Master's thesis

NTNU
Norwegian University of Science and Technology
Department of Engineering Cybernetics

Casper Nilsen

The Cyborg v4.0

Implementing GUI for ROS with real-time monitoring, commanding and controlling capabilities

Master's thesis in MITK - Cybernetics and Robotics
Supervisor: Sverre Hendseth
June 2020

The Cyborg Robot

NTNU
Norwegian University of Science and Technology

Norwegian University of Science and Technology

Masters Thesis

The Cyborg v4.0 - Implementing GUI for ROS with real-time monitoring, commanding and controlling capabilities

---

Author:
Casper NILSEN

Supervisor:
Associate Professor Sverre HENDSETH
Assisting Supervisor:
PhD Candidate Martinius KNUDSEN

Department of Engineering Cybernetics
Faculty of Information Technology and Electrical Engineering
Trondheim, Norway
16.12.2019

Problem Description

The problem to be solved during the specialization project in the Autumn of 2019 and Master's Thesis Spring 2020 was to create a GUI interface to remotely control and monitor the robot. During the specialization project, the aim was to create the foundational deliverables. These deliverables were then to be developed with the necessary capabilities during the Master's Thesis. As written in the specialization project report, the suggested deliverables for the Remote GUI (Graphical User Interface) Interface project were:

- A GUI such as a web app or using ROS-RQT.
- A GUI client ROS node that serves as an interface between the ROS-based system and the GUI.
- Communication between the GUI client ROS node and controller node.
- Optional: A database to store and plot historical data from the robot.

These deliverables were successfully created during the specialization project and serve as the foundations for the work in this Thesis. The result was a Web App using ASP.NET Web Forms, a GUI ROS node that listens and transmits data from the ROS system and a document-based NoSQL cloud database built using MongoDB Atlas. For this Master's Thesis, the aim was to develop the necessary capabilities on top of the deliverables. The suggested capabilities for the GUI were:

- **monitoring:** Ability to view where in the map the robot is. View operating state like running state-machine, manual operation, stopped etc. View status information, like battery charge, motor status. Plot historical information (optional).
- **commands:** Ability to change operating state. Changing state-machine state. Send robot to location.
- **Manual operation:** Ability to view live video feed and control robot using keyboard input. Onboard microphone and speakers for two-way communication.

Abstract

A cyborg or cybernetic organism is the union between mechanical and living biological parts. An interdisciplinary effort by NTNU Cyborg is in the process of developing a robot body that is connected to biological neural networks. This robot, fittingly named *The Cyborg* or *The Robot* aims to have a closed-loop bi-directional data flow through the neurons. Grown from the stem cells of either humans or rats, the neurons adapt to stimuli via electronic circuitry, creating the neural networks. The final goal is to be a continuous system using measurements from the robot's environment to train the cultured networks and using transmitted signals from the networks to control the behaviour of the robot.

In the long term, the plan is for the Cyborg to act as a mascot, roaming the hallways fully autonomously in Glassgården at NTNU Gløshaugen to generate interest for students in its relevant fields like biotechnology, neuroscience, engineering cybernetics and computer and information science. The current cyborg is in version 3.0 and is semi-autonomous. It navigates using a self-made map, is motivated by its behaviour-system and operates according to its state machine. Currently, the robot is at Supervisory Control level 9 since it cannot be left without supervision. To get to level 10, Full Automation, extensive testing must be conducted. The robot can be controlled and monitored remotely on a computer, but the procedure is non-trivial. There is therefore a need for a full-fledged GUI for real-time supervisory capabilities like fine control and comprehensive monitoring to aid in achieving a stable system.

During this Master Thesis a commander module for top-level management of other modules was developed and tested in the robot's main software framework called ROS (Robot Operating System). The full state machine at onset was added to a container and made into the 'behaviour' mode of operation state. Additional states were added to the state machine for modes of operation like manual control, demo and suspension. The GUI in the form of a new web application using Vue and Vue CLI (Command Line Interface) was evaluated, designed, implemented and tested. Various reactive functionalities for robot interaction, stored as separate Vue components was implemented. These functionalities include selecting mode of operation, click-to-navigate using navigation map and manual control using keyboard input or on-screen joystick with robot camera from the Computer vision Module. The GUI allows for monitoring of information like battery charge, motor state, mode of operation, SMACH state and behaviour-system PAD Values. The GUI also got a command tool for changing SMACH states, perform text-to-speech and selecting emotion in emotion-system. Finally, communication between the GUI and ROS system was bridged. Each part of the GUI module was discussed, their limitations defined and suggestions for future work was presented. From the problem description, every suggested feature for the Cyborg GUI was delivered successfully.

Abstrakt

En kyborg eller kybernetisk organisme er fusjonen mellom mekaniske og levende biologiske deler. En tverrfaglig innsats fra NTNU Cyborg er i ferd med å utvikle en robot som er koblet til biologiske nevrale nettverk. Denne roboten, passende nok kalt *Kyborgen* eller *Roboten*, har som mål å ha en lukket sløyfe med toveis kommunikasjon gjennom biologiske nevroner. Nevronene, som er vokst fra enten mennesker eller rotters stamceller, kan tilpasse seg i henhold til stimuli fra mikroelektroder og skape nevrale nettverk. Det endelige målet er et system som kontinuerlig bruker målinger fra robotens miljø til å trene de kultiverte nettverkene og bruke overførte signaler fra nettverkene til å kontrollere oppførselen til roboten.

På lang sikt er planen at Kyborgen skal fungere som en maskot ved at den kjører helt autonomt gjennom gangene i Glassgården ved NTNU Gløshaugen. Dette gjøres for å øke interessen for studentene innenfor relevant forskning som bioteknologi, nevrovitenskap, ingeniørvitenskap og data- og informasjonsvitenskap. Kyborgen er nå i versjon 3.0 og er semi-autonom. Den navigerer ved hjelp av et selvprodusert kart, er motivert av sitt eget atferdssystem og opererer i henhold til egen tilstandsmaskin. For øyeblikket er roboten på "Supervisory Control" nivå 9 siden den ikke kan kjøre uten tilsyn. For å komme til nivå 10, "Full Automation", må omfattende tester bli utført. Roboten kan kontrolleres og overvåkes eksternt på en datamaskin, men fremgangsmåten for dette er ikke-triviell. Det er derfor behov for et fullverdig grafisk brukergrensesnitt med sanntidsovervåkingskapasiteter som presis styring og systematisk overvåking for å oppnå det stabile systemet.

I løpet av denne masteroppgaven ble en kommandomodul for toppnivåstyring av andre moduler utviklet og testet i robotens viktigste programvareramme som heter ROS (Robot Operating System). Tilstandsmaskinen ble konsolidert til en egen 'oppførsel'-tilstand eller modus som en rekke av andre operasjonsmoduser. Ytterligere tilstander ble lagt til tilstandsmaskinen for modusene manuell kontroll, demo og avbrudd. Et brukergrensesnitt i formen av en ny webapplikasjon ble evaluert, designet, implementert og testet for Vue og Vue CLI (Command Line Interface). Ulike reaktive funksjonaliteter for robotinteraksjon, lagret som separate Vue-komponenter, ble implementert. Disse funksjonalitetene inkluderer valg av driftsmodus, navigasjonskart som navigerer robot ved trykk og manuell kontroll ved bruk av tastatur eller joystick på skjermen sammen med video fra "Computer vision" modulen. Brukergrensesnittet gir mulighet for overvåking av informasjon som batterinivå, motortilstand, driftsmodus, tilstanden i tilstandsmaskinen og oppførselssystemets PAD (Pleasure Arousal Dominance) verdier. Det har også et kommandoverktøy som kan endre tilstand, spille av tekst til tale og styre emosjonssystemet. Kommunikasjonen mellom web app og det eksterne ROS-systemet ble også koblet sammen. De enkelte delene av modulen ble diskutert, begrensningene deres definert og forslag til fremtidig arbeid ble presentert. Fra problembeskrivelsen ble alle foreslåtte funksjoner for Kyborgens GUI (grafisk brukergrensesnitt) levert.

Preface

This Master's Thesis was written as part of the Department of Engineering Cybernetics at the Norwegian University of Science and Technology. It serves as the second part of a year-long project assignment together with the specialization assignment in the autumn of 2019. This thesis was made partly to be useful for students who want to understand the cyborg project, its history, its different components and above all, the GUI module.

It is assumed that the reader has some experience with the programming languages Python and JavaScript and has partial understanding of their concepts and syntax.

I would like to thank Associate Professor Sverre Hendseth for his expertise and accommodating approach to our meetings, which were valuable and engaging. I would also like to thank PhD Candidate Martinius Knudsen for his encouragement and facilitation. Unfortunately, because of the global pandemic this Spring 2020, access to the cyborg was almost completely prohibited. In spite of this, I would like to thank my peers working together with me on this project for managing to deliver on their respective modules. As my work was dependent on theirs, this matter was essential for me to complete my work.

Some content from the specialization project this autumn has been purposefully reused in this thesis.

Abbreviations

- AMCL - Adaptive Monte Carlo Localization
- API - Application Programming Interface
- ARIA - Advanced Robot Interface for Applications
- BSON - Binary JavaScript Object Notation
- CLI - Command Line Interface
- CSS - Cascade Style Sheets
- DNS - Domain Name System
- DOM - Document Object Model
- EiT - Experts in Teams
- GUI - Graphical User Interface
- HTML - HyperText Markup Language
- HTTPS - Hypertext Transfer Protocol Secure
- IDE - Integrated Software Environment
- IP - Internet Protocol
- JSON - JavaScript Object Notation
- LED - Light Emitting Diode
- LTS - Long Term Support
- MEA - Micro Electrode Arrays
- NTNU - Norwegian University of Science and Technology
- NoSQL - Not only Structured Query Language
- PAD - Pleasure, Arousal, Dominance (Used in emotional state model)
- ROS - Robot Operating System
- SSH - Secure Shell

- SSL - Secure Sockets Layer
- TCP - Transmission Control Protocol
- UDP - User Datagram Protocol
- UTC - Coordinated Universal Time

Contents

1 Introduction

A Cyborg, or cybernetic organism, is the union between biomechatronic and organic body parts. The NTNU Cyborg project at The Norwegian University of Science and Technology (NTNU) is in the process of creating a Cyborg interfaced with living nerve tissue. At St. Olavs Hospital, Neural cells are seeded on Microelectrode arrays (MEAs) and are cultured and trained to form biological neural networks. The MEAs contains microelectrodes which can capture activity and stimulate the networks. The neural signals are transformed into electrical signals, meaning the MEAs are the interface between neurons and electronic circuits. By connecting these signals to the robot body, real-time bi-directional data flow is enabled. The aim is to make a closed-loop system with information flowing between the robot body and the biological neural network. The goal is for the biological neurons to control simple processes in behaviour and actions of the robot body. In this system, the robot can learn and adapt as a response to electrical stimuli by observing its environment. [1] [2] [3]

Many modules have been developed for the robot body. However there is no real user interface except for directly monitoring the computer that the robot's software runs on. This thesis describes the work of creating a GUI (Graphical User Interface) module for real-time control of the robot.

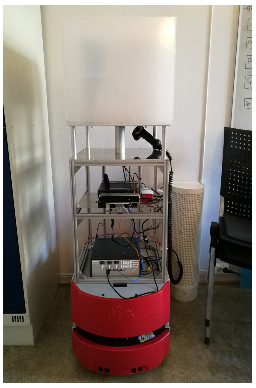
1.1 History of The Cyborg

The Cyborg Project is a collaborative, multidisciplinary effort by NTNU researchers, PhD candidates and Masters Students during specialization projects, Master's Theses and EiT (Experts in Teams) villages. It involves departments of Engineering Cybernetics, Biotechnology, Neuroscience and Computer and Information Science at NTNU. The body of the Cyborg, or just the robot, is built on top of a Pioneer LX robot base which is an autonomous, general purpose, indoor robot base with an integrated computer. The robot base is capable of autonomous navigation using built-in drivers. For the last five years, different components have been mounted on top of the robot base, creating what we now know as the Cyborg. [1]

In 2015, an EiT group fixed the skeleton frame made of aluminium on top of the base for mounting of other equipment. A Kinect module was developed over multiple Master's Theses and the following year the ROS (Robot Operating System) architecture was adopted as the robots main application framework. In 2017, the ZED stereoscopic camera and a developer board to handle visual computing was added. In 2018, two EiT village groups designed respectively the 3d printed casing and the LED dome. Last year the robot got a visual overhaul along with consolidation of its ROS modules.

Through many years of development, the robot is now capable of running autonomously through the halls of the school. It has a built-in state machine and behaviour system which are administered with a controller to decide where to navigate and how to behave. Its LED lights can display text, animations and show an interpretation of recorded MEA data. Some notable iterations are highlighted in the figure below.

(a) 2016:
Skeleton frame


(b) 2017:
Selfie stick, Kinect

(c) 2018:
3D printed casing

(d) 2019:
Dome, paint-job

1.2 Motivation

The motivation for the Cyborg project is to increase interest for students in advanced research fields like engineering cybernetics, neuroscience, computer- and information science and biotechnology. The Cyborg is meant to roam the hallway called Glassgården at Elektrobygget, NTNU Gløshaugen while interacting with students and visitors as a mascot. The Cyborg has huge potential to coordinate students from a wide area of research fields. From AI to philosophy, the hope is for the project to create a platform for interdisciplinary collaboration. Following this idea, the hope is that working on the GUI module that utilize most of the features on the robot may be appealing for many students.

Many modules built for the robot like the Kinect module have become deprecated over the years after additional testing. The robot itself is presentable with the new design and has made great improvements in its underlying systems of navigation, behaviour, visualizations and audio. However there is a need for thorough testing to ensure modules are fully operational and stable when the robot is in its working environment. A GUI would be a valuable aid in this testing process as it could conserve project resources as well as be the platform for future user interfaces with the robot.

Currently the only user interface, other than selecting operation mode at robot startup with the mode selector box, is via the CLI (Command Line Interface). Operating the robot requires accessing the robot directly by typing commands into the CLI via remote desktop tools like TeamViewer. This procedure is non-trivial and requires considerable insight into the specific commands of the robots libraries and tools. A full-fledged GUI with real-time administrative capabilities to replace this procedure has therefore been motivated.

1.3 Goal

The Cyborg Project has major research goals within technological, biomedical and philosophical areas. Through the research, they expect to achieve major conceptual and methodological advances in these areas. For the robot body this is more of an ambition or effect than the specific goal. As it stands, the robot is not connected to the biological neural networks and is not fully autonomous. Before interfacing with the MEAs, the first objective then becomes making the robot fully autonomous and reliable. Mica R. Endsley and David B. Kaber formulates their own Levels of Automation or LOA by designating the degree of human operator and computer control of dynamic control tasks. As it stands, the current robot is at Supervisory Control level 9 when running in its 'behaviour' mode of operation. This means it generates its own options, selects the option to implement and executes the selected action. To reach level 10, Full Autonomy, additional testing is required as previously stated in the motivation. When the robot becomes fully autonomous without the need for intervention, the goal of the Cyborg v4.0 will be achieved.[1] [4]

At level 9 the robot requires remote monitoring, the tools for this task are in the present circumstances not sufficient. The main goal becomes to create the GUI module with all features specified in the problem description. Additional requirements and goals for the GUI module are specified in their relevant implementation sections.

1.4 Related Work

1.4.1 Ongoing Work

Navigation System

Lasse Göncz aims to re-implement the navigation system on the Cyborg and optimize the localization performance. Based on the ROS navigation stack, tasks for mapping, localization, path planning and obstacle avoidance are to be re-implemented. Based on quantitative study of variance convergence in the estimated pose calculated by the Adaptive Monte Carlo Localization algorithm, the localization system is to be improved.

The Computer Vision Module

Ole Martin Brokstad aims to implement a Computer Vision system capable of detecting natural human behaviour, allowing the Cyborg to become a socially intelligent robot. A module detecting such human behaviour using CV technology on images recorded with the first generation ZED Stereoscopic camera, is to be implemented on the Jetson TX1 Developer board.

Behaviour System And Visualizations

Johanne Døvle Kalland aims to explore the possibility of using Behaviour Trees as a means to architect the task-level behaviour of the Cyborg. New visualizations and audio files to increase the presentability of the robot is to be added. The PAD emotional state model that directs the robot's behaviour is also being evaluated.

1.4.2 Previous Work

The Cyborg v3.0 - Finalizing the Foundation for an NTNU Mascot

In the Spring of 2019 Areg Babayan consolidated the modules in ROS and worked on making the Cyborg ready for demonstration. He finished the new design with new bodypaint, hardware rack and LED dome in collaboration with the EiT group. A behaviour module was implemented and the state machine was configured and tested. Babayan's work, together with [5] and [6] have been important research studies for the work in this thesis.

Part I

Background and Theory

2 | System Specifications

2.1 Hardware

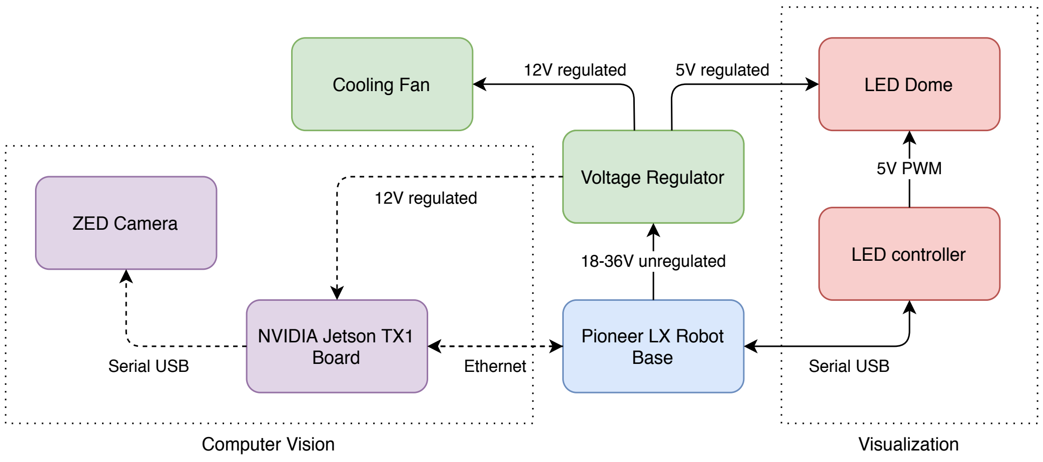


Figure 2.1: Structural overview of the robot hardware. Dotted lines means currently not connected hardware. Modified from [7].

2.1.1 Robot Base

The Pioneer LX robot by Adept MobileRobots is the robot base used for other components to build upon. It has an onboard computer capable of running either Linux or Windows, motors for turning the wheels and laser, sonar, ultrasonic and bumper sensors. Software libraries and tools for navigation and mapping of environment was also supplied. The robot base can support up to 60kg of payload. The Pioneer LX has the following hardware: [8]

- Intel D525 64-bit dual core CPU @1.8 GHz
- Intel GMA 3150 integrated graphics processing unit

- Intel 6235ANHMW wireless network adapter
- Ports for ethernet, RS-232, USB, VGA, and various other analog and digital I/O
- SICK 300 and SICK TiM 510 laser scanner, for navigation and object detection.
- Sonar sensors and a bumper panel.
- Joystick for manual control.
- A 60 Ah battery, can power the robot for up to 12 hours.
- Automated charging station, allows autonomous docking.

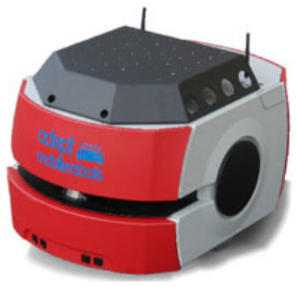


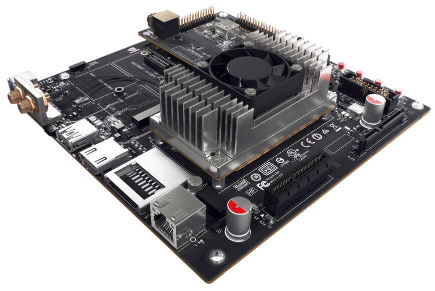
Figure 2.2: The Pioneer LX robot base.

2.1.2 ZED Camera and Jetson TX1 Developer Kit

The ZED stereoscopic 3D camera by Stereolabs shown in Figure 2.3a is used to capture 3D video along with having depth perception. It has dual 4MP (Mega Pixel) cameras, is the worlds fastest depth camera and has third party integration with ROS. The Jetson TX1 Developer kit shown in Figure 2.3b is a fully featured tool for visual computing. Along with the ZED camera it is used in the Computer Vision module to recognize human behaviour. The TX1 has the following hardware specifications. [9] [10]

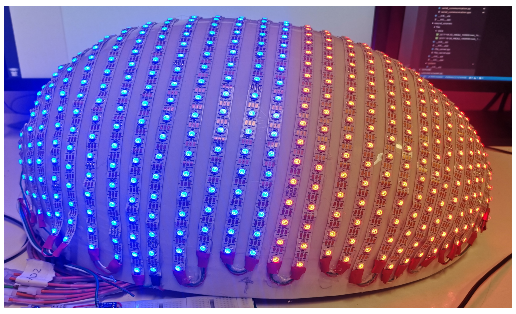
- NVIDIA Maxwell^{TM} GPU with 256 NVIDIA CUDA Cores.
- Quad-core ARM Cortex-A57 MPCore Processor.
- 4 GB LPDDR4 Memory.
- Ports for Gigabit Ethernet, USB 3.0 Type A, HDMI and other I/O.

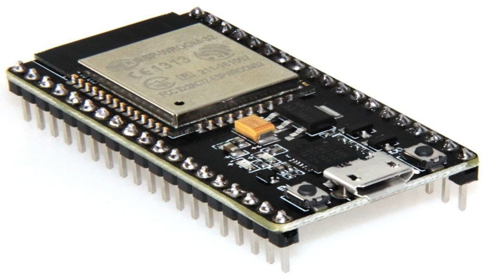
(a) ZED Stereoscopic camera.


(b) Nvidia Jetson TX1 board.

2.1.3 LED Dome

The LED (Light Emitting Diode) dome made of a plastic dome with WS2812B LED strips attached to the surface and another see-through matte frosted plastic materia dome on top. It is used to display visualizations of neural data, text and other animations. The LEDs are controlled by the LED Controller, which consists of a NodeMCU ESP-32S and a 74VHCT125A buffer. The ESP-32S is a development board with a 32-bit dual-core running at 240MHz which is considered extremely fast for a microcontroller. The buffer amplifies the 3.3V signal the ESP-32 provides to a 5V PWM signal the LED strips need. The ESP-32S is connected to the Pioneer LX via USB (Universal Serial Bus). The LED dome and NodeMCU ESP-32S is pictured below. [11] [12]


(a) LED Dome.


(b) NodeMCU ESP-32S.

2.2 Software

2.2.1 Robot Base Software

Althought Adept MobileRobots shut down in 2018, software running on the robot base is still available on the internet. The Pioneer LX robot was delivered with the following preinstalled software: [8]

- **ARIA:** The *ARIA* (Advanced Robot Interface for Applications) library is the core development library or SDK that is used on the robot. It is an open source C++ library used for high-performance access to the robot sensors and effectors. The *ROSARIA* library is used to interface between ARIA and ROS.

- **ARNL:** The *ARNL* (Advanced Robot Navigation and Localization) library by MobileRobots is a development kit for localization of navigation. The ROSARNL library was used to interface with ARIA, but because of its limitations it was switched out for RosAria and the probabilistic localization system AMCL (Adaptive Monte Carlo Localization). [13] [14]

- **Mapper3:** *Mapper3* converts and edits maps for use with ARNL and MobileSim. It was switched out for the gmapping package for creating the map of Glassgården. [14]

- **MobileSim:** *MobileSim* is the simulator for the Pioneer LX robot. ARIA will connect to MobileSim instead of the real robot if MobileSim is running. It uses the generated map (map of Glassgården) to simulate the physical environment the robot will operate in.

- **MobileEyes:** *MobileEyes* is a legacy GUI(Graphical User Interface) software for remote teleoperation and parameter configuration. It is currently a deprecated package.

3 | Background and Theory

This chapter gives some background information on the technologies used on this thesis. Developing the GUI requires knowledge in every facet of the cyborg robot's main software framework, ROS. Section 3.1 introduces ROS and gives relevant information about its concepts, tools and useful commands. The web app was hosted on Azure Cloud Services. In section 3.5 Azure and its services are explained. Lastly, MongoDB and MongoDB Atlas concepts are briefly presented in section 3.6.

3.1 Robot Operating System

Robot Operating System - or ROS - is an assortment of tools, software and drivers for developing software for robots. The purpose of ROS is to give developers the tools to create robust, reusable, general-purpose robot software. ROS is considered an operating system because it provides operating system capabilities like hardware abstraction, low-level device control, message-passing and a package manager. It has a collaboration-oriented ecosystem of volunteers running a documentation wiki with over 3000 ROS packages and has its own dedicated Q&A website. The libraries are mainly implemented with Python, C++ and Lisp using the client libraries *rospy*, *roscpp* and *roslisp* respectively. Although these client libraries are the most popular, ROS is built for programming language independency and so can be implemented with any modern programming language.

ROS is released in distributions every six months and the release only supports the latest Ubuntu LTS (Long Term Support) version. The concept levels of ROS, the *filesystem level* and the *computation graph level* will be summarized below. ROS has a comprehensive wiki on its concepts, abstractations, packages and so on which can be found in the ROS Wiki [15].

3.1.1 The FileSystem Level

The filesystem level involves the resources that users encounter inside the workspace on disk. ROS uses *Catkin* as its build system and it is in the Catkin workspace we access the filesystem level. The figure below shows the context between the different filesystem level concepts.

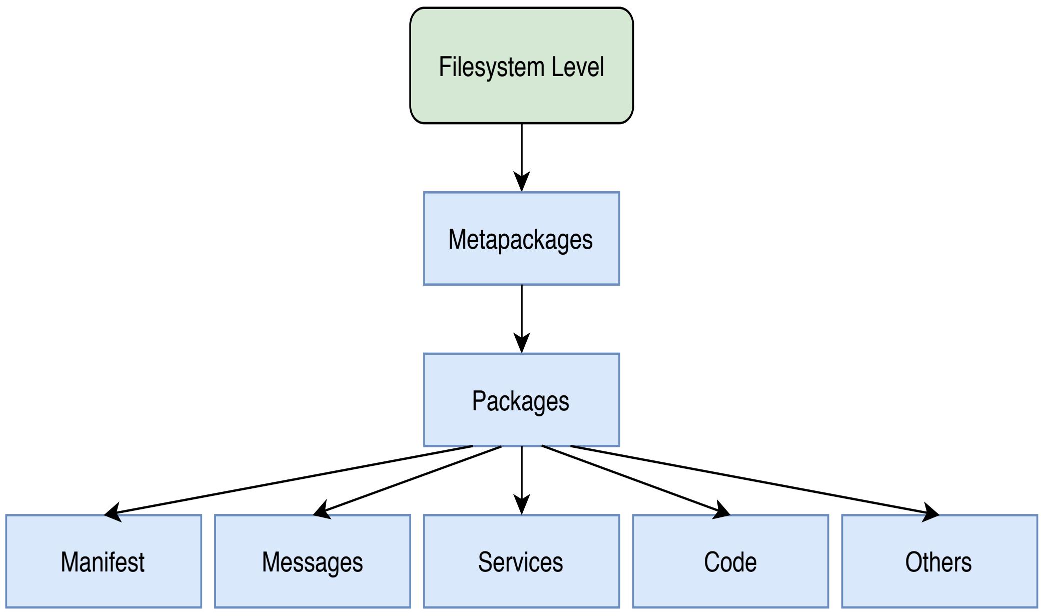


Figure 3.1: ROS Filesystem Level.

Packages

For any given module, the relevant ROS software is stored in *Packages*. A package may contain ROS nodes, non-ROS libraries, datasets, configuration files and anything else the module might need to operate successfully as a module. A package should contain enough software to be useful, but not so much that it becomes heavyweight and difficult to use with other modules. ROS packages usually have a certain folder structure and will often contain some of the following directories.

- src/package_name/: Contains source files of programs.
- include/package_name: Includes header files for C++ nodes of other packages.
- package_name/msg/: Contains Message (msg) files.
- package_name/srv/: Contains Service (srv) files.

- package_name/scripts/: Contains executable scripts.
- package_name/CMakeLists.txt: CMake build file, input to CMake build system to build the package.
- package_name/package.xml: The package manifest, contains properties of package like name, version, authors and dependencies etc.
- package_name/README.md: Github README file that describes usage of package (specifically used in this project).

Metapackages

The representation of a group of packages is called a metapackage. The metapackage does not contain any files and do not install anything other than their package.xml (manifest). A metapackage references the related packages so that the packages can be loosely grouped. A metapackage is a placeholder for the Stack term, e.g the navigation stack.

Package Manifests

Manifests (package.xml) contains the metadata of a package which includes its name, version, description, license information, dependencies etc. The file is required to be inside every ROS package that uses the Catkin build system.

Message types

Message (msg) types are used as descriptions of the data structure of a ROS message. This makes it easy for ROS tools to generate source code for the message type in different programming languages. A *message.msg* file consists of two parts, *fields* and *constants*. Fields are defined as the data that is sent inside the message. Constants are like field descriptions except that it also is assigned a value. ROS comes with some built-in messages called *std_msgs*, but it is also possible to define custom messages.

```
# fieldtype1 fieldname1
int32 x
float32 y
float64 z=1 #Constant
```

Listing 3.1: Example of Message file.

Service types

Service (srv) types are used as descriptions of the request and response data structures of a service. A *service.srv* requires a minimum of two fields or messages, the request field and response field. An example of a custom .srv file, usually found in the subfolder srv/ of a package is shown below.

```
# request msg
string question
---
# response msg
string answer
```

Listing 3.2: Example of Service file.

3.1.2 The Computation Graph level

The computation graph level is the peer-to-peer network of ROS processes running data processes with each other. The figure below shows the concepts of the computation graph level.

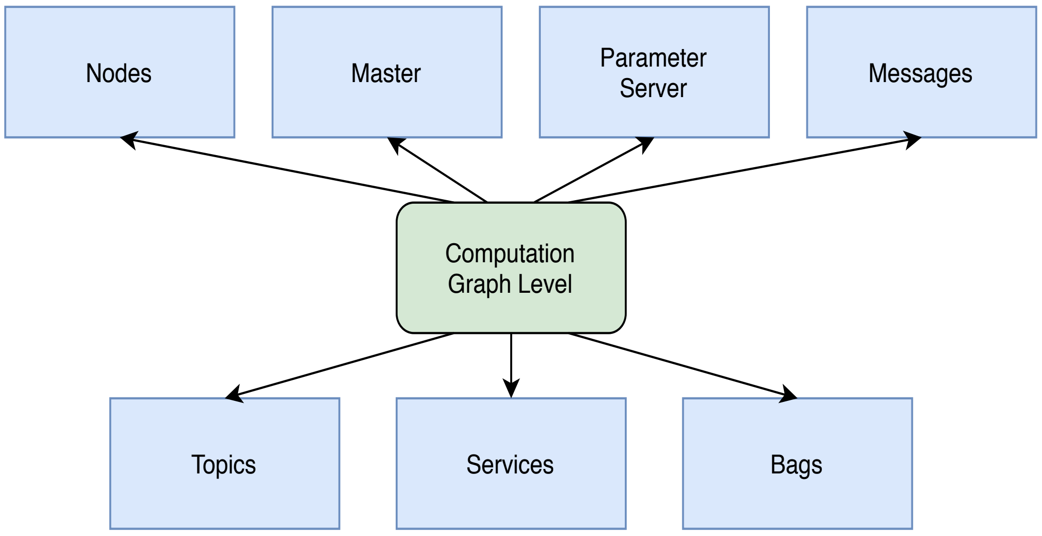


Figure 3.2: ROS Computation Graph Level.

Nodes

Nodes are the primary source of computation, i.e the Python or C++ scripts in the project. They are developed using the client libraries *rospy* and *roscpp* and are designed to be interchangeable and modular. This means that if a node is broken, only the task the node is responsible for is affected. Nodes communicate using streaming topics, RPC services and the parameter server. According to ROS guidelines a node should accomplish one task and have a descriptive name for that task. An example would be for a navigation package. One node could be used for handling the wheels, one node for the localization and another for planning the path. A ROS node for rospy is shown below:

```python
# !/usr/bin/env python
import rospy

# Initializes node, called only once for each rospy process
rospy.init_node("cyborg_nodename")

# Stops thread from shutting down.
rospy.spin()
```

Listing 3.3: Example of rospy node.

ROS Master

The ROS Master provides name registration and lookup for the Computation Graph. This allows nodes to locate each other, exchange messages over topics and make service calls. The ROS Master is started by either running the *roscore* or *roslaunch* commands. The Master is also responsible for hosting the Parameter Server. The figure below illustrates how the ROS Master registers the nodes as advertisers/publishers and subscribers.

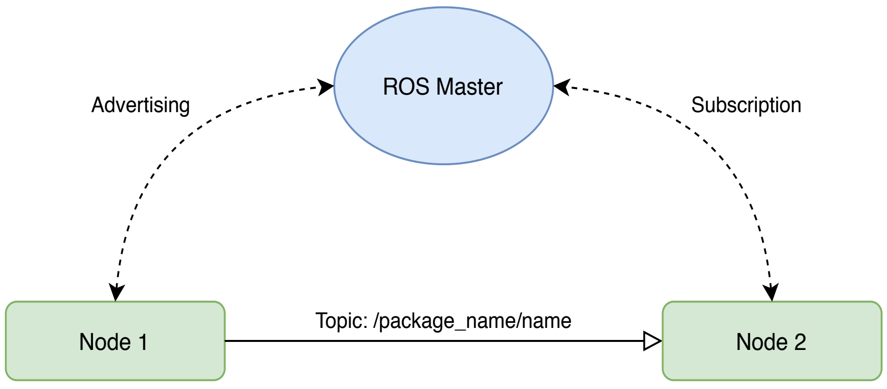


Figure 3.3: Showing how ROS Master locates nodes and allows transmission of messages over topics. This is a simplified version of the publish-subscribe pattern.

Parameter Server

The Parameter Server stores data by unique keys in a central location. It is a dictionary that is accessible to store and retrieve parameters at system runtime and is designed to hold static data like configuration parameters. The parameters are named using the ROS naming convention, just like for topics and nodes.

Messages

Nodes communicate via message-passing by publishing messages to topics. The ROS Master then routes the messages to nodes that are subscribing to that topic. A message is simply a data structure of fields, also called a message type, as explained in 3.1.1. A client calling a service in another node uses both a *request* and a *response* message as part of a ROS service call. Messages also follow standard ROS naming conventions, e.g the built-in ROS message file located in std_msgs/smg/String.msg is called *std_ msgs/String.*

Topics

Messages are passed between nodes via a transport system called topics which uses publish and subscribe semantics. A topic has a name that is used to identify the data of the topic. The publisher node acts as a server, sending the data to the relevant topic sporadically or periodically to be consumed by nodes that subscribe to the same topic. The nodes are generally not aware of which other nodes are listening to the topic, they are just coordinated by the ROS Master and connect directly to each other via TCP/IP (Transmission Control Protocol/Internet Protocol) or UDP (User Datagram Protocol) transports. There can be multiple concurrent publishers and subscribers for the same topic.

Services

For simple request-reply communication, services are used. Services are simply a pair of messages, one for the request and one for the response. Nodes can offer up services under a name in which client nodes can send a request message to and await the response message for. Requests are best suited for non-preemptive and discrete tasks. An example would be to turn a motor on or off. It is also possible to have a *persistent* service which keeps the TCP channel open, enabling higher performance but costing robustness to changes like connection losses.

Actions

Actions can be seen as ROS services with additional features to support long-running tasks. When a service takes too long to respond, the client might want feedback on progress or to cancel the entire service request. This is where actions are better suited for the task. To execute extended, pre-emptive goals with continuous updates using actions, the *actionlib* package is used. Actions are, similarly to services and messages, defined using a .action file. The file contains ROS messages for the *goal*, *feedback* and *result* of the action. These are sent between the two nodes in question, called the *ActionServer* and the *ActionClient*.

The goal message is used to initiate the task and provide the end goal. The feedback message provides updates on progress, often at a set interval and the result message is sent back upon completion of the goal. An example task where actions would be suitable could be when navigating a robot base. The wanted pose would be the goal message, the pose along the path would be the feedback message and the actual final pose would be the result message.

Bags

Bags are used to store and play back ROS message data from topics. They are stored as files in the (.bag) format and in this format can be processed, analyzed and visualized using an assortment of other ROS tools. Bags are used to mimic nodes actually publishing messages to topics, even with the same timestamps.

3.1.3 The SMACH State Machine Library

SMACH (State MACHine) is an open source Python library for creating task-level architectures of complex robot behaviour. It is independent of ROS at its core but can be integrated with ROS using the *executive smach* metapackage. SMACH is used when the robot wants to accomplish a complex plan where all possible states and their transitions can be explicitly stated. It is not recommended for unstructured tasks or for developing low-level systems that need to be very efficient.

The library offers several different types of "state containers" other than the typical state machine container. The *Iterator* container loops through a state or states until the state succeeds. The *Concurrence* container lets two states execute simultaneously and the *Sequence* container is a state machine container where states with auto-generated transitions are executed in sequence. It is possible to get detailed introspection of these state machines visually using another ROS tool called *SMACH Viewer*. The following snippet shows how to create a simple state machine.

```python
# State
class state1(smach.State):
    # Define state initialization
    def __init__(self, outcomes=['outcome1', 'outcome2'],
                        input_keys=["input"],
                        output_keys=["output"]):

    # Define state execution
    def execute(self, userdata):
        if userdata.name == "john":
            return 'outcome1'
        else:
            return 'outcome2'


# State machine
# state2() class is omitted due to repetition.
sm = smach.StateMachine(outcomes=['outcome4','outcome5'])
with sm:
    smach.StateMachine.add('STATE1', state1(),
            transitions={'outcome1':'STATE2',
            'outcome2':'outcome4'})
    smach.StateMachine.add('STATE2', state2(),
            transitions={'outcome2':'STATE1'})
```

Listing 3.4: Creating a simple SMACH state machine.

The above listing 3.4 gives us the following state machine in Figure 3.4:

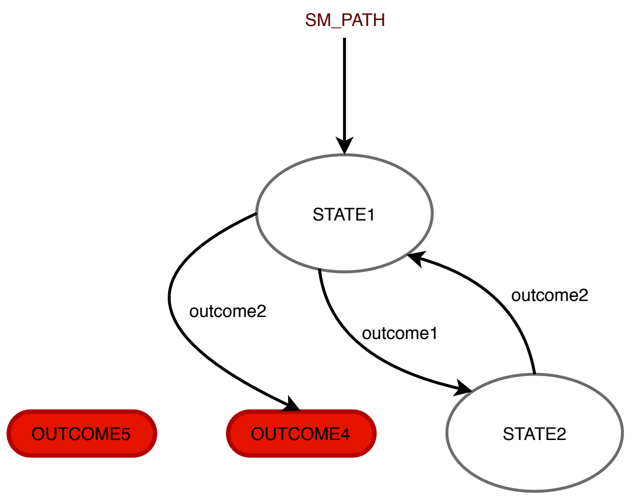


Figure 3.4: Example of SMACH state machine where the arrows are transitions.

3.2 ROS Specific Tools

This section introduces some useful and arguably necessary tools used in the cyborg robot ROS workspace.

3.2.1 SMACH Viewer

The SMACH Viewer visualizes possible SMACH state transitions or outcomes, the active states and data values passed between different states. The viewer is an introspection interface to the state machines that displays either a directed graph view similar to Figure 3.4 or alternatively a tree view.

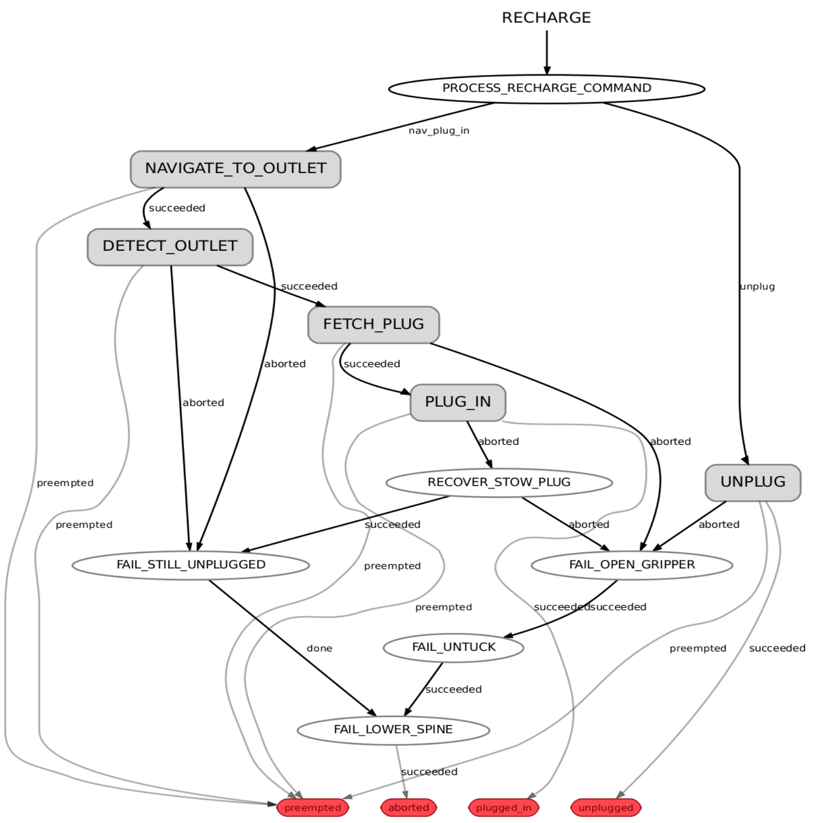


Figure 3.5: Example of SMACH Viewer directed graph. Image courtesy of [16].

3.2.2 RViz

RViz (ROS Visualization) is a 3D visualization tool for visualizing robots in simulated map environments. It has a wide array of plugins that leverage the ROS display types which are ROS compatible interpretations of robot hardware data like laser scans, odometry data, pointcloud data and generated maps. Because RViz supports most standard ROS message types used in navigation, visualization, geometry and sensors, RViz can visualize data from topics regardless of what robot model is used.

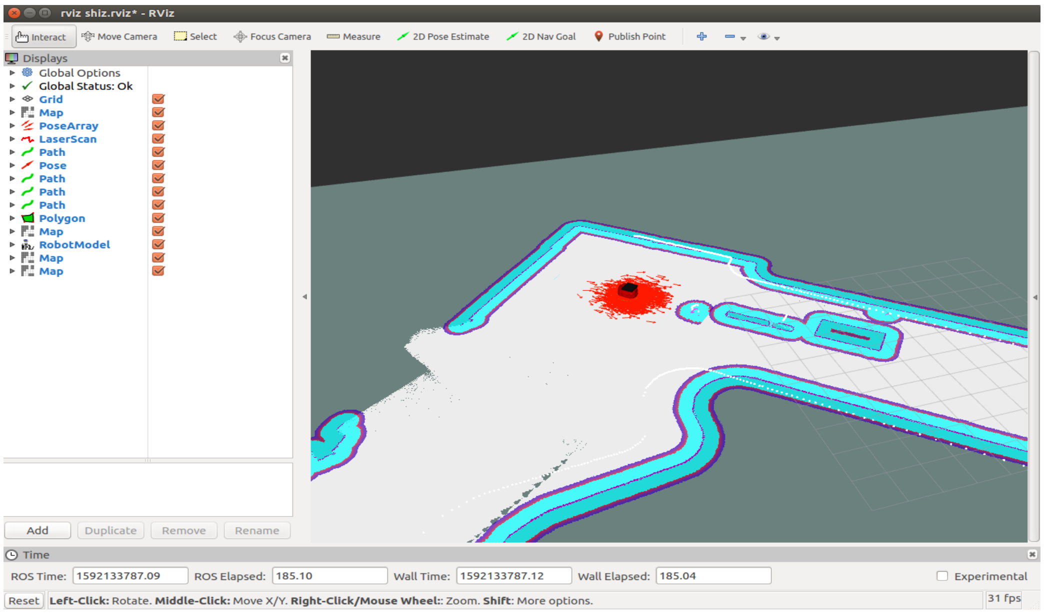
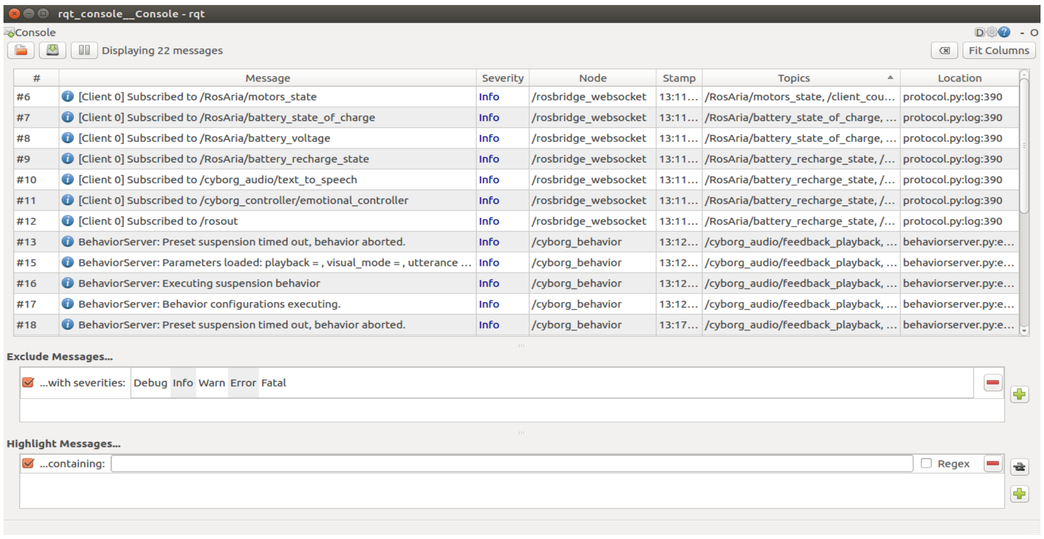


Figure 3.6: RViz showcasing its various display types like costmaps and model of the cyborg robot at runtime.

3.2.3 RQT Console

Part of the rqt commons plugins, rqt console provides a GUI for displaying and filtering of runtime messages. The GUI in combination with the *rostopic* terminal command is useful for targeting bugs in source code.



Displaying 22 messages

#	Message	Severity	Node	Stamp	Topics	Location
#6	[Client 0] Subscribed to /RosAria/motors_state	Info	/rosbridge_websocket	13:11...	/RosAria/motors_state, /client_cou...	protocol.py:log:390
#7	[Client 0] Subscribed to /RosAria/battery_state_of_charge	Info	/rosbridge_websocket	13:11...	/RosAria/battery_state_of_charge, ...	protocol.py:log:390
#8	[Client 0] Subscribed to /RosAria/battery_voltage	Info	/rosbridge_websocket	13:11...	/RosAria/battery_state_of_charge, ...	protocol.py:log:390
#9	[Client 0] Subscribed to /RosAria/battery_recharge_state	Info	/rosbridge_websocket	13:11...	/RosAria/battery_recharge_state, /...	protocol.py:log:390
#10	[Client 0] Subscribed to /cyborg_audio/text_to_speech	Info	/rosbridge_websocket	13:11...	/RosAria/battery_recharge_state, /...	protocol.py:log:390
#11	[Client 0] Subscribed to /cyborg_controller/emotional_controller	Info	/rosbridge_websocket	13:11...	/RosAria/battery_recharge_state, /...	protocol.py:log:390
#12	[Client 0] Subscribed to /rosout	Info	/rosbridge_websocket	13:11...	/RosAria/battery_recharge_state, /...	protocol.py:log:390
#13	BehaviorServer: Preset suspension timed out, behavior aborted.	Info	/cyborg_behavior	13:12...	/cyborg_audio/feedback_playback, ...	behaviorserver.py:e...
#15	BehaviorServer: Parameters loaded: playback = , visual_mode = , utterance ...	Info	/cyborg_behavior	13:12...	/cyborg_audio/feedback_playback, ...	behaviorserver.py:e...
#16	BehaviorServer: Executing suspension behavior	Info	/cyborg_behavior	13:12...	/cyborg_audio/feedback_playback, ...	behaviorserver.py:e...
#17	BehaviorServer: Behavior configurations executing.	Info	/cyborg_behavior	13:12...	/cyborg_audio/feedback_playback, ...	behaviorserver.py:e...
#18	BehaviorServer: Preset suspension timed out, behavior aborted.	Info	/cyborg_behavior	13:17...	/cyborg_audio/feedback_playback, ...	behaviorserver.py:e...

Exclude Messages...

✓ ...with severities: Debug Info Warn Error Fatal

Highlight Messages...

✓ ...containing: ☐ Regex

Figure 3.7: Rqt console displaying messages from robot at runtime.

3.3 Useful ROS Commands

There are many command-line tools defined in the ROS wiki, some are more useful than others. The table below contains the most used commands in the work of this thesis as well as their descriptions. For further information see the ROS wiki [15].

	Command	Description
Common :	*roscd*	Directs to a ROS package directly.
	rosclean	Cleans up filesystem (logs).
	rosdep	Installs system dependencies for a package.
	roslaunch	Launches sets of nodes from the XML configuration file. Used in this project.
	rosmsg	Prints out lists of info about message data structures.
	rosnode	Displays runtime node info and lets one ping nodes to check connectivity.
	rosparam	Used to get and set parameter server values. (must be YAML-encoded text).
	rossrv	Prints out lists of info about service data structures.
	rosservice	Used to call services, list services, find services and list info like type.
	rostopic	Used to echo (print message to screen) topic messages. Can publish data and display info about topics also.
Graphical :	*rqt_bag*	rqt_bag is a graphical tool for viewing data in ROS bag files.
	rqt_graph	rqt_graph displays an interactive graph of ROS nodes and topics
	rqt_plot	rqt_plot creates a plot of numerical ROS topic data.
	rqt_console	rqt_console is a GUI plugin used to display and filter ROS messages.

Table 3.2: ROS commands and their descriptions.

3.4 Web Fundamentals

This section gives a short introduction to the core concepts of web development.

- **HTML:** *HyperText Markup Language* - or HTML - is the standard Markup language for displaying documents in a web browser. HTML documents are comprised of *elements* that are enclosed in *tags*. E.g the entire document is wrapped inside an opening <html> and closing </html> tag. The title is within title tags, paragraphs in paragraph tags and so on. It is the standard markup language for formatting websites in web browsers like Mozilla Firefox and Google Chrome. Inside the document, there are two main elements, the *head* and the *body*, which make up the skeleton of every web page. The head element contains metadata like references to scripts and links to stylesheets, while the body contains the actual content of the document. HTML is currently in its fifth version called HTML5. [17]

- **CSS:** *Cascading Style Sheets* (CSS) support HTML documents with the styling and layout of a webpage. This is achieved by elements inheriting certain *values* for certain *properties* that the CSS document has declared as a *rule*. The latest version of CSS is called CSS3. [17]

- **JavaScript:** JavaScript supports HTML by use of scripts that can manipulate the page, interact with the user or send requests over the internet. JavaScript is an object oriented language where each object can have *properties*, *events* and *methods*. Properties are characteristics of the object to be changed like name or size. Events are simply different occurrences happening to objects, like a mouse click or keyboard press. Finally, instead of an object having variables as parameters it can have a function, these functions are called methods. [18]

3.5 Azure Cloud Services

Azure Cloud Services is a collection of cloud computing services from Microsoft. Azure provides basic physical or virtual computer system resources like computing power and storage remotely over the internet. The aim for the user is to reduce the cost and time of setting up on-premise IT infrastructure by leveraging the *SaaS* (Software as a Service) model. In the SaaS model all hardware and back-end is handled by the third party and the user has little to no information about hardware or software infrastructure. The user simply subscribes to the different services as desired and access their interface via the browser.

Most cloud service providers including Azure offer computing resources for web app hosting and active directories. Using Azure's integrations to other platforms like Github it is possible to identify users on hosted web apps with sign-in pages, increasing security. Other popular cloud computing services include Amazon Web Services and Google Cloud. Documentation on most of Azure's services, guides for setup and articles explaining their functionalities can be accessed in [19].

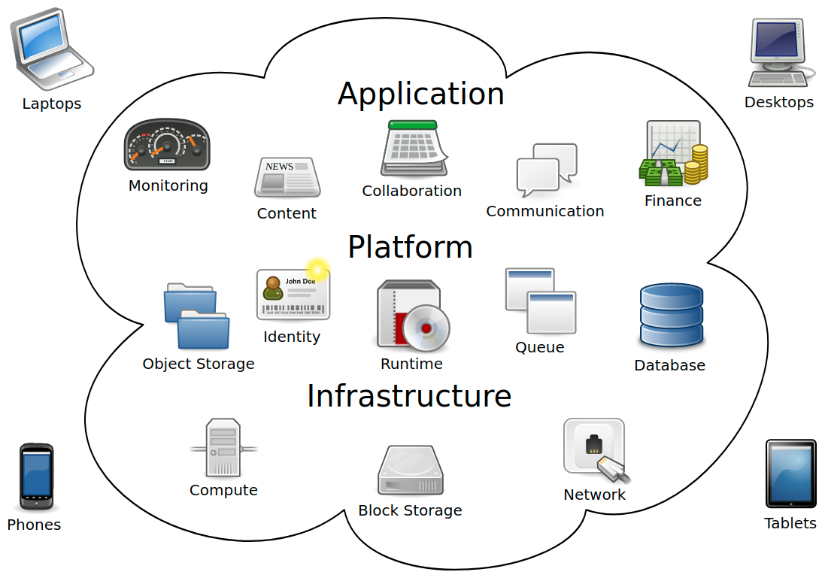
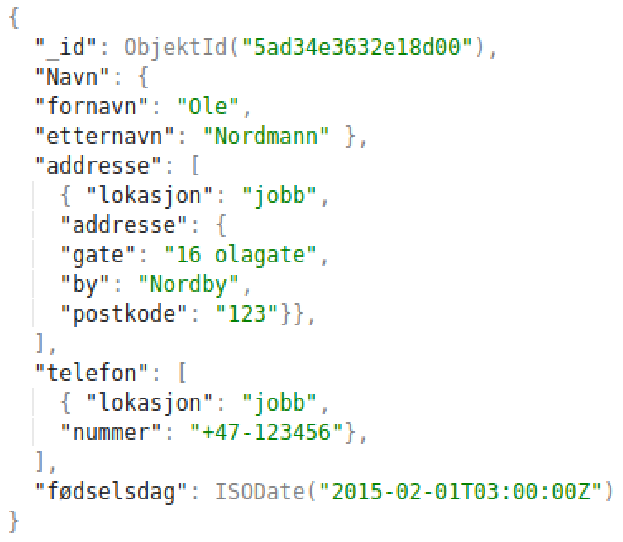


Figure 3.8: Cloud Computing visualization.

3.6 MongoDB and MongoDB Atlas

In the specialization project, MongoDB Atlas was implemented to store historical message data of the robot. MongoDB is a document-based NoSQL (Not only SQL or Structured Query Language) database as opposed to a relational database. This means it stores information in *documents* containing sets of ordered key-value pairs. MongoDB store documents in BSON (Binary JavaScript OBject Notation) format. In the key-value pairs in these documents the key is simply a string that references the particular value. The value can be of data types like Int and Bool but documents can also be nested by using another document as the value.

Each document has no relationship with other documents, but contain an object id for unique identification. In each database there can be a number of *Collections*. Collections are the equivalent of a table in a relational database and a document is equivalent to a row. Collections do not have to be created beforehand, simply inserting a document will create the collection. For more information regarding MongoDB, refer to manuals for MongoDB and MongoDB Atlas [20] [21]. An example of the format for a document in MongoDB is shown below. [22]



```
{
  "_id": ObjektId("5ad34e3632e18d00"),
  "Navn": {
  "fornavn": "Ole",
  "etternavn": "Nordmann" },
  "addresse": [
    { "lokasjon": "jobb",
    "addresse": {
    "gate": "16 olagate",
    "by": "Nordby",
    "postkode": "123"}},
  ],
  "telefon": [
    { "lokasjon": "jobb",
    "nummer": "+47-123456"},
  ],
  "fødselsdag": ISODate("2015-02-01T03:00:00Z")
}
```

Figure 3.9: Example of MongoDB document.

Part II

Implementing the GUI Module

4 | ROS Commander Module

The commander module is the top level coordinator of all other ROS nodes. Its purpose is to handle *mode of operation* requests like manual control from the web app sent to the ROS system. It's also responsible for sending historic data to MongoDB Atlas, which is the cloud database created during the specialization project in Autumn 2019. The commander is *not* responsible for handling any real-time control or monitoring communication between the web app and itself, except for choosing operation mode. When a user selects the operation mode in the app, the commander should handle the appropriate controller modules for hardware components and other subsystems like the emotion system and state machine.

4.1 Preparations for New Software Structure

This section describes the necessary preparations for a new system architecture or software structure that supports the commander module, starting with an evaluation of the current ROS module structure. A new proposed system architecture based on the evaluation is presented. Following the new structure, specifications for the commander module are stated. To clarify, mentions of packages denoted with cyborg means that the package is built by students.

4.1.1 Evaluating Current Software Structure

The first aim is to create new operation modes that interfaces with modules that control audio, visualization and navigation. The current system architecture with the *cyborg controller* package being the top-level node uses *states* defined in the state machine to execute *behaviours*. At the onset there are no states defined for different modes of operation. Fortunately, the previously implemented modules are self-contained following ROS package principles. This means that the audio, visualization and navigation modules offer their functionalities as a self-contained services. On top of these services, the modules *cyborg behaviour*, *cyborg controller*, *cyborg event scheduler* and *cyborg primary states* are all responsible for executing behaviours by calling these services. These modules will referred to as the *behaviour* system. A short summary of the behaviour system is presented for clarity.

- **Cyborg Controller:** The controller module contains the state machine, emotion system and motivator. It is the top-level controller of packages cyborg- behaviour, primary states and event scheduler. It initializes the emotion system with a set of emotions listed in Appendix B.1. This emotion system uses the PAD (Pleasure, Arousal, Dominance) emotional state model and is used to influence the state machine in a dynamic manner. When the behaviour system is started the robot is in the idle state of the state machine, the motivator for the emotion system gradually decrease PAD values over time until it is motivated to execute another behaviour, e.g *astro_ language*. These behaviours are only executed when no external or scheduled events are available.

- **Cyborg Behaviour:** Contains a *behaviour.launch* file with predefined behaviours. As an action server, when cyborg behaviour is launched it creates actions for each of the behaviours in the launch file. In the launch file, the behaviours also have parameters defined for the parameter server. Most notably parameters *visual mode* (visualization module), *utterance* (audio module), *location* (navigation module) and *completion trigger*. These behavioural preset parameters define what modules the specific behaviour activates and how the behaviour is completed. The trigger can be audio playback, navigation goal or a certain amount of time passing. If a state and a behaviour have the same name, the behaviour belongs to that state and the completion of that behaviour will result in a *succeeded* transition in the state machine. If the behaviour fails or is preempted it is also handled since all states must have an abort transition predefined.

- **Cyborg Event Scheduler:** The event scheduler module makes sure the location of the robot matches the desired location of the ongoing event. If the current location doesn't match it publishes a *navigation_ schedular* event to the controller. It's also responsible of notifying the controller module if the battery is running low.

- **Cyborg Primary States:** The primary states module, similarly to the behaviour module is a ROS action server. It handles more complex behaviours for states that cyborg behaviour cannot handle like *idle*, *wandering* and *navigation_planning*.

Figure 4.1 shows a representation of the current software structure.

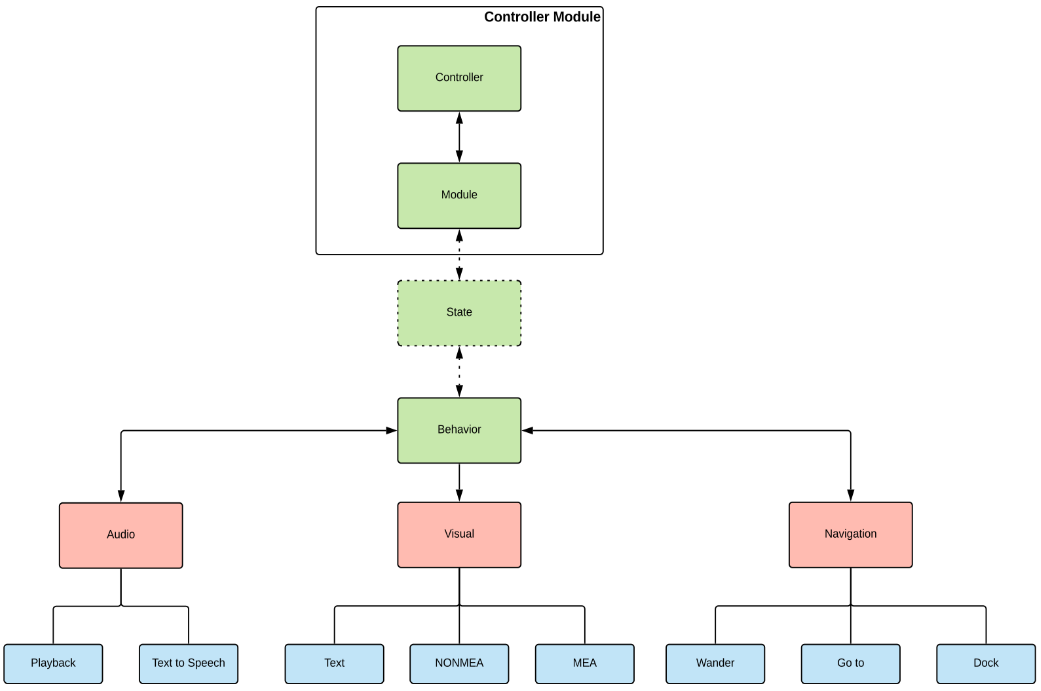


Figure 4.1: Old software structure of the cyborg's ROS system. Red boxes indicate output modules. Blue boxes indicate features of modules. Image courtesy of [7]

4.1.2 Proposed System Architecture

From the evaluation it is concluded that adding operation modes to the existing system is done by modifying the state machine and adding additional behaviour presets in the behaviour launch file. Additionally to handle the behaviour modules a commander module must be added. The author found that the overall software structure can remain since modules in the behaviour system all can effectively be put on hold when the commander requires them to. After some modifications to the controller module, the commander can be built on top of the existing behaviour system. The following figure presents the new proposed structure for the robot.

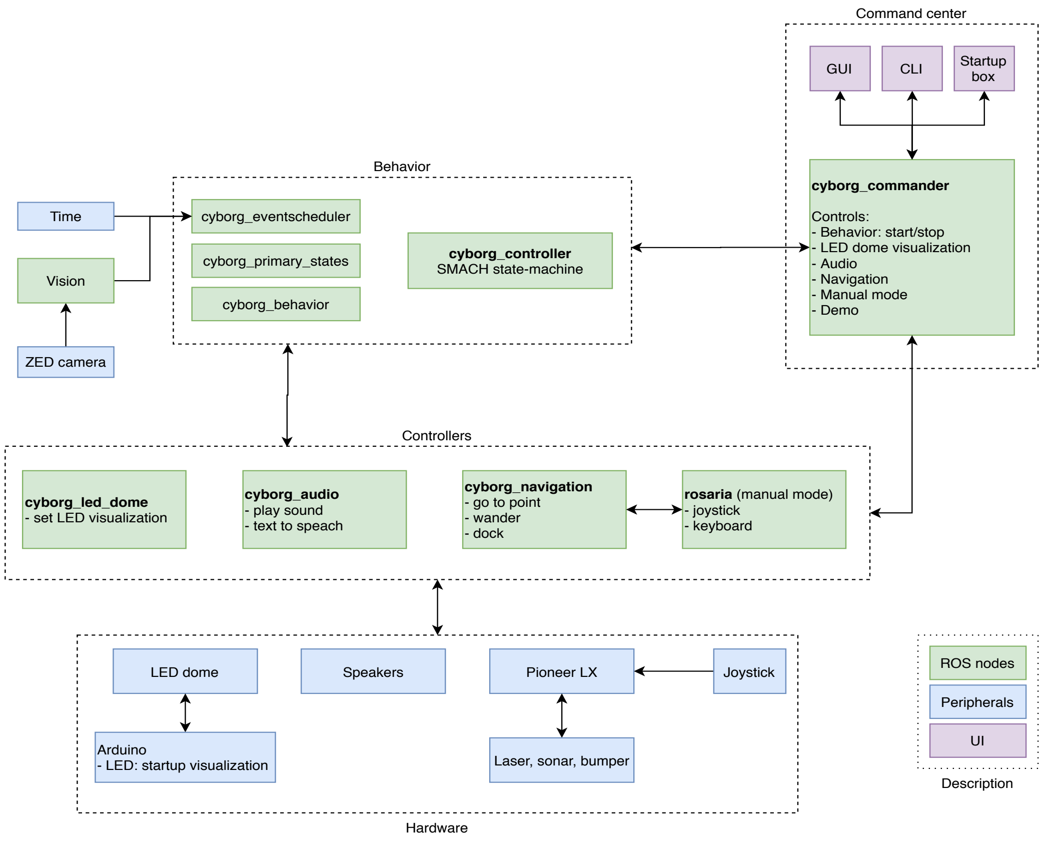


Figure 4.2: Proposed ROS system architecture of the cyborg.

4.1.3 Specifications and Requirements for Commander Module

From the proposed system architecture, the modes of operation are determined to be *manual control*, *suspension*, *demo* and *behaviour*. In effect, the current behaviour system becomes the behaviour mode of operation. The specifications and requirements for the commander module are listed below.

- The commander module must be implemented in ROS.
- Communication between other nodes and the GUI must support ROS protocols.
- The module must work with updated versions for modules stated previously in ongoing work, most importantly the re-sectionimplementation of the navigation module.
- The module must coordinate behaviour system modules when there are state transitions between modes of operation states to ensure no

conflicts. Modes of operation to handle are *suspension*, *manual control*, *demo* and *behaviour*.

- Transmission to MongoDB Atlas must support *all* existing ROS topic data types. This means it should in practice be able to receive all message types from all nodes.
- The interval of transmissions must be adjustable by use of the parameter server.
- Each topic must be inserted into its own collection in MongoDB Atlas.
- The module must support messages being transmitted either periodically or sporadically.

4.2 Commander Module Design

The design for the commander module is based on the proposed system architecture and the specifications. The commander module consists of two ROS nodes, one called *topic transmitter* and the other *topic receiver*. The topic transmitter subscribes to specific topics that are relevant for historic data storage and transmits them to MongoDB Atlas. The node has two types of topic transmitters. The first executes transmission of messages immediately and the second executes transmission of messages periodically with an adjustable interval parameter. The periodic transmitter is useful for topics with messages that arrive at too low or too high of a frequency. The topic receiver subscribes to a (for now empty) *robot mode* topic. It also publishes to every topic it requires in order to effectively control the behaviour system.

4.3 Implementing Commander Module

4.3.1 The Topic Transmitter ROS Node

When working on ROS nodes it can be advantageous to follow ROS best practices. The best practices specify tips for using roslaunch, standard units of measurements and how to the robot system among others and is highly encouraged. It is also encouraged to follow a Style Guide or follow the style of existing code for readability. The PyStyleGuide [23] and PEP 8 [24] were used for Python code in this thesis. The ROS C++ Style Guide [25] or Google C++ Style Guide [26] can be used for C++ code. [27]

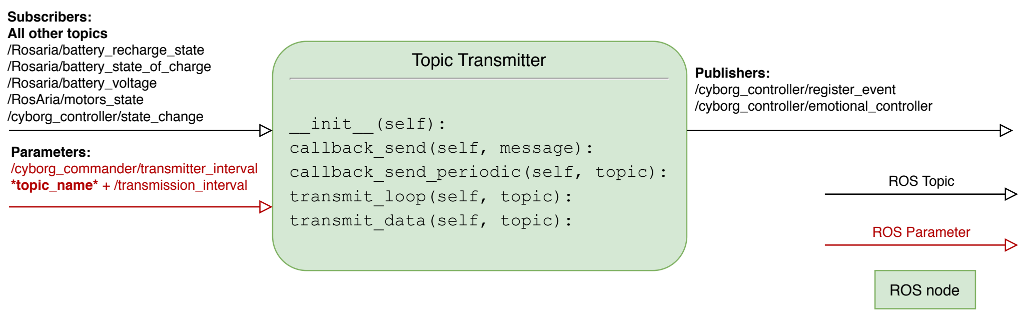

Subscribers:
All other topics
/Rosaria/battery_recharge_state
/Rosaria/battery_state_of_charge
/Rosaria/battery_voltage
/RosAria/motors_state
/cyborg_controller/state_change
Parameters:
/cyborg_commander/transmitter_interval
topic_name + /transmission_interval
Topic Transmitter
__init__(self):
callback_send(self, message):
callback_send_periodic(self, topic):
transmit_loop(self, topic):
transmit_data(self, topic):
Publishers:
/cyborg_controller/register_event
/cyborg_controller/emotional_controller
ROS Topic
ROS Parameter
ROS node

Figure 4.3: Class diagram for topic transmitter ROS node.

The topic transmitter was developed in Python as a ROS node with a single Python class called *Topic_Transmitter*. The class is instantiated by the main node called *commander*. Topic_transmitter uses *pymongo*, which is a python library to interface with MongoDB, and *rospy_message_converter* to convert ROS messages into JSON. Using the parameter server one can define transmission intervals for either specific topics using rosparam on the specific topic transmitter_interval value or on all intervals with the cyborg commander transmitter_interval topic. If both are defined then specific interval is prioritized. Every ROS topic data type is supported including ones from the re-implemented navigation module. The full list is added in Appendix B.2. The class diagram for the topic transmitter ROS node is presented in Figure 4.3. There are four methods in the topic transmitter, they are presented below.

callback_send: Callback for normal send operation. Simply gets name of topic and calls transmit_data function.

callback_send_periodic: Callback for periodic send operation. It saves the message and if there isn't a dedicated thread for the topic it creates a thread with transmit_loop as target method.

transmit_loop: Thread running in loop that executes transmit_data at a set time interval. Also checks with parameter server to see if topic transmission interval changes.

transmit_data: Converts ROS messages to JSON, adds current date and time in UTC (Coordinated Universal Time) time format. Lastly it inserts JSON data into appropriate MongoDB Atlas collections.

The following shows an example conversion as done in the transmit_data method:

```
"""ROS message"""
charging_state: 0 charge_percent: 100

"""Converted message (JSON) with date added"""
{
'date': datetime.datetime(2020, 2, 4, 14, 32, 6, 908047),
u'charging_state': 0,
'_id': ObjectId('5de7b55325eb2f36cbr8b7c5'),
u'charge_percent': 100.0
}
```

Listing 4.1: Example conversion from ROS message to JSON format.

4.3.2 The Topic Receiver ROS Node

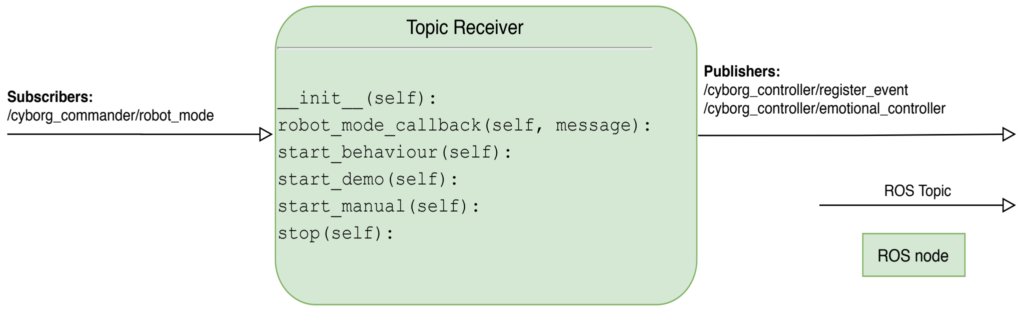


Figure 4.4: Class diagram for topic receiver ROS node.

The topic receiver was developed in Python as a ROS node with a single Python class called *Topic_Receiver*. The class, like the transmitter, is also instantiated by the main commander node. The states *startup*, *suspension*, *demo* and *manual* were added to the state machine and the entire state machine at onset was put into a state machine container. The full state machine is laid out in Appendix A.2. There are four methods in the topic transmitter, they are presented below.

robot_mode_callback: Callback method for robot mode topic. The method checks the topic of type *std_msgs/String* for what mode of operation method to call.

stop: The stop method is called by robot_mode_callback. It turns the emotion system off and puts the state machine into the suspension state. It was found that no preemption was necessary for navigation or audio as the behaviour that controls the hardware components are preempted when the linked state in the state machine is aborted. Because of the circumstances as stated in the preface, preemption of visualizations could not be tested.

start_behaviour: Start_behaviour called if "behaviour" is sent to the robot mode topic. It turns the emotion system on, and puts the state machine into the idle state inside the behaviour state machine container.

start_demo: The start_demo method turns off the emotion system and puts the state machine into the demo state.

start_manual: Similar to start_demo, the start_manual method turns off the emotion system and transitions the state machine to the manual state.

The next step was to link the operation mode states with appropriate behaviours. The following code was appended to the behaviour module's launch file.

```
<rosparam param="startup">
    visual_mode: "startup"
    completion_trigger: "time 15"
</rosparam>
<rosparam param="suspension">
    visual_mode: "suspension"
    dynamic: True
</rosparam>
<rosparam param="demo">
    visual_mode: "demo"
    completion_trigger: "time 30"
</rosparam>
<rosparam param="manual">
</rosparam>
```

Listing 4.2: Behaviour presets for modes of operation.

4.4 Tests and Results

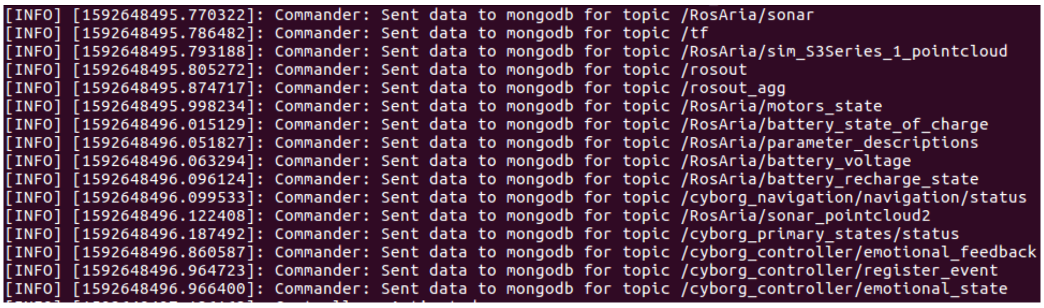
This chapter presents the testing of the topic transmitter, the topic receiver and the corresponding results. Testing the commander module was done in a simulated environment whose goal is to replicate the robot's software environment as closely as possible. In practice the systems software systems are identical except for MobileSim which acts as the physical representation of the Pioneer LX robot base. A computer running Linux distribution Ubuntu 16.04 LTS (Xenial Xerus) OS (Operating System) with the Unity Desktop environment was used. To setup the ROS simulation environment on a Ubuntu computer see appended setup script in Appendix C.1. In the case of outdated guide, refer to updated version on GitHub [28].

4.4.1 Testing Topic Data Type Support

The topic transmitter subscribes to all 98 topics who have a wide variety of data types. The goal of this test is to see what data types are supported by the message converter. The test also included testing change of transmission intervals and support for sporadic or periodic transmissions. Every topic was sent periodically with an interval of 10 seconds.

4.4.2 Result

The test shows that every topic data type listed in Appendix B.2 are supported. Transmission interval parameters for all topics were also tested. An example would be using command rosparam to set the commander module's transmission interval topic to 5, which sets the interval for all topics. Figure 4.5 shows the output log in the terminal.



```
[INFO] [1592648495.770322]: Commander: Sent data to mongodb for topic /RosAria/sonar
[INFO] [1592648495.786482]: Commander: Sent data to mongodb for topic /tf
[INFO] [1592648495.793188]: Commander: Sent data to mongodb for topic /RosAria/sim_S3Series_1_pointcloud
[INFO] [1592648495.805272]: Commander: Sent data to mongodb for topic /rosout
[INFO] [1592648495.874717]: Commander: Sent data to mongodb for topic /rosout_agg
[INFO] [1592648495.998234]: Commander: Sent data to mongodb for topic /RosAria/motors_state
[INFO] [1592648496.015129]: Commander: Sent data to mongodb for topic /RosAria/battery_state_of_charge
[INFO] [1592648496.051827]: Commander: Sent data to mongodb for topic /RosAria/parameter_descriptions
[INFO] [1592648496.063294]: Commander: Sent data to mongodb for topic /RosAria/battery_voltage
[INFO] [1592648496.096124]: Commander: Sent data to mongodb for topic /RosAria/battery_recharge_state
[INFO] [1592648496.099533]: Commander: Sent data to mongodb for topic /cyborg_navigation/navigation/status
[INFO] [1592648496.122408]: Commander: Sent data to mongodb for topic /RosAria/sonar_pointcloud2
[INFO] [1592648496.187492]: Commander: Sent data to mongodb for topic /cyborg_primary_states/status
[INFO] [1592648496.860587]: Commander: Sent data to mongodb for topic /cyborg_controller/emotional_feedback
[INFO] [1592648496.964723]: Commander: Sent data to mongodb for topic /cyborg_controller/register_event
[INFO] [1592648496.966400]: Commander: Sent data to mongodb for topic /cyborg_controller/emotional_state
```

Figure 4.5: Selection of transmitted ROS topics.

4.4.3 Discussion

After running the test several times it was concluded that all data types are converted and transmitted in a desirable fashion. All topics from the new navigation module were transmitted successfully. The change of intervals and the usage of both periodic and sporadic transmission also behaved as expected. MongoDB Atlas has however some operational limitations for the M0 Cluster or free version. The 98 topics transmitted concurrently is barely reaching the 100 operations per second limit. Another limitation to consider is the logical database size, which is fixed at 512MB. This should be sufficient for the use cases of this project if only the topics that are relevant for historic display are transmitted. Lastly the database is limited to transfer 10GB of data in and 10GB of data out per week, which should be plenty for this system.

4.4.4 Testing Modes of Operation

A simple test of changing modes of operation was conducted. The purpose of the test was not to see if modes like demo and manual control operations work but if the state machine activates the proper states and that the behaviour system is robust in different scenarios. The test involves publishing mode of operation commands to the *robot_mode* topic while also publishing to other topics which might interrupt the robot in some way. While testing, the behaviour of state transitions, preemption of navigation, preemption of audio and the resulting debug logs was observed.

4.4.5 Result

The test shows that the state machine behaves as expected for different conditions like wandering emotionally or when navigating to location. For all states in the behaviour system, the state machine correctly transitions to other modes of operation. Figure 4.7 shows an example of the state machine when publishing stop to the /cyborg_commander/robot_mode topic with data type std_msgs/String.

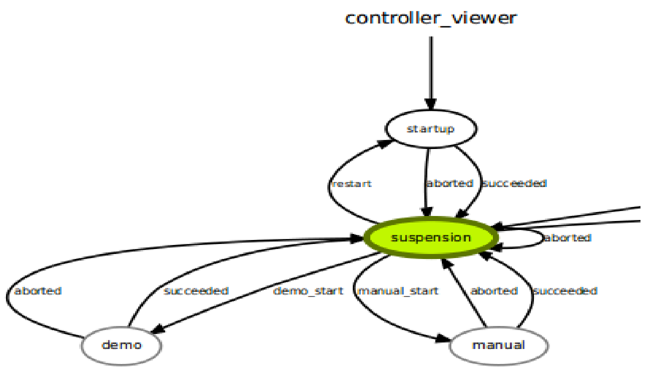


Figure 4.6: Active suspension state.

4.4.6 Discussion

The results from 4.4.5 show that the operation modes perform as expected. The startup finishes after 15 seconds, demo finishes after 30 seconds and manual control runs until otherwise told like suspension. An intuitive way of illustrating the topics for the commander's computation graph can be done by using ROS tool rqt_graph. The tool shows publish-subscribe patterns of topics and packages using directed graphs. The computation graph for the cyborg commander is displayed in Figure 4.7.

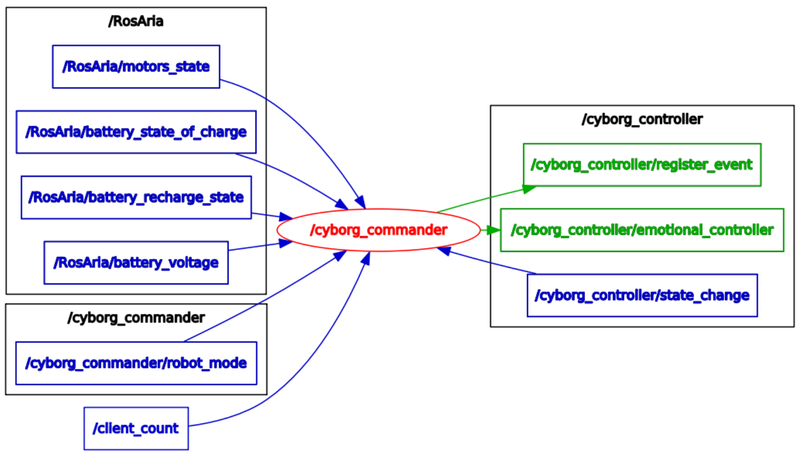

/RosAria
/RosAria/motors_state
/RosAria/battery_state_of_charge
/RosAria/battery_recharge_state
/RosAria/battery_voltage
/cyborg_commander
/cyborg_commander/robot_mode
/client_count
/cyborg_commander
/cyborg_controller
/cyborg_controller/register_event
/cyborg_controller/emotional_controller
/cyborg_controller/state_change

Figure 4.7: Computation graph illustration, boxes are topics and ovals are packages.

5 | Evaluating Web App Solutions

This chapter contain evaluations of different web app frameworks for the GUI. There are many smaller decisions to be made for each framework, however only the individual frameworks and their integration capabilities with ROS along with certain criteria are evaluated upon. The criteria for selection are substantiated in detail in chapter 11, but some noteworthy ones are rendering speed, popularity / community size, licensing, cross-platform OS compatibility, ROS-compatibility and maybe most importantly learning curve.

5.1 Choice of Framework

5.1.1 Django

Django is a server-side, cross-platform web framework for secure and rapid web application development. Django is open source under the BSD license and uses Python as its dedicated programming language. Django is oriented on solving problems with complex database-driven websites. Its principle "don't repeat yourself" says a lot about its emphasis on reusability when creating web components. As a backend (server-side) framework Django has access to a wide variety of frontend (client-side) libraries in Python to build user interfaces and includes built-in features for authentication and administration of their applications. Django could be used to make both a web app and a server to run on the robot to provide API's (Application Programming Interface) for the web app to control the robot. With Django, *roslibpy* could be used to allow remote publishing, subscribing, service calls and other ROS functionality using Python. Roslibpy uses websockets to connect to *ROSBridge Server*, part of the *ROSBridge Suite* of packages.

Figure 5.1: Django.

5.1.2 Vue

Vue (Vue.js) by Evan Vue is a client-side web framework for building powerful user interfaces and SPAs (Single Page Applications). It is an open-source, cross-platform framework that uses JavaScript as its main programming language. Its focus is on allowing developers to create reactive and composable web *components* for rapid prototyping. Its virtual DOM (Document Object Model) mirrors and tracks changes of the actual HTML DOM to re-render only elements on the page that are actually changing as opposed to the entire page. The DOM is an interpretation of the document as a tree structure used to analyse and alter HTML documents. Vue is a lightweight framework for building UIs quickly and has newly released tooling such as the Vue CLI. Vue has a shallow learning curve (easy to learn) when compared to the similar frameworks like Angular and React and arguably has better documentation. Since Vue apps are built with JavaScript, ROS can be interfaced with the web app using server libraries like ROSBridge Server or rosnodejs. To communicate with ROSBridge or rosnodejs, the core JavaScript library for interactions with ROS called *roslibjs* could be used. [29] [30] [31]

Figure 5.2: Vue.

5.1.3 React

React - or React.js - by Facebook is a cross-platform, client-side web framework for building reactive user interfaces. It is similar to Vue in that it's simply a base for building and managing Single Page Applications. It also shares the virtual DOM tree structure with Vue. However, React uses a modification of JavaScript called JavaScriptX (JSX) instead of regular HTML, CSS and JavaScript to develop the web components. Although React uses the JavaScriptX framework, external libraries for ROS integration like *roslibjs* can be used with React with some effort. Although React is easier to learn and use than Angular, mostly because of Angular's sheer size, it does however not match Vue's ease of use.

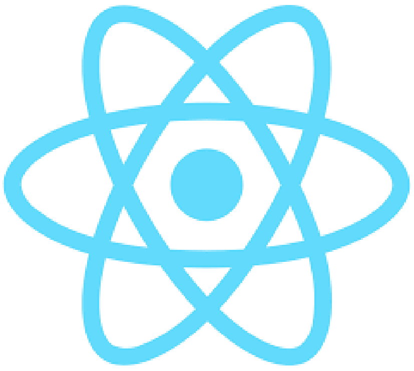


Figure 5.3: React.

5.1.4 Angular

Angular by Google is both a client- and server-side web framework that is used to build Single Page Apps using HTML and TypeScript. TypeScript is an open-source subset of JavaScript from Microsoft with safety features like static type checking that enforce functions to use certain data types like String. Angular is used in larger projects and is a more comprehensive framework that can almost be called a platform. It features, like Vue a dedicated CLI for creating and managing projects and comes with additional features "out of the box". Angular offers the most functionality and is arguably the most powerful framework but has greater complexity and requires one to learn TypeScript since most of its documentation is written for TypeScript. Angular supports roslibjs the same way as React and Vue does by using a package manager, but implementing the library requires some effort.

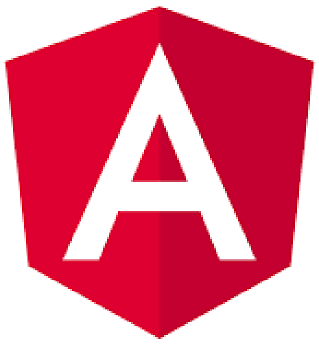


Figure 5.4: Angular.

5.1.5 ASP.NET Web Forms

ASP.NET - or ASP.NET 4.7.2 - is both a client- and server-side framework for building web apps using HTML, CSS, JavaScript and C#. It is the first programming model that was available in ASP.NET. There are different underlying frameworks available for ASP.NET like ASP.NET MVC and ASP.NET Razor Pages. ASP.NET Web Forms has an extensive collection of libraries and tooling and is run on the Visual Studio IDE (Integrated Software Environment). Web Forms features drag-and-drop functionality for creating page elements like text boxes and buttons. Web Forms runs exclusively on Windows and is licensed by Microsoft. The functionality of the page is driven by code written in the code-behind class file (*.aspx) in the C# programming language which acts as the application's server-side code. Usual ROSBridge functionality could be replaced with *ASP.NET SignalR*, a high performance server for real-time applications.

Figure 5.5: ASP.NET logo, Used with permission from Microsoft.

5.1.6 ASP.NET Core

ASP.NET Core - or .NET Core - is a free, cross-platform, client- and server-side framework and is the complete rework of ASP.NET MVC for modern web app development. Microsoft has released many underlying frameworks like ASP.NET Core Blazor, ASP.NET Core Razor pages and ASP.NET Core MVC. The new .NET Core version of SignalR could be used here aswell.

Figure 5.6: ASP.NET Core.

5.2 Method of Choice

Vue was selected as the most suitable option for framework. Although the other frameworks offer solid server-side development capabilities, the time saved from integrating roslibjs or roslibpy in the other frameworks is very valuable. Although a reactive, SPA, frontend centered Python framework would be ideal as to prevent including more than one programming language. The object-oriented programming in JavaScript is very similar to that in Python, which is not the case for C#. ROSBridge will be used as handler between ROS system and GUI.

6 | Specifications and Requirements

As a stand-alone module and possible platform for future user interfaces, the GUI should withstand scrutiny. This chapter contains the specifications and requirements necessary for the web app to perform the functionalities as described in the problem description.

1. The GUI should display information about SMACH states, robot base info like battery charge, PAD values etc.
2. The GUI must use the ROSBridge Server to publish messages to ROS topics, call ROS services etc.
   - 2.1. Topics that are subscribed and published to should be respectively unsubscribed and unadvertised to when exiting the GUI.
3. Elements of the GUI must be contained in modular components and elements should be reactive whenever possible.
4. The GUI must have the ability to send the robot to a location using a 2D map and click-to-navigate functionality.
5. The GUI must support teleop control of the robot.
6. (Optional) The Computer Vision module's videostream should be streamable to the GUI.
7. (Optional) The GUI should be able to show historic graphs of saved messages.

7 | Web App Design

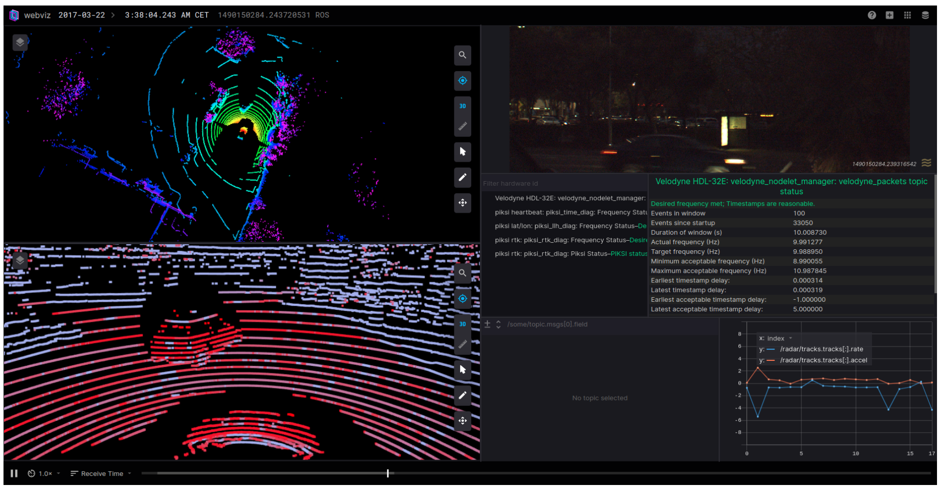
This chapter contains the design process for the web app. Inspiration was taken from other projects and are presented in subsection 7.1. To make the design process of the web components iterative, the page layout was designed using wireframes. The final wireframes for the web app are presented in subsection 7.2.

7.1 Inspirations

There were many inspirations to take from when designing and building the application. The key projects are presented in this section.

7.1.1 RobotWebTools Webviz

Webviz by Cruise Automation is part of the RobotWebTools initiative which is a collection of open-source modules to allow web applications interface with ROS. Their Webviz application allows users to drag-and-drop ROS bags to be played back live for debugging purposes. Figure 7.1 illustrates their demo website where bag files from an autonomous vehicle are played back. [32]


webviz 2017-03-22 3:38:04.243 AM CET 1490150284.243720531 ROS
1490150284.239316542
Filter hardware id
/some/topic.msgs[0].field
No topic selected
x: index
y: /radar/tracks.tracks[:].rate
y: /radar/tracks.tracks[:].accel
1.0×
Receive Time

Velodyne HDL-32E: velodyne_nodelet_manager:
piksi heartbeat: piksi_time_diag: Frequency Statu
piksi lat/lon: piksi_llh_diag: Frequency Status–De
piksi rtk: piksi_rtk_diag: Frequency Status–Desire
piksi rtk: piksi_rtk_diag: Piksi Status–PIKSI status

Velodyne HDL-32E: velodyne_nodelet_manager: velodyne_packets topic status

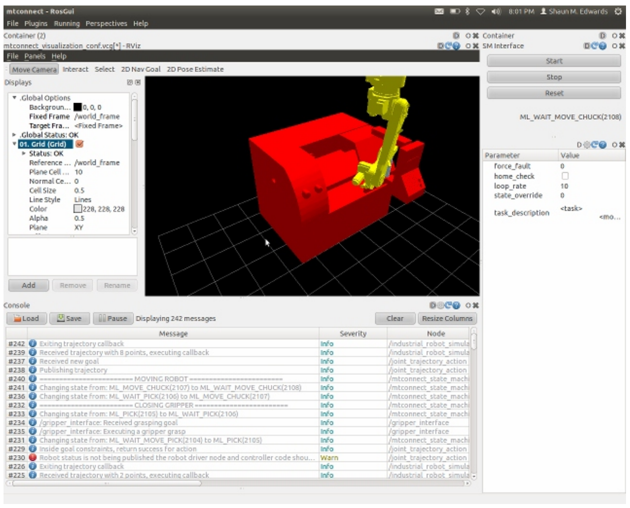
Desired frequency met; Timestamps are reasonable.

Events in window	100
Events since startup	33050
Duration of window (s)	10.008730
Actual frequency (Hz)	9.991277
Target frequency (Hz)	9.988950
Minimum acceptable frequency (Hz)	8.990055
Maximum acceptable frequency (Hz)	10.987845
Earliest timestamp delay:	0.000314
Latest timestamp delay:	0.000319
Earliest acceptable timestamp delay:	-1.000000
Latest acceptable timestamp delay:	5.000000

Figure 7.1: Webviz web-based application for playback of ROS bag files.

7.1.2 ROS Industrial

ROS-Industrial is an open-source collection of libraries, tools and drivers for robotics researchers and professionals to build industrial hardware for manufacturing companies. The program is used for advanced factory automation and is supported by the ROS-Industrial Consortium which is a world-wide membership organization with members like robotics companies ABB and ARM. The program has a wide array of projects that aim to increase productivity for industrial robots by evolving robots to handle a variety of automation tasks. [33]



ML_WAIT_MOVE_CHUCK(2108)

.Global Options	
Backgroun...	■ 0, 0, 0
Fixed Frame	/world_frame
Target Fra...	<Fixed Frame>
.Global Status: OK	
01. Grid (Grid)	✓
Status: OK	
Reference ...	/world_frame
Plane Cell ...	10
Normal Ce...	0
Cell Size	0.5
Line Style	Lines
Color	□ 228, 228, 228
Alpha	0.5
Plane	XY

Parameter	Value
force_fault	0
home_check	□
loop_rate	10
state_override	0
task_description	<task> <mo...

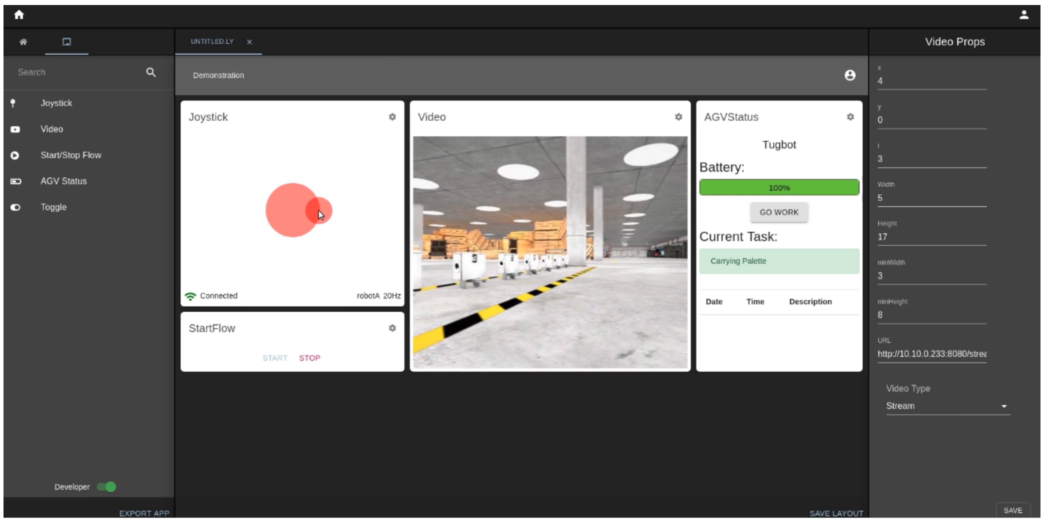
Displaying 242 messages

	Message	Severity	Node
#242	Exiting trajectory callback	Info	/industrial_robot_simula
#239	Received trajectory with 8 points, executing callback	Info	/industrial_robot_simula
#237	Received new goal	Info	/joint_trajectory_action
#238	Publishing trajectory	Info	/joint_trajectory_action
#240	========================= MOVING ROBOT =========================	Info	/mtconnect_state_machi
#241	Changing state from: ML_MOVE_CHUCK(2107) to ML_WAIT_MOVE_CHUCK(2108)	Info	/mtconnect_state_machi
#236	Changing state from: ML_WAIT_PICK(2106) to ML_MOVE_CHUCK(2107)	Info	/mtconnect_state_machi
#232	========================= CLOSING GRIPPER =========================	Info	/mtconnect_state_machi
#233	Changing state from: ML_PICK(2105) to ML_WAIT_PICK(2106)	Info	/mtconnect_state_machi
#234	/gripper_interface: Received grasping goal	Info	/gripper_interface
#235	/gripper_interface: Executing a gripper grasp	Info	/gripper_interface
#231	Changing state from: ML_WAIT_MOVE_PICK(2104) to ML_PICK(2105)	Info	/mtconnect_state_machi
#229	Inside goal constraints, return success for action	Info	/joint_trajectory_action
#230	Robot status is not being published the robot driver node and controller code shou...	Warn	/joint_trajectory_action
#226	Exiting trajectory callback	Info	/industrial_robot_simula
#225	Received trajectory with 2 points, executing callback	Info	/industrial_robot_simula

Figure 7.2: Sample GUI from ROS Industrial project.

7.1.3 MOV AI

MOV AI is a company developing an industry-grade operating system and software development framework using ROS. The use case is for autonomous, collaborative robots like robots in automated storage warehouses. MOV AI aims to shorten return of investment for companies looking to replace manual labor with automation. Their Virtual ROS Launch system replaces ROS tools like roslaunch and they provide their own IDE to make customizable GUIs. [34]



x
4
y
0
i
3
Width
5
Height
17
minWidth
3
minHeight
8
URL
http://10.10.0.233:8080/strea
Video Type
Stream

Figure 7.3: MOV AI GUI displaying their customizable widgets.

7.2 Page Layout

Simplified wireframes or page layout diagrams were made for the GUI to identify necessary components and their placements. To meet the desired functionality three separate pages were found to be required. The final versions of page layouts are laid out in Figure 7.4, Figure 7.5 and Figure 7.6.

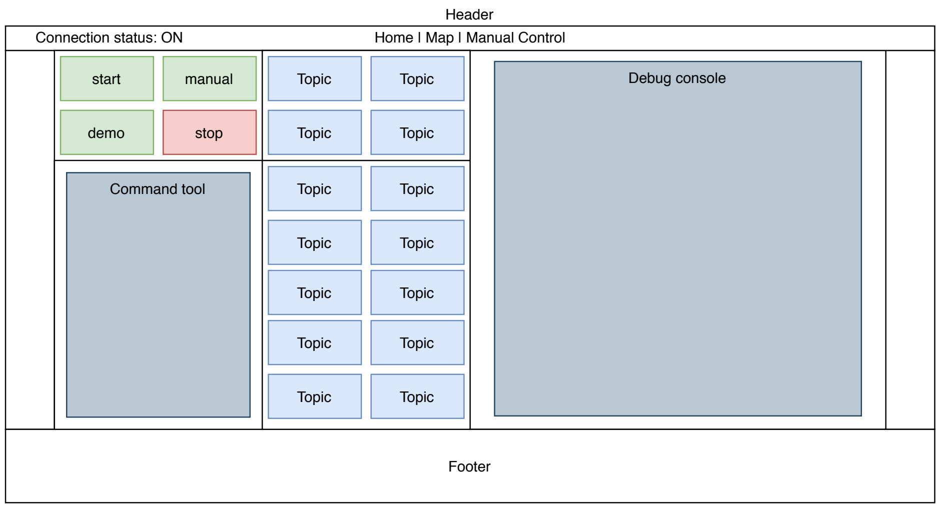


Figure 7.4: Wireframe of Home page. Header and footer elements were designed to persist through each page.

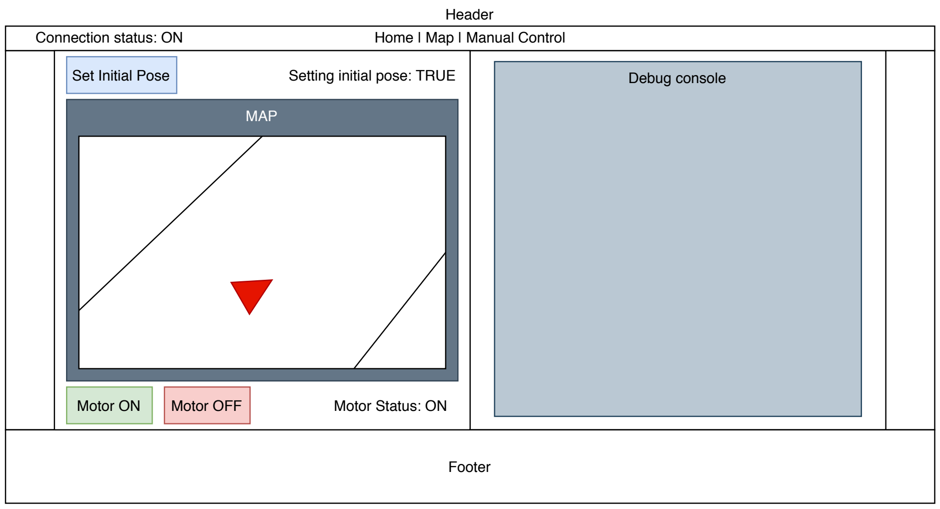


Figure 7.5: Wireframe of Map page.

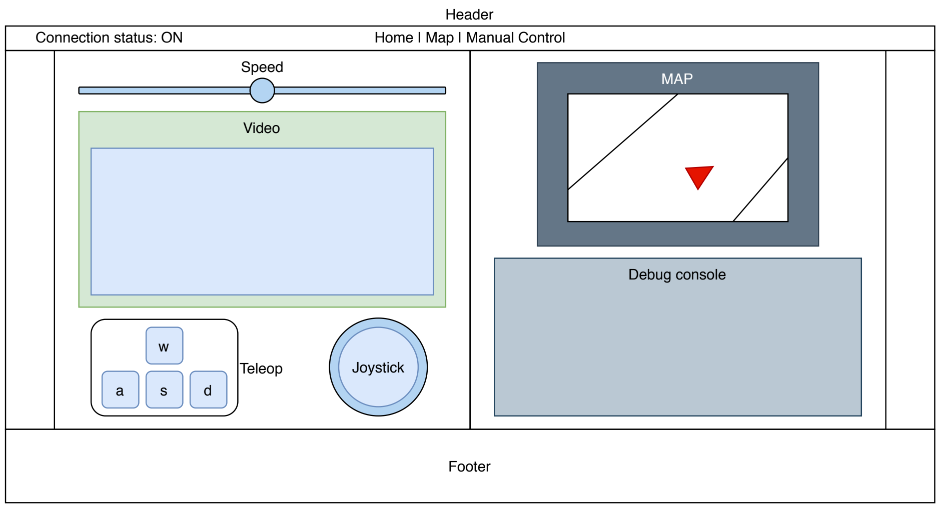


Figure 7.6: Wireframe of Manual Control page.

8 | Implementation

This chapter presents the implementation of the Web App GUI following the previous specifications and design. The chapter starts with explaining the Vue CLI and the vue folder structure in section 8.1. Section 8.2 describes single file components and how they have been been used to create modular components. The components are presented in section 8.3 and the final result of the web application is displayed at the end of this chapter.

8.1 Vue CLI

The Vue CLI is a terminal interface that can be used to quickly stage the application for development. It offers a browser UI to quickly serve, build and inspect the application locally. Vue CLI runs node.js and contains a collection of core plugins like the babel compiler or optional TypeScript. The UI also includes a tab for installing external plugins to add to the project from *npm*, which is the standard package manager Vue. [35]

8.1.1 Folder structure

Like the filesystem level of ROS, Vue has a directory structure that is commonly shared between projects.

- src/ : Contains the source code for the project.
- src/components : Single file components with .vue extension.
- src/views : Also contains .vue files, but a view is the "page" single file components used for routing.
- src/router : In a Vue SPA, pages aren't really pages but rather routed views on the same page. The router directory contains the index.js that dynamically renders the router between views.
- dist/ : Contains minified and compiled project HTML, CSS and JavaScript build content. Entire application in one minified directory.

- assets/ : All Vue assets to be imported into Vue components.
- public/ : Static assets to be copied into the project. Most important is public/index.html, which is the HTML document that the app will be built on.
- node_modules/ : Contains all JavaScript libraries and dependencies used by the npm package manager.
- package.json: Metadata for JavaScript libraries and dependencies, used by npm.

8.2 Single File Components

Using the build tool *Webpack*, it is possible to create single-file components (*.vue extension) in Vue. A component is divided into three parts. The HTML code inside *template* tags, the JavaScript code inside *script* tags and finally CSS stylesheets inside *style* tags. The example below shows homepage.vue.

```
<template>
  <div id="homepage">
    <contactForm></contactForm>
  </div>
</template>

<script>
import contactForm from "../components/page/contactForm";

export default {
    name: "homepage",
    components: {
    contactForm
  }
}
</script>

<style scoped>
#page {
  background-color: #f3f3f3;
}
</style>
```

Listing 8.1: Page view component with a contactForm component.

8.3 Implementation of Components

This section shows the implementation of individual components and their respective results. BootstrapVue was used to integrate Bootstrap into the project. Bootstrap is a front-end CSS library to organize page layouts in grid systems and build page elements such as alerts, tooltips and pop-up modals that fit on mobile devices. More information on BootstrapVue and its use cases can be found in [36].

8.3.1 Connection Status

The reactive connection status component was created using the roslibjs library to establish a connection with the ROSBridge Server locally. ROS-Bridge will be explained later in Section 10.2. Depending on the status of the connection, *loading*, *error* or *active* icons are displayed along with a descriptive tooltip. The component also contains a reconnect button for re-establishing connection. Before the user leaves the page the connection is closed using the *beforeDestroy()* Vue event. Listing 8.2 shows how the connection is established in the script tag.

```
import Vue from "vue";
import ROSLIB from "roslib";

var ros = new ROSLIB.Ros({
  url: "ws://localhost:9090"
});
Object.defineProperty(Vue.prototype, "$ros", { value: ros });
```

Listing 8.2: Establishing connection with ROSBridge and allows connection be accessed with *$ros* object.

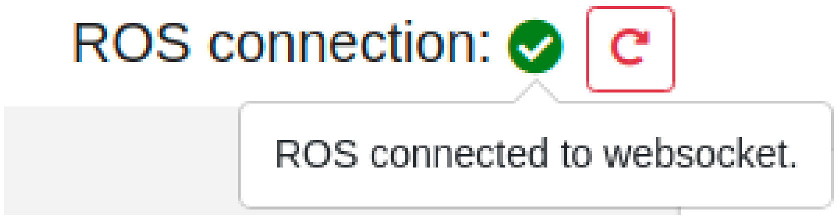


Figure 8.1: Connection status icon with tooltip and reconnect button.

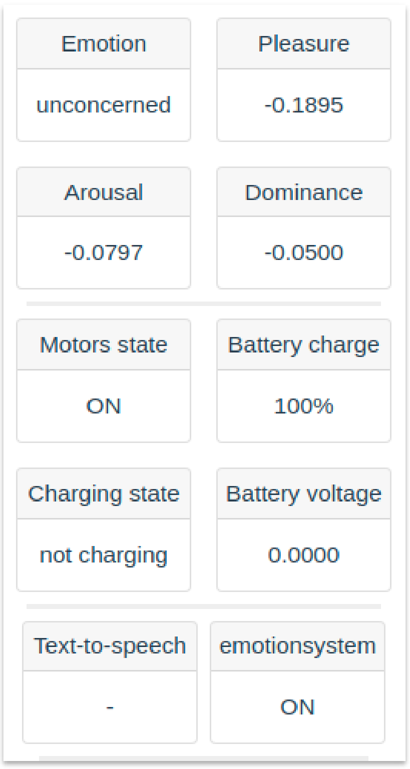
8.3.2 Cards

Three reactive card components were created. One for SMACH states called *SMACHStateCards*, another for RosAria information called *ariaCards* and another for last text-to-speech input and emotion system status called *etc-Cards*. The component uses the established ROS connection to subscribe to their respective topics. When the cards are destroyed (leaving app or changing view) the topics are unsubscribed. A snippet code for subscribing to text to speech is shown in Listing 8.3.

```
<template>
  <div id="etcCards">
    <b-card-text>{{ text_to_speech }}</b-card-text>
  </div>
</template>

<script>
import ROSLIB from "roslib";
export default {
    name: "etcCards",
    data() {
        return {
          text_to_speech: "-"
        };
    },
    created() {
        this.textToSpeechTopic = new ROSLIB.Topic({
            ros: this.$ros,
            name: "/cyborg_audio/text_to_speech",
            messageType: "std_msgs/String"
        });
        this.textToSpeechTopic.subscribe(function(msg) {
            self.text_to_speech = msg.data;
        });
    },
    destroyed() {
        this.textToSpeechTopic.unsubscribe();
    }
}
</script>
```

Listing 8.3: Reactive BootstrapVue Card element updated with text to speech messages.

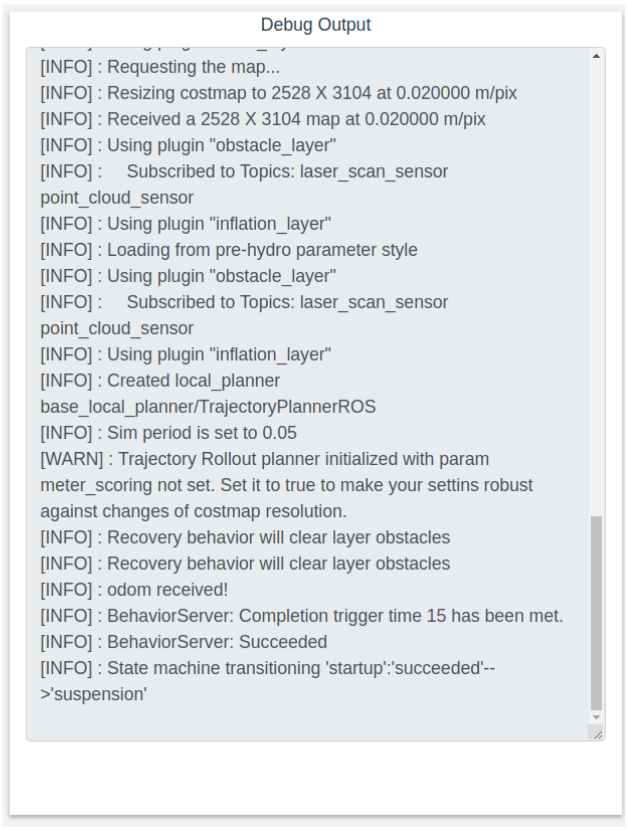


Emotion	Pleasure
unconcerned	-0.1895
Arousal	Dominance
-0.0797	-0.0500
Motors state	Battery charge
ON	100%
Charging state	Battery voltage
not charging	0.0000
Text-to-speech	emotionsystem
-	ON

Figure 8.2: ariaCards and etcCards components.

8.3.3 Debug Console

The debug console subscribes to topic */rosout* of type *rosgraph_msgs/Log*, formats the message and outputs it to a *textarea* element. The debug console component is, like the connection status, shared by import by different views. The debug console is shown in 8.3.

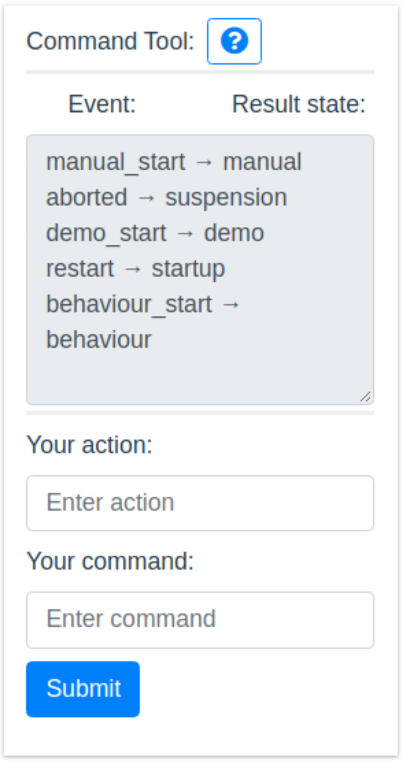


```
[INFO] : Requesting the map...
[INFO] : Resizing costmap to 2528 X 3104 at 0.020000 m/pix
[INFO] : Received a 2528 X 3104 map at 0.020000 m/pix
[INFO] : Using plugin "obstacle_layer"
[INFO] :     Subscribed to Topics: laser_scan_sensor
point_cloud_sensor
[INFO] : Using plugin "inflation_layer"
[INFO] : Loading from pre-hydro parameter style
[INFO] : Using plugin "obstacle_layer"
[INFO] :     Subscribed to Topics: laser_scan_sensor
point_cloud_sensor
[INFO] : Using plugin "inflation_layer"
[INFO] : Created local_planner
base_local_planner/TrajectoryPlannerROS
[INFO] : Sim period is set to 0.05
[WARN] : Trajectory Rollout planner initialized with param
meter_scoring not set. Set it to true to make your settins robust
against changes of costmap resolution.
[INFO] : Recovery behavior will clear layer obstacles
[INFO] : Recovery behavior will clear layer obstacles
[INFO] : odom received!
[INFO] : BehaviorServer: Completion trigger time 15 has been met.
[INFO] : BehaviorServer: Succeeded
[INFO] : State machine transitioning 'startup':'succeeded'--
>'suspension'
```

Figure 8.3: Adjustable debug console component with labels.

8.3.4 Command Tool

A command tool was re-implemented from the cyborg command package created by Thomas Rostrup Andersen in 2016 [6]. The tool displays the current available transitions or events for the state machine and their resulting states. The available actions are *event*, *speech*, *emotion* and *emotionswitch*. Given a state transition as command, the event action will put the state machine into the resulting state. Given any speakable text, the speech action will enable text to speech by using the pyttsx3 engine in the audio module to play the text as sound through the speaker. The emotion action, given emotions like angry or sad, switches the emotion of the emotion system to predefined PAD values listed in appendix B.1. Finally the emotionswitch, with commands on/ON or off/OFF turns the emotion system on or off. A guide for the tool will pop up if the *?* at the top is clicked. The guide is added in Appendix A.1.


Command Tool:

Event: Result state:

manual_start → manual
aborted → suspension
demo_start → demo
restart → startup
behaviour_start → behaviour

Your action:

Enter action

Your command:

Enter command

Submit

Figure 8.4: Command tool showcasing different available transitions/events and their resulting states. Top button opens up user manual in Appendix A.1

.

8.3.5 Navigation Map

A navigation map with click-to-navigate funtionality was added. RobotWebTool's JavaScript libraries *ros2djs* and *nav2djs* were modified to display the robot in the 2d map of its own creation in real time. With an initial press and hold with a following orientation drag and release, a goal is sent to the */move_ base* action server via ROSBridge. A picture of the robot was added and the path of the robot from topic */move_ base/NavfnROS/plan* of type *nav_ msgs/Path* was also added. The map consists of a canvas element, made interactive with JS libraries *easeljs* and *eventemitter2*. Nav2djs extends ros2djs's visualization of a map with navigational capabilities. When the robot is initialized it cannot orientate itself in the world. A button to estimate initial pose was added, instead of navigating to a location a click will set the robot's pose. For this the robot_pose_publisher package was added to the ROS system. Buttons for calling services to turn the motors on or off were also added.

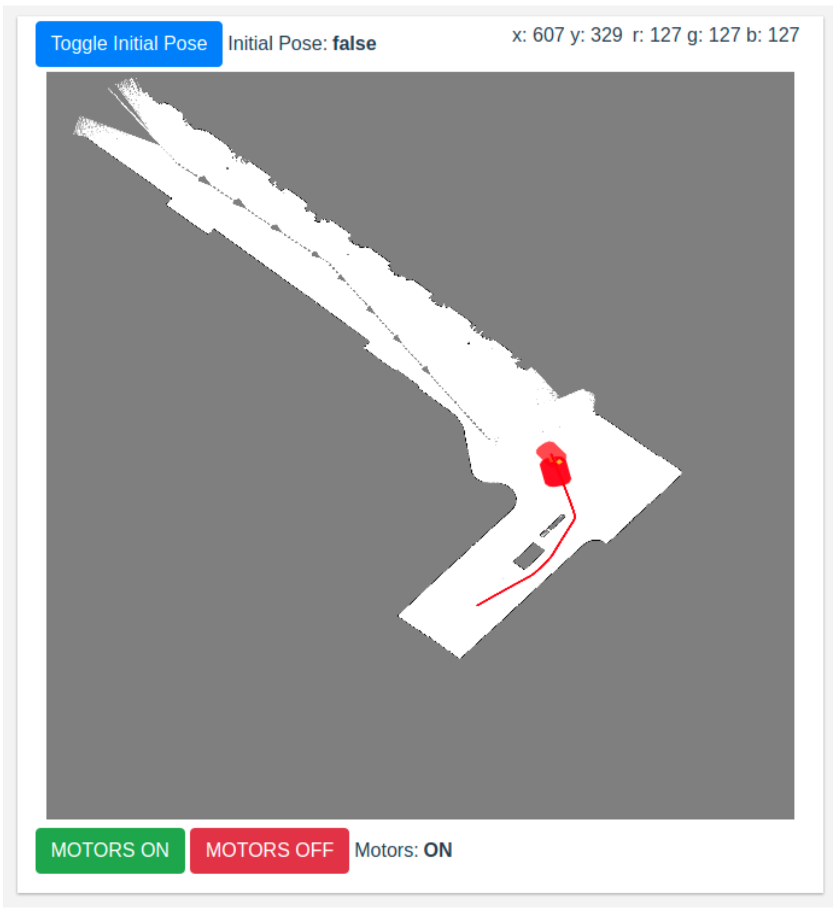


Figure 8.5: Map component with click-to-navigate functionality and with possibility to estimate initial pose. Buttons at bottom turn motors on and off. X, Y coordinates and RGB values in top right corner.

8.3.6 Manual Teleoperation and Joystick

Manual teleoperation using wasd keys or joystick were added as a component. JS libraries *keyboardteleopjs* and *nipplejs* was used for these two elements along with integration of ROS package *web_video_server* in ROS system to handle live streaming video. The component was based on code from [37]. Figure 8.6 showcases the component.

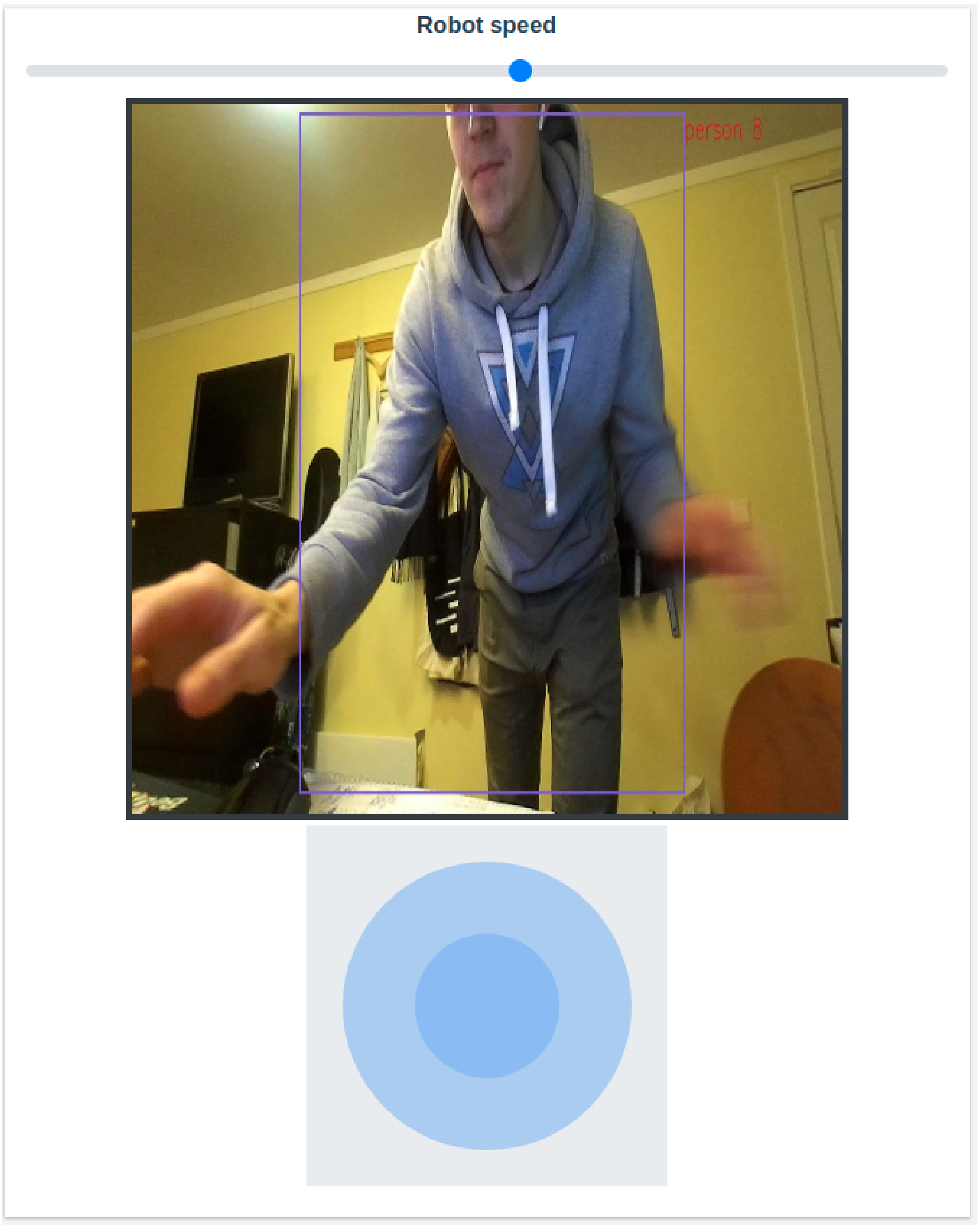


Figure 8.6: Video stream from Computer Vision Module using ZED Stereoscopic camera. Sliding bar for selection of rotation and velocity speed. Joystick for manual control in blue. Teleoperation is also optional using wasd keys.

8.3.7 Historic Graphs

Although optional, historic graphs from the database were added to the web app. The 5 topics listed in Figure 4.3 were considered to be the most relevant topics to store data from. Using *MongoDB Charts*, a visualization tool to create charts for mongoDB data in the browser, an *iframe* element that updates every minute was implemented in the web application. The historic charts were ultimately left out of the final web app as to prioritize the author's time spent developing other integrations. An example chart displayed on the web app is shown in Figure 8.7 and the final web pages are presented below in Figures 8.8, 8.9 and 8.10.

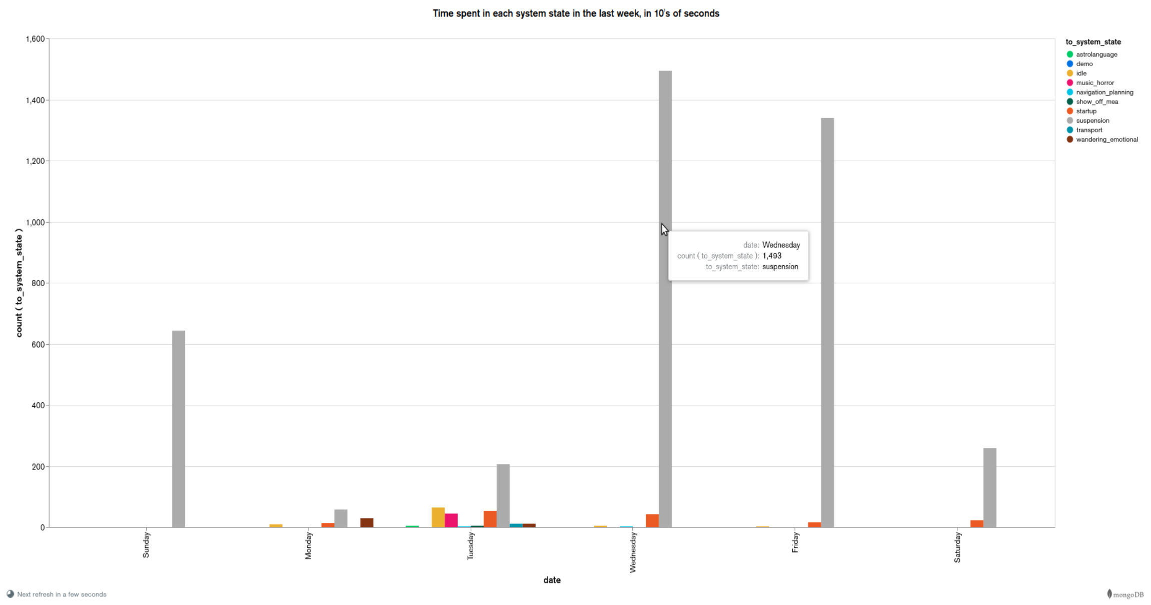
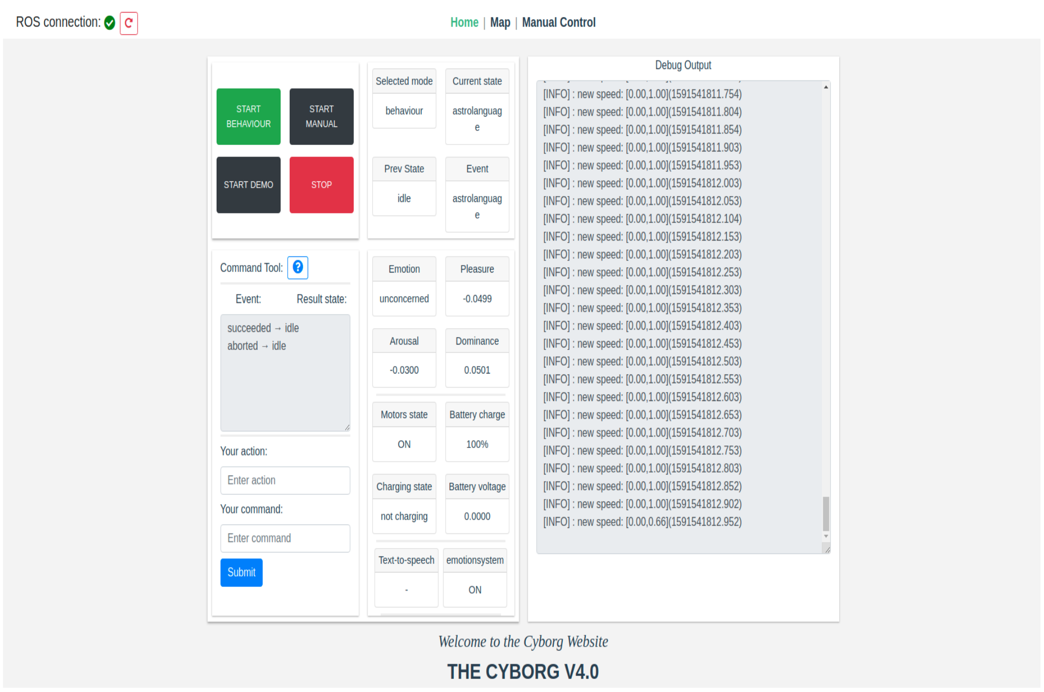


Figure 8.7: Active SMACH state for the last week in amount of 10's of seconds.



Event:	Result state:

succeeded → idle
aborted → idle

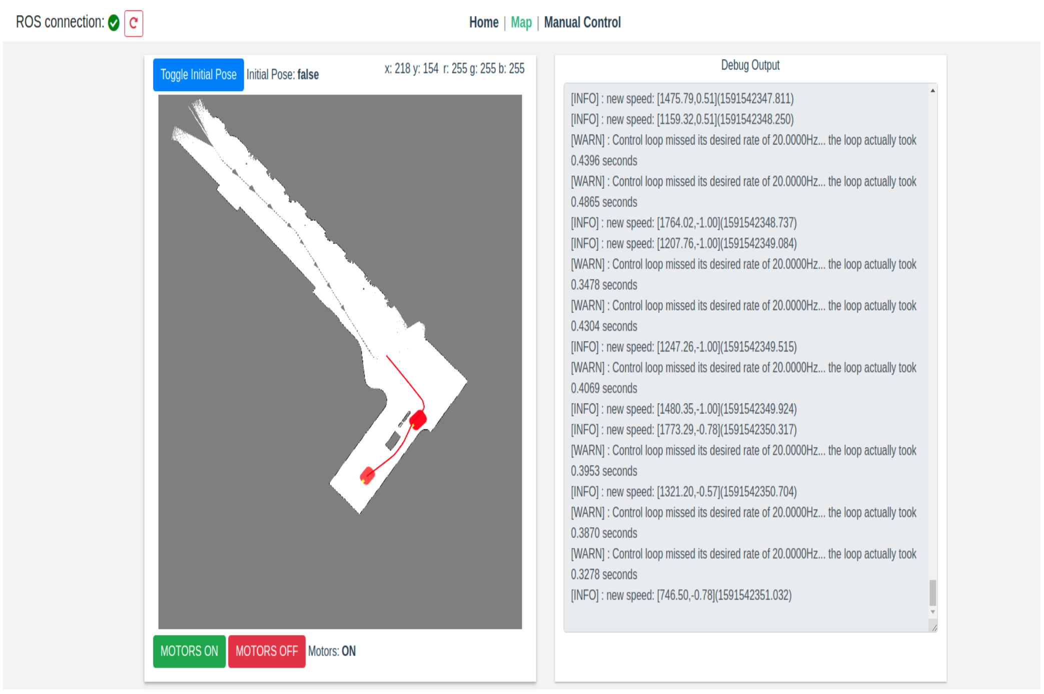
Selected mode	Current state
behaviour	astrolanguage
Prev State	**Event**
idle	astrolanguage
Emotion	**Pleasure**
unconcerned	-0.0499
Arousal	**Dominance**
-0.0300	0.0501
Motors state	**Battery charge**
ON	100%
Charging state	**Battery voltage**
not charging	0.0000
Text-to-speech	**emotionsystem**
-	ON

```
[INFO] : new speed: [0.00,1.00](1591541811.754)
[INFO] : new speed: [0.00,1.00](1591541811.804)
[INFO] : new speed: [0.00,1.00](1591541811.854)
[INFO] : new speed: [0.00,1.00](1591541811.903)
[INFO] : new speed: [0.00,1.00](1591541811.953)
[INFO] : new speed: [0.00,1.00](1591541812.003)
[INFO] : new speed: [0.00,1.00](1591541812.053)
[INFO] : new speed: [0.00,1.00](1591541812.104)
[INFO] : new speed: [0.00,1.00](1591541812.153)
[INFO] : new speed: [0.00,1.00](1591541812.203)
[INFO] : new speed: [0.00,1.00](1591541812.253)
[INFO] : new speed: [0.00,1.00](1591541812.303)
[INFO] : new speed: [0.00,1.00](1591541812.353)
[INFO] : new speed: [0.00,1.00](1591541812.403)
[INFO] : new speed: [0.00,1.00](1591541812.453)
[INFO] : new speed: [0.00,1.00](1591541812.503)
[INFO] : new speed: [0.00,1.00](1591541812.553)
[INFO] : new speed: [0.00,1.00](1591541812.603)
[INFO] : new speed: [0.00,1.00](1591541812.653)
[INFO] : new speed: [0.00,1.00](1591541812.703)
[INFO] : new speed: [0.00,1.00](1591541812.753)
[INFO] : new speed: [0.00,1.00](1591541812.803)
[INFO] : new speed: [0.00,1.00](1591541812.852)
[INFO] : new speed: [0.00,1.00](1591541812.902)
[INFO] : new speed: [0.00,0.66](1591541812.952)
```

Welcome to the Cyborg Website

THE CYBORG V4.0

Figure 8.8: Final home view.



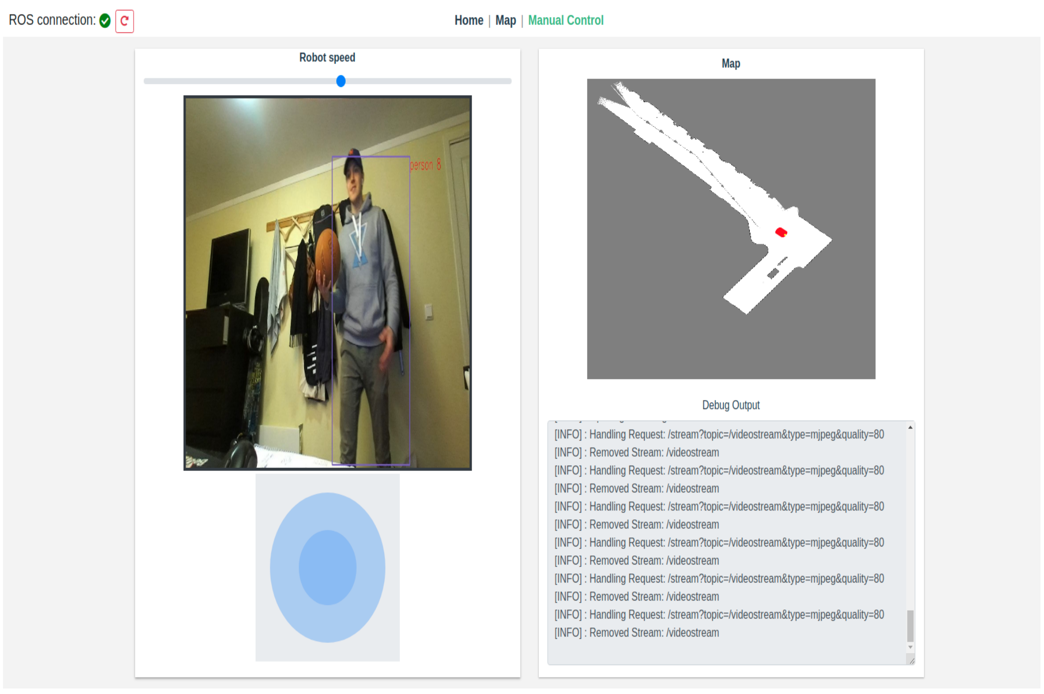
Initial Pose: **false**

x: 218 y: 154 r: 255 g: 255 b: 255

Motors: **ON**

```
[INFO] : new speed: [1475.79,0.51](1591542347.811)
[INFO] : new speed: [1159.32,0.51](1591542348.250)
[WARN] : Control loop missed its desired rate of 20.0000Hz... the loop actually took
0.4396 seconds
[WARN] : Control loop missed its desired rate of 20.0000Hz... the loop actually took
0.4865 seconds
[INFO] : new speed: [1764.02,-1.00](1591542348.737)
[INFO] : new speed: [1207.76,-1.00](1591542349.084)
[WARN] : Control loop missed its desired rate of 20.0000Hz... the loop actually took
0.3478 seconds
[WARN] : Control loop missed its desired rate of 20.0000Hz... the loop actually took
0.4304 seconds
[INFO] : new speed: [1247.26,-1.00](1591542349.515)
[WARN] : Control loop missed its desired rate of 20.0000Hz... the loop actually took
0.4069 seconds
[INFO] : new speed: [1480.35,-1.00](1591542349.924)
[INFO] : new speed: [1773.29,-0.78](1591542350.317)
[WARN] : Control loop missed its desired rate of 20.0000Hz... the loop actually took
0.3953 seconds
[INFO] : new speed: [1321.20,-0.57](1591542350.704)
[WARN] : Control loop missed its desired rate of 20.0000Hz... the loop actually took
0.3870 seconds
[WARN] : Control loop missed its desired rate of 20.0000Hz... the loop actually took
0.3278 seconds
[INFO] : new speed: [746.50,-0.78](1591542351.032)
```

Figure 8.9: Final map view.



```
[INFO] : Handling Request: /stream?topic=/videostream&type=mjpeg&quality=80
[INFO] : Removed Stream: /videostream
[INFO] : Handling Request: /stream?topic=/videostream&type=mjpeg&quality=80
[INFO] : Removed Stream: /videostream
[INFO] : Handling Request: /stream?topic=/videostream&type=mjpeg&quality=80
[INFO] : Removed Stream: /videostream
[INFO] : Handling Request: /stream?topic=/videostream&type=mjpeg&quality=80
[INFO] : Removed Stream: /videostream
[INFO] : Handling Request: /stream?topic=/videostream&type=mjpeg&quality=80
[INFO] : Removed Stream: /videostream
[INFO] : Handling Request: /stream?topic=/videostream&type=mjpeg&quality=80
[INFO] : Removed Stream: /videostream
```

Figure 8.10: Final Manual Control view.

9 | Testing and Results

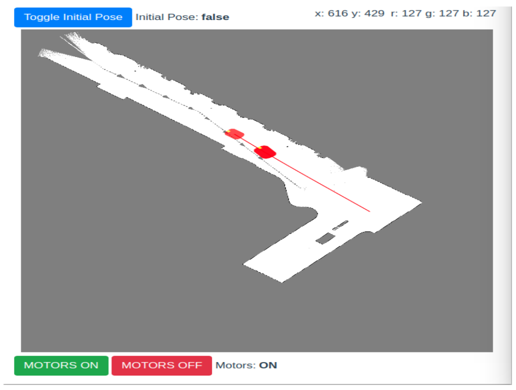
This chapter presents the testing of the Web Application, the following results and a short discussion of the results. The web app was tested with the ROS system running together with MobileSim. The conducted tests can be considered as integration testing since both ROSBridge and the web app are tested simultaneously. The app is served locally using the Vue CLI user interface. To setup the ROS simulation environment on a Ubuntu Xenial computer see setup script in Appendix C.1. To setup development environment read Github README file at [38]. Appendix D also provides notes for developing with Vue.

9.1 Testing Navigation Map

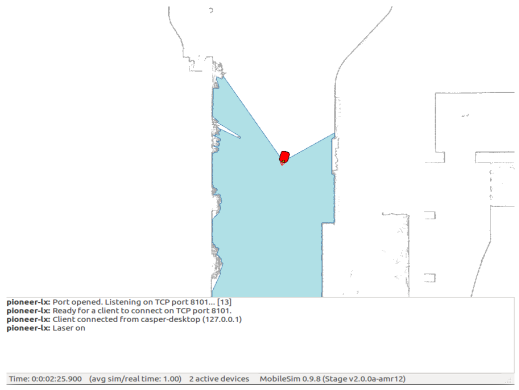
The navigation map in the *Map* view was tested to examine its stability under various circumstances. The test procedure involved among other things checking pose estimation, turning motors on and off while navigating, using joystick or keyboard for control while simultaneously navigating and sending new goals rapidly. The simulator was observed together with the application's map to ensure localization was correct at all times.

9.1.1 Result

The robot was moved around put into different circumstances for an hour. Turning the motors on or off will not stop the navigation module from sending messages to change twist and velocity, but will stop the robot from moving. The commands from the different inputs of joystick and teleop become interweaved without other errors. Figure 9.1a and 9.1b shows the navigation map during testing.



(a) Navigation Map web app.



```
pioneer-lx: Port opened. Listening on TCP port 8101... [13]
pioneer-lx: Ready for a client to connect on TCP port 8101.
pioneer-lx: Client connected from casper-desktop (127.0.0.1)
pioneer-lx: Laser on
```

```
Time: 0:0:02:25.900   (avg sim/real time: 1.00)   2 active devices    MobileSim 0.9.8 (Stage v2.0.0a-amr12)
```

(b) MobileSim.

9.1.2 Discussion

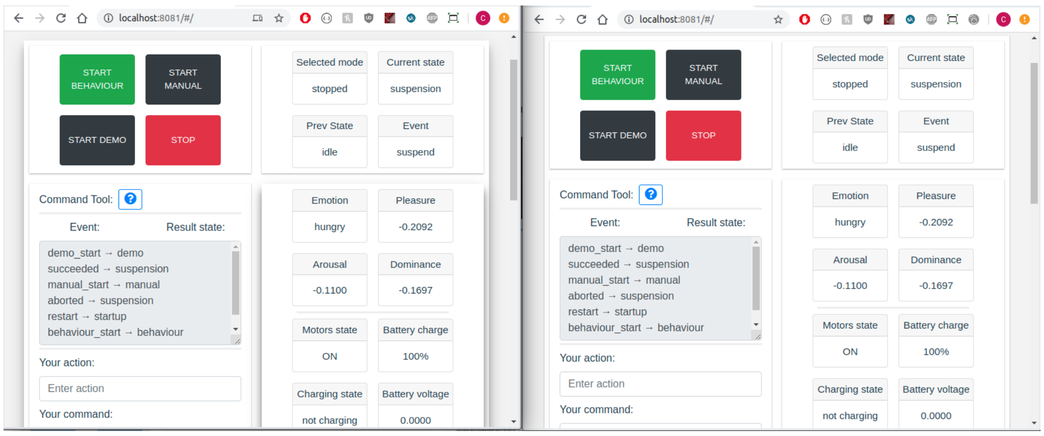
The navigation is robust to disturbances from joystick and teleop control to a certain degree. The pose estimation works as intended and is precise enough for the robot to localize itself after about 3 seconds of navigation. The interweaving between inputs is not unreasonable behaviour and concurrent users all have maps updated in real-time as expected. The loading time on the map is 2ms but the nav2djs library takes about two seconds to display it. This could be optimized by using eventemitter3 instead of eventemitter2 or switching out nav2djs with another library.

9.2 Testing Concurrent Clients

It is important that the web app and ROSBridge can handle multiple users simultaneously. All features were tested with 4 tabs open at the same time providing input, simulating several clients on the app.

9.2.1 Result

Simultaneous audio input resulted in queued audio playback. Simultaneous navigation led to only the latest goal location be used. Input from one browser shows up instantaneously in the other browser. Figure 9.2 shows two users monitoring and controlling the robot at the same time.



Selected mode: stopped
Current state: suspension
Prev State: idle
Event: suspend
Emotion: hungry
Pleasure: -0.2092
Arousal: -0.1100
Dominance: -0.1697
Motors state: ON
Battery charge: 100%
Charging state: not charging
Battery voltage: 0.0000

Event:	Result state:
demo_start	demo
succeeded	suspension
manual_start	manual
aborted	suspension
restart	startup
behaviour_start	behaviour

Your action:

Your command:

Selected mode: stopped
Current state: suspension
Prev State: idle
Event: suspend
Emotion: hungry
Pleasure: -0.2092
Arousal: -0.1100
Dominance: -0.1697
Motors state: ON
Battery charge: 100%
Charging state: not charging
Battery voltage: 0.0000
demo_start → demo
succeeded → suspension
manual_start → manual
aborted → suspension
restart → startup
behaviour_start → behaviour
Your action:
Your command:

Figure 9.2: Two simultaneous users.

.

9.2.2 Discussion

Multiple concurrent users worked with no problems. ROSBridge handles subscribing and publishing as expected. The behaviour of the cyborg modules also matched the expectations.

10 | Bridging and Deployment

After deployment, to enable remote communication between ROS and the web app, some bridging was required. This chapter explains the continuous deployment to Azure, the Rosbridge Suite of packages, the Web Video Server package and how No-ip hostnames with SSL certifications were used to allow for remote interaction with the robot.

10.1 Deploying Web App

An Azure Web App service was created to host the app on the web. The Node 12 LTS runtime stack on a Windows operating system was used. To allow continuous deployment, the deployment center in the Azure portal can interface Azure with GitHub where the code is stored using the *Kudu engine*. Since the entire app build is stored inside the *dist/* folder, to deploy only the contents of dist/ a *.deployment* file is added to the project. It is used by the *Kudu engine* to choose what folder to deploy to the web. There are many guides online describing how to deploy web apps on Azure so it is omitted in this thesis.

10.2 The Rosbridge Suite

The *Rosbridge Suite* by RobotWebTools was used to provide the transport layer between the robot and the website using *websockets*. One of its packages, the *Rosbridge Server*, is used as an interface between client web browsers and a ROS system. This is the server that libraries like *roslibjs* communicates with as mentioned earlier in Section 8.3. Another suite package called *rosapi* is used to retrieve meta-information like lists of services and topics via service calls. [39][30] [40]

These calls must conform to the rosbridge protocol which is the JSON format shown in Listing 10.2.

```
{ "op": "subscribe",
  "topic": "/cmd_vel",
  "type": "geometry_msgs/Twist"
}
```

The RosBridge Server listens for any attempted connections, handle them as clients and serve them information. The server handles interactions on the GUI like ROS service calls, 2D nav goals and keyboard events in the teleop by converting JSON messages to ROS calls using the *Rosbridge Library*. The rosbridge server listens on port *9090* by default, handling any websocket requests containing JSON data matching the rosbridge protocol. Rosbridge was installed and added to the *cyborg.launch* roslaunch file in the *Cyborg Launch* package. A context diagram for rosbridge is presented below. [32]

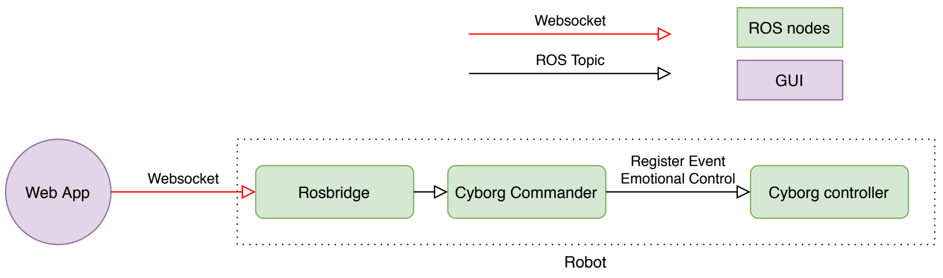


Figure 10.1: Context diagram for Rosbridge, between ROS nodes, the topics go through the ROS Master.

10.3 The Web Video Server

For live videostreams, another server was added to the ROS system. The web video server allows for live streaming of ROS images to clients on local or remote browsers. Similarly to Rosbridge, it opens a local port that listens for incoming requests. Using the Jetson TX1 Developer board, the Computer Vision module handles video from the ZED Stereoscopic camera. This video is published to the topic */videostream*, which the server broadcasts on the local *8080* port. The server supports two basic types of streams, image and video. A context diagram for the server is shown below. [41]

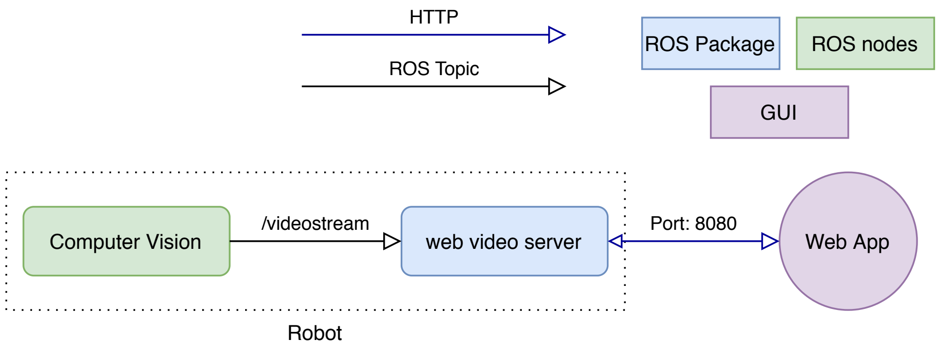


Figure 10.2: Context diagram for web video server, the specific node the web video server runs depends on choice of stream encoding.

10.4 No-ip Hostname and SSL authentication

Because the Eduroam school network dynamically changes the ip of the robot when it changes Wi-Fi access points, communicating with the robot from the internet is not possible. To solve this problem, the No-ip dynamic DNS (Dynamic Name Server) update client was installed and a No-ip online hostname created. A context diagram for No-ip is presented in figure 10.3.

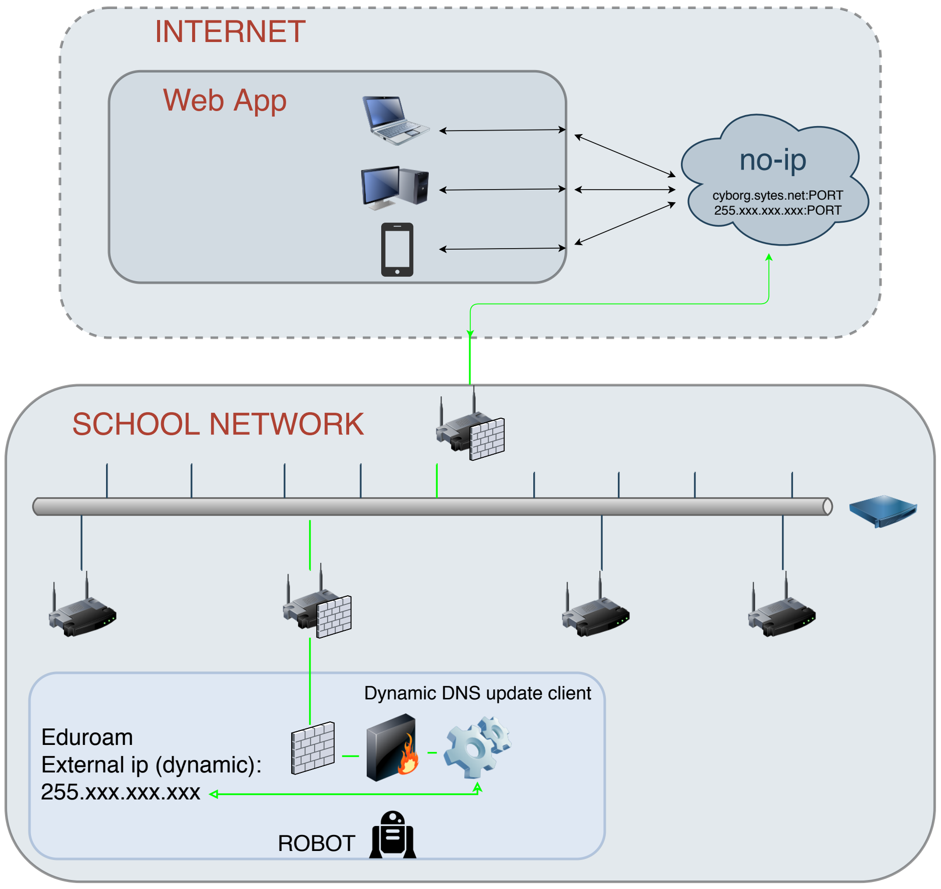


Figure 10.3: Context diagram for web video server, the specific node the web video server runs depends on choice of stream encoding.

As the robot switches Wi-Fi access points when wandering the hallways, its internal IP address changes dynamically. The DDNS client tracks the ip and updates the static domain name *cyborg.sytes.net*. To install the No-ip update client, follow the guide in Appendix C.2. Listing 10.1 shows how the hostname replaces the local connection.

```
var ros = new ROSLIB.Ros({
  // url: "ws://localhost:9090"
  url: "ws://cyborg.sytes.net:9090"
});
```

Listing 10.1: Replacing local communication.

The website runs on the HTTPS protocol. Websites running in HTTPS only accept resources over secure https connections. This is a problem for our hostname, which runs on http. The hostname website was therefore authenticated for secure socket connections and rosbridge was modified to handle these connections. Since image and video files are considered passive elements in a browser, we don't have to authenticate for the web video server to retrieve http requests on the web app. A guide to setup SSL authentication for a No-ip hostname is added to Appendix C.2. The changes made to the rosbridge launch file is shown below in Listing 10.2.

```
<include file="$(find rosbridge_server)/launch/ \\
    rosbridge_websocket.launch">
  <arg name="certfile" default="$(find cyborg_commander) \\
   /cyborg_sytes_net_ee.crt" />
  <arg name="keyfile" default="$(find cyborg_commander) \\
  /cyborg_sytes_net.key" />
  <arg name="ssl" default="true" />
  <arg name="authenticate" default="false" />
</include>
```

Listing 10.2: Forcing ssl over Rosbridge.

11 | Discussion

The focus of this thesis has been to develop a GUI for real-time monitoring, commanding and controlling capabilities. As a collaborative project, the author went beyond the problem description to build a platform for students to build user interfaces by iterative work for years to come. For in the future to visually broadcast the capabilities of the robot's MEA communication, Computer Vision module's object recognition software and its emotions was an important secondary focus.

Deciding which framework to go for was therefore not a simple task, there were differing opinions for each framework, however the most important factors would be the frameworks learning curve and modularity considering further development. The documentation for each framework was gone through, their benchmarks and popularity was explored and compatible libraries with ROS were examined. Vue was found to be the best in those regards. After using Vue, I am confident it was the right choice of framework.

Due to the circumstances, testing on the physical robot was not possible. This is unfortunate since checking the software load on the real robot could be useful to check for hardware limitations, which was already a suggestion of future work from Areg's thesis (Computer upgrade, p.117) [7]. The situation led to testing in simulated environments, which limited other tests for remote connectivity. However, the specifications were met and exceeded, and the GUI is ready to be used as a stand-alone module.

12 | Future Work

The most important suggestions for future work are presented in the list below:

- **No-ip -** Most important of all is to get the DDNS client up and running so that the GUI can access the robot remotely. Please see guides in appendix C.2. No-ip has been proven to work on the Eduroam network previously according to a conversation with [42].
- **Admin and user login page -** Separate administrator and regular user single sign-on pages was explored with Azure active directory b2c that could be integrated with the NTNUCyborg GitHub organization.
- **Mobile Support -** The web app functions on mobile devices, but some proportions are wrong. Minor modification to BootstrapVue's size properties would fix this.
- **Test live camera -** The web video server should be tested with live video from the stereoscopic camera.
- **Domain -** A domain name could be provided by NTNU.
- **Send to location in command tool -** Sending the robot to the predefined locations in the database, could be a useful debugging tool.
- **Computer upgrade -** As stated in the discussion, the computer in the robot base struggles with navigation and visualization, due to slow performance. A new or several distributed computers could solve this problem.
- **Software upgrades -** Updating the system to Ubuntu 18.04 or 20.04 with ROS 2 and Python 3 could help explore more of the robot's potential.

13 | Conclusion

The ROS Commander node was developed and integrated and tested with the ROS system. A component-based, reactive Vue SPA was designed, its specifications stated and its components implemented and tested. Components such as a click-to-navigate navigation map, on-screen joystick and teleop controls for manual control and command tool for changing states in state machine. Various modules like the controller and were modified to work with the GUI. Communication between the web app and ROS computer was established with secure HTTPS protocols using No-ip. This means all suggested features specified in the Problem Description were implemented fully and tested on the website.

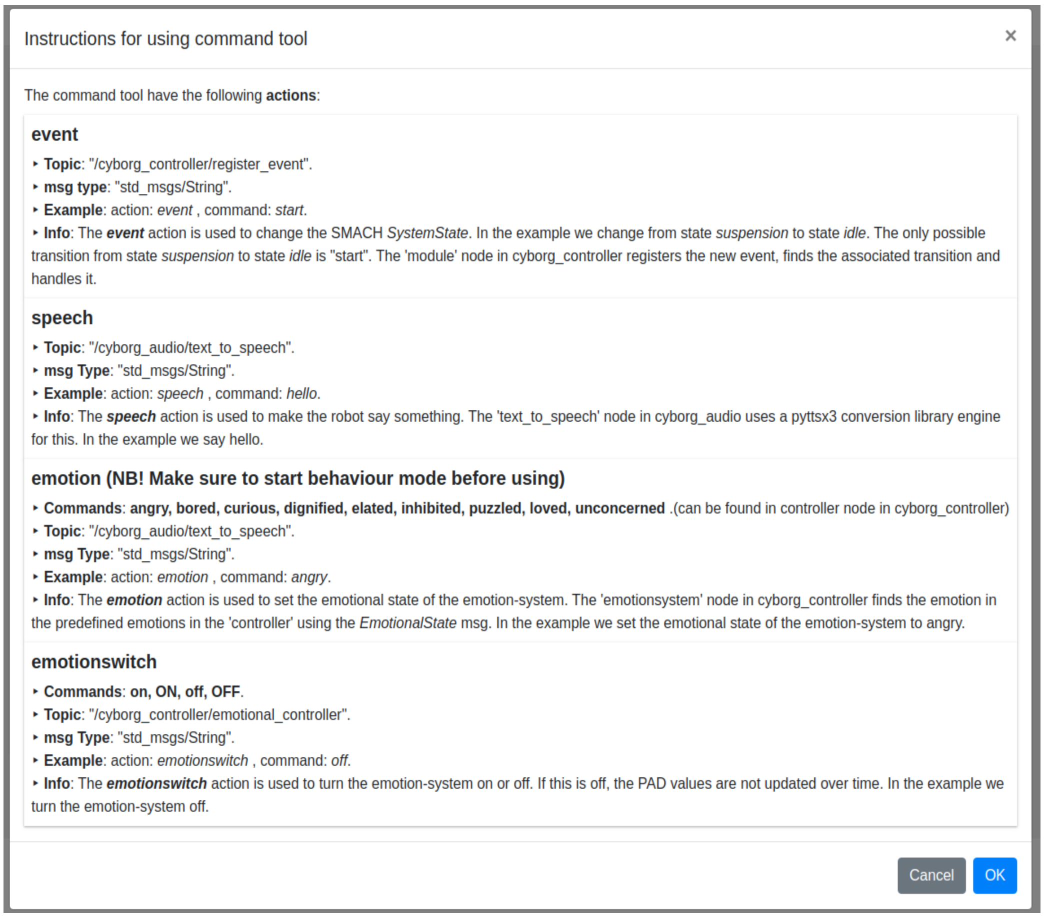
Commander node for top-level management of ROS package nodes and communication between the robot and the cloud. Videostream from the robot camera of the Computer Vision Module was achieved on the website. Historical data from MongoDB Atlas was implemented with MongoDB Charts on the website, but was deemed unnecessary for the time being.

Send to location, monitor behavioursystem, alter state-machine states, select robot mode of operation. Change behaviour state, text-to-speech. Accessible on a webpage for computers and cellphones.

Appendices

A | Diagrams and figures

Appendix A.1 Command Tool Manual


Instructions for using command tool

The command tool have the following **actions**:

event

- **Topic**: "/cyborg_controller/register_event".
- **msg type**: "std_msgs/String".
- **Example**: action: *event* , command: *start*.
- **Info**: The ***event*** action is used to change the SMACH *SystemState*. In the example we change from state *suspension* to state *idle*. The only possible transition from state *suspension* to state *idle* is "start". The 'module' node in cyborg_controller registers the new event, finds the associated transition and handles it.

speech

- **Topic**: "/cyborg_audio/text_to_speech".
- **msg Type**: "std_msgs/String".
- **Example**: action: *speech* , command: *hello*.
- **Info**: The ***speech*** action is used to make the robot say something. The 'text_to_speech' node in cyborg_audio uses a pyttsx3 conversion library engine for this. In the example we say hello.

emotion (NB! Make sure to start behaviour mode before using)

- **Commands**: **angry, bored, curious, dignified, elated, inhibited, puzzled, loved, unconcerned** .(can be found in controller node in cyborg_controller)
- **Topic**: "/cyborg_audio/text_to_speech".
- **msg Type**: "std_msgs/String".
- **Example**: action: *emotion* , command: *angry*.
- **Info**: The ***emotion*** action is used to set the emotional state of the emotion-system. The 'emotionsystem' node in cyborg_controller finds the emotion in the predefined emotions in the 'controller' using the *EmotionalState* msg. In the example we set the emotional state of the emotion-system to angry.

emotionswitch

- **Commands**: **on, ON, off, OFF**.
- **Topic**: "/cyborg_controller/emotional_controller".
- **msg Type**: "std_msgs/String".
- **Example**: action: *emotionswitch* , command: *off*.
- **Info**: The ***emotionswitch*** action is used to turn the emotion-system on or off. If this is off, the PAD values are not updated over time. In the example we turn the emotion-system off.

Figure A.1: Instructions for event, speech, emotion and emotionswitch commands.

Appendix A.2 State Machine

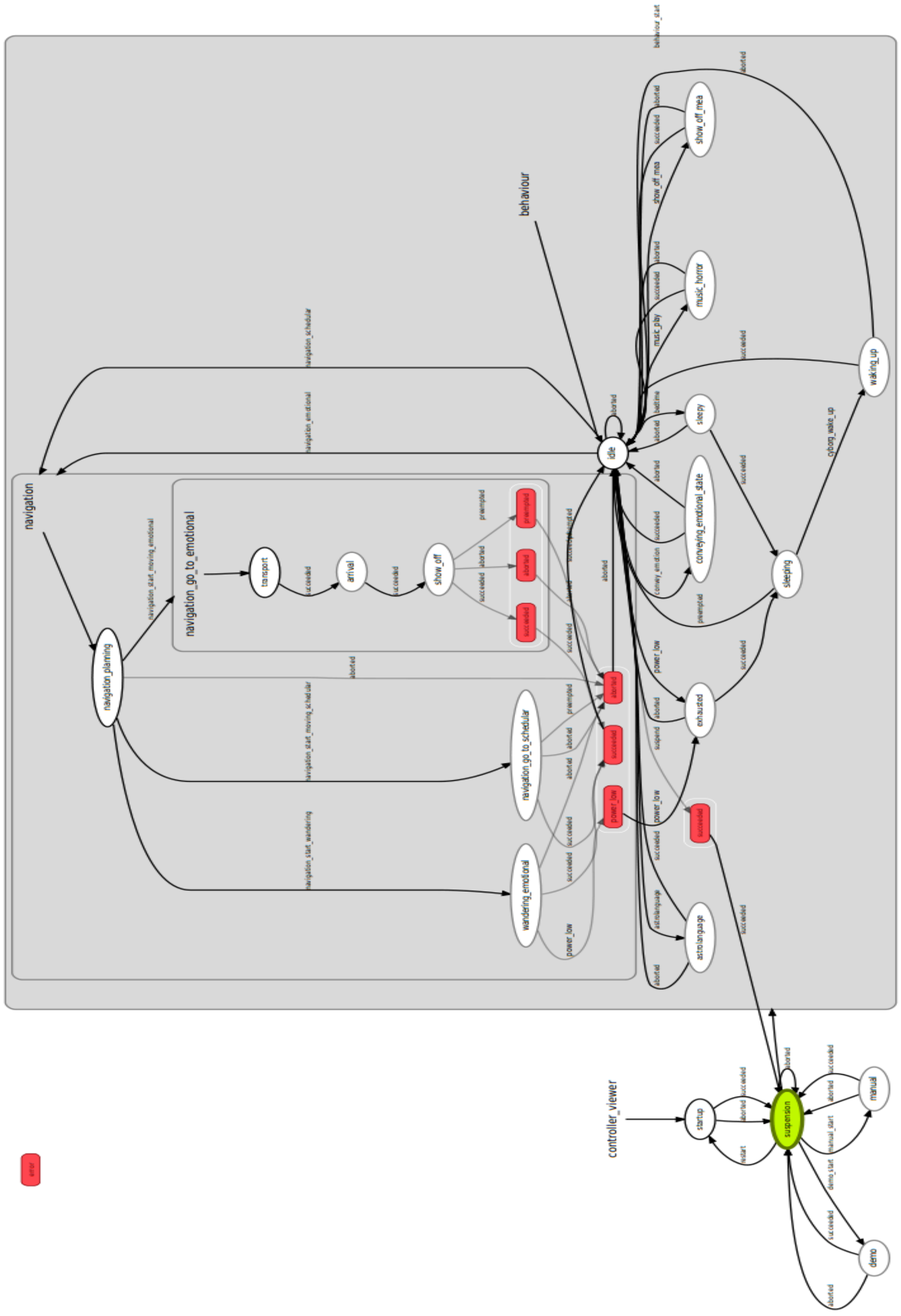


Figure A.2: State machine visualized in Smach Viewer.

B | Tables

Appendix B.1 Table of Presets for PAD Emotion Model Emotions

	Name	Pleasure	Arousal	Dominance
Emotions :	*angry*	-0.51	0.59	0.25
	bored	-0.65	-0.62	-0.33
	curious	0.22	0.62	-0.10
	dignified	0.55	0.22	0.61
	elated	0.50	0.42	0.23
	inhibited	-0.54	-0.04	-0.41
	puzzled	-0.41	0.48	-0.33
	loved	0.89	0.54	-0.18
	unconcerned	-0.13	-0.41	0.08
	hungry	-0.44	0.14	-0.21
	sleepy	0.20	-0.70	-0.44

Table B.2: Presets of emotions found in controller node, used by emotion system node. PAD values range from -1 to 1.

Appendix B.2 Table of Supported ROS Topics and Data Types

Name	**Type**
/amcl/parameter_descriptions	dynamic_reconfigure/ConfigDescription
/amcl/parameter_updates	dynamic_reconfigure/Config
/amcl_pose	PoseWithCovarianceStamped
/clicked_point	PointStamped
/client_count	Int32
/cmd_vel	Twist
/connected_clients	ConnectedClients
/controller_viewer/smach/container:	
_init	SmachContainerInitialStatusCmd
_status	SmachContainerStatus
_structure	SmachContainerStructure
/cyborg_audio:	
/feedback_playback	String
/feedback_text_to_speech	String
/playback	String
/text_to_speech	String
/cyborg_behavior:	
/cancel	GoalID
/command_location	String
/dynamic_behavior	String
/feedback	StateMachineActionFeedback
/goal	StateMachineActionGoal
/result	StateMachineActionResult
/status	GoalStatusArray
/cyborg_controller:	
/emotional_controller	String
/emotional_feedback	EmotionalFeedback
/emotional_state	EmotionalState
/register_event	String
/state_change	SystemState

/cyborg_navigation:	
/current_location	String
/navigation/cancel	GoalID
/navigation/feedback	NavigationActionFeedback
/navigation/goal	NavigationActionGoal
/navigation/result	NavigationActionResult
/navigation/status	GoalStatusArray
/cyborg_primary_states:	
/cancel	GoalID
/feedback	StateMachineActionFeedback
/goal	StateMachineActionGoal
/result	StateMachineActionResult
/status	GoalStatusArray
/cyborg_visual/domecontrol	String
/initialpose	PoseWithCovarianceStamped
/joint_states	JointState
/map	OccupancyGrid
/map_metadata	MapMetaData
/map_updates	OccupancyGridUpdate
/move_base/cancel	PoseStamped
/move_base/current_goal	PoseStamped
/move_base/feedback	MoveBaseActionFeedback
/move_base/global_costmap:	
/costmap	OccupancyGrid
/costmap_updates	OccupancyGridUpdate
/footprint	PolygonStamped
/inflation_layer/parameter_descriptions	dynamic_reconfigure/ConfigDescription
/inflation_layer/parameter_updates	dynamic_reconfigure/Config
/obstacle_layer/parameter_descriptions	dynamic_reconfigure/ConfigDescription
/obstacle_layer/parameter_updates	dynamic_reconfigure/Config
/parameter_descriptions	dynamic_reconfigure/ConfigDescription
/parameter_updates	dynamic_reconfigure/Config
/static_layer/parameter_descriptions	dynamic_reconfigure/ConfigDescription
/static_layer/parameter_updates	dynamic_reconfigure/Config
/move_base/goal	MoveBaseActionGoal

/move_base/local_costmap:	
/costmap	OccupancyGrid
/costmap_updates	OccupancyGridUpdate
/footprint	PolygonStamped
/inflation_layer/parameter_descriptions	dynamic_reconfigure/ConfigDescription
/inflation_layer/parameter_updates	dynamic_reconfigure/Config
/obstacle_layer/parameter_descriptions	dynamic_reconfigure/ConfigDescription
/obstacle_layer/parameter_updates	dynamic_reconfigure/Config
/parameter_descriptions	dynamic_reconfigure/ConfigDescription
/parameter_updates	dynamic_reconfigure/Config
/move_base/NavfnROS/plan	Path
/move_base/parameter_descriptions	dynamic_reconfigure/ConfigDescription
/move_base/parameter_updates	dynamic_reconfigure/Config
/move_base/result	MoveBaseActionResult
/move_base_simple/goal	PoseStamped
/move_base/status	GoalStatusArray
/move_base/TrajectoryPlannerROS:	
/cost_cloud	PointCloud2
/global_plan	Path
/local_plan	Path
/parameter_descriptions	dynamic_reconfigure/ConfigDescription
/parameter_updates	dynamic_reconfigure/Config
/odom	Odometry
/particlecloud	PoseArray
/robot_pose	Pose
/RosAria:	
/battery_recharge_state	Int8
/battery_state_of_charge	Float32
/battery_voltage	Float64
/bumper_state	BumperState
/motors_state	Bool
/parameter_descriptions	dynamic_reconfigure/ConfigDescription
/parameter_updates	dynamic_reconfigure/Config
/sim_S3Series_1_laserscan	LaserScan
/sim_S3Series_1_pointcloud	PointCloud
/sonar	PointCloud
/sonar_pointcloud2	PointCloud2

/rosout	Log
/rosout_agg	Log
/tf	TFMessage
/tf_static	TFMessage
/status	GoalStatusArray

Table B.3: Names and Types of supported ROS topics.

C Guides

Appendix C.1 Setup Robot Simulation

```
#!/bin/bash

echo "Setup script running..."
echo "WARN - Specific versions of packages are needed in this
project and when installing warnings about deprecated or old
software may show up. This is expected."
read

# This may not be complete, and may be missing some libraries
# or install commands
# All commands should also be updated to include versions.


echo "---------- General installs and setup ----------"
echo "to continue press enter"
read

sudo apt-get update
sudo apt-get install git
sudo apt install python2.7
sudo apt install python-pip
pip install --upgrade pip==20.0.2


echo "---------- Install ROS ----------"
echo "to continue press enter"
read

## ROS
# Setup your computer to accept software from
# packages.ros.org:
sudo sh -c 'echo
"deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc)
main" > /etc/apt/sources.list.d/ros-latest.list'
```

```
# Set up your keys
sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80'
--recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654

sudo apt-get update
# Full ROS installation
sudo apt install ros-kinetic-desktop-full
# Find avaliable packages
apt-cache search ros-kinetic
read -p "Available packages (above).
Take a note if some are needed. Press ENTER to continue."
# Initialise rosdep
sudo rosdep init
rosdep update
# Setup the environment
echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
source ~/.bashrc
# Dependencies for building packages
sudo apt install python-rosinstall python-rosinstall-generator
python-wstool build-essential
# Create a workspace
mkdir -p ~/catkin_ws/


echo "---------- Clone Cyborg repo from Git ----------"
echo "to continue press enter"
read

## Clone Cyborg Repo from git, and place it in the right
directory
cd ~/catkin_ws
git clone https://github.com/thentnucyborg/CyborgRobot.git
#clones the master branch

cd ./CyborgRobot
echo "Use the master branch? Please type the branch name or
'yes' if you wish to use the master branch."
read branchname
if [ "$branchname" == "yes" ]
then
  echo "Using the master branch"
else
  git checkout $branchname
fi

mv ~/catkin_ws/CyborgRobot/* ~/catkin_ws/
#move all files and folders to the workspace
mv ~/catkin_ws/CyborgRobot/.* ~/catkin_ws/
#move all hidden files and folders to the workspace.
```

```
Ignore the message saying . and .. cannot be moved
echo "Please ignore the message saying . and ..
cannot be moved"
rm -rf ~/catkin_ws/CyborgRobot  #delete the now empty folder



echo "---------- Install Arnl and ARIA .debs ----------"
echo "to continue press enter"
read

echo "Check that you have downloaded the Arnl and Arnl-base,
Aria and MobileSim .deb files into Downloads from Box to be
able to continue"
read

cd ~/catkin_ws/setup/installs
sudo dpkg -i arnl-base_1.9.2+ubuntu16_amd64.deb
sudo dpkg -i libarnl_1.9.2a+ubuntu16_amd64.deb
sudo dpkg -i libaria_2.9.4+ubuntu16_amd64.deb
sudo dpkg -i mobilesim_0.9.8+ubuntu16_amd64.deb


echo "---------- Installs for New Navigation ----------"
echo "to continue press enter"
read

# Installs for GUI
sudo apt-get install ros-kinetic-rosauth
sudo apt-get install ros-kinetic-rosbridge-server

## Installs for Navigation stack (may be more)
cd  ~/catkin_ws/src
sudo apt-get install ros-kinetic-navigation
sudo apt-get install ros-kinetic-tf2-sensor-msgs
sudo apt-get install libsdl-dev
sudo apt-get install libsdl-image1.2-dev
sudo apt-get install libbullet-dev
git clone https://github.com/ros-visualization/rviz.git
-b kinetic-devel
git clone https://github.com/ros-planning/navigation.git
-b kinetic-devel
# Install move-base
sudo apt install ros-kinetic-move-base


echo "---------- Install SMACH ----------"
echo "to continue press enter"
read
```

```

## Install SMACH
sudo apt-get install ros-kinetic-executive-smach
sudo apt-get install ros-kinetic-executive-smach-visualization

sudo apt-get install python-pyqt5
sudo apt-get install python-qt-binding


echo "---------- Install Aduio ----------"
echo "to continue press enter"
read

## Install for Audio node
pip2 install -Iv pyttsx3==2.7 #-I ignores installed packages,
-v prints/verbose
pip2 install python-vlc==3.0.7110

echo "---------- Install Command node ----------"
echo "to continue press enter"
read

## Install for Command node
pip2 install npyscreen


echo "---------- Install Controller ----------"
echo "to continue press enter"
read

## Install for Controller node
sudo apt install graphviz-dev
pip2 install pygraphviz
#Alternatively, run: sudo apt-get install python-pygraphviz


echo "---------- Install LED Dome node ----------"
echo "to continue press enter"
read

## Install for Led Dome node
pip2 install colour==0.1.5
pip2 install numpy==1.16.6
pip install pandas==0.24.2
pip2 install pyserial==3.0.1
pip2 install pyopengl
pip2 install pyopengl-accelerate
pip2 install pytz

```

```

echo "---------- Install Behaviour Trees ----------"
echo "to continue press enter"
read

## Behavior Trees
pip2 install networkx==2.2
# install nodejs for behavior3editor
curl -sL https://deb.nodesource.com/setup_10.x | sudo -E
bash - sudo apt-get install -y nodejs # also installs npm
#install bower
sudo npm install -g bower
# install dependencies for behavior3editor
cd ~/catkin_ws/src/behavior3editor
npm install
bower install
sudo npm install --global gulp@3.9.1
# install b3 module
cd ~/catkin_ws/setup/installs/behavior3py
sudo python setup.py install


echo "---------- Setup UDEV Rules ----------"
echo "to continue press enter"
read

## Set up UDEV rules
sudo cp ~/catkin_ws/setup/90_cyborg_usb_rules.rules
/etc/udev/rules.d/
sudo udevadm control --reload
sudo udevadm trigger


echo "---------- other ----------"
echo "to continue press enter"
read

## Other
sudo apt-get install sqlitebrowser  #tool for editing
databases

# Make  python  and  bash  scripts  executable
find ~/catkin_ws/src/ -name '*.py' -exec  chmod +x {} \;
find ~/catkin_ws/src/ -name '*.sh' -exec  chmod +x {} \;


echo "---------- Finish setting up catkin_ws ----------"
echo "to continue press enter"
read
```

```

# Finish setting up the workspace
source ~/.bashrc
source /opt/ros/kinetic/setup.bash
cd ~/catkin_ws
catkin_make
echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
# Opening new terminal runs source command,
# so we dont have to source workspace each time.
# source devel/setup.bash


## Base requirements
sudo usermod -a -G dialout $USER  #add user to dialout group
sudo apt autoremove



echo "Setup script ended..."
echo "---------------------"

# Relogin is required for last cmd to take effect
echo "You must logout and back in for userprivileges
to take effect..."

echo "There are still some things to install:"
echo " - If changes to the code on the led-controller are
needed (the NodeMCU ESP32), follow the install instructions
in cyborg_ros_led_dome/README.md"
```

Appendix C.2 Setup No-ip Hostname with SSL and DDNS Client

The rosbridge server and web video server grants local requests of clients on port 9090 and 8080 respectively by default.
Setup No-ip and check open ports:

1. Create hostname on no-ip.com.
2. Go to https://www.portchecktool.com/ to check for open port on external ip.
   Check that port 9090 is open.
3. If it is not open, make inbound rule in router and check firewall settings of computer until it is.
4. Set external ip for hostname in no-ip.com.

   Setup DDNS:
5. Download Dynamic Update Client. Follow instructions on https://bit.ly/2YlwQRo.
6. 'sudo /usr/local/bin/noip2 -S' to check if noip2 service is running.
7. Run robot and check http://*hostname*:9090

SSL certificate for hostname:
The noip domain uses insecure http requests which the cyborgGUI website rejects. To get https requests, an SSL certificate is needed on the cyborg.sytes.net domain.

1. Follow guide on: https://bit.ly/3eoQP7r
2. In step4, create a CSR: Use OpenSSL CSR Creation Wizard:
   https://www.digicert.com/easy-csr/openssl.htm
3. Run the resulting command in cd: /catkin_ws/src/cyborg_commander on robot computer.
4. Copy the csr key from the folder where the command was run and add it to noip. (make sure this key is referenced in cyborg.launch file).
5. Select approver email: 'admin@cyborg.sytes.net' and fill in rest.

6. Approve request and wait for email from rapidSSL to arrive. Follow installation instructions. For Geotrust certificate, I followed this guide: https://knowledge.digicert.com/solution/SO15168.html

7. Run the cyborg, check https://cyborg.sytes.net:9090, it should connect to the Autobahn WebSocket Endpoint.

D | Notes for Developing GUI

Some important notes for developing the GUI are noted below.

- IP address must be whitelisted in MongoDB Atlas. A range of addresses from Eduroam should be used. more info at https://bit.ly/2YYd3GL.
- To deploy from Azure, the repository will not show unless you are an owner or have permission from the GitHub NTNUCyborg organization.
- On creation of Azure Web App Service, **Windows OS is required**.
- Vue devtools in Chrome can be very useful for debugging.

  Run the following commands, each in its own terminal, to initialize simulation environment:

  1. MobileSim -m /catkin_ws/src/navigation/map_server/maps/ glassgarden.map -r pioneer-lx –start -29500,8000,0
  2. roslaunch cyborg_launch cyborg.launch
  3. First: export PATH= /.npm-global/bin:$PATH, then: vue ui –dev

References

[1] Martinius Knudsen. *NTNU Cyborg About the Project.* 2020. URL: `https://www.ntnu.edu/cyborg/about` (visited on 03/25/2020) (cit. on pp. 1–3).

[2] Loanna Sandvig. *Biological neural networks.* 2020. URL: `https://www.ntnu.edu/cyborg/bioneuro` (visited on 03/25/2020) (cit. on p. 1).

[3] Author Unknown. *Neural interface and system overview.* 2020. URL: `https://www.ntnu.edu/cyborg/neurorobotics` (visited on 03/25/2020) (cit. on p. 1).

[4] Mica R. Endsley. "Level of automation effects on performance, situation awareness and workload in a dynamic control task." In: *Ergonomics* 42.3 (Mar. 1999), pp. 464–465 (cit. on p. 3).

[5] Jørgen Waløen. *The NTNU Cyborg v2.0: The Presentable Cyborg.* Master's Thesis. Trondheim, Norway: Faculty of Information Technology and Electrical Engineering, 2017 (cit. on p. 5).

[6] Thomas Rostrup Andersen. *Controller Module for the NTNU Cyborg.* Master's Thesis. Trondheim, Norway: Faculty of Information Technology and Electrical Engineering, 2017 (cit. on pp. 5, 54).

[7] Areg Babayan. *Finalizing the Foundation for an NTNU Mascot.* Master's Thesis. Trondheim, Norway: Faculty of Information Technology and Electrical Engineering, 2019 (cit. on pp. 7, 29, 69).

[8] Adept MobileRobots. *PioneerLX User Guide.* 2020. URL: `https://www.ntnu.no/wiki/display/cyborg/Pioneer+LX` (visited on 03/29/2020) (cit. on pp. 7, 10).

[9] Stereolabs. *The camera that senses space and motion.* 2020. URL: `https://www.stereolabs.com/zed/` (visited on 03/29/2020) (cit. on p. 8).

[10] NVIDIA. *Jetson TX1 Developer Kit.* 2020. URL: `https://developer.nvidia.com/embedded/jetson-tx1-developer-kit` (visited on 03/30/2020) (cit. on p. 8).

[11] Eskil Hatling Hølland et al. *Design and production of the LED-dome for the NTNU-Cyborg.* EiT project report. Trondheim, Norway, 2018 (cit. on p. 9).

[12] Johanne Døvle Kalland. *Exploring Visualisations and Behaviour.* Specialization Project. Trondheim, Norway: Faculty of Information Technology and Electrical Engineering, 2019 (cit. on p. 9).

[13] Adept MobileRobots. *ARNL Documentation.* 2020. URL: `http://vigir.missouri.edu/\~gdesouza/Research/MobileRobotics/Software/ARNL-SONARNL/Arnl-1.7.0+gcc41/docs/ARNL-Reference/main.html` (visited on 03/29/2020) (cit. on p. 10).

[14] Lasse Göncz. *Reimplementing the Navigation Stack on the NTNU Cyborg in a Simulated Environment.* Specialization Project. Trondheim, Norway: Faculty of Information Technology and Electrical Engineering, 2019 (cit. on p. 10).

[15] TullyFoote. *Documentation.* 2018. URL: `http://wiki.ros.org/` (visited on 02/09/2020) (cit. on pp. 11, 22).

[16] JosephHirschfeld. *Getting Started with smach.* 2020. URL: `http://wiki.ros.org/smach/Tutorials/Getting\%20Started` (visited on 03/08/2020) (cit. on p. 20).

[17] Jon Duckett. *HTML & CSS.* Indianapolis, Indiana: John Wiley & Sons, Incorporated, 2014 (cit. on p. 23).

[18] Jon Duckett. *JAVASCRIPT & JQUERY.* Indianapolis, Indiana: John Wiley & Sons, Incorporated, 2014 (cit. on p. 23).

[19] Microsoft. *Azure documentation.* 2020. URL: `https://docs.microsoft.com/en-us/azure/?product=featured` (visited on 03/10/2020) (cit. on p. 24).

[20] MongoDB. *The MongoDB 4.2 Manual.* 2020. URL: `https://docs.mongodb.com/manual/` (visited on 04/02/2020) (cit. on p. 25).

[21] MongoDB. *MongoDB Atlas.* 2020. URL: `https://docs.atlas.mongodb.com/` (visited on 04/02/2020) (cit. on p. 25).

[22] Jeang Kuo Chen and Wei Zhe Lee. “An Introduction of NoSQL Databases Based on Their Categories and Application Industries.” In: *Algorithms* 12.5 (May 2019), pp. 1–16 (cit. on p. 25).

[23] jihoonl. *PyStyleGuide.* 2014. URL: `http://wiki.ros.org/PyStyleGuide` (visited on 03/27/2020) (cit. on p. 31).

[24] Coghlan. Nick van Rossum. Guido Warsaw. Barry. *Style Guide for Python Code*. 2013. URL: `https://www.python.org/dev/peps/pep-0008/` (visited on 03/27/2020) (cit. on p. 31).

[25] PaulBouchier. *ROS C++ Style Guide*. 2018. URL: `http://wiki.ros.org/CppStyleGuide` (visited on 03/27/2020) (cit. on p. 31).

[26] Google. *Google C++ Style Guide*. 2019. URL: `https://google.github.io/styleguide/cppguide.html` (visited on 03/28/2020) (cit. on p. 31).

[27] IsaacSaito. *ROS Best Practices*. 2018. URL: `http://wiki.ros.org/BestPractices` (visited on 10/20/2020) (cit. on p. 31).

[28] A-make, JohanneLun, and paytience. *setup.sh*. 2020. URL: `https://github.com/thentnucyborg/CyborgRobot/blob/master/setup/setup.sh` (visited on 03/31/2020) (cit. on p. 35).

[29] ArvidNorlander. *roslibjs*. 2019. URL: `http://wiki.ros.org/roslibjs?distro=kinetic` (visited on 04/27/2020) (cit. on p. 39).

[30] baalexander. *rosbridge server*. 2020. URL: `http://wiki.ros.org/rosbridge_server?distro=kinetic` (visited on 04/26/2020) (cit. on pp. 39, 64).

[31] IanMcMahon. *rosnodejs*. 2017. URL: `http://wiki.ros.org/rosnodejs?distro=kinetic` (visited on 04/28/2020) (cit. on p. 39).

[32] Russell Toris et al. *Robot Web Tools: Efficient Messaging for Cloud Robotics*. paper. 2015 (cit. on pp. 44, 65).

[33] GvdHoorn. *Industrial*. 2020. URL: `http://wiki.ros.org/Industrial#ROS-Industrial\_Overview` (visited on 05/24/2020) (cit. on p. 45).

[34] mov.ai. *SOLUTION OVERVIEW*. 2020. URL: `https://mov.ai/solution-overview/` (visited on 05/24/2020) (cit. on p. 46).

[35] Vue. *Overview*. 2019. URL: `https://cli.vuejs.org/guide/` (visited on 05/03/2020) (cit. on p. 49).

[36] Unknown Author. *Getting Started*. 2020. URL: `https://bootstrap-vue.org/docs` (visited on 04/24/2020) (cit. on p. 51).

[37] Dominik Nowak. *Bootstrap 4 + ROS: creating a web UI for your robot*. 2020. URL: `https://medium.com/husarion-blog/bootstrap-4-ros-creating-a-web-ui-for-your-robot-9a77a8e373f9` (visited on 04/23/2020) (cit. on p. 56).

[38] paytience. *CyborgGUI*. 2020. URL: `https://github.com/thentnucyborg/cyborgGUI` (visited on 03/31/2020) (cit. on p. 61).

[39] GvdHoorn. *rosbridge suite*. 2017. URL: http://wiki.ros.org/rosbridge_suite?distro=kinetic (visited on 04/26/2020) (cit. on p. 64).

[40] baalexander. *rosapi*. 2020. URL: http://wiki.ros.org/rosapi?distro=kinetic (visited on 04/26/2020) (cit. on p. 64).

[41] MatSadowski. *web video server*. 2019. URL: http://wiki.ros.org/web_video_server?distro=kinetic (visited on 04/23/2020) (cit. on p. 65).

[42] Martinius Knudsen. personal communication. 2020 (cit. on p. 70).