including several angles (of sensors) and offsets (sensors not located above the robot's center of mass).

`TransformationAlg.java` is credited to Lars Marius Strande [35], but is located in this namespace. It has various operations on matrices, such as finding the center of mass of a given matrix, getting the translation matrix, getting the rotation, and more.

`no.ntnu.et.navigation`

`CollisionManager.java` this class checks if robots are about to collide with walls or other robots, and attempts to navigate around such events.

`MapNode.java` this class represents a node in a graph, and is used for pathfinding. It is a combination of a linked list and cost functions between nodes.

`NavigationController.java` this class can send commands to robots, stop robots, resume robots, add and remove robots, controls the collision manager, and similar.

`NavigationRobot.java` this class contains a list of the waypoints for a robot. It also has a concept of `priorityCommands`, and has flags for setting and getting collision status for a given robot.

`PathPlanningFunctions.java` this class has functions for pathfinding. Thus it can accept a grid map, a starting location, and a target location, and generate a path of waypoints from these parameters.

`RobotTaskManager.java` this class finds target locations for robots to move towards.

`SlamNavigationController.java` this class is attributed to Geir Eikeland [9]. It is a simple handler of robots and measurements.

`SlamPathPlanningFunctions.java` is another set of functions for planning robot paths. It largely overlaps with the previous path planning code.

`SortedMapNodeList.java` deals with keeping a sorted list of nodes based on their cost.

`no.ntnu.et.simulator`

`BoundaryFollowingController.java` sets the simulated robots movement. It randomises some behaviour. It can calculate shortest distances and headings.

`Drone.java` simulates a drone, with fake headings and movement.

`Feature.java` implements features. A feature is a wall or obstacle. It is characterised by its start and end positions; it is a line. It is *paintable* i.e. it can be shown graphically.

`GraphicContent.java` shows all graphical content. This includes estimated poses, targets, sensor beams, the map, robots, and similar.

`InitialPoseDialog.java` is a class which describes a dialog which pops up and prompts the user for an initial pose of a robot.

`SimRobot.java` simulates a robot, with fake sensors, measurements, and communication.

`Simulator.java` has a run loop which controls robots to some degree, and adds simulated messages to the system's *inbox* (to be seen later). It can pause and unpause robots, set robot commands, turn on and off estimations, and more. It also controls some aspects of the GUI.

`SimulatorGUI.java` represents the user interface of the simulator. Handles mouse events, buttons, panels, and other related user interface features.

`SimWorld.java` has a list of robots, can create robots, can find intersections, read maps from a file name, add features for borders, and more.

`SlamRobot.java` is a class inheriting from `SimRobot` but does nothing more.

4.1.2 Namespace: `no.ntnu.hkm`

This namespace has the single sub-namespace `particlefilter`.

`no.ntnu.hkm.particlefilter`

`MapMatching.java` accepts two hash maps (with strings as keys and integers as values), and computes a score for how correlated they are. In other words, finds out how similar two maps are.

`MapMerger.java` has algorithms and functions for taking two maps and merging them into a single map.

`Particle.java` implements particles. The description in the source code comments is "Each particle represents a pose and path hypothesis for the robots". The class has a pose, a vector of previous poses, a global map, a local map, can match maps, and find a location given a map.

`Particlefilter.java` implements a particle filter. This class is the filter which seeks to improve robot poses. It can find distances between points, propagate particles forward in time (i.e. predict) via Gaussian noise, print to CSV (comma separated value) file, create maps, and has a runner function which runs in a thread.

4.1.3 Namespace: `no.ntnu.tem`

This namespace is further divided into `application`, `communication`, `gui`, and `robot`.

`no.ntnu.tem.application`

`RobotController.java` keeps track of robots in the system. Adding a robot requires specifying address (as an integer, i.e. an ID), a string name, width, length, update rate, offset of the axle (and sensors), and sensor headings. There are functions which only apply to a drone-type robot. There is also some functionality related to battery management.

`Installer.java` has functionality for copying a file, and copying some `.dll` files to some predefined paths.

`Application.java` is the main class of the program. It has global flags such as if the simulator is active, if the application is paused, and if the particle filter is active. It has a navigation controller, a mapping controller, a robot controller, and the world map. The GUI is also a member here, and the communication. It can also open a PDF file, write commands to robots, set serial communication ports, and more.

`no.ntnu.tem.communication`

`ARQProtocol.java` is a class implementing a protocol for sending and receiving messages. It also maintains a map of integer IDs keys to connections. It expects a reference to a network, and an inbox. It can be used to create connections, send data, receive data, listen for new connections, and more.

A second class, `ARQConnection`, represents a stateful connection. It can contain incomplete messages, has an integer address associated with it, has a timeout, can receive and send, and more. Messages can be fragmented and have to be reassembled when entirely received. Also, sequence numbers associated with messages must be maintained, and acknowledge responses are expected.

A third class, `ARQSegment`, wraps itself around the actual payload being sent to a reciever. This segment part indicates the intent of the payload, and has a type field which is one of `Data`, `Ack`, `Syn`, `SynAck`, or `Alive Test`.

`BatteryMessage.java` is a simple class holding a battery level and an array of bytes.

`CRC8.java` computes 8-bit checksums for arbitrary data.

`CobsUtils.java` encodes and decodes byte arrays with respect to the COBS scheme.

`Communication.java` contains references to the protocols used in the Java application, serial communication, message inboxes, an inbox reader, and more. It also has functionality for displaying the available serial ports on the host machine, sending orders to robots, confirming *handshakes*, pausing robots, and more.

`DroneUpdateMessage.java` parses a message on the format expected of a drone robot.

`Frame.java` represents a *frame* as used in the custom protocols used in the application. A frame has a sender, a receiver, a protocol, data, and a checksum.

`HandshakeMessage.java` contains the definition of a *handshake message* which is a message type exchanged when robots first connect as per the Java application. It involves several fields: Robot name, width, length, offsets, headings, and a deadline.

`InboxReader.java` implements an inbox. This class runs in a separate thread when initiated. It reads from the system's inbox. If a message is available, it is parsed based on type. The different types handled here are *battery update* messages, *idle* messages, *handshake* messages, and several others. It also adds a robot to the system when it sees a successful handshake.

`ListPorts.java` lists the available serial ports on the host computer.

`Message.java` defines the available message types sent between the Java application and the robots. These types are: `Handshake`, `Update`, `Order`, `Idle`, `Pause`, `Unpause`, `Confirm`, `Finish`, `Ping`, `Ping Response`, `Debug`, `Drone Update`, and `Battery Update`.

`Network.java` is responsible for sending messages over the serial communication port. It also pops messages from the *frame* inbox, and receives them.

`Protocol.java` is a protocol interface composed of the two functions *send* and *receive*.

`SerialCommunication.java` puts bytes onto the output stream, which results in putting actual data onto the serial port. It also listens to the serial port, and streams incoming bytes into an array until delimited. Then the COBS layer is decoded, and the message is put onto an inbox.

`SimpleProtocol.java` is a replacement protocol which can be used instead

of `ARQProtocol`. It has functions for sending, receiving, and reassembling messages.

`StatusMessage.java` contains definitions for status messages. The possible types here are `Pause`, `Unpause`, `Idle`, `Confirm`, `Busy`, and `Finish`.

`UpdateMessage.java` is the type expected regularly from a previously connected robot during its lifetime. It contains the robot pose and various fields related to sensor readings (data and angles).

no.ntnu.tem.gui

`RobotPanel.java` relates to Figure 4.1. When a new robot connects to the application, an entry such as seen in the figure is added.

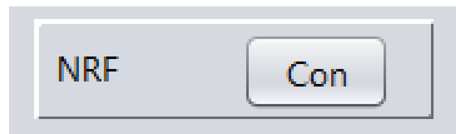


Figure 4.1: Robot entry in list.

`RobotInfoGUI.java` relates to a panel specific to a single robot after it has connected and started running. See Figure 4.2.

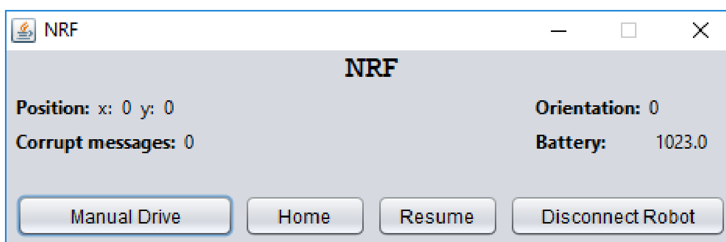


Figure 4.2: Robot info panel.

`ParticleMap.java` relates to rendering a particle map.

`ParticleFilterOptions.java` relates to the various settings available for the particle map, as seen in Figure 4.3.

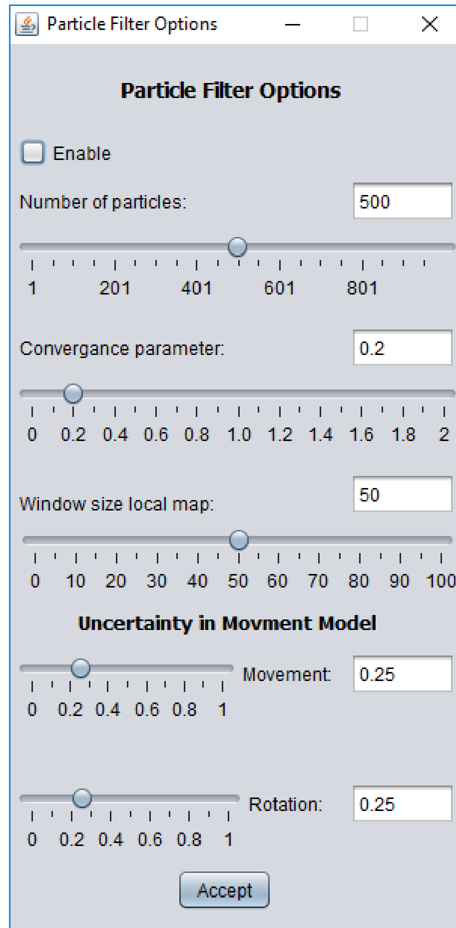


Figure 4.3: Particle map settings.

`ModeSelectionGUI.java` relates to a panel shown at application start. See Figure 4.4.

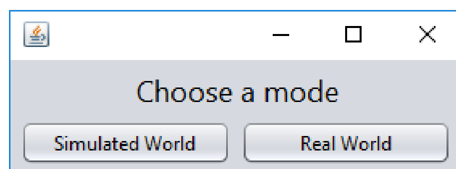


Figure 4.4: Choosing mode.

`mapMergeOptions.java` relates to controlling the merging of maps.

`MapGraphic.java` controls the rendering of the map itself. An example is given in Figure 4.5.

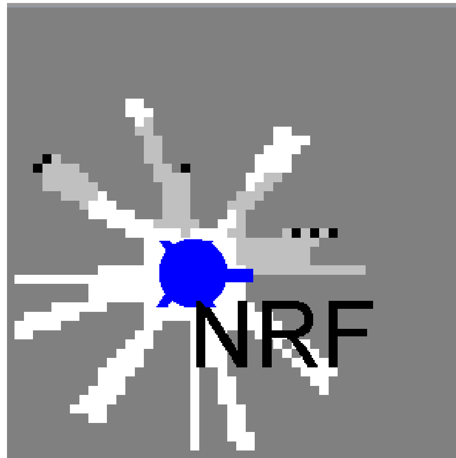


Figure 4.5: Map with a robot and some data.

`InitialRobotParameterGUI.java` prompts the user for setting initial robot parameters, see Figure 4.6.

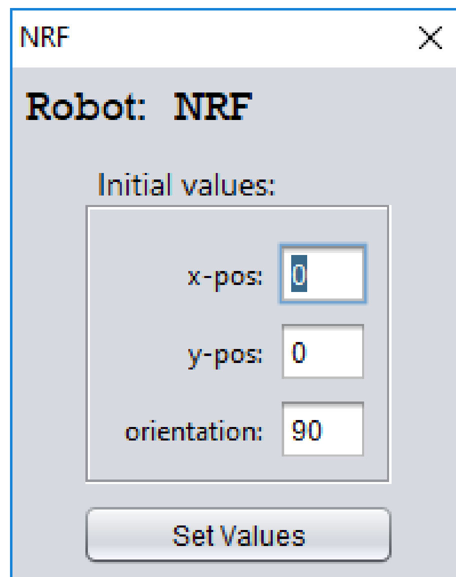


Figure 4.6: Setting initial robot pose.

`ManualDrivePolarGUI.java` chooses a point to manually drive to, in polar co-

ordinates. Similar to Figure 4.6.

`ManualDriveCartesianGUI.java` chooses a point to manually drive to, in Cartesian coordinates. Similar to Figure 4.6.

`MainGUI.java` holds the various elements of the main application. See Figure 4.7.

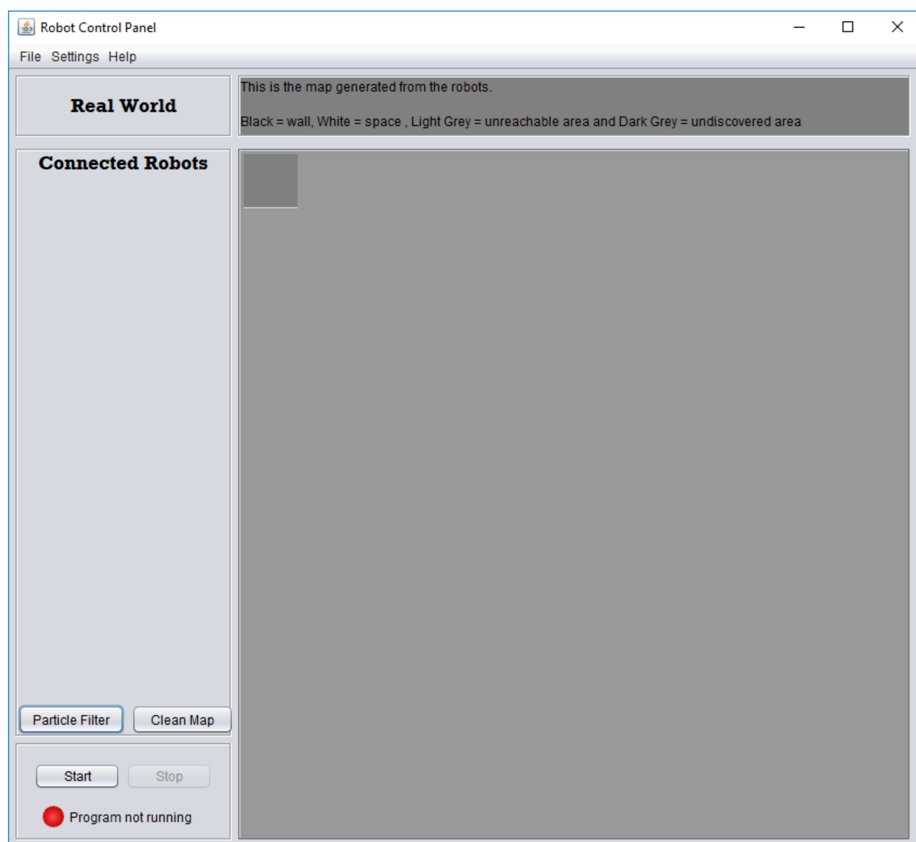


Figure 4.7: Main window.

`AboutGUI.java` concerns the *About* entry under the *Help* entry found on the toolbar as seen in Figure 4.8. The menu shown is seen in Figure 4.9.

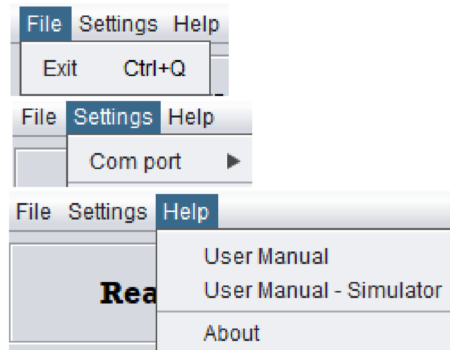


Figure 4.8: Toolbar entries.

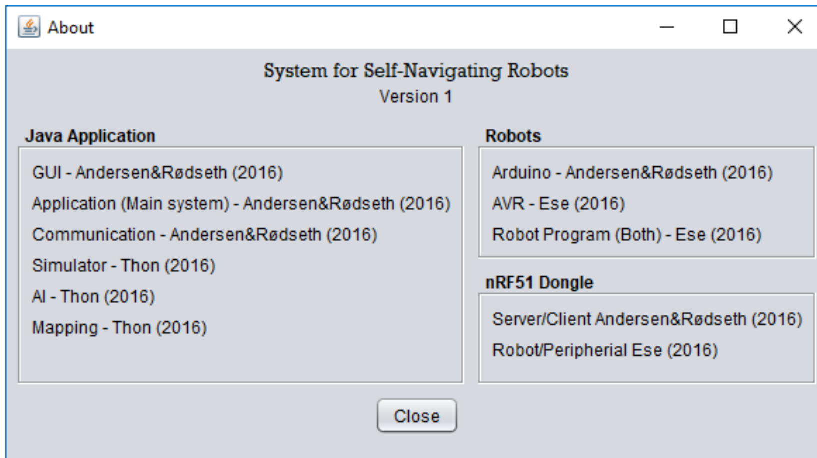


Figure 4.9: About menu.

`no.ntnu.tem.robot`

`Robot.java` is a monolithic class (over 40 member variables) for representing a robot. Some of these include: An integer ID, a string name, offset of the axle, offset of sensors, the infrared sensors, measurements, concurrency locks, flags, and more. Member functions include adding measurements, getting transformation matrices, getting alignment, polling whether the robot is stuck, and more.

`Measurement.java` this class represents readings from sensors of a given robot. It stored the orientation of an infrared sensor at the time of measuring, the data itself, and robot pose.

`IR.java` is an abstraction of the infrared sensors which some of the robots are equipped with. It stores the current heading of the sensors, the number of infrared sensors, and more.

4.2 High-Level Overview of Features

4.2.1 Graphics, User Interface

Graphically, the application is quite simple. See Figure 4.5. Another example is seen in Figure 4.10.

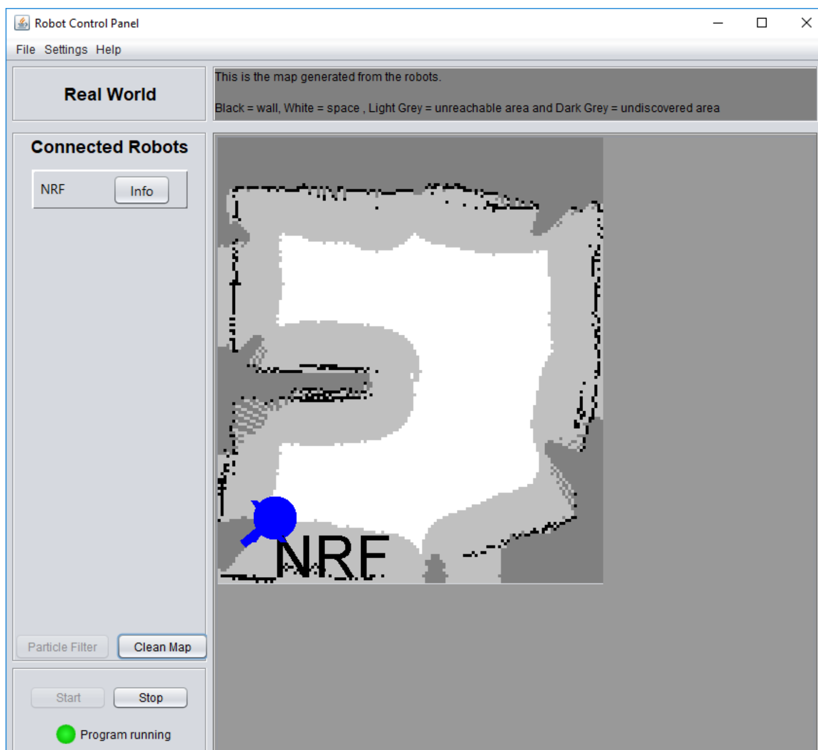


Figure 4.10: Robot in the process of mapping.

Here we see that generally, robots are circles with a rectangle indicating the robot's last known heading. Robots are labelled with their name. The white area has been explored, grey areas are areas which the robots should not traverse (e.g. due to the proximity to a wall), and black pixels indicated measurements

from sensors, i.e. obstacles. The map can be zoomed in and out.

The user interface is unsurprising. It consists of labelled buttons, sliders, and input fields. A toolbar is also present.

4.2.2 Robot Handling

Robots are appended to a list of *Connected Robots*. Robots can be disconnected, paused, resumed, told to drive home, and sent to manual locations. Miscellaneous information is available, such as position, orientation, and battery level if applicable.

4.2.3 SLAM

Some higher level SLAM features have been attempted and are available. Robots are automatically sent to explore new locations via the communications protocol and internal algorithms. There have been experiments with using particle filters and merging maps for raising the level of quality of the mapping process (as opposed to displaying only raw data from robots). The results have been mixed [19].

4.2.4 Simulation

The simulation features include a whole separate graphical interface and several in-depth features such as placing a robot within a map which has hidden obstacles. The simulated robot will find these obstacles when traversing the simulated world, and can be set to have errors in its pose and sensor outputs. Simulated maps can be saved and loaded from disk in a textual format.

4.2.5 Communication

The communication layer is complex. Both the server and robots need to have a physical dongle connected, providing Bluetooth connectivity. There is no support for using Bluetooth capabilities within a desktop computer, or Bluetooth chips on the robots other than from Nordic Semiconductor, because a proprietary protocol is used for sending data over Bluetooth. Additionally, the use of one of these dongles impose a limit of connecting three robots at the most at any one time [28]. All robots, dongles, and servers will need to maintain and implement checksumming, COBS encoding and decoding, and the ARQ proto-

col. These must be maintained in several programming languages, as both C and Java is used in this communications stack.

In short, the checksum is generated by a stateful mathematical operation over the contents of a message to be sent.

COBS encoding and decoding is necessary due to the fact that a serial communication is involved in the communication stack at several levels. The serial communication end-points do not use control signals when transferring messages, and thus need to be delimited by a special character to indicate when a message is completely transferred. However, as serial data streams are sent byte-for-byte, only the values 0 to 255 are available, and all can potentially represent valid data. Therefore, COBS transforms the data into a new format which does not use the data value 0. Thus the value 0 can be used to signify the end of a message.

The ARQ protocol is concerned with retaining a mapping between senders and receivers to their respective connections in the application. It also initiates sending data, receives data, divides data into several fragments, reassembles data, puts type information onto payloads, retransmits lost data, verifies sequence numbers, and more. The division of data into several packets is necessary due to the severe limitation of only sending 20 bytes in any one packet. Additionally, a maximum of 60 bytes can be sent in total, which will then be fragmented into three packets [28].

4.2.6 Miscellaneous

There are a number of miscellaneous functionality within the application. They are mostly used internally and do not affect the high-level overview in any meaningful way. However, they mentioned here for completeness. More details can be found in Section 4.1.

- Various functions for dealing with angles, lines, coordinate conversions (polar to Cartesian) exist.
- Representations of graph features such as vertices and nodes exist.
- Abstractions of the map into a grid with cells exist.
- Representations of robot features such as sensors and physical parameters exist.
- Various operations on matrices exist.

4.3 Discussion

Graphically, the Java application is quite simplistic. Some more options to tune the visual experience during the SLAM session could be beneficial. Panning and changing the appearance of measurements and robots would improve the experience. Visualisation of the path intended to take by robots could also be useful for the SLAM server operator. The user interface works generally well.

The handling of robots could be improved. In source code, there are many classes which reference or interact with the robots. As such, it is hard to get an overview of the control flow of robot handling. Moreover, there are several instances where hard-coded pieces of code targeting only a single type of robot. This makes source code less general, less readable, and does not scale well. The actual mapping of sensor data and pathfinding for robots seems to work well in general.

Communication is the weakest part of the Java application. The complexity is too high and there are too many limitations. The small message size possible without fragmenting messages and the limitation of only three robots is severe. Additionally, proprietary protocols are used at both the server side and on the robots. This means the robots are tied to a single closed ecosystem. It is thus unlikely that the robots are able to cooperate with other smart devices using Bluetooth. There are also several thousand lines of user-made code related to maintaining the communication stack in two different programming languages. This is error prone and not very maintainable. As such, the communication stack should be carefully considered before rewriting it in the new application—a new approach is likely better.

Chapter 5

Case Study: Thread

Aim of This Chapter

- Introduce an alternative method of robot-server communications—Thread.
- Detail the general workings of the Thread protocol stack.
- Present a widely used implementation of the Thread specification—OpenThread.
- Introduce an application-layer messaging protocol which can run on top of Thread—MQTT.
- Present the case study which takes all the above points into practice.

The state of the SLAM project at the start of this thesis has a heavy reliance on Bluetooth [1] and vendor-specific communications (Nordic UART Service [31]). A case study was made in order to look at a viable alternative, possibly decoupling the software from having to use specific protocols, and not having to rely on specific hardware.

This chapter will introduce an alternative to Bluetooth, and a well-established reliable communication protocol. The pros and cons of this approach will be discussed, as well as a possible migration plan. An example conducted early in the thesis will be presented.

5.1 Thread

Thread [14] is a networking protocol intended for wireless low-power devices. It is royalty free. The protocol specification is closed unless paid membership is

acquired with the Thread Group alliance.

Some features are [40]:

- IP-addressing: Through border routers, each device can be addressable through the internet via its IPv6 address.
- Security: Authentication and encrypted communications are default.
- Reliability: Mesh networking is default, and devices in a Thread network will automatically changes roles in order to *heal* itself after e.g. devices disconnect.
- Low power: Up to years on normal household batteries.
- Reliable transmission: Each *hop* in the mesh network is *ACKed* and retried if none is seen.

There are some considerations to be made based on the application which Thread is intended to be used on. Bandwidth is limited to 250 kbps. Distances of over 30 metres between devices is unlikely to perform well [12].

Consider Figure 5.1. It shows a simple overview of how Thread relates to the Open Systems Interconnection (OSI) model [23].

A brief bottom-up explanation of how Thread relates to the model follows:

1. Physical: This layer is concerned with receiving and transmitting raw bits over the available underlying physical medium. Thread does not specify this layer, but uses IEEE 802.15.4 PHY here.
2. Data Link: This layer is concerned with the flow of data frames, ACKing, retransmission in case of issues, error checking, and so on. Similarly, Thread does not specify this layer but uses IEEE 802.15.4 MAC.
3. Network: This layer represents the network, and deals with address mapping, routing, reassembly, and so on. Thread specifies this layer and uses IPv6 over 6LoWPAN here.
4. Transport: User Datagram Protocol (UDP) is used for the transport layer. This is a lightweight protocol with less overhead than Transmission Control Protocol (TCP). UDP uses checksums (mandatory for IPv6, optional for IPv4) to ensure data integrity on this layer.

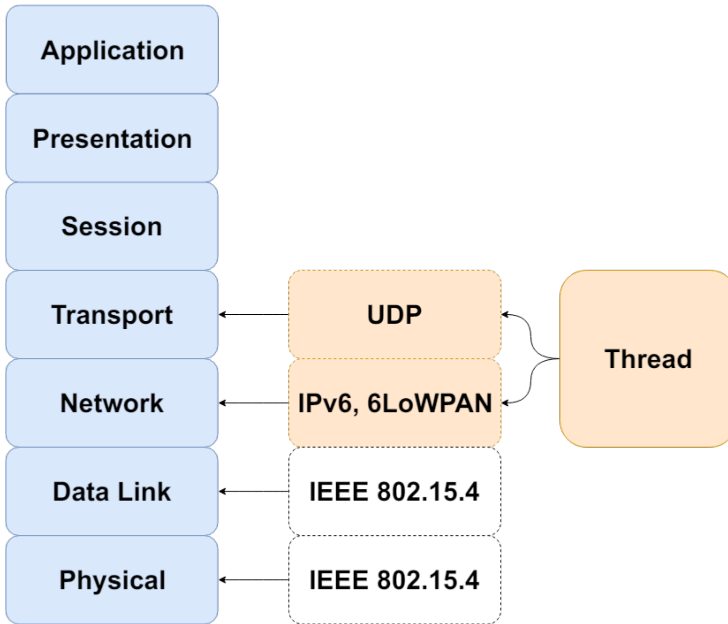


Figure 5.1: Simple Thread OSI model.

5. Session: This layer represents sessions between applications. A well-known example of such a session is via sockets. Thread does not make assumptions on this layer.

6. Presentation: This layer deals with serialising and deserialising data, such that the above application can use its native data representations. Thread does not interact with this layer.

7. Application: This layer deals with higher level software, such as e-mail, databases, and so on. Thread does not specify this layer.

5.1.1 IEEE 802.15.4

IEEE 802.15.4 is a standard which has specifications on the data link- and physical layers of the OSI model. It differs from e.g. WiFi which uses the set of IEEE 802.11 standards specifying the same layers. WiFi was not developed with the same low-power, low-bandwidth intentions which IEEE 802.15.4 has been.

Some interesting features of this standard:

- Guaranteed time slots: This feature enables real-time applications by guarantees of transmitting and receiving data.

- Collision avoidance. An algorithm sensing when others are speaking on our channel is used. If idle, send our frame. If not, wait a random period of time and retry. Control signals (ready to send, clear to send) might be used.

- Several frequency bands. 868 MHz is used in Europe, 915 MHz in North America, and 2540 MHz worldwide.

- Focus on power efficiency. Has a multitude of strategies to save on power when possible.

The physical layer

Data transmission occurs on this layer. Different channels on the chosen frequency are managed here. Signal strength and energy usage and monitoring also happens here.

The data link layer

MAC frames are sent here. The layer acts as a higher level manager of the physical layer, controlling access. Beaconing is also performed here. Beacons are sent and received in order to exchange information about the network.

5.1.2 IPv6

The Internet Protocol version 6 (IPv6) is used in Thread. This means devices may have global unique addresses which are reachable on the global internet. An overview of the fixed IPv6 header [13] is given in Table 5.1. Note the next header can also be used to point to another header—an extension header. Extension headers can again point to other extension headers. This allows an arbitrary number of headers to be used.

IPv6 allows Thread to use a standardised way of communicating with the outside world such that routers, computers, smart-phones, and other devices can understand it.

Field Name	Length (bits)	Explanation
Version	4	This field states the version of the IP used; for IPv6 this is simply the constant 6.
Traffic Class	8	This field can label the type of traffic categorically. For example, it can specify the data as multimedia streaming, telephony, low-priority, and more.
Flow Label	20	A flow label can be used to group packets into a *flow*. This can be used by routers for optimisations and/or avoiding reordering and more.
Payload Length	16	A number indicating the size of the payload given in number of whole bytes.
Next Header	8	This number indicates the type of header following the current one. For example, the value 17 indicates an UDP header.
Hop Limit	8	This number is decremented each time the packet is forwarded. If it reaches 0, the packet is discarded.
Source Address	128	IPv6 address of the sender.
Destination Address	128	IPv6 address of the recipient.

Table 5.1: IPv6 header format.

5.1.3 6LoWPAN

IPv6 over Low-Power Wireless Personal Area Networks is shortened to 6LoWPAN. 6LoWPAN bridges devices using 802.15.4 radios with IPv6, allowing the devices to both send and receive IPv6 packets [16]. Similar to IPv6 headers, 6LoWPAN uses stacked headers. IPv6 packet payloads might be too large for use with 6LoWPAN. 6LoWPAN introduces fragmentation headers for this. This allows large packets to be sent in pieces. 6LoWPAN then reassembles the packet at the receiver. 6LoWPAN also compresses IPv6 headers as much as possible for efficiency in use for low-power devices. 6LoWPAN also supports a mesh header, which is utilised for very efficient link-layer forwarding between devices in a mesh.

5.1.4 Nodes and Devices Types

Consider Figure 5.2. It shows two nodes found in a Thread network. A distinction is made between devices in a network based on how they act with respect to packet transmission, mainly. An end device—or child—communicates with

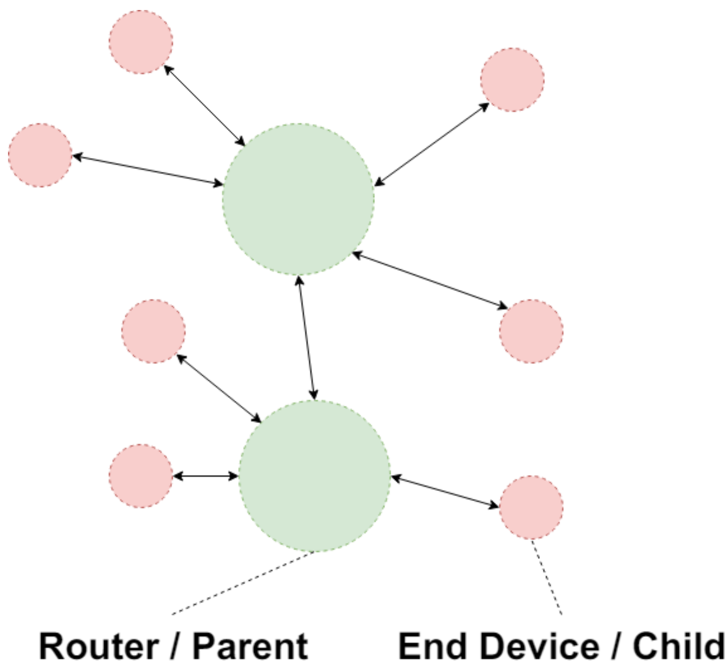


Figure 5.2: Thread nodes overview.

one router. Thus it can not forward packets from other routers or end devices. This allows end devices to disable their transceivers, enabling high efficiency and low power consumption for these devices. On the other hand, routers can and will transmit—or forward—packets to other routers and end devices. The transceiver is always on for a router. Additionally, routers allow new devices to securely connect to the mesh network.

Now consider Figure 5.3. It shows the different types nodes can be. There are four ways to be categorised an end device. You can be a full end device, a router eligible end device, a minimal end device, or a sleepy end device. The full Thread device router node is the only type which is not an end device.

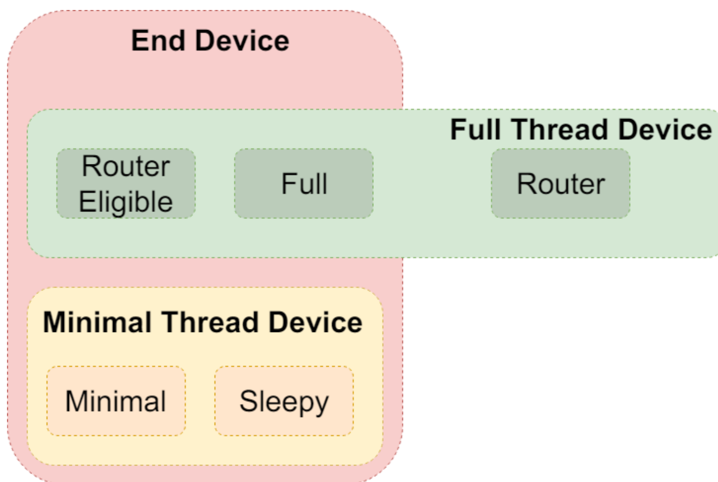


Figure 5.3: Thread device types relationship.

Full Thread devices have their radio on at all times, are accessible through a special multicast address, and contain mappings of various IPv6 addresses to devices. The *router eligible* type can be promoted to a router if needed. The full end device can not. Thus full Thread devices might take on the role of both parent and child. When a new end device intends to join the network, a router eligible node might be upgraded to a router to accommodate this. In contrast, a router which has no children might dynamically become an end device.

Minimal Thread devices must pass all messages to parents and do not receive multicast messages. A minimal end device keeps its transceiver on and thus gets messages as soon as they are ready. On the other hand, sleepy end devices are normally asleep, and must wake up periodically to ask their router for messages via polling.

Another role is the **Thread leader**, which is a single router of the set of all routers. It manages the routers. A device can also be a **border router**, which allows the Thread network to interact with other types of networks, such as the global internet.

Figure 5.4 shows the case of a Thread network after a crucial link has lost its connection. The network is now partitioned into two pieces. If the connection is restored, the network is automatically merged back together. After a partition, new roles are automatically assigned as needed. Thus new Thread leaders might be assigned, and nodes might become routers, or routers might become end devices.

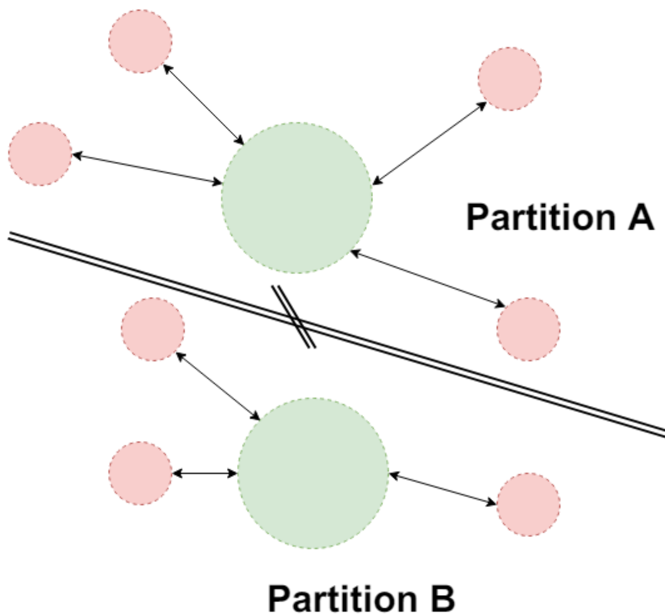


Figure 5.4: Thread network partitioned.

A Thread network is compromised of a single leader, up to 32 routers, and up to 511 end devices per router. This limits the network to about 16,000 devices.

5.1.5 Addressing

A look into how Thread addresses its devices brings more clarity to the how the protocol works.

Figure 5.5 show the possible scopes available in a Thread network. The link-local scope only needs a single radio transmission in order to pass on a message. A mesh-local scope is the scope that encompasses an entire Thread network. The global scope includes all devices that might be accessible from outside the Thread network. The link-local scope uses IPv6 addressing prefix `fe80::/16`

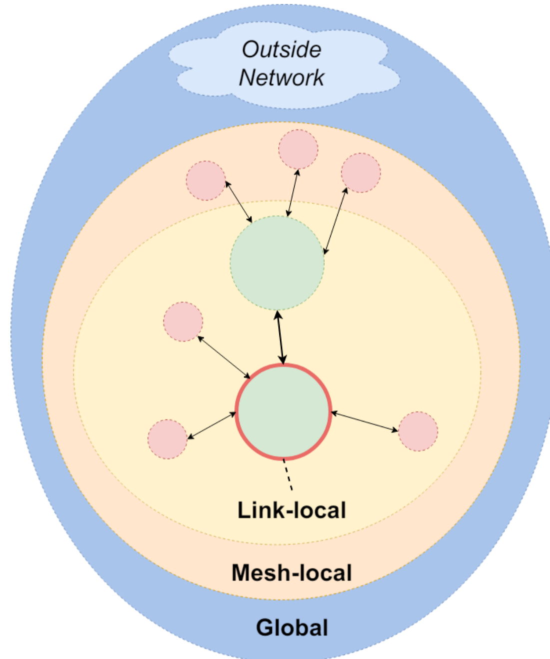


Figure 5.5: Thread network scopes.

and mesh-local uses `fd00::/8`, while the global prefix is `2000::/3`, which is the address used when connecting to a device publicly outside the network itself.

Other addresses include a *routing locator* address which identifies a device within the network based on the current topology (the parent of a child is observable through the child's address), a mesh-local endpoint identifier address which identifies a device independent of the topology, and more.

It is also possible to send messages via multicast. The multicast address differs based on if the user intends to send a message to all full thread devices and minimal thread devices, only full thread devices, and if the message should be link-local or mesh-local.

5.1.6 Network Creation and Joining

There is a slew of information attached to the Thread network itself. Three identifiers are used. These are the personal area network ID (the PAN ID), the extended personal area network ID (the XPAN ID), and the human readable name which is much like the SSIDs known from WiFi. The PAN ID is specified to be two bytes long, while the XPAN ID is eight bytes long.

A device scans for networks by performing the operation as shown in Figure 5.6. Notice the figure has labelled three steps.

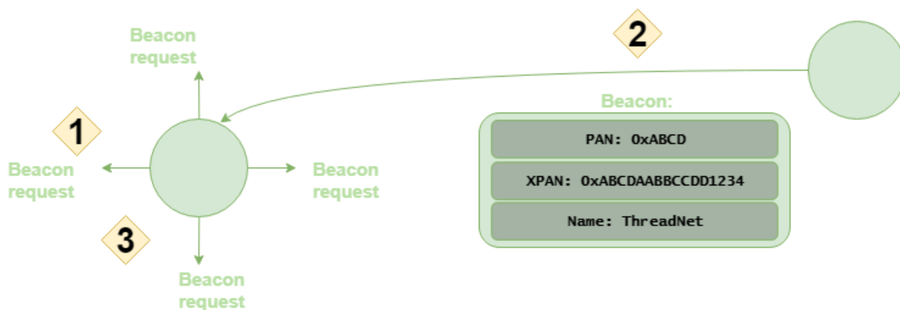


Figure 5.6: Thread network discovery.

1. A device sends out beacon requests through its radio on a single channel. The intent is to illicit responses from all other devices in range.

2. A device was in range. It responds to the beacon request with a beacon. A beacon contains the PAN ID, the XPAN ID, and the human readable name of the network.

3. The device has collected responses from all devices in range. It starts the process over on step 1 on a new channel until it has exhausted its number of channels.

Given the information about neighbouring networks, the device might not create a new network or join one of the discovered ones. If a new network is created, the device is now a leader and a router in the created network. The network is automatically put on an appropriate channel—one which is not busy. The PAN ID is also chosen such that it does not collide with other PAN IDs the device has knowledge of.

Joining a network happens through a more involved handshake process. Figure 5.7 illustrates this process. Note that the figure moves in time in the

downwards direction: there is a single green node at three points in time, and two blue nodes at two points in time. The steps in more detail are as follows:

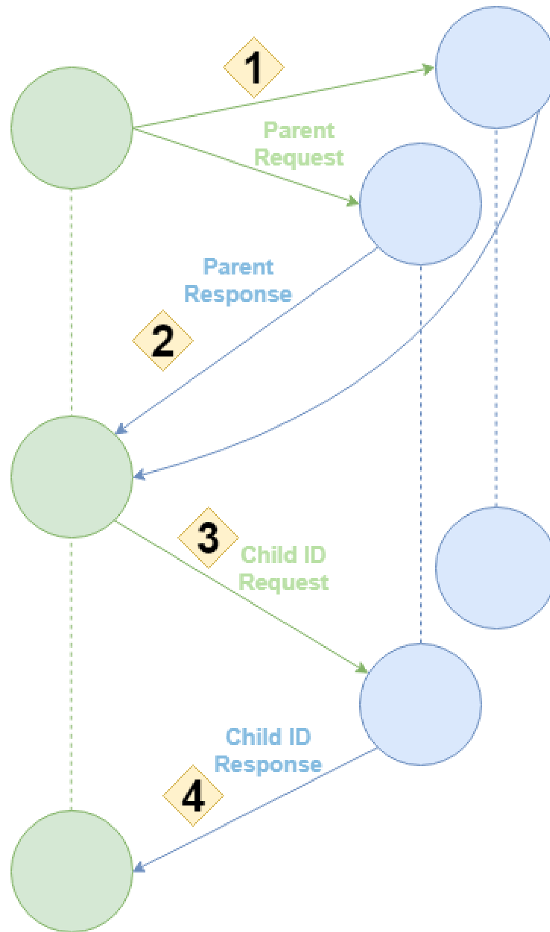


Figure 5.7: Thread network joining.

1. A parent request is sent to all eligible devices in range via a multicast. This includes routers and/or router eligible devices.

2. A parent response is sent to the attaching device via a unicast. The message contents include information about the Thread protocol version in use, a copy of the original request, the address of the device sending the response, information about the leader, and more.

3. The attaching device now sends a child ID request, where the intent is to

establish a more permanent link. Message contents can now also include a description of the child, a keep-alive duration, addresses to register, and more.

4. The child ID response confirms the link. The address of the child is given, timeout duration information, miscellaneous network information, and more.

There are more details of how Thread works. This includes how roles dynamically change within the network (e.g. how routers are selected), how commissioning works (the secure acceptance of new devices), how the topology is structured, and more. However, the explanation given thus far is sufficient for the purposes of this thesis.

5.2 OpenThread

OpenThread is an implementation of Thread. OpenThread is open-source. OpenThread is certified against the Thread 1.1.1 Specification. OpenThread is used by several major actors in the industry, including ARM, Cascoda, Nordic Semiconductor, NXP, Qorvo, Silicon Labs, and Zephyr [24].

The goal of OpenThread is to let developers use an open implementation of Thread on embedded devices. An overview of a general OpenThread setup is seen in Figure 5.8. OpenThread provides software for a border router which bridges the gap between WiFi and Thread networks. The figure indicates two common situations. The NCP (Network Co-Processor) setup indicates that the hardware with WiFi capabilities does not have the radio hardware necessary to communicate with Thread. Therefore, a co-processor with such a radio onboard is used to mitigate this. If the hardware is present, the NCP is not necessary.

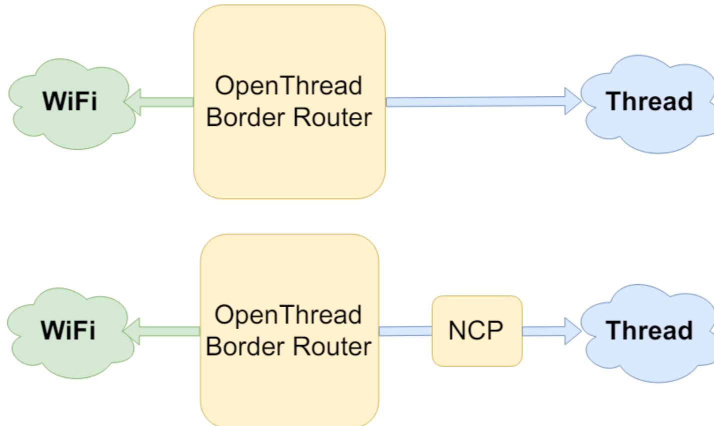


Figure 5.8: OpenThread setup.

5.3 MQTT

MQTT is a messaging transport protocol. It lets clients *subscribe* to topics, and then receives messages if other clients *publishes* to such a topic [26]. Topics are strings, usually delimited by forward slashes. A typical example could be `sensors/kitchen/temperature/1` or `mailbox/backend/update` and so on. A major advantage of using such a protocol is the ease of many-to-one and one-to-many communications. An MQTT *broker* ensures message delivery to all interested parties. This means that if a large number of clients subscribe to some topic, they will all receive messages published to this topic. And a single client subscribed to some topic will get messages from any number of publishers to this topic.

MQTT uses TCP/IP to transport its messages. The overhead of the protocol itself is designed to be small, with IoT in mind.

5.3.1 Quality-of-Service

MQTT has a concept of QoS (quality-of-service). There are three levels of QoS:

- QoS 0/"At most once": This level indicates that messages are delivered at *best effort*. Messages can be lost. This level has the least amount of effort and traffic associated with it.

- QoS 1/"At least once": This level assures that messages are delivered—however more than one copy might arrive at the destination.

- QoS 2/"Exactly once": This level ensures that messages arrive and only once.

5.3.2 MQTT-SN

MQTT for Sensor Networks (MQTT-SN) is a close relative to MQTT which has low-cost, low-power, low-bandwidth wireless devices in mind [34]. Emphasis is placed on the fact that it is intended for wireless environments where bandwidth is low, error rates are higher, and packet sizes are smaller. MQTT-SN was created with the already discussed 802.15.4 standard in mind.

Differences to MQTT

The most relevant differences compared to MQTT are as such:

- UDP is now normally used instead of TCP, for reasons of efficiency and reducing overhead.
- The topic of a message must be a part of every message from end-to-end in MQTT. This can be a string of tens of bytes. If upwards of tens or even hundreds of messages are sent per second this is a large overhead. Therefore, MQTT-SN uses short two byte aliases (a *topic ID*) for topics.
- MQTT clients need to program the address of the MQTT broker into a device's firmware, or somehow communicate it to the device at runtime. With MQTT-SN this is no longer needed—a discovery procedure now lets devices find a MQTT-SN *gateway* dynamically.
- MQTT has a concept of a *keep-alive* duration. If a broker does not receive any communication from a client within such a duration, it is considered disconnected. MQTT-SN has extended this feature to accommodate *sleeping* clients, which embedded battery powered devices often are.

MQTT-SN introduces the concept of an MQTT-SN *gateway*. This acts as an intermediary between the actual MQTT broker. Figure 5.9 shows an overview of a setup of both MQTT and MQTT-SN clients connected to a broker—notice MQTT-SN clients speak to their gateways instead of the broker directly.

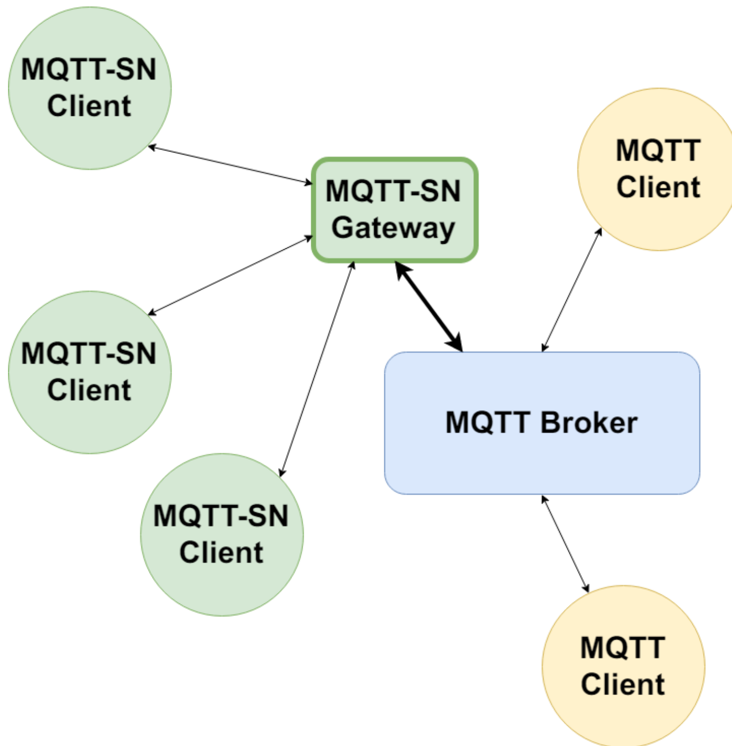


Figure 5.9: MQTT and MQTT-SN overview.

5.4 In Practice: Nordic Semiconductor SoC and Raspberry Pi

This section explains how the use case was done in practice, and what the results were.

Nordic Semiconductor has a new lineup of SoCs which include radios supporting the IEEE 802.15.4 standard—the nRF52840 [30]. This is the newest SoC in the same family as the ones already in use for their Bluetooth technology in the robots for the SLAM project. They are using the nRF52832, or older (some are using the nRF51 family of products).

This new SoC can be used via either developing a custom printed circuit board, or by using Nordic Semiconductor's development kit as seen in Figure 5.10, or in a smaller form factor as seen in Figure 5.11. Note that any SoC which supports a 802.15.4 radio can be used. Nordic Semiconductor was chosen for its accessibility and the familiarity of its software stack based on personal

experience.

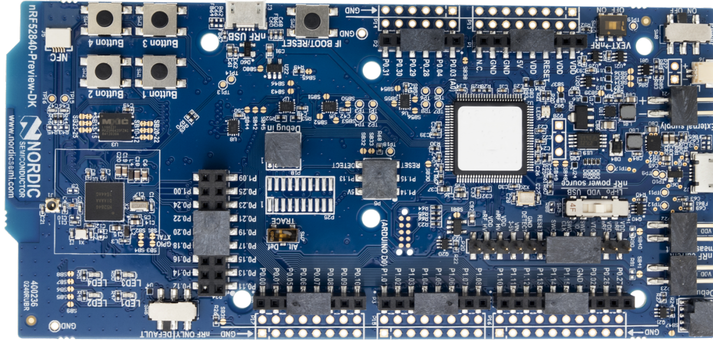


Figure 5.10: Nordic Semiconductor nRF52840 Development Kit.

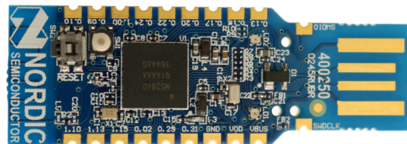


Figure 5.11: Nordic Semiconductor nRF52840 Dongle.

The OpenThread border router software supports any Linux machine. A Raspberry Pi 2 was setup to act as the border router. The Raspberry Pi 2 does not feature a 802.15.4 radio. Therefore, an NCP setup was necessary. Additionally, the plan was to communicate with robots using the MQTT messaging protocol. Nordic Semiconductor provide examples of MQTT-SN client example firmware for their nRF52840 SoC. Therefore, a MQTT-SN gateway was required. Nordic Semiconductor has released an image of a Linux installation targeted to run on a Raspberry Pi. It fulfils our requirements as it comes with the OpenThread border router software pre-installed. Out of the box it has an MQTT-SN gateway which relays messages to a publicly (and freely) available MQTT broker. If an nRF52840 dongle is placed in the Raspberry Pi's USB port, it will act as a NCP for the Thread network (the dongle must be flashed with NCP firmware, also provided by Nordic Semiconductor). The complete ready-to-use setup is shown in Figure 5.12.

Note that the on-board WiFi of the Raspberry Pi can be used for internet

connectivity, but likely requires additional configuration. Thus the steps can be summarised as:

- Flash an nRF52840 dongle with Nordic Semiconductor provided firmware.
- Flash an SD card with the Nordic Semiconductor provided image.
- Optionally change configuration files on the Raspberry Pi to use a custom MQTT broker, or change Thread network configuration options.

When this setup is ready, Thread capable devices can join the network, and MQTT-SN clients can subscribe and publish to topics on demand.

With this setup installed, Nordic Semiconductor's MQTT-SN client examples from their *nRF5 SDK for Thread and Zigbee v3.0.0* [29] were flashed onto a Nordic Semiconductor nRF52840 development kit. This allowed verifying Thread connectivity and MQTT usage. MQTT was verified by both publishing and subscribing to various topics from a desktop computer running an MQTT client, and inspecting this via logging on the development kit. Similarly, the development kit firmware both published to topics and subscribed to topics, which was all successful. Lastly, the Raspberry Pi was instructed to use a locally installed MQTT broker instead of the default publicly available one. This was faster, as the publicly available one receives large amounts of traffic. This concludes the use case of Thread.

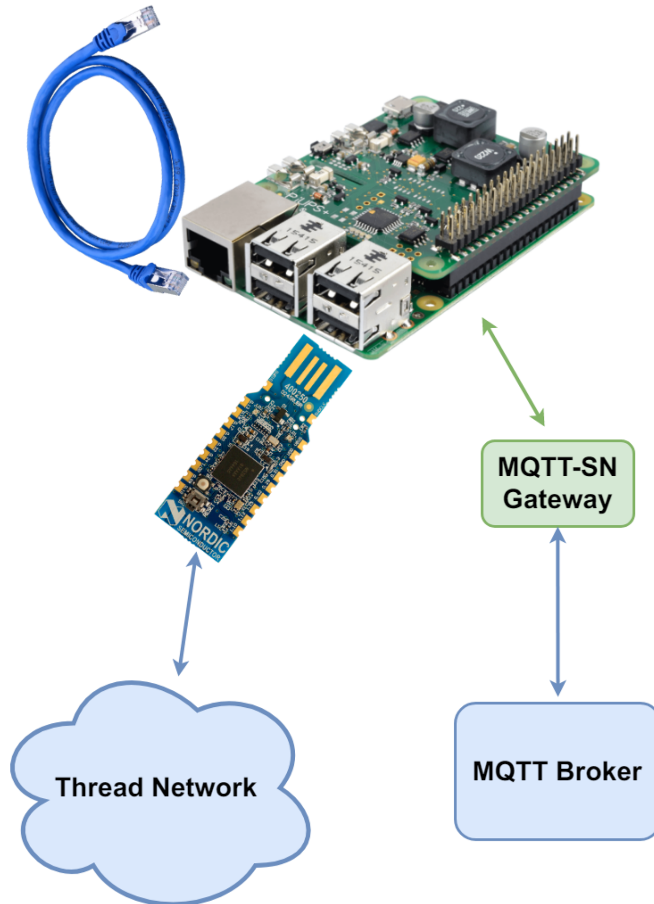


Figure 5.12: OpenThread setup with devices. Note that the desktop running the server application can be used instead of a dedicated Raspberry Pi.

Chapter 6

Design of New Server

Aim of This Chapter

- Provide an outline of the major goals of the new server application.
- Outline some guidelines for software design patterns to follow in the source code.
- Provide a general graphical representation of the intended design.

6.1 Major Goals

Based on the insight of what the Java application offers along with the general context of the project as a whole, several goals can be stated. The major points are:

- Plan some design patterns in advance to ensure code quality in the codebase.
- Improve on the graphical aspect of the SLAM application. A user experience with more options graphically is sought.
- Keep the user interface at a similar quality level as the Java application—clean, tidy, and self-explanatory.
- Provide a new communication stack based on the successful case study seen in Chapter 5.
- Implement a way to automatically find paths given an map with obstructions and robots.

6.2 Design Patterns

Having laid out a few design patterns can help guide the process of creating a clean code-base.

6.2.1 Use Callbacks Generated By Events

It is desired to decouple the various classes and functions of the implementation. This means the various classes are as independent as possible. It is undesirable to have a long chain of classes where one function or result leads to the invocation of other functions which leads to deep nested code and *spaghetti code*. One way to achieve this is to use events. If classes offer events, other classes can listen to events via callback functions. This means that any class can independently generate events when interesting things happen, and other classes can choose to retrieve relevant data from such events or ignore it completely. This means we have achieved a sense of independence in classes, since classes now have no assumptions on how their events are going to be used.

Examples of interesting events can be mouse clicks, robots sending data, button clicks on the user interface, and so on.

6.2.2 Use Inheritance Where Appropriate

The main uses of inheritance is to group objects together by some relationship, and to derive more complex classes while reusing some or all of the super-class functionality, or even customising super-class functionality by overriding functionality [27][37].

During implementation, well-suited uses of inheritance should be looked for. Refactoring of code into general base classes should be done whenever applicable. When this is done, general principles are collected in general interfaces, and specialisations come in the form of small specialised classes. This keeps the sizes of individual classes low. A low line count in a class makes it more maintainable and readable, and keeps the scope of the class focused.

6.2.3 Single Responsibility

The inspection of the Java codebase showed that while many classes were small and had a single responsibility, there were also several which had many responsibilities and as such low cohesion.

Here, a *single responsibility* refers to a clear single intent of the class. For example a parser should only parse, and not interact with a network. Or a network protocol should only deal with its inner workings as a protocol, and not interact with the application layer (as in the OSI model [23])—which the ARQ protocol seen in Chapter 4 is an offender of doing. There were other cases of source files in the Java application which had tens of private variables, flags, and functions in a single class. This is a counterexample of this design pattern.

Given a single responsibility, high cohesion is also sought. This means the member functions within a class should be closely related. As a counter-example, a class named `math` should not have the member functions `add`, `multiply`, and `calculateSalary`. The last member function involves mathematics, but is too unrelated to the other member functions and therefore lowers the cohesion.

6.2.4 Modern C++

As seen in Chapter 2, modern C++ brings many tools to aid code quality and safety. Some key points to keep in mind while implementing the application:

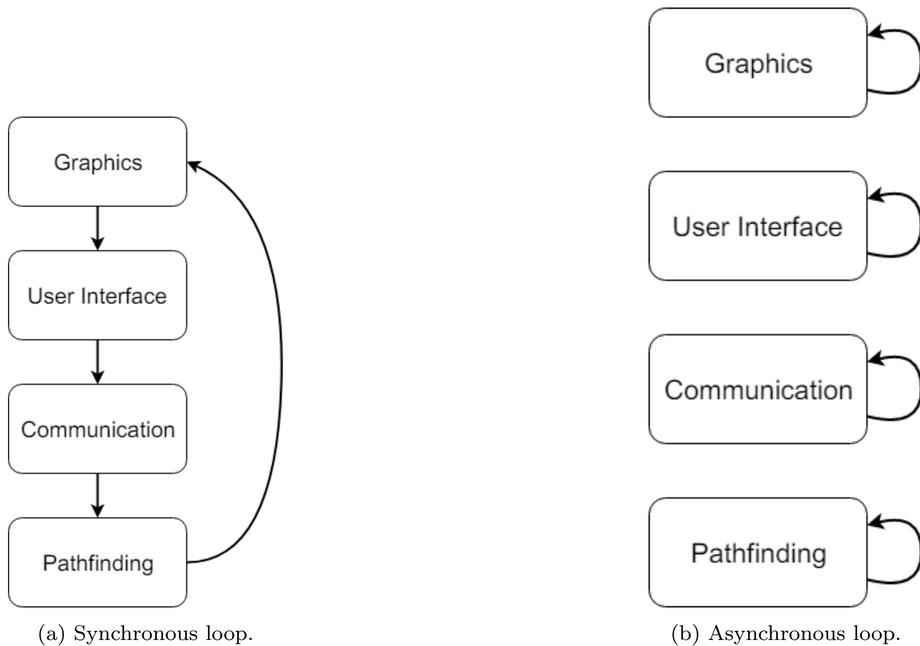
- Wrap resource management in smart pointers [36] to ensure no leaks.
- Use lambdas when appropriate for readability, and to avoid creating unnecessarily large scopes for functions.
- Use the keyword `auto` when appropriate to make more terse and readable code.
- The new template types `std::tuple` and `std::pair` can be valid alternatives to named structures.
- Use class enumerations instead of plain enumerations when possible. This makes enumerations strongly typed.
- `std::optional` is a new feature which allows return optional values from functions and is immensely useful.
- `std::any` can be used to allow having arbitrary parameter types. `std::variant` can be used if the possible set of types is known.
- Structured bindings (for example unpacking a right-hand-side pair into two left-hand-side variables) can improve readability.
- Using nested namespaces in a single declaration removes clutter.

6.2.5 Asynchronous Single-Threaded Until Avoidable

Experts argue that multi-threaded code should be avoided until it is *actually necessary* [36]. However, for the sake of structuring code in a readable way, small cohesive tasks should be created and should run as independently as possible. This is achievable without using multiple threads. Consider Figure 6.1a and Figure 6.1b. The synchronous loop is easy to read and understand and translates well to code in simple cases. However, it has a few issues. If tasks get complex and long, readability quickly withers. Secondly, if any task has a blocking call, this would interrupt the whole user experience. As an example this could arise due to wanting to wait for an event on a timeout. Therefore, the asynchronous approach should be used. In an asynchronous context, a task wanting to wait for a timeout can *yield* and let other tasks run in the meantime. This approach both avoids the issue of waiting on events, and avoids the complexities of mutual exclusion on shared data. The downside of the asynchronous approach is that there are now several running contexts active at more or less the same time. This can make it harder to get an overview of the application's execution flow. Note that the approach described here assumes the asynchronous tasks run on the same thread, which simplifies design but is not strictly necessary.

If at any point calculations which take several tens of milliseconds (or so) to perform are necessary, it is imperative to move towards a multi-threaded approach. As such, the asynchronous approach should be implemented in such a way that using multiple threads does not need any major rework. If a multi-threaded approach is migrated towards in the future, there are several things to consider. All shared data between threads must be validated to accessed properly. For example, a `std::lock_guard` can be used to safely and easily access data in a multi-threaded context. Also, any access to shared data of a temporal nature should have timeouts associated with it to mitigate live-locks.

As a last note, it is worth mentioning that the term *asynchronous* is used loosely in this context— since tasks yielding execution on a single thread could be argued to be synchronous. However, from the perspective of the programmer the end result is largely the same (until a performance bound is reached).



(a) Synchronous loop.

(b) Asynchronous loop.

Figure 6.1: Contrasting loop approaches.

6.3 Main Ties Functionality Together

The role of `main.cpp` should be clearly defined. Using the notion of events, callbacks, and asynchronous tasks, the role of `main.cpp` will be to tie functionality together. That is, `main.cpp` will contain all the asynchronous tasks. Also, most events should be handled here. `main.cpp` should orchestrate responses to events and initiate further action, while classes provide the functionality which is possible to initiate. The rationale of this is to have a defined place to put the application logic and program flow, and not disperse it over many classes. This was arguably a weak point of the Java application, where the equivalent to `main.cpp` started various threads in various classes—thus making the developer need to traverse many classes before getting an overview of the application flow.

One pitfall of having several tasks running in `main.cpp` is the danger of global variables in order to share data between tasks, this should be avoided [37]. One way to avoid this is to use a form of *channels*, which can pass data to tasks, and retrieve data from tasks. This is also thread safe, which sets the stage for later optimisation via multi-threading. Thus a way to visualise the design

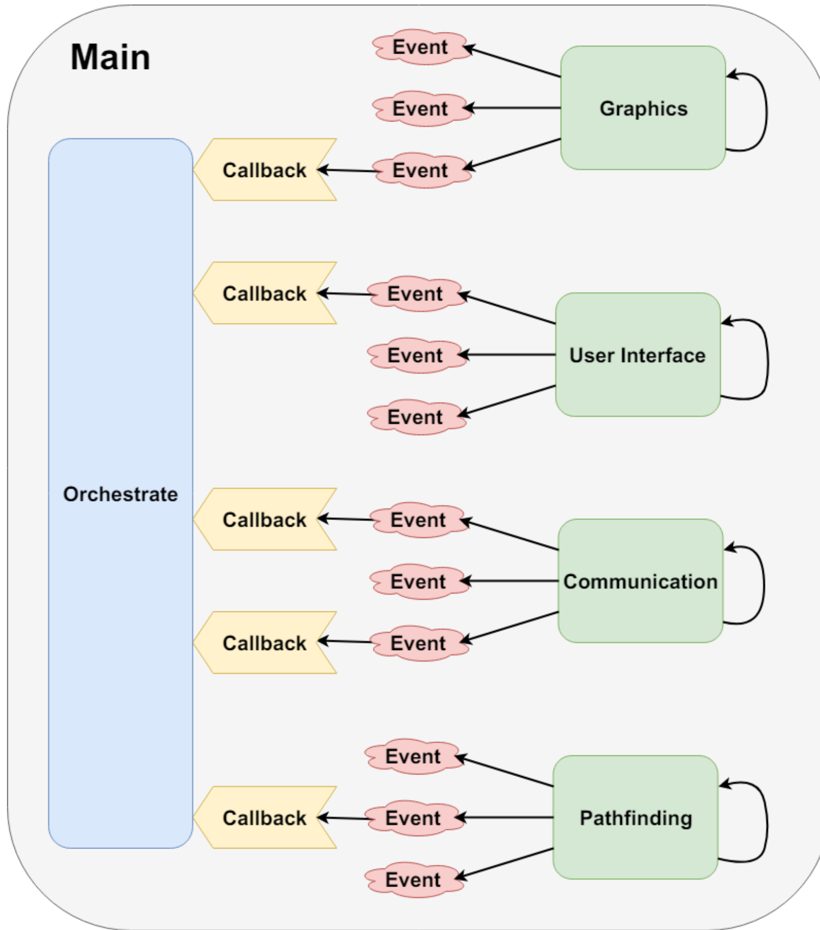


Figure 6.2: Design overview.

is given in Figure 6.2. The figure shows the nature of using classes as services which provide events. Only events which are interesting are registered with callbacks. Also, the various example tasks encapsulate the generation of the events themselves, hiding unnecessary complexity from users of the code-base.

Chapter 7

Results

Aim of This Chapter

- Introduce the implementation via a general introductory overview.
- Showcase the implementation with regards to each major goal given in the design—Chapter 6.
- Provide an insight into the entire implemented codebase, as was done for the Java application in Section 4.1.
- Where needed, provide additional information and guidance with regards to the implementation—this will help future developers understand intent.

7.1 Overview

The codebase and various features will be explained in some depth. A general overview of the C++ application is presented here to set the context of what will be explained in more depth in sections to come.

A quick look into the graphical presentation is shown in Figure 7.1. It shows a great deal of the features implemented in the new application:

- A control panel with various tabs. Currently the *Robots* tab is selected, and a single robot sub-tab is shown.
- A simulated robot has added some data—256 points of Cartesian coordinates—and these are displayed as circles in the same colour as the robot with a given configurable radius.

- The application grid is shown as a collection of green circular nodes, these are traversable by pathfinding. If one or several measurements fall within such a node it turns red—it is now obstructed and will not be considered for pathfinding.

- A node has been set as target for the robot, and pathfinding has been turned on. As such, a path connected by straight lines is shown for the robot. The lines are connected by circles—these are the waypoints issued by the application to the robots.

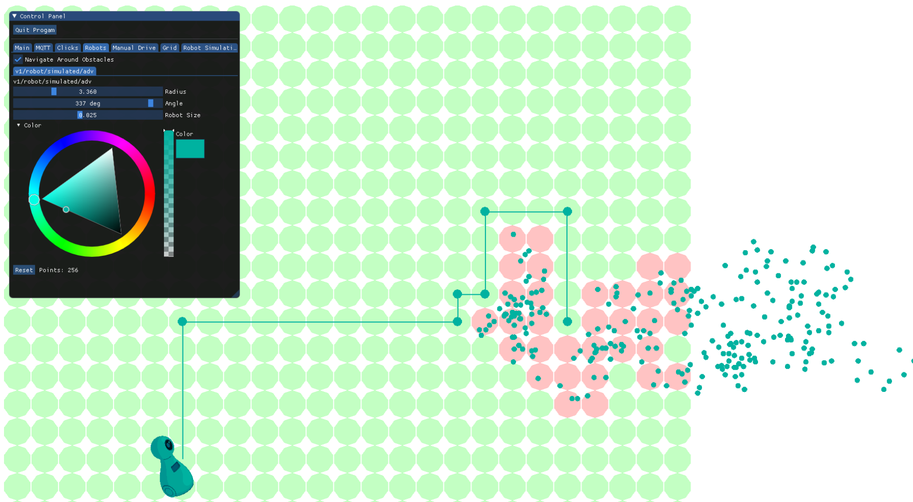
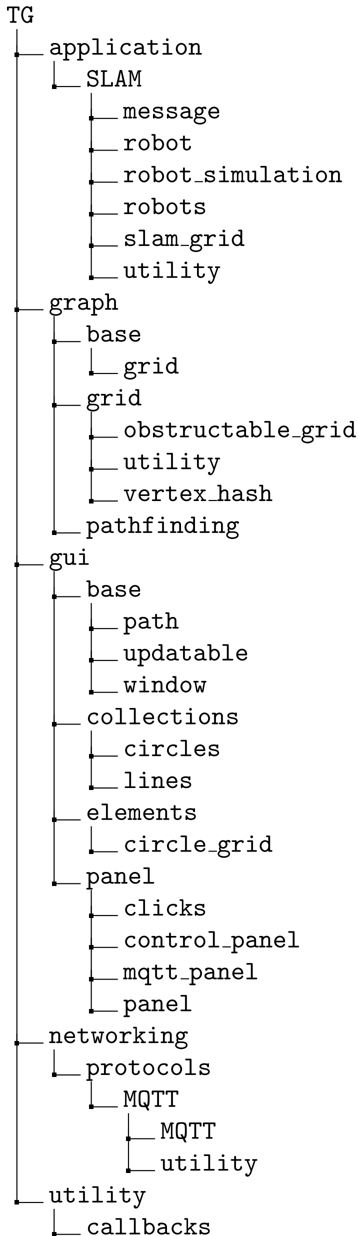


Figure 7.1: A general overview.

7.2 Codebase Structure

The application is structured via namespaces. These are as follows:



7.2.1 Namespace Explanations

The intention of the major namespace groups are explain here.

Namespace: `TG::application`

For uses directly concerned with the actual application. Uses many of the implemented classes in conjunction.

Namespace: `TG::graph`

Deals with *graphs*, in the sense of nodes, vertices, and algorithms using graph constructs.

Namespace: `TG::gui`

Everything related to graphics is sorted here, including drawing of all primitives, and all handling of panels (with buttons, sliders, and so on).

Namespace: `TG::networking`

All protocols and their usages are sorted under this namespace.

Namespace: `TG::utility`

Utility functions and classes of higher abstractions—not sorting under any other namespace—sort here.

7.3 Codebase Walk-through

This section is thorough, which is provided as a service to future developers of the application. A second purpose is to provide background to some of the rationale for design choices explained in following sections.

7.3.1 Namespace: `TG::gui`

`TG::gui::base::window`

This file provides the actual window in which all the graphics is shown, including primitives and panels. It allows a user to add panels, drawables, and updatables to the window via polymorphism. If any of these are added via a single function call, the given element will be shown as a part of the window.

Another part of the window is to generate events. Many events are available, including an event if any keyboard key is pressed, a mouse button is pressed or released, the window is resized or moved or closed, and more.

`TG::gui::base::updatable`

This file is a header-only class. It is a simple abstract class with a single virtual function that must be defined in child classes: `update`. This function is given a time delta since the last time it was called. By inheriting from this interface, any class can add itself as an *updatable* via the `TG::gui::base::window` class, and is then automatically updated each time the window refreshes. This allows doing time-dependent work such as animations.

`TG::gui::base::path`

This class is graphically drawable. The path is composed of lines (via `TG::gui::collections::lines`) which connects waypoints which are circles (via `TG::gui::collections::circles`). Its intention is to graphically show a *path*.

The main use-case intended is that a given robot can such a path associated with it—showing the path of waypoints laid out for the robot.

`TG::gui::collections::circles`

This class is a helper class wrapping a collection of the basic drawable circle. It has some functionality for acting on the whole collection, such as setting colours and alpha values, which apply to the whole collection. Attributes such as radius and point resolution of circles are settable.

`TG::gui::collections::lines`

Similar to `TG::gui::collections::circles`, this wraps a collection of lines. Attributes such as line thickness and colours are settable.

`TG::gui::elements::circle_grid`

This class represents a drawable grid of circles. It allows creating a grid with the amount of rows and columns wanted, and a given *separation*. Separation indicates the space between rows and columns, essentially being the diameter of a given circle in the grid. The class can also have customisable colouring and alpha values.

`TG::gui::panel::panel`

This class acts as a parent class for all panels, allowing them to be grouped together via polymorphism. If the panel class is constructed with a title (that is, a name as a string) it will become a stand-alone panel with its own window. If no title is given, it is expected to be embedded within another panel. Therefore, the user of panels should have one top-level panel with a title, and can optionally embed sub-panels within these. After a panel is constructed, the panel contents can be added as a function with `ImGui` calls within (see Section 2.4.3 or Chapter 9).

During runtime, the panels are shown via calling the member function `TG::gui::panel::panel::show`. This is automatically handled if the panel is given to `TG::gui::base::window`, which is the intention in this application.

`TG::gui::panel::clicks`

This class is a panel used for control of what the mouse clicks do during the runtime of the application. It can generate two events. The events are generated when the left-click or right-click mode changes. The data passed to a registered callback function is one of:

- `nothing`: Meaning the respective click should do nothing.
- `set_target`: Meaning the respective click should set a target node for a robot.
- `obstruct`: Meaning the respective click should obstruct a node in the grid.

`TG::gui::panel::mqtt_panel`

This class is a panel used as a helper for the MQTT communication in the application. It allows manually publishing messages to arbitrary topics, manually subscribing to topics, and monitoring the flow of in- and outbound messages.

`TG::gui::panel::control_panel`

This panel holds other panels. The main feature of this class is that when a panel is added to it, the child panel is automatically placed within a new tab in the control panel.

7.3.2 Namespace: `TG::graph`

`TG::graph::base::grid`

This is the base grid abstraction in the context of graphs. It has a simple interface. It can be constructed from a given number of rows and columns. It can also be reset to other sizes during runtime.

`TG::graph::grid::obstructable_grid`

This grid inherits from the base grid. The specialisation it offers is to allow marking nodes as *obstructed*. Querying whether a node is obstructed is possible, as well as unobstructing nodes.

`TG::graph::grid::utility`

This namespace has free functions relating to grid graphs. It contains a utility function for converting integer row and columns accesses into vertex descriptors as used internally by the `Boost.graph` library.

`TG::graph::grid::vertex_hash`

`Boost.graph` requires a hashing function for placing vertices into an unordered map, which is done as part of the pathfinding algorithm. This namespaces provides this requirement.

`TG::graph::pathfinding`

This namespace is used by the file `grid_path_solver.cpp`. It has two free functions.

1. `solve(...)` takes a grid graph, and two points in a grid. It returns a vector of points which represents the points in the solved path between the two points given. The returned vector is optional—if no path is possible no vector is given.

2. `reduce(...)` takes a vector of points, and reduces it to simple lines. This simplification means that any straight line will only have two points after the reduction. This is used because the solved path might return a solution containing any number of points on a straight line—this can be inefficient when issuing waypoints to robots.

7.3.3 Namespace: `TG::networking`

`TG::networking::protocols::MQTT::MQTT`

This class implements all the MQTT functionality. When searching for MQTT C++ libraries, it was found that most clients use C libraries. Therefore, this class wraps an asynchronous MQTT C library. It exposes a number of events which can be used with callbacks: `connect_response`, `disconnect_response`, `subscribe_response`, `publish_response`, `connection_lost`, `message_arrived`, and `delivery_complete`.

The member functions of the C++ class have been reduced to the set of most important functions such as connecting, disconnecting, setting the quality of service, subscribing, and publishing. An important note of this class is that the underlying C library uses threads to achieve asynchronicity. This means that all callbacks are given in a multi-threaded context—which means special care has to be taken when using these events.

`TG::networking::protocols::MQTT::utility`

This MQTT utility namespace introduces a structure representing an MQTT topic. This application uses topics which are composed of a version, a sender, an ID, and a command. This structure is arbitrary and need not be used in the future if another scheme is desired. A function `parse_topic(...)` takes a string and parses it into the structure defined above. The output stream operator `<<` is also overloaded to accept such a topic structure and print it out in a nice fashion.

7.3.4 Namespace: `TG::utility`

`TG::utility::callbacks`

This class is inherited by other classes in order to enable the event-callback pattern. It is explained in detail in Section 7.4.

7.3.5 Namespace: `TG::application`

`TG::application::SLAM::message`

This class encapsulates the messages sent from robots. Messages include a position (of the robot), and an optional arbitrary number of obstacles. Positions

and obstacles are given by their *(x, y)* coordinate in the form of two `int16_t`s.

`TG::application::SLAM::robot`

This class represents the abstraction of robots. They are drawable and intended to be shown graphically. Robots can be added obstacles to (as a response to a robot having sent a message with an obstacle to the server), and a path can be set for showing the intended waypoints to follow. Other functionality is minor and includes colouring and other minor features. All the added obstacles can be cleared during runtime (essentially resetting the robot history), which will produce an event `POINTS_CLEARED` which can be used for callbacks.

`TG::application::SLAM::robot_simulation`

This class simulates a robot. It does this by publishing messages (in the format of `TG::application::SLAM::message`) exactly the same as a physical robot would. The simulation works by simply randomising the robot position and the measured obstacles in small steps. This simulation has been valuable in testing the implementation of the server application, as the server application sees no difference between a simulation and a real physical robot—this ensures testing is as close to reality as possible.

`TG::application::SLAM::robots`

A robot is defined by the topic to which it publishes messages to. Messages are not parsed directly in `main.cpp`. Messages are fed to this robots class, where they are parsed. A new robot might then be found at any time, and this class adds robots dynamically.

Functionality offered includes getting all the obstacles found by all robots, getting all robot IDs, getting the target point of a given robot, and setting the path of a robot.

There are some events put out by this class. These include `ROBOT_MOVED`, `ROBOT_FOUND_OBSTACLE`, and `ROBOT_CLEARED_POINTS`. These can be assigned to callbacks as normal.

`TG::application::SLAM::slam_grid`

This is the high-level grid used in the application. It combines
`TG::graph::grid::obstructable_grid` and

`TG::gui::elements::circle_grid`. The result is a grid which can be shown and represented graphically, and also be obstructed and navigated via pathfinding.

`TG::application::SLAM::utility`

This namespace has a collection of various free utility functions. Functions include converting to and from the global coordinate system and the rows and columns found in the application grid drawn in the application, getting random numbers and random colours, and converting from bytes to `int16_t`.

7.4 Callbacks and Events

The pattern of using callbacks based on events was readily used, and was one of the design goals of the application. It is in use in the following classes:

- `TG::application::SLAM::robot`
- `TG::application::SLAM::robots`
- `TG::application::SLAM::slam_grid`
- `TG::gui::panel::clicks`
- `TG::gui::panel::control_panel`
- `TG::gui::panel::mqtt_panel`
- `TG::networking::protocols::MQTT::MQTT`

Thus the pattern is a large component of the application.

7.4.1 Enabling Class

`TG::utility::callbacks` is the enabling class which provides a simply shortcut to allow any class to use the pattern of event-callback. The enabling class is very short; it is shown in its entirety (excluding preprocessor directives) in Listing 7.1.

Listing 7.1: The `callbacks` class.

```
1 namespace TG::utility
2 {
3 template<typename Key, typename FunParam = std::any>
4 class callbacks
```

```
5 {
6   using Fun = std::function<void(FunParam)>;
7
8 public:
9   void enable_callback(Key key, Fun fun) { cbs_.insert_or_assign(↩
        key, fun); };
10
11  void call_callback(Key key, FunParam param) {
12    if (auto cb = cbs_.find(key); cb != cbs_.end())
13      cb->second(param);
14  };
15
16  // Class should be inherited, not instantiated on its own.
17  virtual ~callbacks() = 0 {};
18
19 private:
20  std::map<Key, Fun> cbs_;
21 };
22
23 }
```

An explanation follows. For usage see Chapter 9. The class uses templates. Thus when inheriting the callbacks class, the type of the template parameter `Key` has to be specified. This parameter is intended to be e.g. an enumeration class which specifies the type of events available to enable callbacks for. An example is how `TG::application::SLAM::robots` inherits the callbacks class. It specifies `robots_events` as the type, which is an enumeration class consisting of the events `ROBOT_MOVED`, `ROBOT_FOUND_OBSTACLE`, and `ROBOT_CLEARED_POINTS`.

The second template parameter allows the user to specify the type which is passed to the assigned callback function. This is by default `std::any`, which allows the class producing the events to pass anything to the user callback.

A user of `TG::application::SLAM::robots` can then call `enable_callback` for any of the three available events, and pass the function to be called on those events. This is done for all the events provided by `TG::application::SLAM::robots` (and many of the other event-based classes) in `main.cpp`.

7.5 Various Design Goals

This section comments on the status of various design goals stated in Section 6.1.

7.5.1 Inheritance

Inheritance has been used to a large degree, and has been very helpful. At the most basic level, the library used for drawing graphics—`SFML`—provides an abstract class `sf::Drawable` which several of the classes in this application has inherited from. This allows user-classes to be drawn to the screen.

Also, all classes which represent user interface panels, all classes which use

callbacks, and the application grid which inherits the two other types of grids all use inheritance.

7.5.2 Single Responsibility

The single responsibility adage has been followed as closely as possible. By using namespaces with relatively deep nesting, a clear picture of each source file's role has been made clear. This has helped keep the single responsibility goal in mind during development. However, this is a hard goal to maintain. By identifying and reflecting on which classes have followed this goal well and which have not, target classes for later refactoring are found.

Examples of classes which have a single responsibility and do nothing else are the GUI collection classes, the panels, the MQTT wrapper, the abstract callbacks template class, and more. An example of a class which should perhaps be refactored is the robots class. It consumes messages arriving via MQTT, adds robots, is drawable, can set robot paths, has a boolean flag which indicates if robots should navigate towards paths or not, and some other functions. The class is not huge (it only has two private member variables), but could possibly be cleaned up as it is not as cohesive as it could be.

7.5.3 Modern Codebase

The codebase is modern—it can not be compiled with anything less than C++17 support. It has used several modern features:

- Smart pointers are used exclusively; no naked calls to `new` or `delete`.
- The keyword `auto` is used extensively.
- Structured bindings are used extensively.
- Lambdas are used extensively.

`main.cpp` uses lambdas very frequently in order to setup callbacks, fibers, and various small anonymous functions. This language feature has been a major contributor to the readability of the implementation.

The oldest part of the codebase is due to the dependency of `paho-mqtt`—the MQTT C library. This part of the code has been encapsulated as much as possible by wrapping it in a C++ class such that the application developer does not need to interact with the legacy C code at all.

7.6 Asynchronous And Single-Threaded

The application has remained single-threaded, with the exception of the MQTT class which has a multi-threaded MQTT C library at its base.

The application uses `Boost.fiber` to create small tasks which explicitly yield their execution to other tasks when appropriate. Five tasks run in `main.cpp`:

1. GUI task. This task simply updates the graphical user interface, updating animations, elements, panels, and also triggers any events related to UI.

2. MQTT task. Connects to the MQTT broker, and subscribes to a pre-defined wildcard topic `v1/robots/#`. This means that anything published to a topic starting with `v1/robots/` is listened to by default. Messages arrive by callback. This task also has two channels associated with it. One channel has messages which are to be published, the other channel is topics that the client should subscribe to.

3. Robots inbox task. This task checks if a message has arrived on its message channel. If so, feed it to the handler of robots—which is the only consumer of messages in the application.

4. Robots outbox task. This task publishes messages to robots known to the system once a second. The message published is simply where the system wants the robots to go next. If no such point exists, the task sleeps another second.

5. Robot simulation task. Simply runs the robot simulation once, then yields. Running it once means the simulated robot is updated by moving a small bit, and adding a new obstacle.

7.7 Graphical Implementation

This section shows the various implemented graphical aspects of the new application.

Figure 7.2 shows the application after starting it and folding the control panel. The application grid is shown in green. If robots publish measurements within this grid, the corresponding node will get obstructed. Then, pathfinding will navigate around this node.

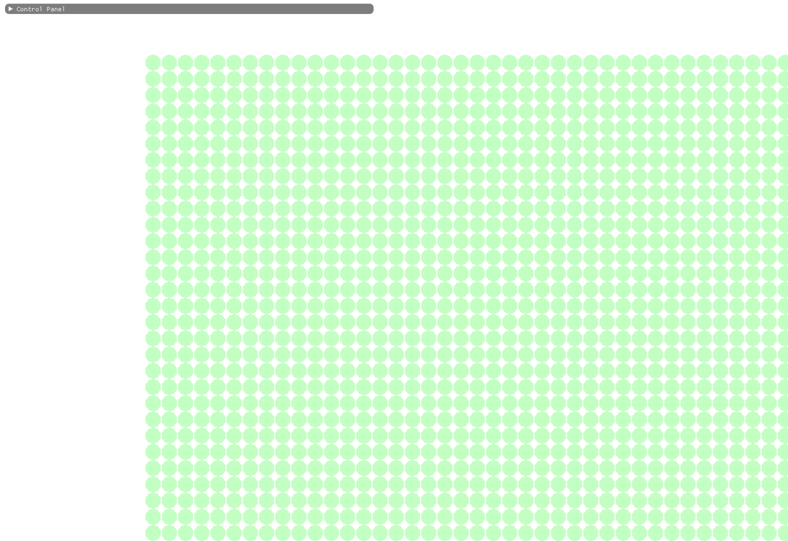


Figure 7.2: Overview of application after start.

Figure 7.3 shows the application after a robot has reported some measurements via MQTT. The points reported by the robot are added as small dots in the same colour as the robot itself. The points—if not outside the grid—will obstruct any node that correspond to the point. These nodes are then shown as red.

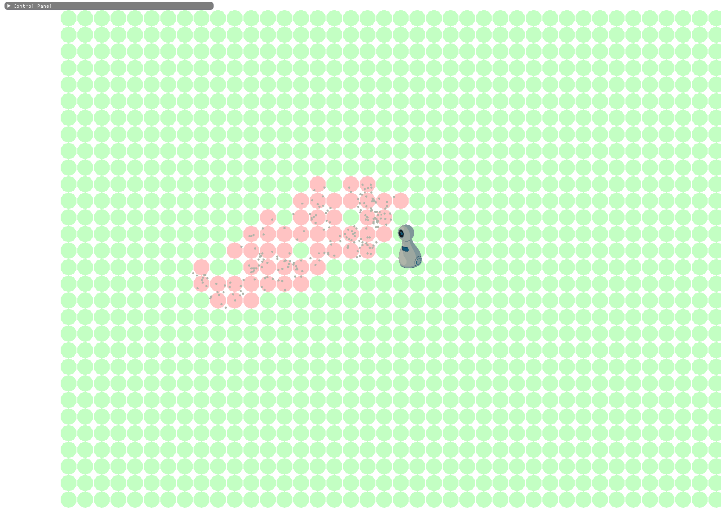


Figure 7.3: Application after robot has reported measurements.

Figure 7.4 shows the result of setting a node as the target node for pathfinding. The figure shows the obstructed nodes are avoided.

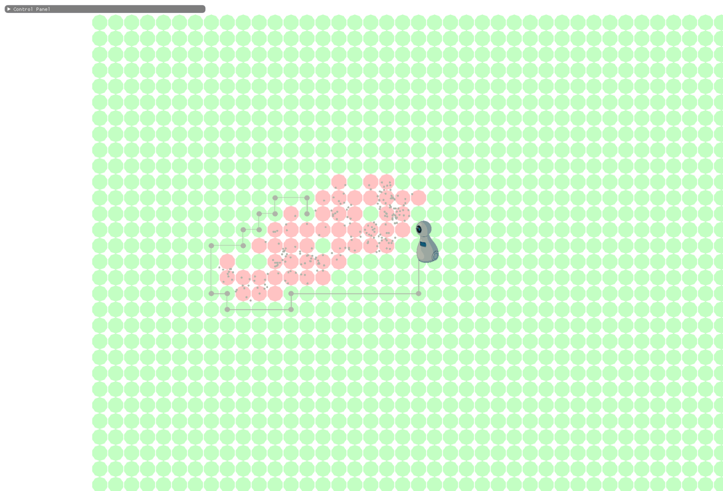


Figure 7.4: A path of waypoints for a robot shown.

Figure 7.5 shows the same target node but without pathfinding turned on. This can be useful for testing purposes. The obstructions in the grid are now ignored, and the robot receives a waypoint directly towards the intended target.

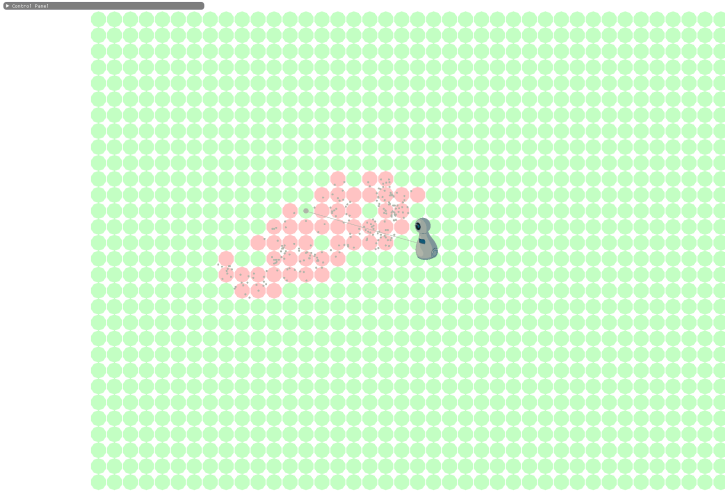


Figure 7.5: A straight manual path is shown.

Customisation of robot appearance is possible. Figure 7.6 shows the same robot but with its size increased, the measurement points with increased radius, and the colour set to a semi-transparent pink. All these options are available in real-time while the application is running, and specified individually per robot.

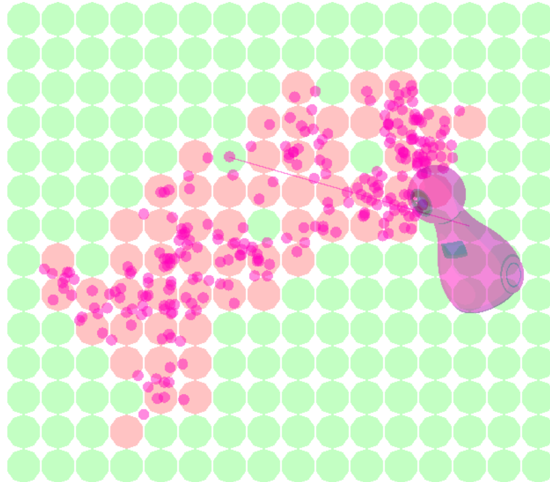


Figure 7.6: Robot with changed appearance.

The grid can also be changed dynamically in various ways. Figure 7.7 shows the same robot and the same data, but the grid has been changed to have less

separation (and more rows and columns). The grid colour alpha value has also been increased in order to decrease the transparency. The robot colour was changed in order to be more visible.

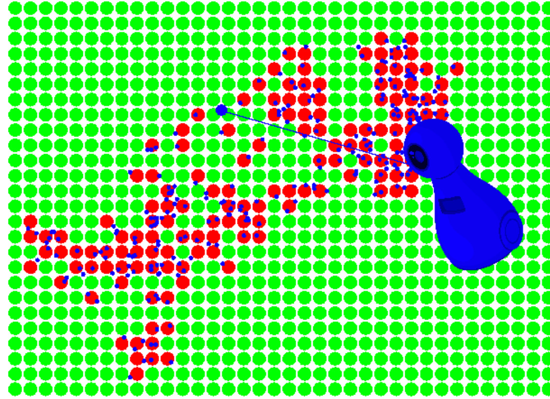


Figure 7.7: Grid with changed appearance.

Figure 7.8 shows the effect of turning the grid alpha to zero. This can be useful in order to get the clearest view of only the robot measurements and not the underlying grid.

Figure 7.8: Grid with full transparency.

Lastly, Figure 7.9 shows the vertical axis flipped, the entire view rotated, the application a bit zoomed, and a large grid separation. This shows that the drawing of graphics is quite controllable in the new application.

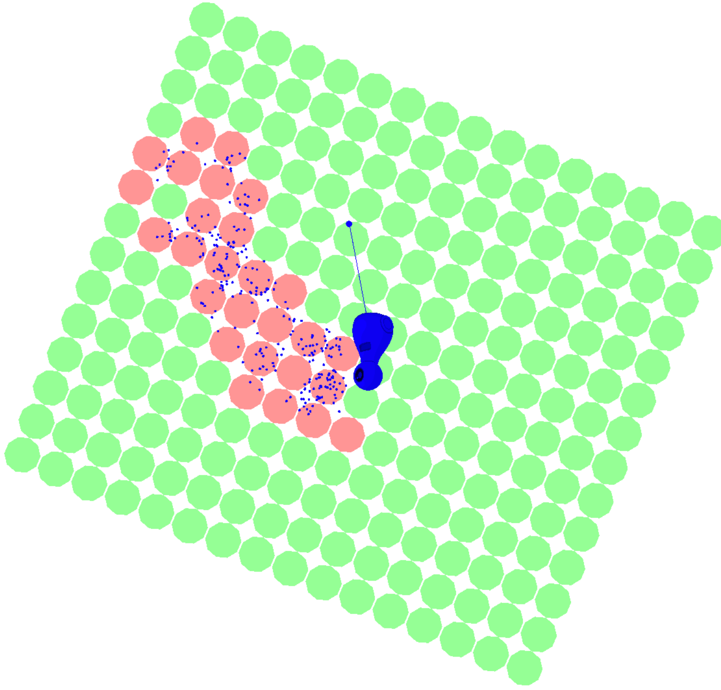


Figure 7.9: The application view can be controlled freely.

7.8 User Interface

Figure 7.10 shows the user interface via use of the control panel. The control panel has seven tabs available for controlling various features of the application. These will now be shown.

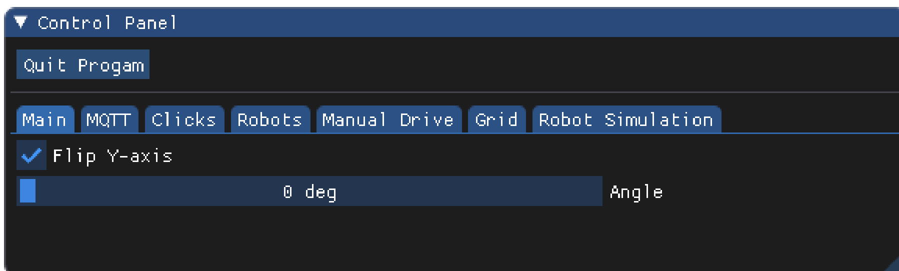


Figure 7.10: User interface overview.

Figure 7.11 shows the MQTT overview, showing four new tabs.

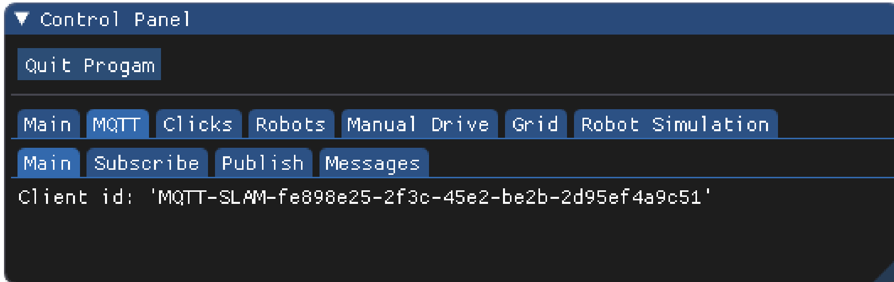


Figure 7.11: MQTT panel overview.

Figure 7.12 shows the MQTT subscribe tab. As shown, `v1/robot/#` is subscribed to by default. The topic `test` was subscribed to during runtime for demonstration purposes. As more topics are subscribed to (if needed), the list will dynamically grow.

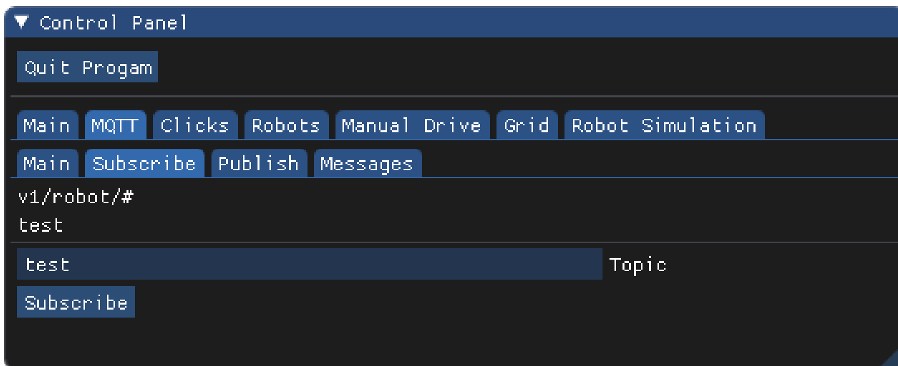


Figure 7.12: MQTT subscribe tab.

Figure 7.13 shows the MQTT publish tab. A decision was made to not make this tab a general MQTT client, as this can easily be achieved by any MQTT client for desktop PCs, smart-phones, and similar. Rather, this tab lets a user publish a valid SLAM message to an arbitrary topic, for purposes of testing.

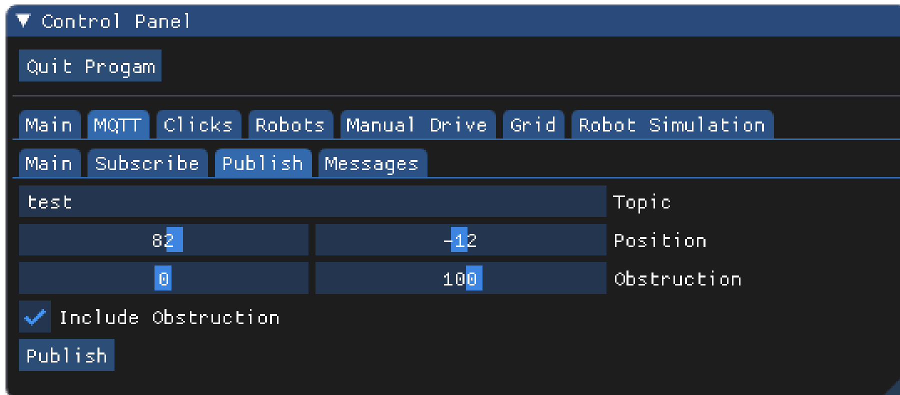


Figure 7.13: MQTT publish tab.

Figure 7.14 shows the MQTT messages tab. The figures have demonstrated that we are subscribed to the topic `test`, and for demonstration the message shown in Figure 7.13 was published via the interface. Since we were subscribed to the same topic we published to, the message arrives in both the *Incoming* and *Outgoing* interface windows. Two other messages were published to topics not subscribed to—these only show up in the *Outgoing* section. Another useful feature of this tab is the option to display the incoming messages as *Raw data*. This is the default. This provides the output seen in the figure which is a byte for byte interpretation of the data displayed as hexadecimal numbers. This was shown to be very useful when interfacing with robots, as a debugging tool.

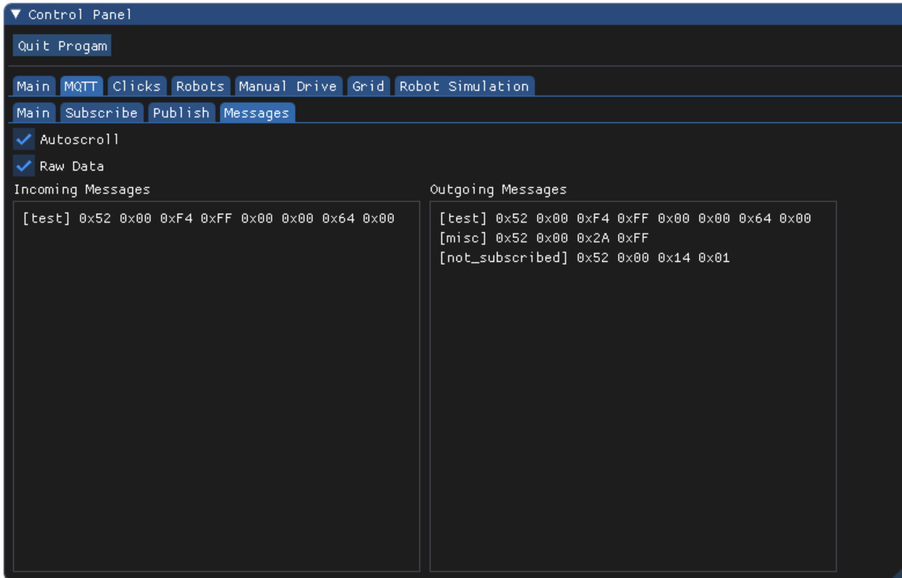


Figure 7.14: MQTT publish tab.

Figure 7.15 shows the tab providing control over what mouse-button clicks do. Currently, it allows choosing to set a button for making robots go towards a target (in the grid), or manually obstructing nodes for testing purposes. Note that the application has hard-coded the functionality of the mouse wheel. Pressing the mouse wheel and moving the mouse moves the view around. Scrolling the mouse wheel zooms the view.

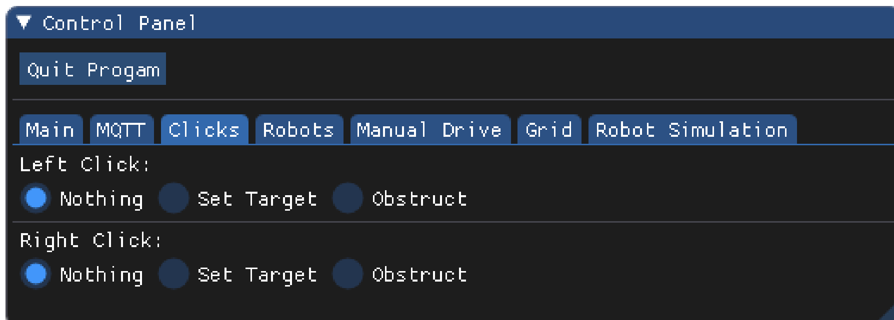


Figure 7.15: Clicks tab.

Figure 7.16 shows the robots tab. It has a global checkbox to select whether robots are to move directly towards a point (the default option), or to move via

pathfinding (that is, navigate around obstructed nodes).

A single tab named `test` is displayed, which is the topic previously published to, which was also subscribed to. As such, it was interpreted as a robot which presented a measurement, and is thus added to the grid as shown. Each robot is added dynamically into a tab side-by-side. The *Radius* slider controls the visual representation of the points measured by the robot. They are by default quite small, and can be increased here for higher visual fidelity. The *Angle* slider refers to the angle of the robot sprite. In the current application, this slider automatically adjusts when a robot adds a measurement, such that the robot looks towards the new measurement. The *Robot Size* slider adjusts the visual size of the displayed robot sprite. The colour wheel adjusts the colour of the robot and all its measurement. This is especially useful when several robots are adding measurements at the same time, as contrasting colours can be chosen. Another approach is to make a robot partly or completely transparent to reduce visual noise when many robots are plotting simultaneously. Lastly, a *Reset* button is available for removing all measurements added by a robot thus far. The application grid is readjusted to unobstruct all affected points.

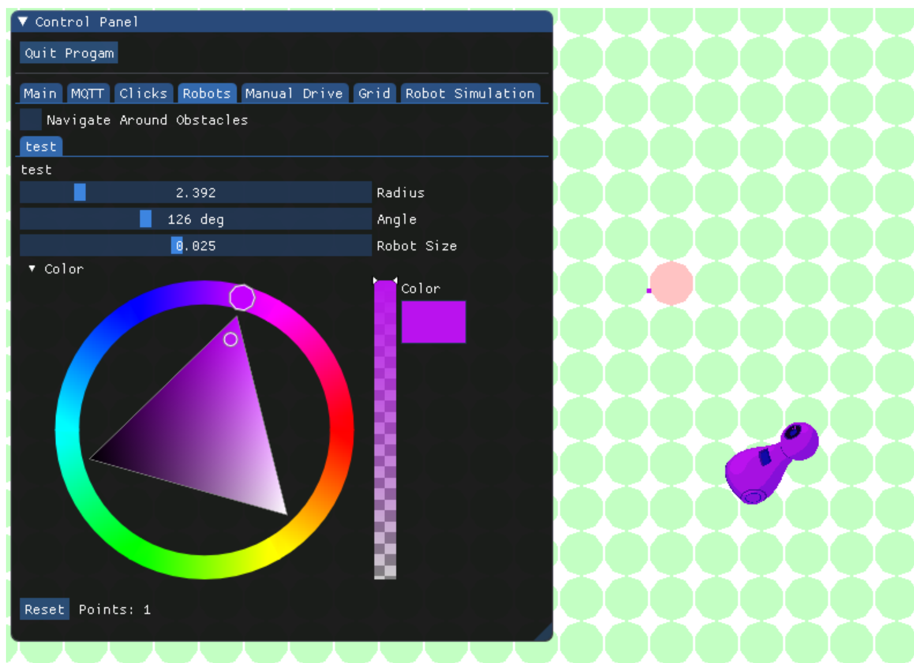


Figure 7.16: Robots tab.

Figure 7.17 shows the manual drive tab. This was added by request of one of the robot developers. A common task for the robots is to drive a simple square path in order to test the calibration of on-board sensors (gyroscopes, accelerometers, and similar). Therefore such a panel was created. A checkbox enabling the feature can be checked when desired. This will overwrite the manual targeting and pathfinding. The next checkbox makes it such that the path of the square has a corner at the coordinate *(0, 0)*. If unchecked, the square's centre point will be *(0, 0)* instead. The size of the square path can be adjusted at will. Clicking one of the corner buttons will apply the point specified as the next waypoint for robots. Lastly, *Manual Input* allows the user to specify precisely a given coordinate to set as waypoint.

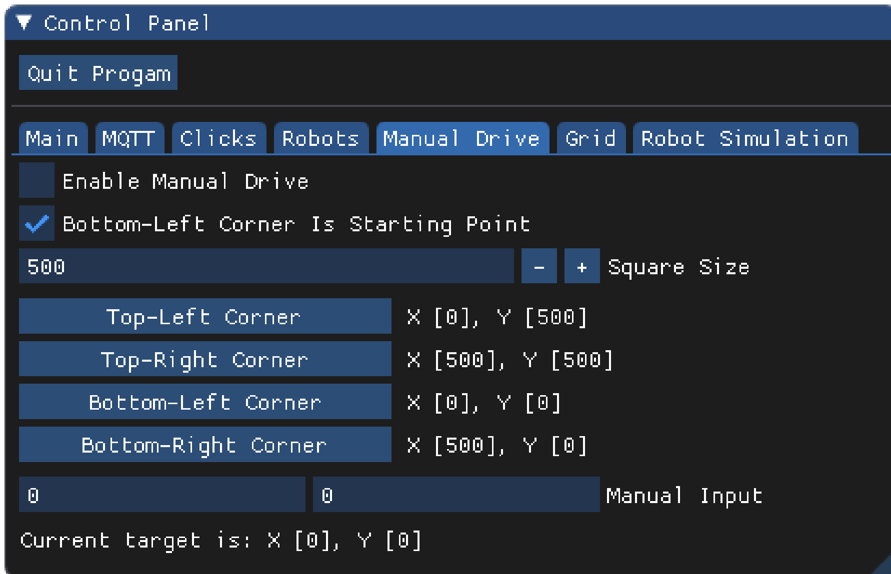


Figure 7.17: Manual drive tab.

Figure 7.18 shows the grid panel. It simply allows customisation of the underlying application grid during runtime. Note that this is less trivial than it appears. If the number of rows, columns, or the separation changes, the underlying abstract obstructable grid might be invalid. The simplest solution to this was to destroy and recreate the whole grid and recalculate all obstructions if any of these parameters change. When a large number of measurements exist in the application, dragging these sliders can be noticeably *laggy*, however this is only a minor nuisance and does not affect the application in any meaningful

way, as the sliders should be tuned once and do not make sense to continuously move around during the application's lifetime. Also, after the sliders have been set, no performance impact occurs. Naturally, changing the alpha of the grid does not warrant recreating it—only the visual appearance changes.

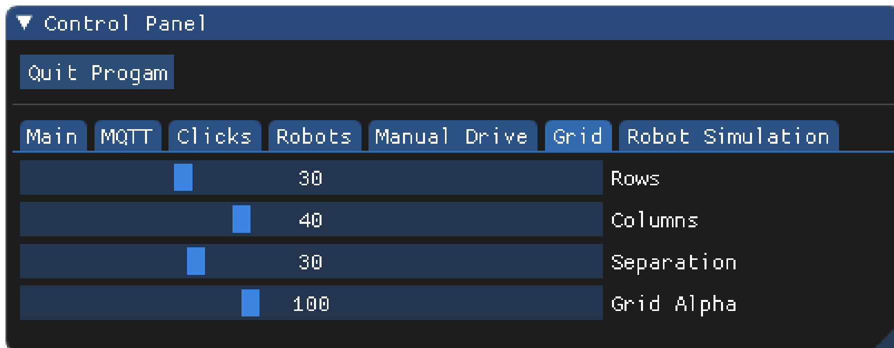


Figure 7.18: Grid tab.

Figure 7.19 shows the simulation tab. It simply has a checkbox to turn on a simulated robot. When checked, a simulated robot starts moving around randomly, with random measurements added as well. The simulated robot sends a message every 10 milliseconds, to allow testing at high load. The simulated robot sends valid messages to MQTT, and acts just as a physical robot would from the perspective of the server application.

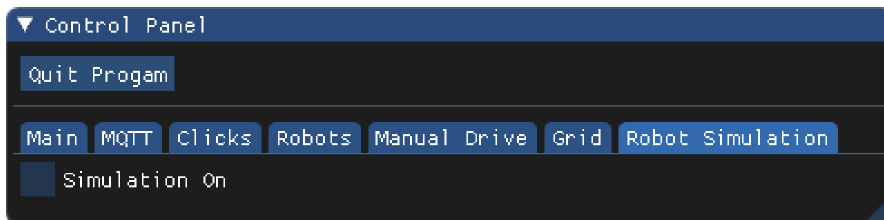


Figure 7.19: Simulation tab.

7.9 Communication Stack

The way communication happens with robots is the most drastic change of the new application versus the Java application. Chapter 5 explains the background, setup, and installation of the new communication stack. This setup is assumed,

but the whole stack is not strictly required. As the server application uses MQTT—an application layer protocol—as long as robots are able to use this protocol somehow the application will be compatible. Currently, a Raspberry Pi is used to act as the Thread border router. Naturally, any (Linux) machine could be used.

7.10 Finding Paths

Pathfinding can be seen as a higher level SLAM task. The groundwork has been implemented by using the abstraction of an obstructable resizable grid, which is overlaid the actual physical geometry which the robots traverse. Since this grid is implemented as a graph, algorithms for finding paths are readily available. The pathfinding used in the implementation is uses the `Boost.Graph` library, which does not only provide abstraction for graphs but also some common algorithms. This library is extremely flexible. When an algorithm is applied to a graph, the library allows callbacks to be issued at every vertex visited during the course of the algorithm. Also, heuristics for determining what constitutes a *good* path is customisable. The current implementation uses a A* search algorithm—available in the library—and specifies that the distance from a given vertex and the goal/target vertex as the heuristic. Thus, short paths are favoured as solutions.

Chapter 8

Testing and Profiling

Aim of This Chapter

- Inspect the finished server application's performance.
- Show some insight into the CPU usage.
- Show some insight into the memory usage.

8.1 Testing

The feature of running a simulated robot proved useful as a means of testing the server by adding many measurements and running tests for a long duration of time.

Figure 8.1 shows the simulated robot after running several hours. The *Robots* panel shows the simulated robot has added well over 270 000 measurements to the application. Notice the simulated robot roams freely well outside the original application grid and adds measurements very far away due to the nature of its randomness. Notice the figure is zoomed far out—the robot is barely visible. Do not mind the small control panel text, the point is to give an impression of the large amount of points added to the application. This points to the application being able to run for long amounts of time with stability.

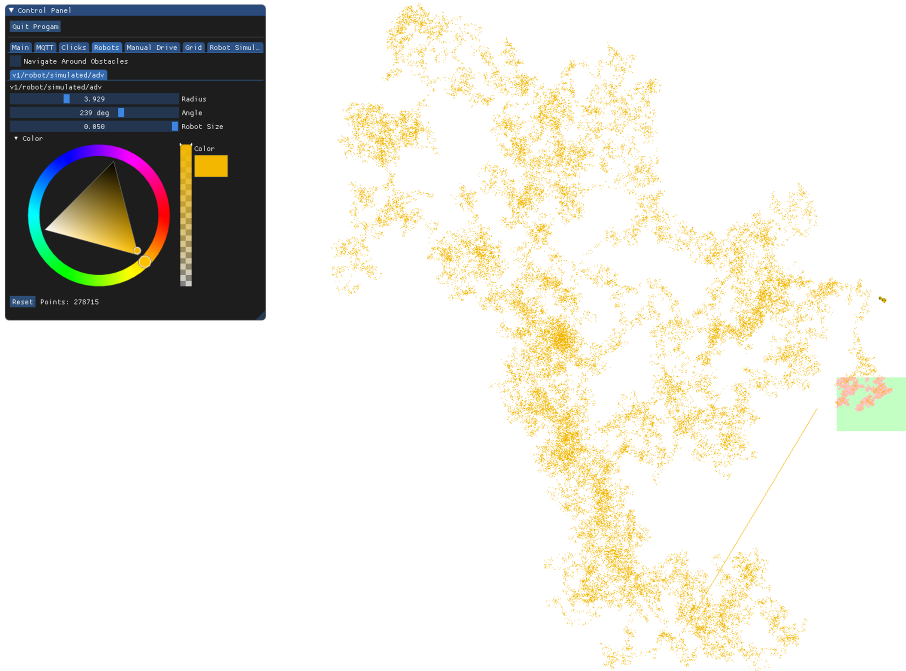


Figure 8.1: Testing a high number of measurements.

8.2 CPU Usage

Figure 8.2 shows a Visual Studio report of the CPU usage during the application runtime. The report shows that the vast majority of CPU time goes to drawing the robots and their measurements. As much as 96.75% of the used CPU time goes to this drawing. Note that this is relative to the usage of the CPU of the application itself, meaning that it always adds up to 100%—even if the application as a whole does not demand a lot from from the total available system CPU.

CPU Usage

Current View: Call Tree

Function Name	Total CPU [unit, %]	Self CPU [unit, %]	Module
SLAM-application.exe (PID: 14796)	4763 (100,00 %)	0 (0,00 %)	Multiple modules
make_fcontext	4676 (98,17 %)	0 (0,00 %)	boost_context-vc1...
boost::context::detail::fiber_entry<boost::context::detail::fiber_r...	4676 (98,17 %)	0 (0,00 %)	SLAM-application....
boost::fibers::worker_context<<lambda_39970ec82f414fe96...	4673 (98,11 %)	0 (0,00 %)	SLAM-application....
TG::gui::base::window::run	4673 (98,11 %)	0 (0,00 %)	SLAM-application....
sf::RenderTarget::draw	4656 (97,75 %)	0 (0,00 %)	sfml-graphics-2.dll
TG::application::SLAM::robots::draw	4608 (96,75 %)	0 (0,00 %)	SLAM-application....

Figure 8.2: CPU relative usage.

Figure 8.3 shows a snapshot of the total CPU usage in relation to the total available in the system, after starting the application. The graph generally shows use between 0% and 7% during application runtime. Note that this is not straight-forward to interpret. The desktop machine used for this testing has a CPU with 12 logical cores. This means full utilisation of one logical core would amount to 8.33% use in the chart shown. Additionally, the MQTT clients run mostly on separate threads. As an indication, usage close to or at 8% over longer periods of time would indicate that the application has hit a CPU bottleneck.

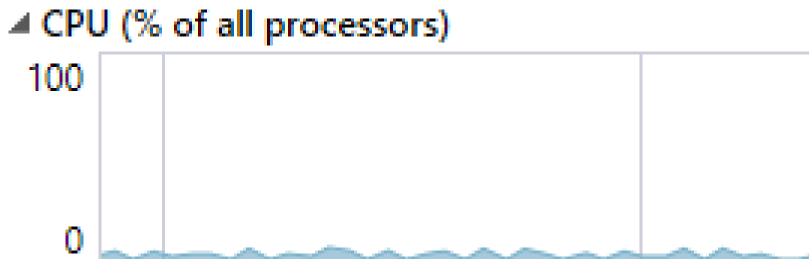


Figure 8.3: CPU usage versus total available in system after starting the application.

Figure 8.4 shows CPU usage after the application has run for some time, and has accumulated close to 20 000 measurements. The chart now shows use between 5% and 9%. This is closer to the maximum available CPU use from a single logical core (with some overhead added from other miscellaneous threads).

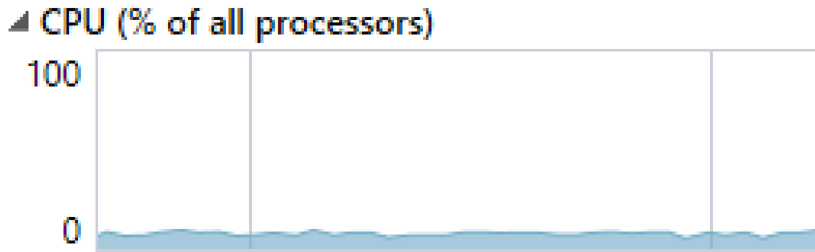


Figure 8.4: CPU usage versus total available in system after some time.

Thus we see that CPU usage is mild at the early stages of the application, and increases steadily with the addition of (thousands of) measurements.

8.3 Memory Usage

Figure 8.5 shows a built-in tool in Visual Studio aimed at inspecting the memory usage of applications. It works by creating a snapshot of the memory heap in at a given point in time, then creating another snapshot at a later point in time, then inspecting the difference. The figure shows the heap difference of about a minutes worth of running the simulated robot. Notice that the difference in allocated memory is about 105 KB, and almost all of it is due to new allocated circles—which essentially are the measurements added by the simulated robot.

Identifier	Count Diff.	Size Diff.	Count
Heap	+417	+105 392	881
boost::context::detail::fiber_entry<boost::context::detail::fiber_record<boost::context::f...	+417	+105 392	828
boost::fibers::worker_context<<lambda_9842c8bb0f93242587fb94e19418cea6> >...	+411	+100 492	493
[Unknown Frame]	+411	+100 492	493
<lambda_9842c8bb0f93242587fb94e19418cea6>::operator()	+411	+100 492	493
TG::application::SLAM::robots::feed_message	+413	+100 528	493
TG::application::SLAM::robot::add_obstacle	+413	+100 528	493
TG::gui::collections::circles::add	+421	+101 040	489

Figure 8.5: Memory snapshot heap difference.

Figure 8.6 shows a memory snapshot after stopping the simulated robot from sending messages, then taking a second snapshot at a time later. The figure shows that no allocations are made, and as such the total heap size is unaffected as well. This indicates that the application in its idle state seemingly does not leak any resources.

4	16,072.13s	4 526 (+3 645 ↑)	1 495,47 KB (+1 342,33 KB ↑)
5	16,161.14s	4 526 (+0)	1 495,47 KB (+0,00 KB)

Figure 8.6: Memory snapshot heap difference—simulation stopped.

Thus we see that the memory footprint of the application is very proportional to the amount of measurements added to the application.

Chapter 9

Extending and Using The New Server

Aim of This Chapter

- Instruct users and future developers on how to run the developed server application.
- Instruct future developers on what is needed to compile the server application.
- Show future developers how four major tasks can be achieved when developing new features: Drawing graphics, adding new user interface panels, using the pattern of callbacks for events, and running tasks in user-threads.
- Outline the work done in connecting a legacy robot to the server application.

9.1 Running the Server

See Appendix A for an overview of the included files. The `bin/` folder includes a `.zip` file which when extracted contains an executable (the `.exe` file). Running this on a Windows operating system should start the server. The server should look like the figures shown in Chapter 7. If the server does not start, a requirement needed to run compiled applications on Windows might be missing. This framework is called `NET Framework` and should be bundled with Visual Studio [21]. This means users not having Visual Studio might need to download it despite not intending to develop the application further. Alternatively, the

framework can be downloaded separately.

9.2 Extending the Server

The `src/` folder contains (after unzipping) the Visual Studio Solution file (ending in `.sln`). Opening this file in Visual Studio presents the project ready to build, run, and extend upon. Within Visual Studio, Debug (32-bit) and Release (32-bit) configurations have been used.

Compiling the project will be successful if the dependencies are installed. The project's dependencies are:

- Boost
- SFML
- Thor
- imgui
- paho-mqtt

The recommended way to install these dependencies are via the tool `vcpkg`. This method is explained in Section 2.3.1.

A few examples will now be presented. The examples show how to create a drawable rectangle on the screen, how to create new panels, how to use the user-created abstract callback base-class, and how to run new tasks in userland-threads. These examples provide the base understanding for how the application is designed. Understanding the examples will provide an easier transition into understanding and extending upon the application source code.

9.2.1 Example: Adding Drawables

The application creates a window which displays all content. This is done via the class `TG::gui::base::window`. A fiber runs a continuous loop calling this class' member function `TG::gui::base::window::run`, which draws and renders all given drawables (and panels, etc.). Using this scheme, this example shows how to add a new type of drawable to the screen.

Consider Listing 9.1.

Listing 9.1: Adding a drawable.

```
1 // Should split into e.g. rectangle.h and rectangle.cpp
```

```
2 class Rectangle : public sf::Drawable {
3 public:
4     Rectangle() : rect_({ 100, 200 }) {
5     rect_.setFillColor(sf::Color::Blue);
6     rect_.setPosition({ -200, 100 });
7   }
8
9   void draw(sf::RenderTarget& target, sf::RenderStates states) ↩
        const {
10     target.draw(rect_, states);
11   }
12
13 private:
14   sf::RectangleShape rect_;
15 };
16
17 // Within main.cpp
18 struct Rectangle rect;
19 win.add_drawable(&rect);
```

A class intending to be drawn inherits from SFML's `sf::Drawable` class. A requirement is to implement (override) the `draw(...)` function with the prototype as shown in the example. A reference to a render target is given, which we can draw to. Here we draw all our drawables. This example has a single draw call; a rectangle. This rectangle will be shown as a blue rectangle with the position and size as given in the class' constructor. See Figure 9.1. A rectangle is displayed along a grid and a robot having found some obstacles.

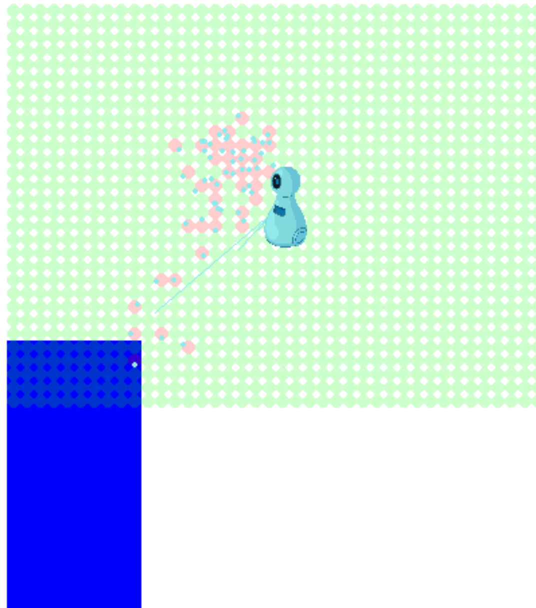


Figure 9.1: Drawing a blue rectangle onto the application.

The *states* argument carries with it the transformations of this class' par-

ents. This means if we apply transformations to the whole window (e.g. a zoom), the zoom effect will carry over to this class. This implies we can ignore transformations to our parents by omission of this argument to the draw call if this is our intention.

After we create a class implementing the required function, we can simply add it to our window after creating an instance of it. This concludes the simple drawable example. For a more comprehensive example, browse the files implemented in this thesis which are drawable. Examples are `circles.cpp`, `lines.cpp`, and `circle_grid.cpp`.

9.2.2 Example: Adding a Panel

There are two main ways of adding a panel to the project.

Adding to the Control Panel

A convenience class `TG::gui::panel::control_panel` has been created for ease of adding new panels in an orderly manner. Consider Listing 9.2. Here, a simple panel with a button and a slider is created. This panel is embedded onto the existing control panel. It will then automatically be placed in a new *tab* in the control panel, as shown in Figure 9.2. Note that other panels have been embedded which is not shown in the example source code presented.

Listing 9.2: Embedding a panel.

```
1  TG::gui::panel::panel example_panel;
2  example_panel.set_fun([&] {
3    static int slider_value = 0;
4    ImGui::SliderInt("Some Value", &slider_value, -100, 100);
5
6    if (ImGui::Button("Click me!"))
7      std::cout << "I was clicked! Slider has value: " << ↩
         slider_value << "\n";
8  });
9
10 TG::gui::panel::control_panel ctrl_panel;
11 ctrl_panel.embed_panel(&example_panel, "Example");
12 // Other panels embedded below ...
```

Adding a Separate Panel

The second way is to make a separate panel, which will not be embedded within another panel. This is done in a very similar way. Consider Listing 9.3. Notice the instantiation of the panel now requires a title. This tells the `TG::gui::panel::panel` class that this panel is to have a separate window.

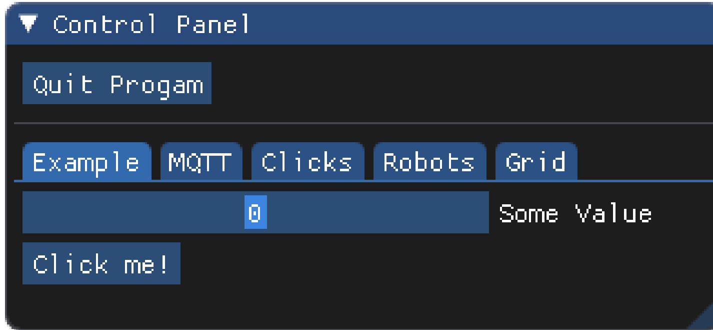


Figure 9.2: Panel embedded onto the control panel.

The other difference is that this panel is now added to the main window, instead of embedded onto the control panel. Figure 9.3 displays this new panel.

This concludes examples of creating panels and adding them to the existing project. See e.g. classes `slam_grid.cpp` and `MQTT_panel.cpp` for more detailed examples.

Listing 9.3: A separate panel.

```
TG::gui::panel::panel example_panel("Example Own Window");
example_panel.set_fun([&] {
  static int slider_value = 0;
  ImGui::SliderInt("Some Value", &slider_value, -100, 100);

  if (ImGui::Button("Click me!"))
    std::cout << "I was clicked! Slider has value: " << ↩
        slider_value << "\n";
});

TG::gui::base::window win;
win.add_panel(&example_panel);
// Other panels embedded below ...
```

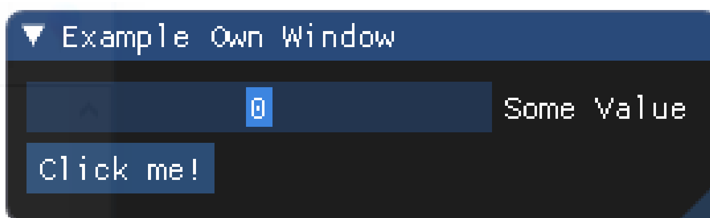


Figure 9.3: A panel separate from other panels.

9.2.3 Example: Adding Callbacks to a Class

Callbacks are widely used in the project. Callbacks allow for decoupling classes. A class can emit events when interesting things happen, and the user of class can attach functionality to these events from the outside. A template class `TG::utility::callbacks` was created to let classes easily support this design pattern. A class inherits this template class with an event type as template argument. This allows the user of this new class to enable callbacks for these events. The new class must then call these events at appropriate times. Consider Listing 9.4. A simple class is created. It inherits from `TG::utility::callbacks`, and specifies the data passed to callbacks will be of type `std::string`. This can be skipped; the default type passed is `std::any`.

Listing 9.4: Adding callbacks.

```
1  // simple_class.h
2  enum class simple_class_events {
3    FOO_EVENT,
4    OTHER_EVENT,
5  };
6  struct simple_class : TG::utility::callbacks<enum ↩
       simple_class_events, std::string> {
7    simple_class() {};
8    void foo() {
9      call_callback(simple_class_events::FOO_EVENT, "Foo called!");
10   };
11 };
12
13 // main.cpp
14 simple_class simple;
15 simple.enable_callback(simple_class_events::FOO_EVENT, [&](std::↩
       string message) {
16   std::cout << "I am a callback. I was passed a message: " << ↩
         message << "\n";
17 });
18
19 simple.foo();
```

The simple class calls the event `simple_class_events::FOO_EVENT` if an outside user calls the member function `simple_class::foo`. If the event has not been enabled, the call to `TG::utility::callbacks::call_callback` is ignored.

When the line `simple.foo()` is called, the message `I am a callback. I was passed a message: Foo called!` is printed to console, indicating a successful callback.

This concludes the example of adding callbacks to new classes created for the project.

9.2.4 Example: Running a Task

Tasks are run in userland-threads via the library `Boost.Fiber`. These threads have very little overhead. They do not run on different OS threads unless

explicitly coded as such. They schedule cooperatively. This means they must actively yield their execution to other userland-threads. An example is given in Listing 9.5.

Listing 9.5: Adding a task.

```
auto example_ch_1 = boost::fibers::buffered_channel<std::string↩
    >(8);
auto example_ch_2 = boost::fibers::buffered_channel<int>(8);

fiber example_f([&] {
  for (;;)
  {
    auto result = example_ch_1.push("Data for you.");
    if (result != boost::fibers::channel_op_status::success)
      std::cerr << "Could not push onto channel 1.";

    auto next_time = system_clock::now() + milliseconds(1000);
    sleep_until(next_time);

    int pop_value = 0;
    const auto pop_wait_time = milliseconds(250);
    auto result2 = example_ch_2.pop_wait_for(pop_value, ↩
        pop_wait_time);
    if (result2 != boost::fibers::channel_op_status::success)
      std::cerr << "Could not push onto channel 2.";

    yield();
  }
});

example_f.join();
```

This example task is given as a lambda function during the instantiation of a fiber. Communication to the outside world is done via channels. The *push* and *pop* operations will instantly succeed if there is space on the given channel, else they yield the fiber. The function *sleep_until* will also yield until the specified duration has expired, after which the fiber is put into a ready state. The explicit call to *yield* naturally yields the execution to other fibers. Notice that the lambda function in the example captures variables in the parent scope via the & operator in the capture list. This is only safe on a single threaded application. If several threads access variables used in a fiber, shared-access protection must be used. See `MQTT_panel.cpp` for an example of this using `std::lock_guard` for protection. Also note that using channels is safe even on multiple threads.

This concludes the example of adding a new simple task which is able to safely pass data to outside channels.

Chapter 10

Using a Legacy Robot

Aim of This Chapter

- Explain the need for special treatment of legacy robots.
- Present a legacy layer enabling legacy robots.
- Present an overview of the inner workings of the legacy layers via figures.
- Discuss the benefits and drawbacks of using this approach.
- Show how communication in action works using the legacy layer.
- Present results of real world use.

10.1 Connecting a Legacy Robot

This section refers to a robot without the capabilities to connect to OpenThread as a legacy robot. This means the required radio hardware is not present, and this is the case for all robots as of the start of this thesis.

The complete source code running on the legacy layer can be found in the delivered files. See Appendix A.

10.1.1 Overview

As this thesis created a server which is intended to connect to robots via "next-gen" technology, a way to use the old robots before porting them to new hardware (or adding radios) is sought. As such, this thesis included helping create a legacy layer which allows this. Note that the solution is intended to bridge the gap between the technologies, and the bridge is obsolete as soon as the robots

have the required hardware on board. As a consequence of this, only a limited time was spent on ensuring robustness and code quality. This means future generations should sooner port the hardware than to keep this solution. The main reason for not using this solution is that it increases the overall complexity of the project as a whole, whereas porting the hardware simplifies it.

Figure 10.1 shows an overview of the implemented solution.

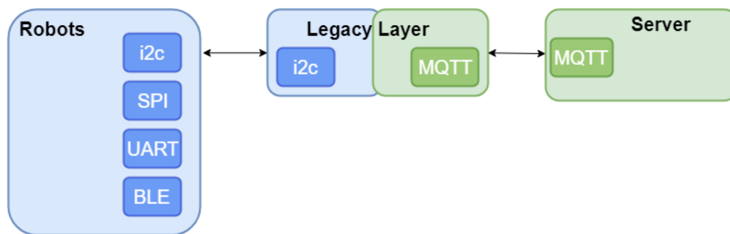


Figure 10.1: Legacy layer overview.

The robot which this layer targeted had many peripherals on board, and I2C was the most available communications bus. Using SPI or UART was determined to be of higher complexity, either due to hardware complexities (soldering, adapting to an already used bus) or software complexities (e.g. UART without control signals), or both.

10.1.2 Implementation

Figure 10.1 shows the legacy layer which bridges the gap between the technologies. For this to work, an MQTT broker has to be available, and hardware communicating via OpenThread is needed. As such, the solution relies on a setup similar to the one already discussed in Chapter 5. This setup is now assumed available. Now, an overview of the source code that was implemented for the legacy layer is given. Note that the implementation is largely callback-driven, and as such asynchronous by nature. Thus state diagrams are represented in separate disconnected trees, as this reflects reality more closely than a single connected state tree.

Figure 10.2 shows the initialisation of MQTT on the legacy layer. Note that error handling is not done. Debugging information is presented if a debugger is connected.

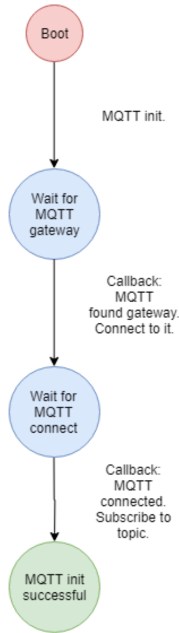


Figure 10.2: Initialisation of MQTT.

This implies the legacy layer assumes working conditions, and if issues arise a restart is likely required.

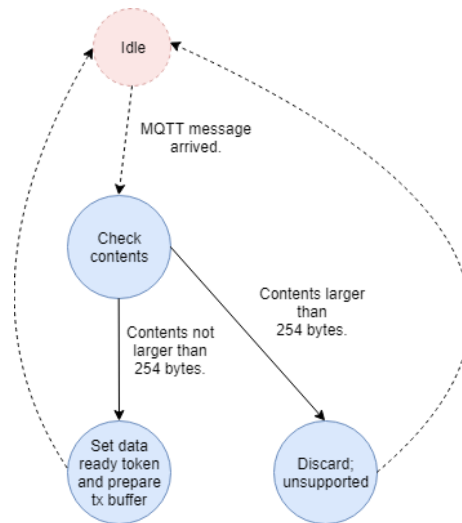


Figure 10.3: MQTT loop lifetime.

Figure 10.3 shows the handling of MQTT during the lifetime of the legacy layer. It is equally simplistic, and does likely not handle corner cases well. One example is if several MQTT messages arrive before the master (i.e. the connected robot) reads. The data will be overwritten as soon as a new message arrives, and the previous content is then lost. The consequence is likely minimal as the only data presented to the robot is the next coordinate to move towards, but the limitation exists.

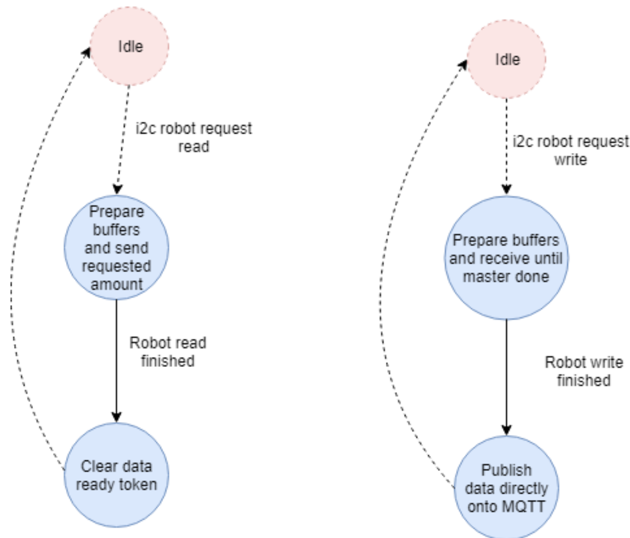


Figure 10.4: I2C loop lifetime.

Figure 10.4 shows how the legacy layer responds to a robot requesting to read or write on the I2C bus. As the legacy layer acts as an I2C slave and no additional control lines were available (for e.g. indicating new data available), new data being available is indicated through the ”data ready token”, which is the first byte presented before the data itself when the robot reads from the legacy layer. This explains the setting of the token on MQTT message arrival and clearing it on successful reads. When the robot I2C master writes onto the I2C bus, the bytes are accepted and directly sent published onto MQTT as soon as the I2C transmission finishes successfully.

10.1.3 Discussion

Table 10.1 shows an overview of the pros and cons of using such a legacy layer. The additional increase in complexity should be weighed heavily. The one-time

Pros	Cons
Allows robots with missing hardware to use OpenThread indirectly. Can possibly be made robust once and then used forever, as the responsibilities are quite small. Does not require porting the robot to a new chip.	Adds complexity to the project. Removes many of the benefits associated with a native OpenThread solution. Introduces additional hardware. Needs a communication bus connected to the additional hardware. Source code must be recompiled and reflashed in order to change configuration parameters. The legacy layer source code is rudimentary. It needs testing and higher code quality and likely contains bugs. If future generations wish to use other protocols than MQTT, this is difficult as the legacy layer is tied to MQTT. It is simpler when OpenThread is run natively.

Table 10.1: Pros and cons of using the legacy layer.

cost of porting the robots to a chip which has the required radio hardware is strongly advised.

10.2 Communication Protocol

As an additional layer of communication is present—I2C—communication is not as simple as a native Thread approach is. A simple approach between server and robots was sought in order to not complicate the task more than necessary. Therefore, a very simple approach was taken:

- All communication happens in forms of coordinate pairs *(x, y)*.
- Whenever a robot sends a message to the server, the first coordinate pair is the position of the robot itself (in its own frame of reference, naturally).
- Whenever a robot sends more than a single coordinate pair, all pairs except the first ones are parsed as positions of obstacles.
- The only message the server sends to a robot (or robots) is a single coordinate pair: This is where the server wishes a robot to go to—a *waypoint*.

By request of one of the robot implementers, coordinates are stored in the `int16_t` data type. This implies that the minimum message payload size is 4 bytes—2 bytes for x and 2 bytes for y. Note that all systems are *little endian*.

Thus an overview of this communication model is given in Figure 10.5. Here, the pair (x_w, y_w) constitutes the waypoint issued to the robot from the server. The pair (x_r, y_r) is the position of the robot as sent from the robot to the server.

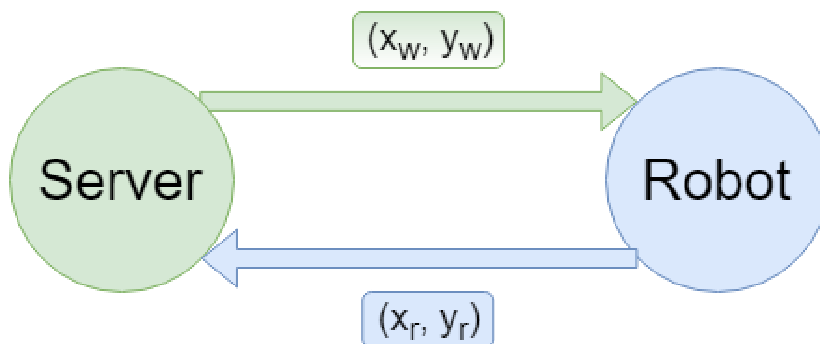


Figure 10.5: Server and robot communication overview.

Another example is given in Figure 10.6. Two additional pairs are now part of the payload sent from the robot. These will be added as obstacles in the server application map.

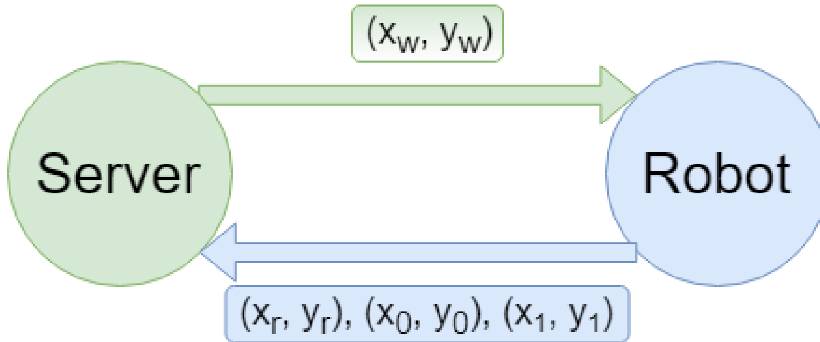


Figure 10.6: Robot sends optional obstacles.

A couple of notes:

1. The number of obstacles that can be sent from the robot is dynamic. 0 is a valid number of obstacles, and 10 at a time should be fine as well. Currently, the dongle does not accept messages over 255 bytes at the I2C level. A native Thread approach is not practically limited: The MQTT specification allows up to 256 MB of data in a single message. Buffer sizes in embedded devices are likely the limiting factor at that point.

2. Two-way communication is not a requirement. The robot can only listen to messages if desired, or completely ignore the server and only send messages.

10.2.1 I2C

The dongle acts as a I2C slave at address `0x72`. Always read 5 bytes from this address when reading. The first byte is a byte indicating if new data has arrived since the last read. If not, this byte will read `0x00`. If new data is present, it will read `0x72`. This byte can be ignored at the receiver if desired. The next four bytes are the two `int16_t` values composing the coordinate pair (x, y), in that order.

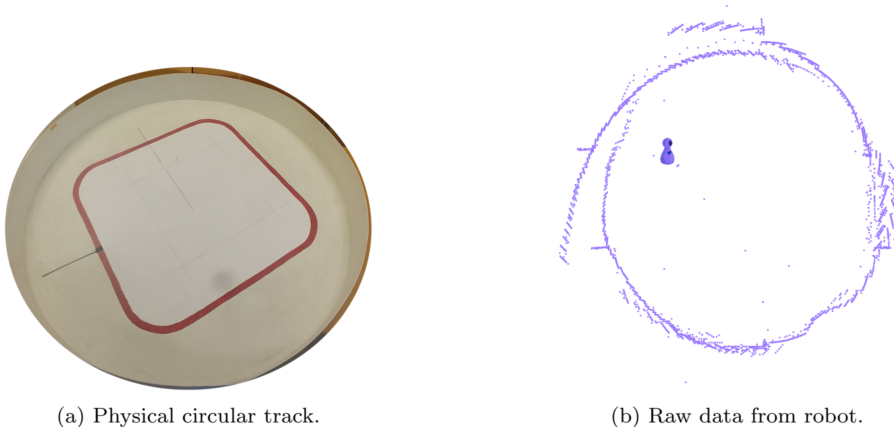
10.2.2 MQTT

Messages from the robot via I2C are published to MQTT topic `v1/robot/Endre/adv`. Messages from the server arrive via the topic `v1/server/Endre/cmd`. This scheme is arbitrary and need not be followed in future generations, but the intention behind the fields was as follows:

- `v1`: This hints at the version of the communication scheme. This is not uncommon in MQTT. If breaking API changes are made, a natural consequence is to raise this version level. Clients then need to adapt to the new version and update their subscriptions.
- `robot/server`: This field indicates the originator of the message.
- `Endre`: Robot name. This legacy layer was made with the Endre Leithe's robot in mind, and as such this field identifies his robot.
- `adv/cmd`: This field hints to the intent of the message. When a robot sends `adv`, it *advertises* a new measurement. When a server sends a `cmd`, a *command* is issued.

10.3 Real World Use: Detecting a Circular Track

The scheme laid forth in this chapter was put to use on a circular test track. This track can be seen in Figure 10.7a. Figure 10.7b shows the raw data as reported by the robot's sensors during the duration of the test.



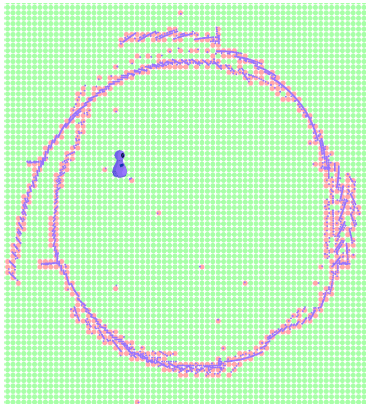
(a) Physical circular track.

(b) Raw data from robot.

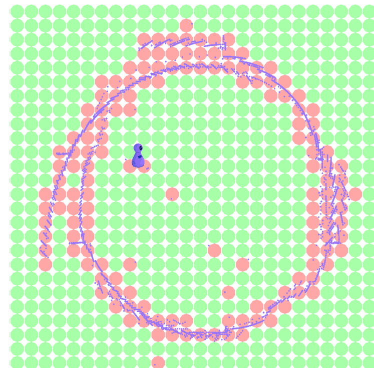
Figure 10.7: Physical and digital representation of a track.

Figure 10.8a shows the underlying grid along with the robot's reported data. Obstructed grid points are red, while unobstructed points are green. Notice the grid serves its purpose: To downsample the resolution of the possible paths the robot can be issued to travel to via pathfinding. This is made apparent by several close points reported by the robot ending up inside a single grid-point.

Figure 10.8b shows the underlying grid after increasing the separation between adjacent grid entries. This effectively decreases the resolution of the grid further. This shows the dynamic nature of the application, as this can be configured in real-time while the application is running without affecting any data. There is a slight optical illusion when the resolution is decreased like this: Notice the distances between the raw data-points issued from the robot have not changed in the shown figures.



(a) Data from robot with underlying grid.



(b) Grid with lower resolution.

Figure 10.8: Dynamically changing grid properties.

Chapter 11

Discussion and Future Work

Aim of This Chapter

- Contrast the state of the new application versus the old.
- Comment on weak points of the new application.
- Identify what to improve on concerning the new implementation as future work.
- List some major goals to add to the application as future work.

11.1 Discussion

11.1.1 C++ Application Versus Java Application

Graphical Presentation

The Java application and the C++ application have both similarities and differences. Graphically, the C++ implementation has more flexibility and freedom. Robots can be changed in various ways visually, as can measurements. Also, the C++ version uses a *sprite* (essentially an image with a transparent background) to display robots. This can be swapped for other images—even images of the real robots. The Java application uses black dots to place measurements, and grey areas to indicate zones where the robot should not traverse due to proximity to objects. The C++ application uses the grid instead, which blocks out a radius equal to the size of a grid node when a measurement falls within the node—this point is now obstructed. The C++ application is dynamic in

that the nodes of the grid can be changed during runtime. The result is shown visually, as well as having an actual effect on the pathfinding of the system.

User Interface

The two applications use different approaches when it comes to the user interface. The Java application uses a more traditional approach of menus, several windows, a toolbar, and pop-up windows. The visual theme of the interface is close to what a typical user is familiar with from experience using other desktop applications. The C++ application user interface collects everything into a common *control panel*. The style and theme is likely more unfamiliar. The different options are grouped into logical categories and all categories can be found in the same place via tabs. The control panel is very easily extended, and this has been documented via examples. The programmatic approach to extending the user interface in the C++ application is contrasted with the Java application—it used a user interface editor which uses a *drag-and-drop-programmatic* mixed approach. Which approach to programming user interfaces is better is up to personal preference.

Code Quality

The C++ application uses a modern and safe programming style benefiting from the newer C++ standards. This includes a code style aimed at readability and safe resource management. Other main points have been to make classes have a single responsibility, and try to achieve decoupling through an event based system. The Java codebase walkthrough showed a varying degree of quality. Several classes were small, readable, and had a single responsibility. Several classes also had a large number of dependencies, private variables, and had too much functionality. Too many classes had references to robots, and sensors, and the control flow of the application is hard to get a sense of via its entry point. The added complexity due to the communication layer degrades the quality of the application.

Communication

The new approach to communication allows much more freedom and ease of development. All the complexity of implementing a custom user protocol, maintaining it in several programming languages, and having to reprogram every device when a new message type is sought is no longer relevant. This is due

to using a standardised protocol widely used in industry—MQTT. MQTT is most easily supported via the Thread stack. This communication stack offers features such as default encrypted messaging and automatic mesh topology.

Another important benefit of using MQTT is the intrinsic pros of the *publish-subscribe* model. Without changing the server application, robots (or any other device supporting MQTT) can publish data to any topic. The server will not notice—the previous ARQ protocol would have to handle any and all messages somehow. Thus the server application and robots can be developed in parallel without unnecessary overlap. When a new message type or a new data feature is ready for use, the server application developer can be noticed. Then the relevant topic can be subscribed to, and message parsing added for the new feature. The same applies for robots—they will always be unaffected by changes done to the server (as long as the server respects the data format a robot expects on a given topic).

The most pressing issue with using the Thread approach is that the robots do not currently have the necessary hardware to use it. As such, either *legacy layers* which enable old hardware to use the new communication stack are needed, or upgrading the robots to use new hardware—which is the more sane approach. The Thread setup has the need for a *border router* as well, which ties together the Thread network with the global internet. This has been shown possible via a Linux machine (a Raspberry Pi) and a dongle providing the Thread radio. All the firmware and installation files for this is provided externally, and minimal setup is needed. This is contrasted by the old communication stack, where any change in the stack would require flashing several dongles, whereas the Thread stack only needs to be setup once. This issue highlights a key difference between the old stack and the new. The old stack intermingles the application layer and the deeper OSI model layers. A pure networking protocol should be indifferent to the application layer—this is the case for the Thread stack. The Java application ARQ protocol has types `DATA`, `ACK`, `SYN`, `SYNACK`, and `ALIVE_TEST`. As an example, robots must be sure their implementation of the protocol responds correctly to `ALIVE_TEST` type messages, and also be sure the sequence number is correct and so on.

The limitations of short messages before fragmentation and the low amount of robots that could be connected simultaneously also disappear using the new stack. The issue of robots randomly disconnecting [19] has not been observed with the new server application. It is also noted that users of the Java application noticed complexity and instability from the Java application, and the

application throwing errors during running [9].

It is therefore argued that a key concern of the project is to overhaul the robots by porting them to the new hardware which natively supports Thread networking.

Features

The Java application has been developed by several generations of the SLAM project, and has accumulated several features which the new C++ application does not yet have. Some major ones are:

- Algorithms for steering robots towards points in an intelligent manner—leading to room exploration. Currently, the C++ version allows manual targeting, square path targeting, and click targeting.
- An intricate simulation. The Java robot simulation allows the user to set a predefined map of the environment which the robot must navigate through and discover, with configurable errors in pose and sensors. The C++ application has a simple randomised robot behaviour with no set map.
- Experiments with pose filtering. The Java application has seen use of higher level SLAM techniques such as using a *particle filter*. The C++ application currently displays only raw measurements.

Thus, the Java application has more intricate features as of present. Arguably, the C++ application has a better foundation for solving SLAM problems, as it has greater flexibility graphically, a more solid communication stack, and is written in a programming language more familiar to future developers—but clearly there is a large amount of work to be done.

Still, there are some points the Java server had issues with that the current application solves. One example is that is has been reported an issue that robots can overwrite the other robots' measurements. This happened because the previous implementation erased data when old measurements were in between a robot and a new measurement [19]. Also, the previous server application has run into both memory and CPU issues [19]. This happened when having hundreds- to thousands of measurements ongoing in the application. The new application has no memory issues, and only encounters issues when having hundreds of thousands of measurements simultaneously—see Chapter 8. Users of the Java application have reported an issue around the bad map resolution—the

smallest point-resolution is 1 centimetre [35]. The new application has no such limitation—the application currently makes no assumption on physical units.

11.2 Future Work

This section suggests some key points for the project to work on in the future.

- The most important point is to focus on the SLAM task, and try to implement some higher-level features related to this. This means dealing with issues concerning errors in pose- and sensor estimation. Thus particle filters (or other approaches) should be considered.
- The major features (Section 11.1.1) of the Java application which are not in the C++ application should be considered implemented.
- Setting the target for a robot applies to every robot—there is currently no way to set a target for a *single* robot.
- Measurements should be abstracted into a class. Currently they are simply added to a robot's vector of measurements. This means the position of the robot during measurement is not recorded. Such information can prove useful and support is currently lacking.
- The user interface can be cleaned up a bit—the *Main* tab, *Manual Drive* tab, and *Robot Simulation* tabs could be placed into separate files. They currently live inside `main.cpp`.
- Some panels can be expanded with more features. The *Main* tab could have a lot more information about general application statistics. Messages sent, received, up-time, number of robots, and so on. The *Clicks* tab could have more features and more buttons, and so on.
- There are some minor graphical issues. Graphical elements' positions are normally defined by their top-left corner in their invisible bounding box. The effect is that e.g. a circle might visually look like it is not centred at the intended point. This can be mitigated by redefining the local origin of the element to the element's centre.
- Performance profiling showed the application becomes CPU bound when several tens of thousands of points are to be drawn simultaneously. This situation can be improved in two ways. One is to move drawing to another

thread. This would allow drawing to be handled on a separate core, thus other operations are unaffected. However, that core would eventually be CPU bottlenecked as well. A solution is to draw the last n measurements of the robots only.

- The current robots should be upgraded to newer hardware stacks such that they can take advantage of the recent developments in technology, and benefit from the Thread stack. As soon as robots are connected via Thread, they are accessible through the global internet. This means the possibility of developing useful monitoring tools for smart-phones, web interfaces, and other desktop applications becomes instantly available.

- Consider a more intricate simulation of robots. By having a hidden map of walls and obstacles for simulated robots to discover, the SLAM techniques are more easily tested.

- `slam_message.cpp` now parses messages where the payload is robot position and an optional amount of obstacles. All information is given via Cartesian coordinates. Consider using the MQTT topic structure more heavily in order to support new future messages. Example could be to have robots publish to topics like `/robot/point/self`, `robot/point/obstacle`, `robot/line/obstacle`, `robot/cartesian/obstacle`, `robot/polar/line` and so on.

- The use of namespaces in the code-base is an important step in having well-organised code. A review of the current namespace ordering should be considered (Section 7.2). For example, `TG::graph::base::grid` can perhaps be moved to `TG::graph::grid::base`, and `TG::gui::base::path` should probably be an element, placed in `TG::gui::elements::path`. The prefix `TG` could be renamed to e.g. `NTNU` or `SLAM` or similar, in order to not alienate future developers.

Bibliography

[1] *Bluetooth Technology Website*. URL: https://www.bluetooth.com/.

[2] *Boost C++ Libraries*. URL: https://www.boost.org/.

[3] *Boost Fiber*. URL: https://www.boost.org/doc/libs/1_70_0/libs/fiber/doc/html/index.html.

[4] *Boost Graph*. URL: https://www.boost.org/doc/libs/1_70_0/libs/graph/doc/index.html.

[5] *C++ language*. URL: https://en.cppreference.com/w/cpp/language.

[6] Omar Cornut. *Dear ImGui*. Apr. 2019. URL: https://github.com/ocornut/imgui.

[7] Elias Daler. *Dear ImGui SFML Binding*. Apr. 2019. URL: https://github.com/eliasdaler/imgui-sfml.

[8] H. Durrant-Whyte and T. Bailey. "Simultaneous localization and mapping: part I". In: *IEEE Robotics Automation Magazine* 13.2 (June 2006), pp. 99–110. ISSN: 1070-9932. DOI: 10.1109/MRA.2006.1638022.

[9] Geir Henning Eikeland. "Implementation of Mapping and Navigation on an Autonomous Robot ". MA thesis. NTNU, 2018.

[10] *Free IDE and Developer Tools — Visual Studio Community*. URL: https://visualstudio.microsoft.com/vs/community/.

[11] *Git*. URL: https://git-scm.com/.

[12] Humberto Gonzalez Gonzalez. "Study of the protocol for home automation Thread." Master's Thesis. Catalonia, Spain: Polytechnic University of Catalonia, 2017.

[13] Rick Graziani. *IPv6 Fundamentals: A Straightforward Approach to Understanding IPv6*. Cisco Press, 2017. ISBN: 1587144778.

[14] Thread Group. *Thread*. URL: https://www.threadgroup.org/.

[15] Konstantino Zuraris Helders. "Real-Time System Implementation on Autonomous Lego-Robot". MA thesis. NTNU, 2017.

[16] Silicon Labs. *UG103.11: Thread Fundamentals*. Tech. rep. URL: https://www.silabs.com/documents/public/user-guides/ug103-11-appdevfundamentals-thread.pdf.

[17] Kristian Lien. "Embedded utvikling på en fjernstyrt kartleggingsrobot". MA thesis. NTNU, 2017.

[18] *MATLAB*. URL: https://www.mathworks.com/products/matlab.html.

[19] Henrik Kaald Melbø. "Autonomous Multi-Robot Mapping". MA thesis. NTNU, 2017.

[20] Microsoft. *Microsoft/vcpkg*. Apr. 2019. URL: https://github.com/Microsoft/vcpkg.

[21] Microsoft. *.NET — Free. Cross-platform. Open Source.* URL: https://dotnet.microsoft.com/.

[22] Microsoft. *PowerShell Documentation*. URL: https://docs.microsoft.com/en-us/powershell/.

[23] Microsoft. *Windows Network Architecture and the OSI Model*. URL: https://docs.microsoft.com/nb-no/windows-hardware/drivers/network/windows-network-architecture-and-the-osi-model.

[24] Nest. *OpenThread*. URL: https://openthread.io/.

[25] Simen Nilssen. "Navigation and control with Arduino robot". MA thesis. NTNU, 2018.

[26] OASIS. *MQTT Version 3.1.1*. Tech. rep. Oct. 2014. URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf.

[27] Fedor G. Pikus. *Hands-On Design Patterns with C++: Solve common C++ problems with modern design patterns and build robust applications*. Packt Publishing, 2019. ISBN: 1788832566. URL: https://www.amazon.com/Hands-Design-Patterns-reusable-maintainable/dp/1788832566.

[28] Mats Gjerset Rødseth and Thor Eivind Svergja Andersen. "System for Self-Navigating Autonomous Robots". MA thesis. NTNU, 2016.

[29] Nordic Semiconductor. *nRF5 SDK for Thread and Zigbee v3.0.0*. Tech. rep. URL: https://infocenter.nordicsemi.com/topic/struct_sdk/struct/sdk_thread_zigbee_latest.html.

[30] Nordic Semiconductor. *nRF52840*. Tech. rep. URL: https://infocenter.nordicsemi.com/pdf/nRF52840_PB_v1.0.pdf.

[31] Nordic Semiconductor. *UART/Serial Port Emulation over BLE*. URL: https://infocenter.nordicsemi.com/index.jsp?topic=/com.nordic.infocenter.sdk5.v14.0.0/ble_sdk_app_nus_eval.html.

[32] *Simple and Fast Multimedia Library*. URL: https://www.sfml-dev.org/index.php.

[33] *Smart Pointers (Modern C++)*. URL: https://docs.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp?view=vs-2019.

[34] Andy Stanford-Clark and Hong Linh Truong. *MQTT For Sensor Networks (MQTT-SN)*. Tech. rep. Nov. 2013. URL: http://www.mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf.

[35] Lars Marius Strande. "Autonom retur og dokking av AVR robot". MA thesis. NTNU, 2017.

[36] Bjarne Stroustrup. *A Tour of C++ (2nd Edition) (C++ In-Depth Series)*. Addison-Wesley Professional, 2018. ISBN: 0134997832.

[37] Bjarne Stroustrup. *Bjarne Stroustrup's C++ Glossary*. URL: http://www.stroustrup.com/glossary.html.

[38] Herb Sutter. *Writing modern C++ code how C++ has evolved over the years*. Oct. 2011. URL: https://www.youtube.com/watch?v=Kghns7c8Ij8.

[39] Eirik Thon. "Mapping and Navigation for collaborating mobile Robots." MA thesis. NTNU, 2016.

[40] *Thread Stack Fundamentals*. Tech. rep. Thread Group, July 2015. URL: https://www.threadgroup.org/Portals/0/documents/support/ThreadOverview_633_2.pdf.

[41] Sebastian Thrun and John J Leonard. "Simultaneous localization and mapping". In: *Springer handbook of robotics* (2008), pp. 871–889.

Appendix A

Files Overview

This overview shows the contents of the `zip`-file delivered alongside this document. Archives are used to preserve the complete integrity of all necessary files.

```
master_thesis_torstein_grindvik.zip
├── application
│   ├── src
│   │   └── src.zip
│   └── bin
│       └── bin.zip
├── documents
│   └── thesis.pdf
├── legacy_layer
│   └── legacy_layer.zip
└── java_nilssen_2018
    └── java_nilssen_2018.zip
```