Institute of Mechanism Theory, Machine Dynamics and Robotics

RWTH AACHEN UNIVERSITY

This thesis was submitted to the Institute of Mechanism Theory, Machine Dynamics and Robotics

Cross-Compiling ROS2 Humble to WebAssembly

Master Thesis

by:

Isabel Paredes B.Sc.

Student number: 415723

supervised by:

Dipl.-Ing. Martin Mustermann

Examiner:

Univ.-Prof. Dr.-Ing. Dr. h. c. Burkhard Corves

Prof. Dr.-Ing. Mathias Hüsing

Aachen, 31 March 2023

Master Thesis

by Isabel Paredes B.Sc.

Student number: 415723

Cross-Compiling ROS2 Humble to WebAssembly

The issue will be inserted here after being drafted and provided by the supervisor beforehand. The issue should contain a detailed list of all work packages. It should not exceed one page and the version handed to the students has to be signed by the professor.

Supervisor: Dipl.-Ing. Martin Mustermann

Eidesstattliche Versicherung

Isabel Paredes Matrikel-Nummer: 415723

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Master Thesis mit dem Titel

Cross-Compiling ROS2 Humble to WebAssembly

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, 31 March 2023 _____
Isabel Paredes

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vor-schriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Aachen, 31 March 2023 _____
Isabel Paredes

The present translation is for your convenience only.
Only the German version is legally binding.

Statutory Declaration in Lieu of an Oath

Isabel Paredes Student number: 415723

I hereby declare in lieu of an oath that I have completed the present Master Thesis titled

Cross-Compiling ROS2 Humble to WebAssembly

independently and without illegitimate assistance from third parties. I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Aachen, 31 March 2023 Isabel Paredes

Official Notification:
Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whosoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.

Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1)If a person commits one of the offences listed in sections 154 to 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly. I have read and understood the above official notification:

Aachen, 31 March 2023 Isabel Paredes

Contents

List of abbreviations

General abbreviations

GUI	Graphical User Interface
LIFO	Last In, First Out
ROS	Robot Operating System
UI	User Interface
URDF	Universal Robotic Description Format
WASM	Web Assembly
emsdk	Emscripten SDK

1. Introduction

1.1. Robot Operating System 2

Robot Operating System (ROS) 2

1.2. Motivation

Web Assembly (WASM)

2. Literature Review

2.1. State of the Art

2.1.1. ROS on Web

The closest representation of the intended project is the work produced by Michael Allwright known as *ROS on Web*. Allwright shares the same goal as the author to develop the technology which allows for running ROS nodes entirely on the browser by cross-compiling C++ code to WebAssembly and using web workers to handle the internal communication [All23].

Equivalently, Allwright targeted the ROS 2 distribution. It is suspected that the *galactic* version was used for the demonstrations. The main demonstration of a running publisher and subscriber is illustrated in Figure 2.1.

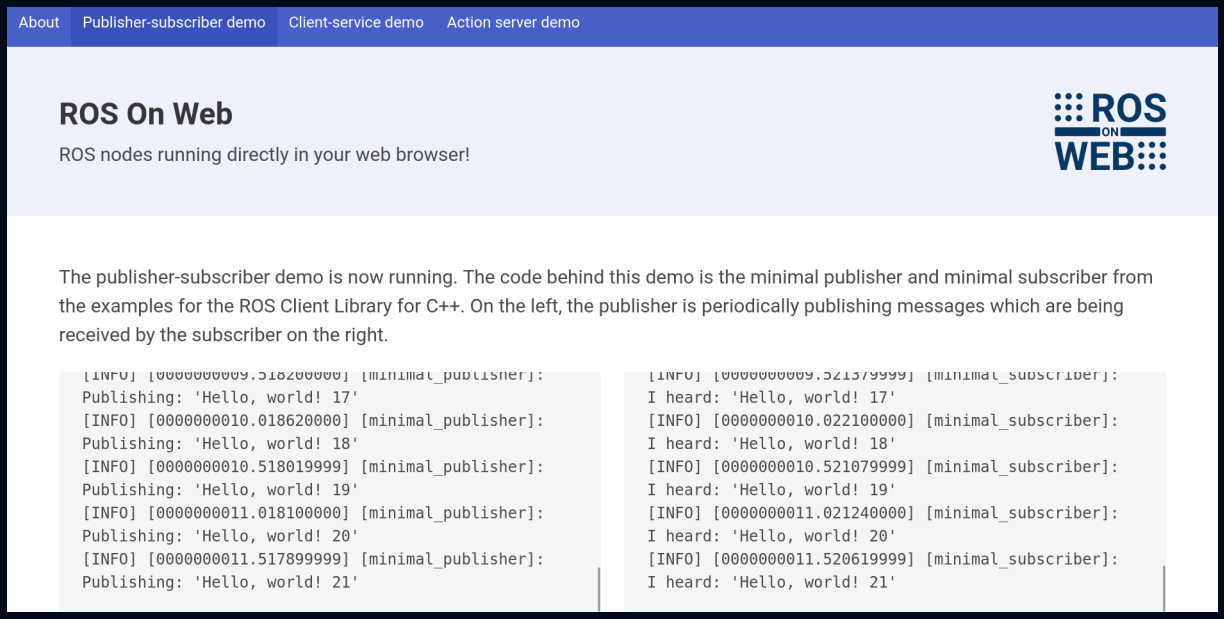


Figure 2.1. *ROS on Web* publisher and subscriber demonstration.

Nonetheless, the greatest disadvantage of *ROS on Web* lies in the fact that the project is not open source. Very little can be derived about how Allwright was able to achieve the demonstrations represented on the website. A few hints are given in the introductory page such as the replacement of the middleware with a custom design and the use of web workers. However, it is not possible to determine the manner in which these technologies were used. Hope remains that in the near future, the repositories for *ROS on Web* become publicly available as an extension of the ROS open source ecosystem.

2.2. Relevant Works

2.2.1. ROSbridge

2.2.2. ROS Control Center

2.2.3. ROSboard

2.2.4. ROSlink

2.2.5. Foxglove Studio

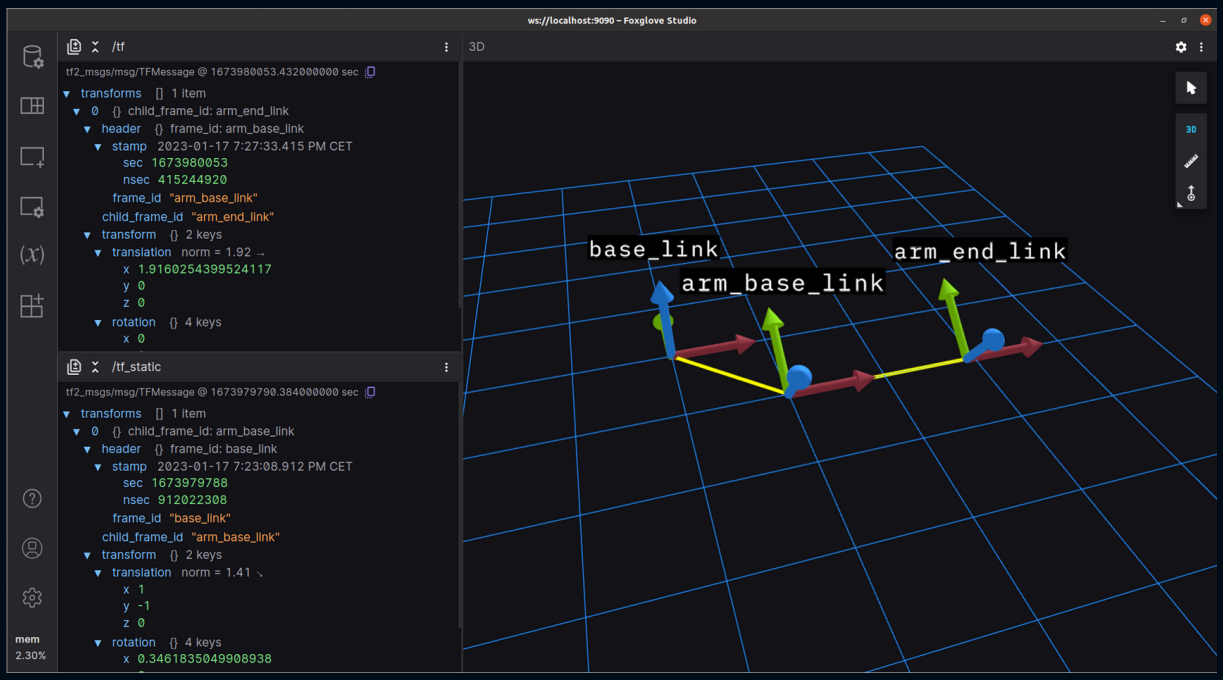


Figure 2.2. Visualizing ROS 2 Transforms with Foxglove Studio

2.3. State of WASM

2.3.1. Unity in WebAssembly

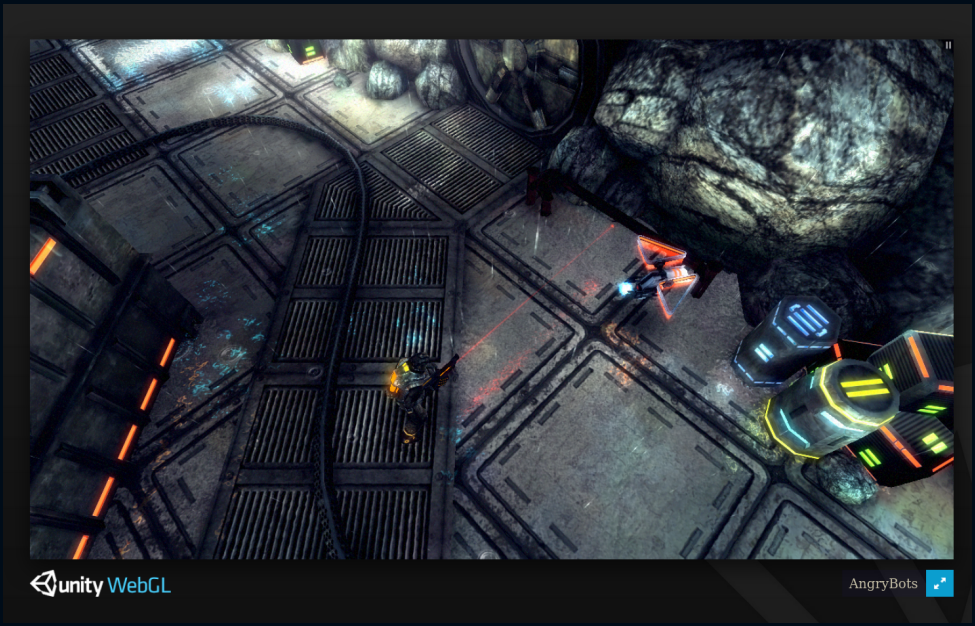


Figure 2.3. Demo of Angry Bots in Unity WebGL

3. Concept Realization

This section provides the major milestones from the project beginning with a brief description of the overall concept solution to the challenges presented in the Introduction, followed by the layers of implementation accomplished during the development phase.

3.1. Target Scenario

To introduce the concept, a "target scenario" is first considered. In this scenario, an intermediate ROS user should be able to reach a high level of usability with the tools developed in this project. First, an intermediate user is described as an individual who is familiar with the ROS ecosystem but does not have the need to maintain or test ROS packages across different platforms. In the target scenario, this intermediate user will be capable of performing the following tasks:

- install pre-compiled ROS 2 packages in the browser
- launch nodes including publishers, subscribers, servers, and clients
- interact with the environment to obtain information about running nodes, this would include echoing topics, listing parameters, reviewing log files, etc.
- visualize Universal Robotic Description Format (URDF) files, transforms, point clouds, markers, etc.
- play and record bag files
- connect with robots via bluetooth

Outside of this scenario, another goal for this project includes making the developed tools available to the general public by distributing them as open-source software. This will allow other roboticists to compile their own packages and share them on the web.

3.2. Implementation Layers

The development of this project is subdivided into multiple levels for the users, the interactions that the users have with the tools developed, and the technical difficulty of developing the tools. These subdivisions are beneficial in providing the reader with an illustration of the progressing stages of development of this project.

Note

If the reader would like to follow along with the demonstrations provided in the following pages, it is recommended to visit ros2wasm.dev. Throughout the text, links will be provided to redirect the reader to specific examples.

3.2.1. User Levels

For the purpose of establishing target users for the developed tools, potential users were categorized based on expertise level with ROS and programming in general. A summary of these levels can be observed in Table 3.1.

Table 3.1. Target users categorized by expertise level.

	User	Description
1	Beginner	Complete beginners who have never used ROS or programmed in any language.
2	Student	University students with basic programming experience.
3	ROS User	Students and researchers who actively use ROS for projects.
4	Roboticist	Robotics software developers including contributors to the ROS ecosystem.

Commencing with Level 1, the *Beginner* category is reserved for students in secondary education who have had little to no experience with programming, and therefore are not familiar with ROS. The tools developed in this project would serve as an initial introduction to robotics for this category of users.

Level 2 consists of university students who have completed elementary programming courses but have not yet been introduced to ROS. For this type of user, this project will provide essential tutorials to become acquainted with the inner workings of ROS.

With a slightly higher level of expertise, Level 3 comprises students or other enthusiasts who are already familiar with ROS and have collaborated in projects which use ROS as the main system to handle communications of multiple robotics elements. This ROS user is equivalent to the intermediate user described in the target scenario (Section 3.1).

Lastly, the highest level of experience is dedicated to roboticists who actively use ROS and contribute to its development. For this category of users, the intention of this project will be to involve more contributors in order to more promptly meet the needs of most ROS users.

3.2.2. Interaction Levels

The Graphical User Interface (GUI) is an essential element in the development of this project because it determines the benefits the users will receive by utilizing these tools. Similarly, the interface the user experiences with the tools has been categorized in increasing levels of interaction. These categories are summarized in Table 3.2.

Table 3.2. User Interface (UI) segmented based on the level of interaction.

	Interface	Description
1	Non-interactive	A publisher runs automatically as soon as the site is loaded.
2	Minimal	User can start/stop a publisher by pressing a button.
3	Basic	User can select and run publisher and subscriber nodes simultaneously.
4	Intermediate	Publishers, subscribers, and services are available, and the user can request basic information about the environment.
5	Advanced	A complete GUI where the user has full control of the environment, can start/stop nodes, modify parameters, manage bag files, and visualize robots.
6	Complete	All ROS 2 features are available and packages can be built on the browser, plus the user can directly connect and interact with external robots.

As the name implies, the *non-interactive* Level 1 does not offer the user any sort of interaction with the ROS environment. With this non-interactive interface, the user can do nothing more than load and reload the page. As soon as the user loads the page, a node which has been pre-compiled will automatically start running. In the simplest case scenario, a publisher node would run and the published messages would be displayed on the window. Because the user has no ability to interact with the environment, this publisher node will continue to run uninterrupted. The scenario described is illustrated in Figure 3.1.

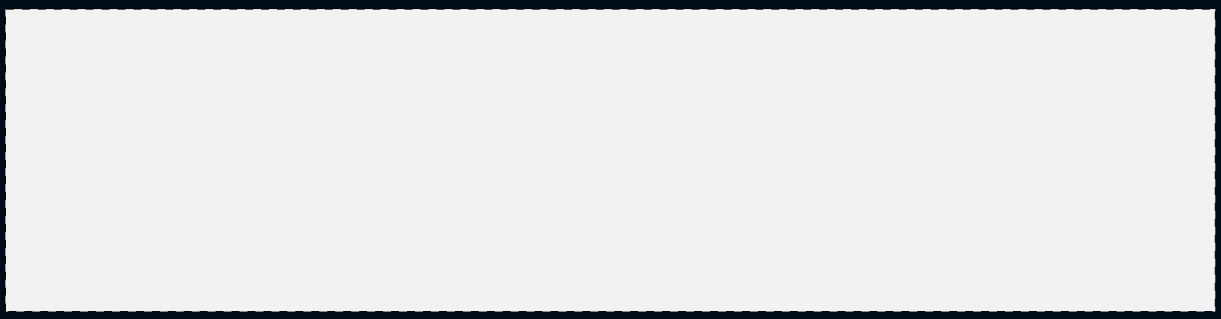


Figure 3.1. TODO Level 1 image

Example 1

A demonstration of a *non-interactive* user interface (Level 1) can be found at ros2wasm.dev/pages/demo01
NOTE: The page must be reloaded to restart the node.

By marginally expanding the interface, the *minimal* Level 2 provides the user with the ability to start and stop a pre-compiled node. This stage is accomplished by the addition of buttons to the website. Continuing with the example previously described, in this level the user can press a button to start a publisher node, view the output of any published messages, and stop the node at any point. Level 2 is exhibited in Figure 3.2

Figure 3.2. TODO Level 2 image

Example 2

A demonstration of a *minimal* user interface (Level 2) can be found at ros2wasm.dev/pages/demo02

The *basic* Level 3 offers the user increasingly more control over the environment. In this level, the user has the ability to run more than one node simultaneously. This makes it possible to have publishers sending messages to a particular topic and subscribers retrieving the published messages accordingly. Nonetheless, the nodes are still pre-compiled and thus the user does not have the ability to change the topic names or any other parameters of the nodes. Figure 3.3 demonstrates a snapshot of Level 3.

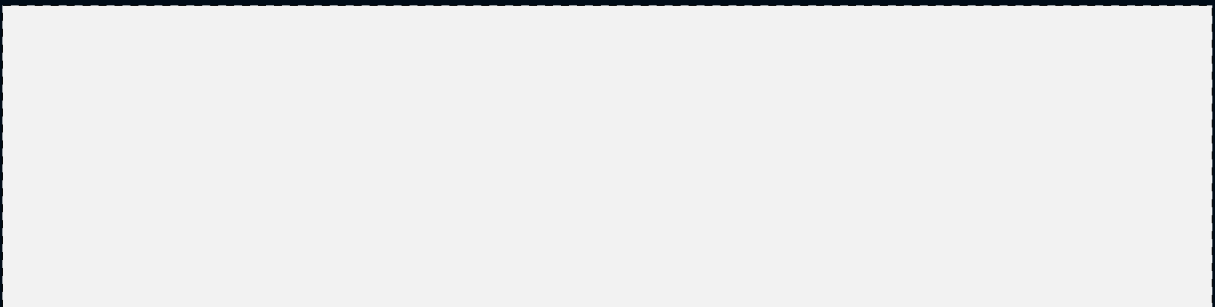


Figure 3.3. TODO Level 3 image

Example 3

A demonstration of a *basic* user interface (Level 3) can be found at ros2wasm.dev/pages/demo03

Stepping further into the list, the *intermediate* Level 4 introduces services, these include both the servers and the clients. Additionally, with this level the users now possess the ability to request information from the environment such as the type of nodes which are running at a given time, or which topics are available. A depiction of Level 4 is shown in Figure 3.4

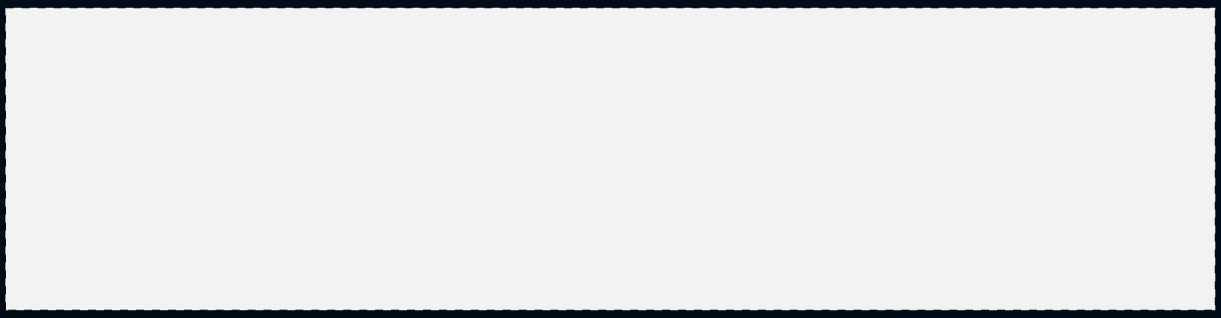


Figure 3.4. TODO Level 4 image

Example 4

A demonstration of an *intermediate* user interface (Level 4) can be found at ros2wasm.dev/pages/demo04

Finally arriving at the *advanced* Level 5, this level is more prominent because it introduces the integration of JupyterLite with the ROS environment. With JupyterLite, the user can directly interact with the ROS environment by using the ROS client libraries such as `rclpy`. A typical workspace in JupyterLite is pictured in Figure 3.5.

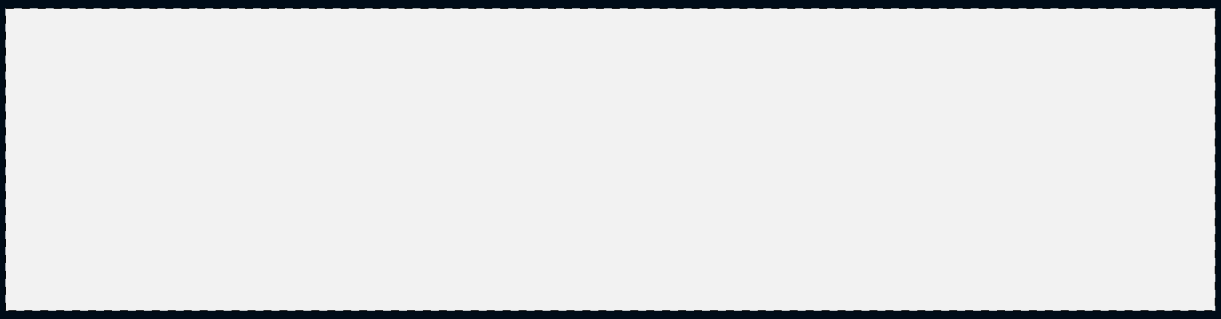


Figure 3.5. TODO Level 5 image

Example 5

A demonstration of an *advanced* user interface (Level 5) can be found at ros2wasm.dev/pages/demo05

Lastly, Level 6 consists of a *complete* ROS environment also made available to the user through JupyterLite. The user would be able to install any additional ROS packages from `emscripten-forge`. An extension for JupyterLite could enable communications with external robots by using the Web Bluetooth API. And a web compiler could be implemented to directly build packages within JupyterLite.

3.2.3. Complexity Levels

Table 3.3. Implementation categories with increasing technical complexity.

Level	Description
1	A publisher is displayed.
2	A publisher and subscriber can communicate with each other.
3	Multiple nodes and distinct topics can run simultaneously.
4	Graphical display and interaction with a ROS client library.
5	Manipulation of a physical robot via bluetooth or wifi.
6	Visualization of a robot with Zethus.
7	Simulation of a robotics scenario with Gazebo.
8	Development workspace for creating and debugging ROS packages.

4. Methodology

In the spirit of reproducibility, this chapter describes the tools and procedures used in the development of this project. Starting with the hardware, the requirements for this project were minimal. An Intel® Core™ i7-1065G7 CPU 1.30 GHz processor with a memory capacity of 15.2 GiB of RAM was used. The operating system consisted of Ubuntu 22.04. A system with substandard specifications can also be adequate for development.

4.1. Development Environment

For the sake of simplicity and to isolate the development environment from global dependencies, a conda environment was created. This conda environment was used to build the ROS packages required. The essential packages installed in the development environment are shown in Table 4.1. For quick installation, an environment yaml file is provided at TODO

Table 4.1. Development environment dependencies

Package	Version
python	3.10
setuptools[1]	58.2.0
numpy	1.24
lark[2]	1.1
rosdep	0.22
cmake	3.25
colcon-core	0.12
colcon-ros	0.3
colcon-package-selection	0.2
colcon-devtools	0.2
nodejs	18.12

4.2. Compilation Tools

Given that the colcon package is already well adapted to build ROS packages, colcon was widely used throughout this project with a few customizations. In order to cross-compile the ROS C++ and C packages to WebAssembly, the Emscripten SDK (emsdk) version 3.1 was installed. The Emscripten toolchain was then provided to colcon as a cmake argument (See Appendix C.1 Line 109).

[1] Version 58.2.0 of setuptools is the highest version that supports setup.py installs which many of the core ROS packages depend upon.
[2] The lark package is required for builtin_interfaces

4.3. Package Building Process

Before any packages could be built, all of the ROS 2 core packages were cloned in a local workspace. To prevent compilation issues, packages related to the default middleware implementations were excluded by adding a COLCON_IGNORE file in their respective root directories; these include iceoryx, cyclonedds, fastrtps, and connextdds. A secondary purpose for excluding these packages is to ensure that the custom middleware implementation is the only implementation available at runtime. This custom middleware implementation must also be included in the current workspace before proceeding.

Additionally, a "blasm" script was created to aid in the building of packages given that the number of arguments quickly became exceedingly lengthy for manual input. The entire script is included in Appendix C.1 for reference. With this script, it is possible to build any package and its dependencies, or to build a package individually. Care must be taken to ensure that the environment variables for the tools such as EMSDK_DIR are set accordingly. The list of options for this script are shown in Figure 4.1.

Figure 4.1. Blasm script options.

If the package compiles without errors, any executables in the package will be converted to *.wasm* and *.js* files to be run on the browser.

4.4. Debugging Tools

When working with new ROS packages, there are two main sources of errors: errors during compilation and errors at runtime. To tackle compilation errors, the first step is to obtain a full description of the error. The "blasm" script has the option to dynamically activate verbose (-v) mode as needed.

Secondly, for debugging errors at runtime, one of the most effective tools is the use of logs. Core ROS packages such as rcutils and rcpputils already include logging functionalities. Adapting these ROS logs into any newly created packages allows for efficient and quick debugging. Alternatively,

customized logging functions could be implemented for an in-depth analysis at the expense of increasing the complexity of the package in question.

4.5. Post Processing

Once the executables have been successfully compiled, the generated JavaScript files are augmented with additional functions to enable communication between the main thread and the ROS packages. The added functions are described in Appendix C.2. (TODO: OR provide link to file on GitHub)

4.6. Testing Environment

Lastly, there are two phases of testing. The first phase consists of testing the packages directly from the terminal. For the early stages such as during the replacement of the middleware implementation, cross-compilation is not yet required; thus, it is possible to locally test the middleware packages by comparing their behavior with the default middleware implementations. This can be achieved in a separate `conda` environment with ROS 2 packages preinstalled by creating and installing an overlay which only includes the customized middleware implementation.

And the second phase involves testing the packages on a web browser. For this project, only Firefox and Chrome were subject to testing due to their popularity. The tools developed in this project may be suitable for other browsers, however, their full functionality is not guaranteed.

5. Middleware Implementation

- What does the middleware do?

5.1. DDS Middleware

ROS supported middleware implementations

5.1.1. FastDDS

default

5.1.2. Eclypse

5.1.3. Gurum

5.2. Custom Middleware

Why it needs to be replaced

5.2.1. Email

5.2.2. Zenoh

Minimal implementation (minimal set of functions)

5.3. Substituting ROS 2 Middleware

At run time

At build time

5.4. Custom Middleware Design

Design of middleware packages (tree diagram or something)

5.4.1. wasm_cpp

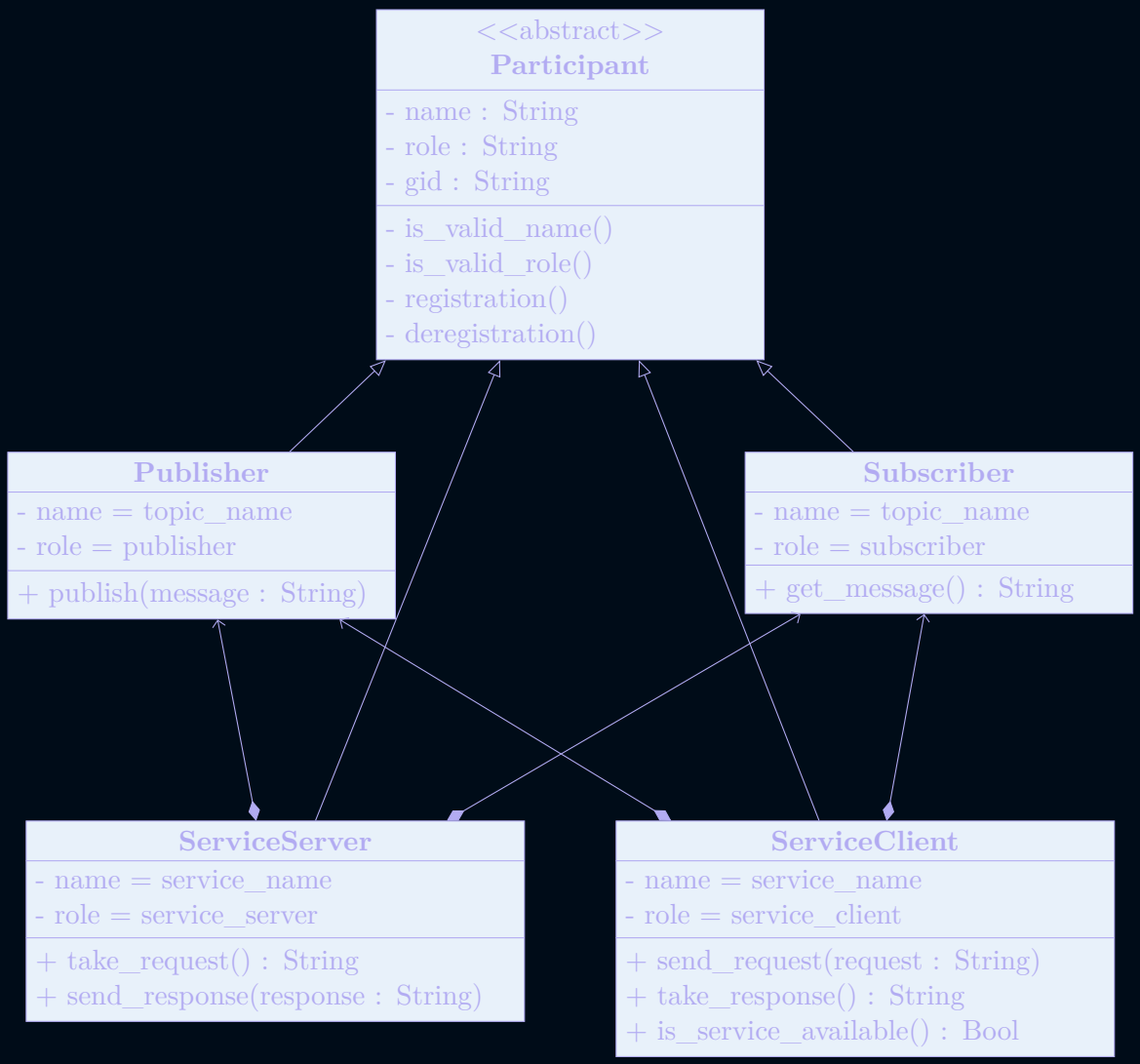
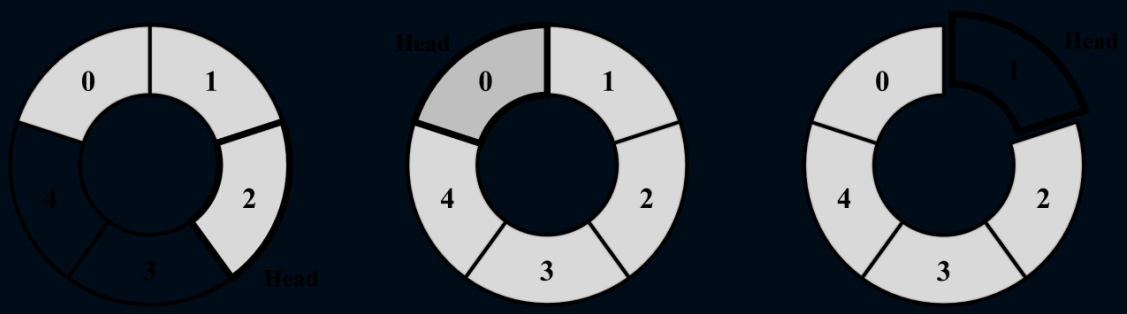


Figure 5.1. A class diagram

6. Design of Web Elements

6.1. Web Workers

6.2. Message Stacks



(a) Filling stack **(b)** Overwriting stack **(c)** Reading stack

Figure 6.1. Modified Circular Stack, Last In, First Out (LIFO)

- Web workers, what are they? why are they needed? - Communication channels - Registry of topics/subs/pubs - Message handling width

7. Package Management and Distribution

- Automating package building - Pipelines - robostack?

8. Concept Assessment

- Survey - Performance measures - Limitations

9. Summary

10. Outlook

- Compiling on the browser - Packaging Gazebo - WASI

Bibliography

[All23] Allwright, M.
ROS On Web
https://rosonweb.io/ (visited on 2023).

List of Tables

List of Figures

A. Illustrations

B. Tables

C. Code

C.1. Build Script

```
#!/bin/bash

#------------------------------------------------------------------
# HELP
#------------------------------------------------------------------
Help()
{
    echo -e "Options:"
    echo "-h     help"
    echo "-c     clean workspace"
    echo "-d     activate cmake debug mode"
    echo "-u     build up to package"
    echo "-s     build selected package"
    echo "-i     ignore \'listed packages\'"
    echo "-p     install emscripten python"
    echo "-v     verbose"
    echo -e "\n"
}

#------------------------------------------------------------------
# INSTALL PYTHON
#------------------------------------------------------------------
InstallPython()
{
    # Install emscripten python
    CONDA_META_DIR="${PWD}/install/conda-meta"
    [[ -d "${CONDA_META_DIR}" ]] || mkdir -p "${CONDA_META_DIR}"
    micromamba install -p ./install python --platform=emscripten-32 \
        -c https://repo.mamba.pm/emscripten-forge -y
    mv ./install/bin/python3 ./install/bin/old_python3
}

#------------------------------------------------------------------
# VARIABLES
#------------------------------------------------------------------
VERBOSE=0
EMSDK_VERBOSE=0
verbose_args=""
package_args=""
package_ignore="--packages-ignore rosidl_generator_py"
debug_mode=OFF
```

```
42
43 export RMW_IMPLEMENTATION="rmw_wasm_cpp"
44
45 #-----------------------------------------------------------------
46 # OPTIONS
47 #-----------------------------------------------------------------
48
49 while getopts "hcvpu:s:i:" option; do
50     case $option in
51         h) # Display help
52             Help
53             exit;;
54
55         c) # Clean workspace
56             [[ -d "${PWD}/install" ]] && rm -rf "${PWD}/install"
57             [[ -d "${PWD}/build" ]]   && rm -rf "${PWD}/build"
58             [[ -d "${PWD}/log" ]]     && rm -rf "${PWD}/log"
59             exit;;
60
61         d) # Activate cmake debug
62             debug_mode=ON
63             echo "[BLASM]: CMake debug mode activated.";;
64
65         v) # Make verbose
66             VERBOSE=1
67             EMSDK_VERBOSE=1
68             verbose_args="--event-handlers console_direct+"
69             echo "[BLASM]: Verbose activated.";;
70
71         p) # Install emscripten python
72             echo "[BLASM]: Installing python."
73             package_ignore=""
74             InstallPython;;
75
76         u) # Build up to package
77             package_args="--packages-up-to ${OPTARG}"
78             echo "[BLASM]: Build up to ${OPTARG}.";;
79
80         s) # Build selected package
81             [[ -d "${PWD}/build/${OPTARG}" ]] && rm -rf "${PWD}/build/${
    OPTARG}"
82             package_args="--packages-select ${OPTARG}"
83             echo "[BLASM]: Build only ${OPTARG}.";;
84
85         i) # Ignore "given packages"
86             package_ignore="--packages-ignore ${OPTARG}"
87             echo "[BLASM]: Ignore packages ${OPTARG}.";;
```

```
88
89          \?) # Invalid option
90              echo "Error: Invalid option."
91              Help
92              exit;;
93      esac
94  done
95
96  #-----------------------------------------------------------------
97  # MAIN
98  #-----------------------------------------------------------------
99  [[ -z "${package_args}" ]] && { echo "No args given."; exit 1; }
100 [[ -d "${PWD}/src" ]] || { echo "Not a workspace directory"; exit 1; }
101
102 echo "[BLASM]: Commencing build."
103
104 colcon build \
105     ${package_args} ${package_ignore} ${verbose_args} \
106     --packages-skip-build-finished \
107     --merge-install \
108     --cmake-args \
109         -DCMAKE_TOOLCHAIN_FILE="${EMSDK_DIR}/upstream/emscripten/cmake/
    Modules/Platform/Emscripten.cmake" \
110         -DBUILD_TESTING=OFF \
111         -DBUILD_SHARED_LIBS=ON \
112         -DCMAKE_VERBOSE_MAKEFILE=${debug_mode} \
113         -DCMAKE_FIND_ROOT_PATH_MODE_PACKAGE=ON \
114         -DCMAKE_CROSSCOMPILING=TRUE \
115         -DCMAKE_FIND_DEBUG_MODE=${debug_mode} \
116         -DFORCE_BUILD_VENDOR_PKG=ON \
117         -DPYBIND11_PYTHONLIBS_OVERWRITE=OFF
```

C.2. JavaScript Functions

```
1  // The Module object: Our interface to the outside world. We import
2  // and export values on it. There are various ways Module can be used:
3  // 1. Not defined. We create it here
4  // 2. A function parameter, function(Module) { ..generated code.. }
5  // 3. pre-run appended it, var Module = {}; ..generated code..
6  // 4. External script tag defines var Module.
7  // We need to check if Module already exists (e.g. case 3 above).
8  // Substitution will be replaced with actual code on later stage of the
       build,
9  // this way Closure Compiler will not mangle it (e.g. case 4. above).
10 // Note that if you want to run closure, and also to use Module
11 // after the generated code, you will need to define   var Module = {};
12 // before the code. Then that object will be used in the code, and you
```

```
// can continue to use Module afterwards as well.
var Module = typeof Module != 'undefined' ? Module : {};

function sleep(ms) {
     new Promise(resolve => setTimeout(resolve, ms));
}

let lastMessage = "data: empty";
let receivedNewMessage = false;
let topic = "";

self.onmessage = function(event) {
    // When a new message is received from main
    lastMessage = event.data;
    receivedNewMessage = true;
}

Module["registerParticipant"] = function registerParticipant(topic_name, role)
{
    topic = topic_name;
    let gid = Math.random().toString(16).slice(2)

    // Register new participant with main
    self.postMessage({
        command: "register",
        topic:   topic_name,
        role:    role,
        gid:     gid
    });

    return gid;
}

Module["deregisterParticipant"] = function deregisterParticipant(gid)
{
    // Deregister participant from main
    self.postMessage({
        command: "deregister",
        topic:   topic,
        gid:     gid
    });

    return;
}

Module["publishMessage"] = function publishMessage(message, topic_name)
```

```
{
    // Send message to main
    if (message.startsWith("data:")) {
        self.postMessage({
        command: "publish",
        topic:    topic_name,
        message: message
        });
    }

    // Assume it gets published
    return true;
}

Module["retrieveMessage"] = async function retrieveMessage(topic_name)
{
    receivedNewMessage = false;
    // Trigger main to send new message
    self.postMessage({
        command: "retrieve",
        topic:    topic_name
    });

    await sleep(100);

    return ( receivedNewMessage ? lastMessage : "" );
}
```