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 for the Development of a Web Browser Supported Robotics Environment

Master Thesis

by:

Isabel Paredes B.Sc.

Student number: 415723

supervised by:

M.Sc. Markus Schmitz and M.Sc. Wolf Vollprecht

Examiner:

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

Prof. Dr.-Ing. Mathias Hüsing

Aachen, 22 April 2023

Master Thesis

by Isabel Paredes B.Sc.

Student number: 415723

Cross-Compiling ROS2 Humble to WebAssembly for the Development of a Web Browser Supported Robotics Environment

---

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: M.Sc. Markus Schmitz and M.Sc. Wolf Vollprecht

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 for the Development of a Web Browser Supported Robotics Environment

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, 22 April 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, 22 April 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 for the Development of a Web Browser Supported Robotics Environment

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, 22 April 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, 22 April 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
SDK	Software Development Kit
JSON	JavaScript Object Notation
API	Application Programming Interface
VM	Virtual Machine
D3wasm	Doom 3 WebAssembly
MVP	Minimum Viable Product
DDS	Data-Distribution Service
OMG	Object Management Group
HTML	HyperText Markup Language
IDL	Interface Description Language
TCPROS	Transmission Control Protocol ROS
RTI	Real-Time Innovations
RTPS	Real-Time Publish Subscribe Protocol
DCPS	Data-Centric Publish Subscribe
QoS	Quality of Service
WebDDS	Web-Enabled DDS
RMW	ROS Middleware

YAML	YAML Ain't Markup Language
LED	Light Emitting Diode
VCS	Version Control System

1. Introduction

1.1. Robot Operating System 2

Robot Operating System (ROS) 2

1.2. Motivation

Web Assembly (WASM)

2. Relevant Works

TODO: Some intro

2.1. 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.

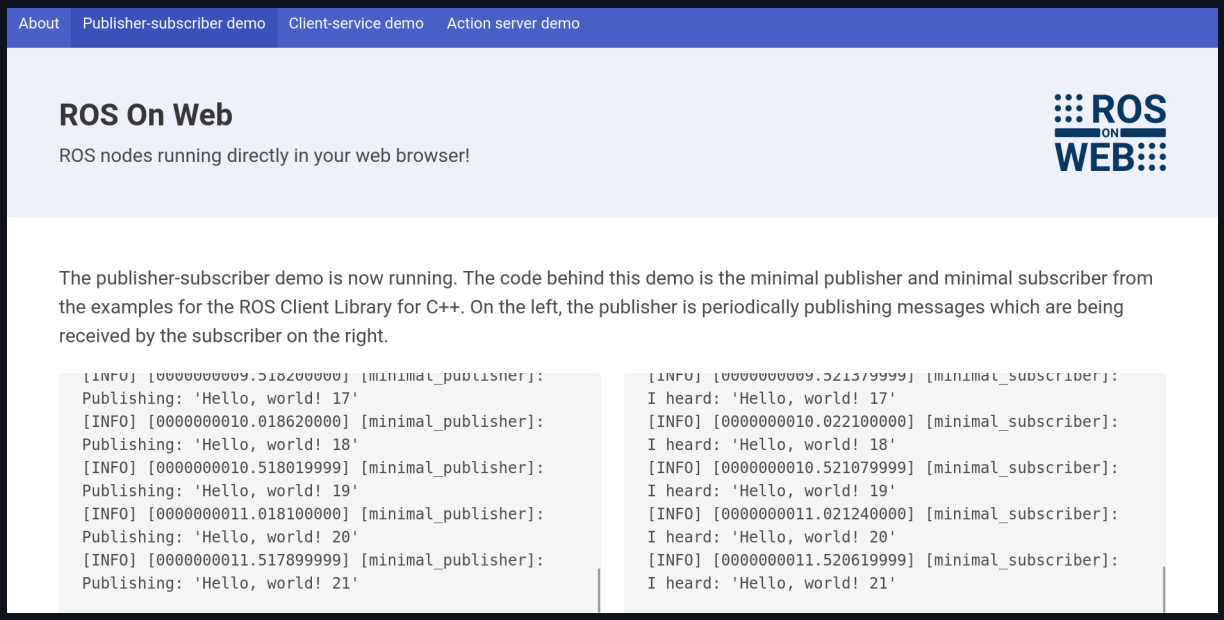


Figure 2.1. *ROS on Web* publisher and subscriber demonstration.

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. roswasm_suite

Another project closely related to the author's developments is roswasm_suite currently maintained mainly by Nils Bore. This suite is a set of libraries which help the user to crosscompile C++ ROS nodes to WebAssembly [Bor]. It also includes a library of helpers to write web Graphical User Interface (GUI)s for ROS; one such example can be observed in Figure 2.2.

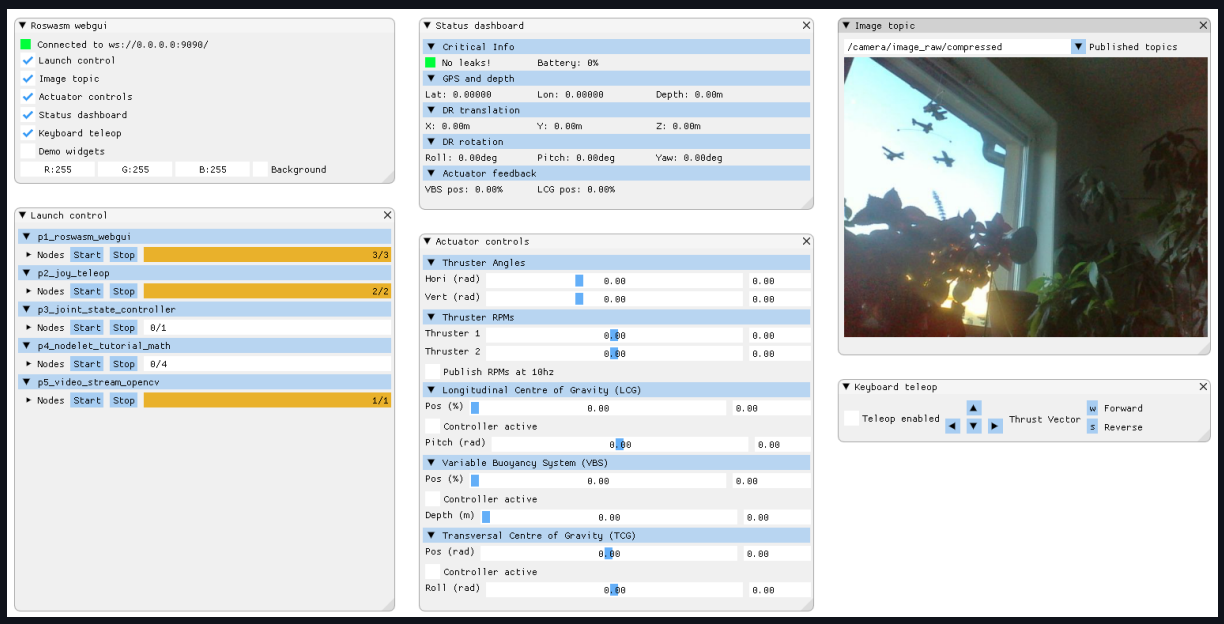


Figure 2.2. Example GUI for ROS using roswasm_gui.

Some of the biggest advantages of this roswasm_suite are that it is open source and actively maintained. The developers have a long history of continuous improvements to the packages, and as of 2023, they have managed to publish ten releases.

Nevertheless, these libraries do come with their set of disadvantages. First, the suite targets a ROS 1 distro and it depends on catkin tools to build the packages. Thus, the user is required to have a ROS 1 installation and the Emscripten Software Development Kit (SDK) before being able to use these libraries to launch ROS nodes on the browser.

2.3. Related Projects

There are several other works which do not directly align with the goals for this project but are pertinent in regards to the idea of developing a robotics environment which can be used on a web browser. Robot Web Tools maintains a collection of open source libraries and tools which can be used for robotic frameworks on the web [Ban].

2.3.1. Foxglove Studio

One of the most notorious examples of robotics applications on the browser consists of Foxglove Studio. The main focus of Foxglove Studio is visualization and debugging of robotic tasks [Sht]. Given that their products are based on observability and not on direct control of robotic systems, this allows them to expand their service area to a wider range of systems and platforms. For example, Foxglove Studio supports both ROS 1 and ROS 2 distributions and the application can be installed on Linux, Windows and MacOS or run directly in Google Chrome.

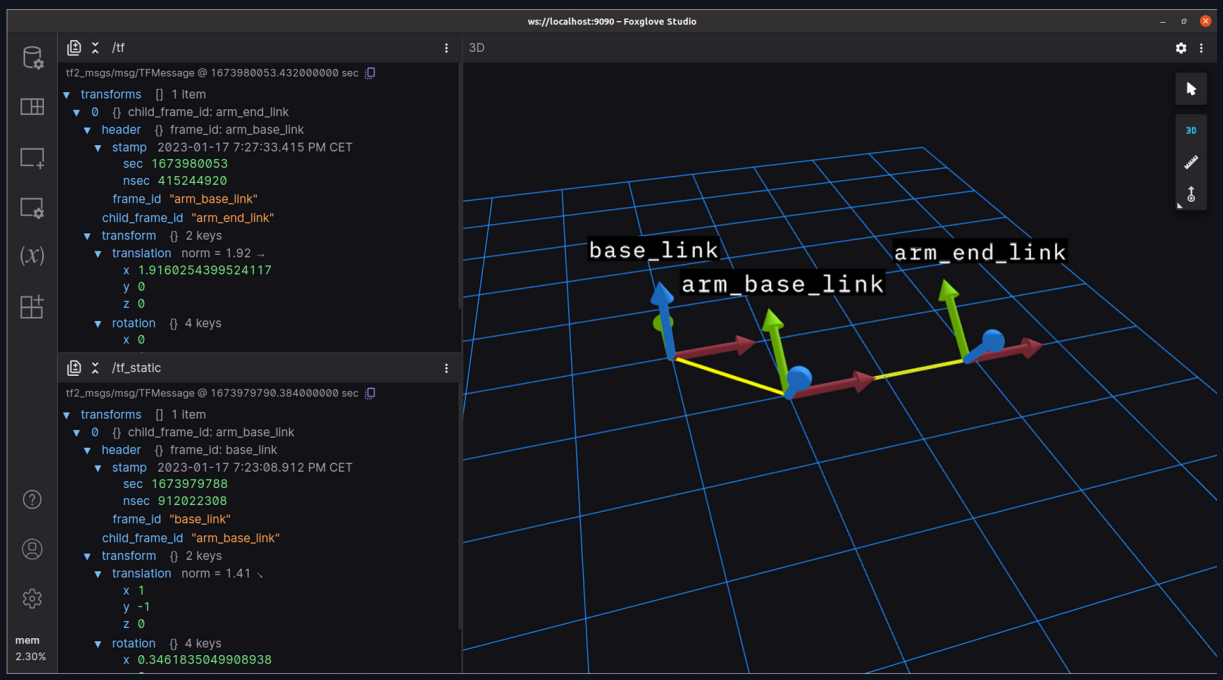


Figure 2.3. Visualizing ROS 2 Transforms with Foxglove Studio [Mil23b].

The wide range of visualization panels is one of the greatest strengths of Foxglove Studio. An example of a panel is shown in Figure 2.3 where ROS 2 transforms are drawn in a graphical display. Additional examples of panels include support for visualizing raw messages, image topics, plots, parameters, Universal Robotic Description Format (URDF) models, etc.

Another advantage of Foxglove Studio is that it is open source and actively maintained by community members. The application is also highly extensible with plugins to fit any specific needs of the users. Nevertheless, as a side effect of concentrating on observability, the application works as an add-on to an existing ROS installation and not as a replacement.

2.3.2. rosbridge_suite

On a similar note, the rosbridge_suite provides a set of packages which allow the user to interact with a ROS system via a JavaScript Object Notation (JSON) interface [Sch22]. Hypothetically, with these libraries, any non-ROS systems could interact with ROS by sending JSON messages. An instance of the rosbridge protocol can be seen in Figure 2.4.

```
{
    "op": "subscribe",
    (optional) "id": <string>,
    "topic": "/cmd_vel",
    "type": "geometry_msgs/Twist"
}
```

Figure 2.4. Example of rosbridge protocol emphasizing the JSON format.

The rosbridge_suite consists of four main packages:

- rosbridge_suite
- rosbridge_library
- rosbridge_server
- rosapi

From these packages, the rosbridge_library performs the conversions from JSON to ROS messages and vice versa. While the rosbridge_server provides the browsers with a WebSocket connection. A handful of Application Programming Interface (API)s enable clients to communicate with rosbridge; these APIs are implemented in JavaScript, Java, Python, and Rust.

Analogous to Foxglove Studio, rosbridge relies on an existing ROS installation. However, the capabilities of rosbridge are extensive and can be applied to real world scenarios; it also supports ROS 1 and ROS 2 distributions. Additionally, the community of contributors is highly involved in the maintenance of the client libraries.

2.3.3. ROS Control Center

The ROS Control Center is another web application which uses the rosbridge_suite to establish a WebSocket connection to a robotics system running ROS [Ber18]. A view of the control center is depicted in Figure 2.5.

With this ROS Control Center, the user is able to display information about running nodes, publish and subscribe to topics, call services, and change ROS parameters. Despite its potential, the control center does not offer explicit support for interacting with ROS 2 systems.

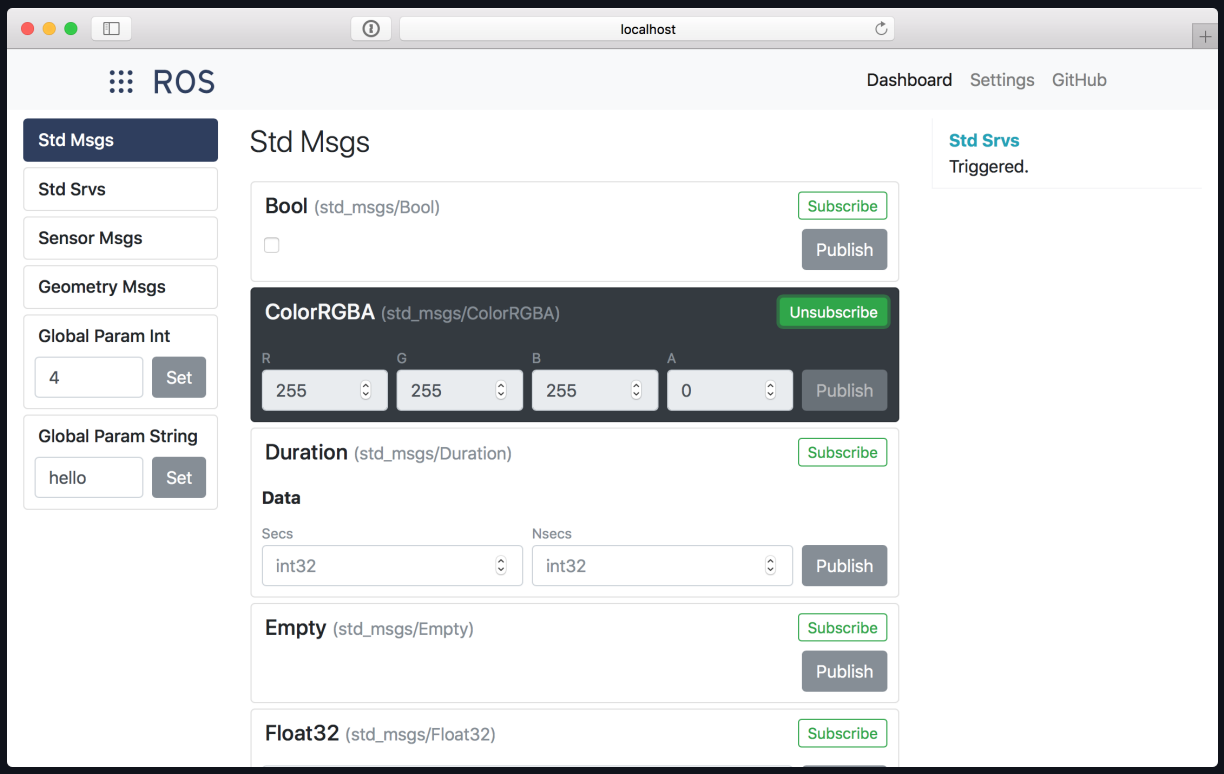


Figure 2.5. ROS control center running locally.

2.3.4. ROSWeb

ROSWeb combines all of the ROS widgets collected by Robot Web Tools into a single web application [Arr22]. As in the case of ROS Control Center (2.3.3), ROSWeb also depends on rosbridge to interact with ROS systems. Figure 2.6 demonstrates the ROSWeb application running on a Windows platform and communicating with a ROS system in a virtual machine running Ubuntu.

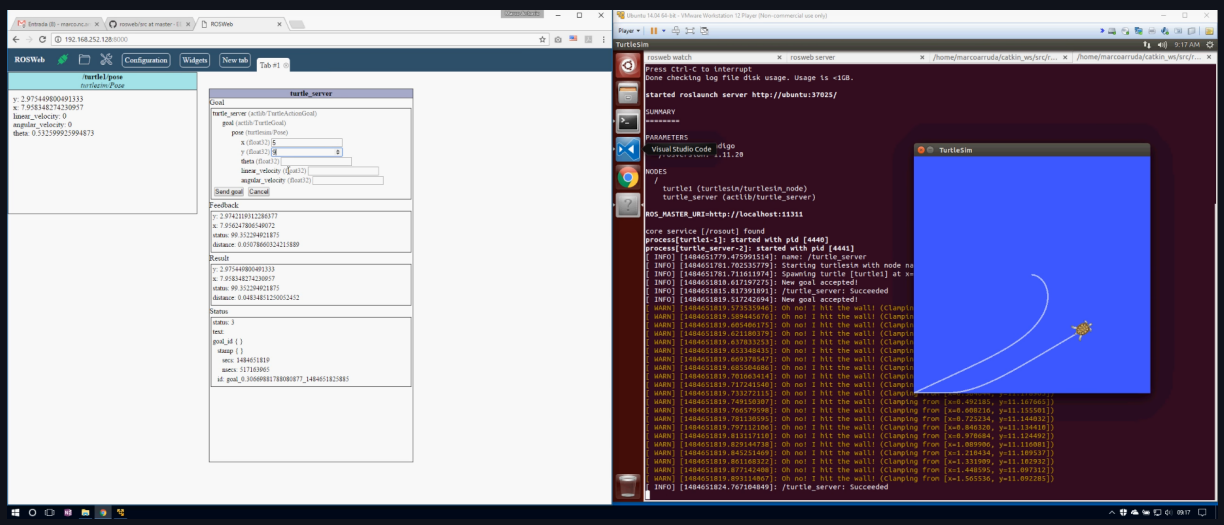


Figure 2.6. ROSWeb application interacting with a Virtual Machine (VM) running ROS.

A major advantage of ROSWeb application is its simplicity. The ROS widgets are interactive

and require minimal programming experience from the users. It also enables the users to easily save and modify workspaces to match their needs.

2.3.5. ROSboard

In contrast to the previous two models, ROSboard does not rely on rosbridge_suite to interact with ROS systems; instead, it has a custom Tornado implementation to function as a web server and WebSocket server [Ven21]. Figure 2.7 shows and example of the ROSboard application with different visualization widgets.

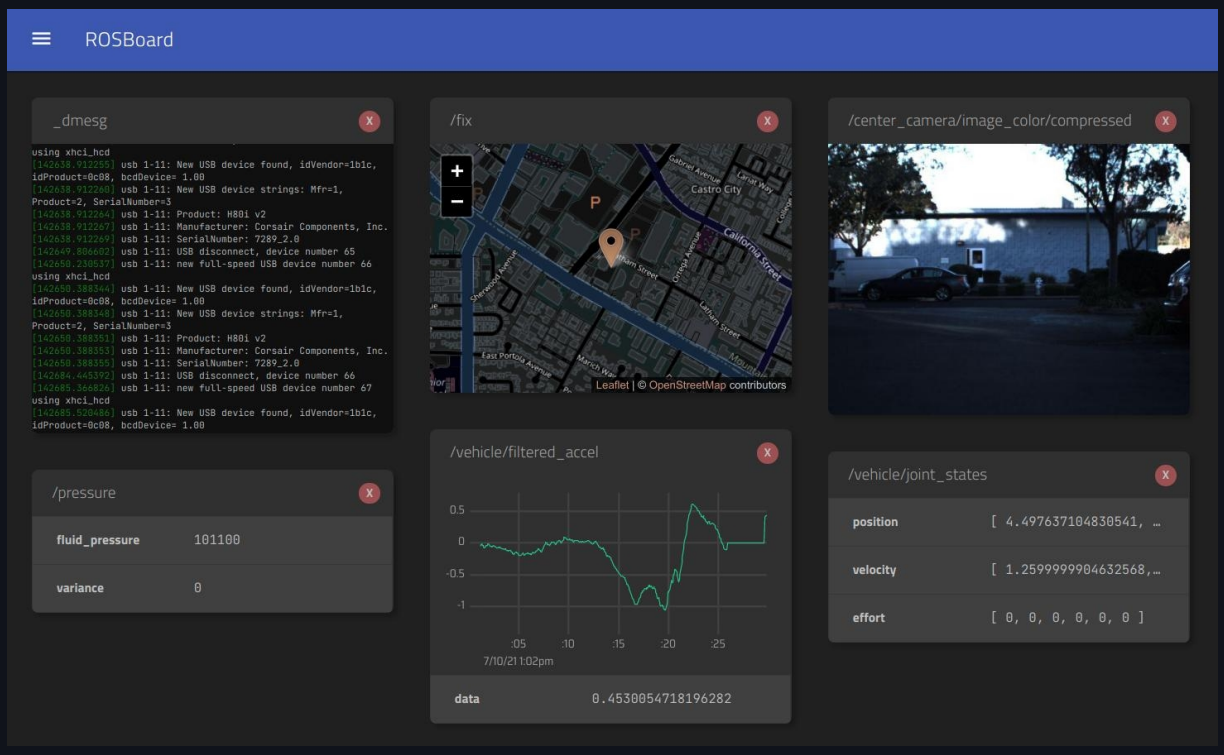


Figure 2.7. ROSboard application visualizing multiple message types.

ROSboard offers support for both ROS 1 and ROS 2 systems thanks to its tailored rospy2 library which is based on ROS 1's rospy but modified to ensure compatibility.

2.3.6. ROSLink

TODO: maybe too old to be relevant

2.4. WebAssembly

WebAssembly, also commonly referred to as simply WASM, is one of the newer technologies to enter the web arena. It was initially released in 2017 after representatives from the major browsers, Google Chrome, Microsoft Edge, Mozilla Firefox, and WebKit, agreed that the design of a Minimum Viable Product (MVP) was completed [Kri17].

In short, WebAssembly is a binary code format for a stack-based virtual machine [Ros23]. It is designed to be fast, safe, efficient, and portable; and it can be used inside or outside the browser. Inside the browser, WebAssembly works as a complement to JavaScript. Some of the use cases include: games, scientific visualizations, platform simulations, interactive educational software, virtual machines, developer tools, etc. [Tho23]. Because of its capabilities, it is believed to be an ideal target for the development of a robotics environment on the browser.

2.4.1. Applications

Before WebAssembly was officially released, one of the first demonstrations of its potential involved the game AngryBots developed by Unity Technologies. As illustrated in Figure 2.8, AngryBots is a first-person shooter game set in a space station filled with menacing robots.

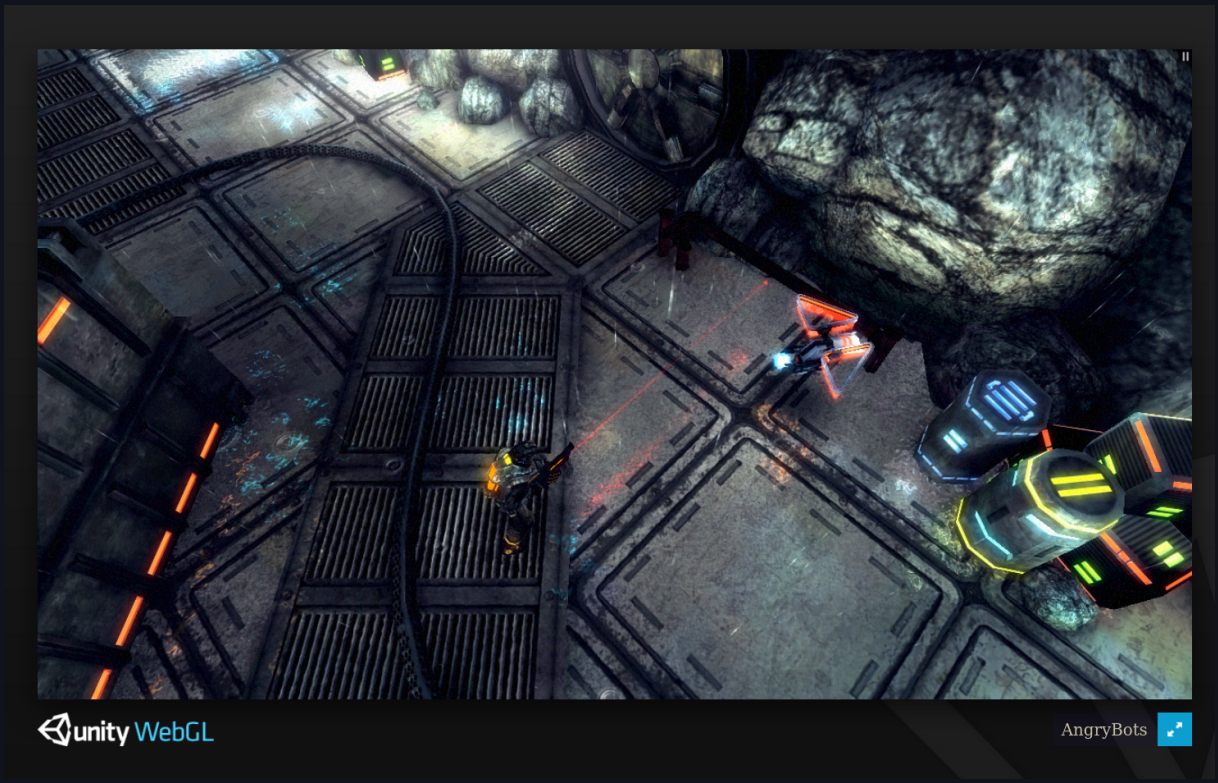


Figure 2.8. Demonstration of Angry ots in Unity WebGL [Ech16].

Similar to AngryBots, the Doom 3 Engine was ported to WebAssembly by using Emscripten. This experiment showed that very large and demanding C++ programs can be run inside a web

browser [Cuv22]. The game, pictured in Figure 2.9 is currently supported by all major desktop web browsers.

Figure 2.9. Online demonstration of the Doom 3 WebAssembly (D3wasm) project.

On a more classic note, the traditional Pong game has also been successfully ported to WebGL using Emscripten [Ben20]. Only a single player is supported in the current version of the game, as seen in Figure 2.10.

Aside from video games, WebAssembly has also been proven to work for other applications such as the web-based LaTeX editor shown in Figure 2.11, which uses a LaTeX compiler built with Emscripten; and the atmospheric simulation displayed in Figure 2.12. There exists several other use cases involving scientific visualizations and rendering.

All of the mentioned examples emphasize the ability of WebAssembly to be adapted to a wide range of applications. This feature is one of the most appealing characteristics for merging WebAssembly and robotics.

Figure 2.10. Classic Pong running on the browser.

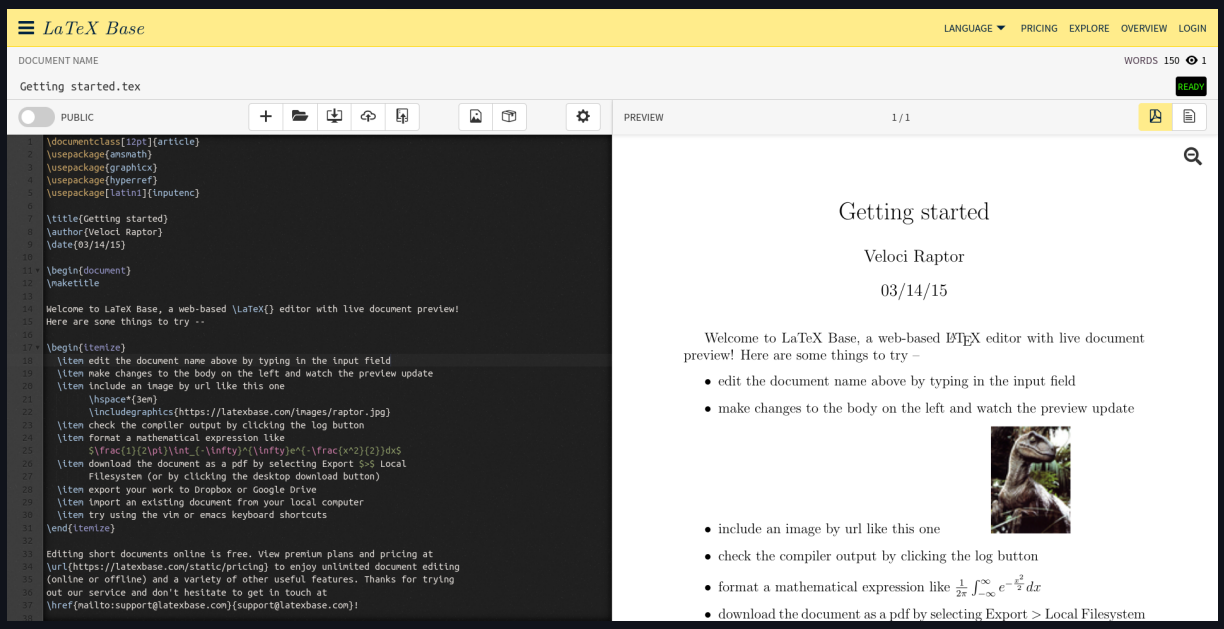


Figure 2.11. Web-based LaTeX editor built with Emscripten.

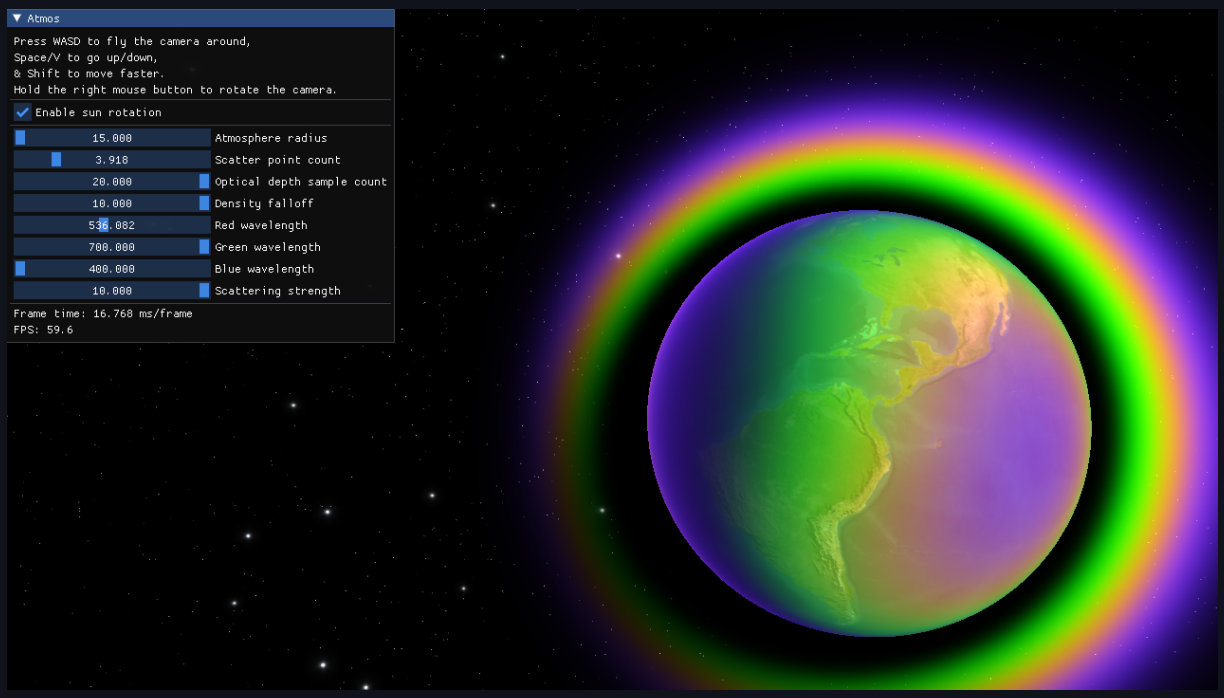


Figure 2.12. Atmospheric simulation [Mil23a].

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 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.

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 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.

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

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.

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.

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.

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.

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.

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`.

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

In regards to technical complexity of the project, a series of complexity levels is outlined in Table 3.3. The first four levels (1-4) must be accomplished in strict order, while levels 5-8 do not have any dependencies on the preceding levels to be able to function.

Table 3.3. Implementation categories with increasing technical complexity.

Level	Description
1	Replacement of the ROS middleware implementation.
2	A custom package and its dependencies can be cross-compiled to WebAssembly.
3	A publisher and subscriber can communicate with each other on the browser.
4	Multiple nodes and distinct topics can run simultaneously.
5	Manipulation of a physical robot via bluetooth or wifi.
6	Interaction with a ROS client library from JupyterLite.
7	Visualization of a robot with Amphion and Zethus.
8	Simulation of a robotics scenario with Gazebo.
9	Development workspace for creating and debugging ROS packages.

Although creating a replacement for the ROS middleware implementation denotes the lowest level on the list, its complexity must not be underestimated. Level 1 consists of creating custom middleware packages to handle discovery and communications between nodes. All other ROS packages depend on the middleware implementation, and because of this, creating a working replacement is crucial before any other features can be implemented.

Once the middleware packages have been introduced, the next level ensures that the core ROS packages can be cross-compiled. Level 2 involves the creation of a recipe to build all packages consistently. Minor modifications are needed in a few of the core packages to prevent compilation errors; for details, refer to TODO: add mods to appendix. And in order to set the custom middleware implementation as default, packages belonging to the other implementations must be ignored or removed.

Continuing the progression, Level 3 involves cross-compiling a package containing two executables, a publisher and a subscriber. Once these nodes are successfully loaded on the browser, communication between the nodes must be established by passing messages between web workers and the main thread.

Level 4 is simply an expansion of Level 3. In this case, the system handling communication between nodes can accommodate a multitude of different topics, message types, and node types. In other words, the communication system must be able to create and remove message queues at the time when they are requested by the individual nodes.

Starting with Level 5, the project begins to develop more practical applications. Manipulating a robot from the browser involves the addition of external libraries which enable communications to the hardware components of the robot. For example, a library which uses the Web Bluetooth API to establish a connection to a robot and sends commands commands to activate the robot's motors. This level also involves creating an interface which translates ROS messages to commands the robot can receive.

Level 6 requires Python support on the browser. This step consists of cross-compiling the ROS client library for Python (`rclpy`) in order to be able to import it from JupyterLite. The benefit of importing `rclpy` is that nodes could be created directly from the browser as shown in Figure 3.1.

```
import rclpy
web_node = rclpy.create_node("web_node")
```

Figure 3.1. Example of creating a node with rclpy.

Visualization of robots begins with Level 7. For this step, the Amphion and Zethus libraries are adapted to receive information from the ROS 2 nodes running on the browser. This entails replacing or modifying the dependency on `rosjslib` (the ROS 1 JavaScript library) which interacts with a native ROS 1 environment from the browser. Currently, `roslibjs` does not actively support ROS 2, but this is likely to change in the near future as more systems transition to ROS 2.

Unlike Amphion and Zethus, recent versions of Gazebo do support simulations for ROS 2 ecosystems. However, for Level 8, Gazebo's tools and libraries would need to be modified to run on the browser or communications must be established to Gazebo's cloud servers.

Lastly, Level 9 culminates with the implementation of all the previous levels plus additional developer tools such as compilers and debuggers running on the browser which would be equivalent to developing ROS packages in a local machine.

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

TODO: more details

Given that the colcon package is already well adapted to build ROS packages, colcon was widely used throughout this project with a few customizations.

There are currently four ways to port projects to WebAssembly [Ste23]:

[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

- Writing WebAssembly directly
- Using AssemblyScript
- Targeting WebAssembly as output for Rust applications
- Using Emscripten for C/C++ applications

Since most ROS 2 packages are written in C/C++, the easiest solution was to use Emscripten. Emscripten is a compiler toolchain which takes C/C++ source code and outputs a wasm module, the JavaScript *glue* code and optionally an HyperText Markup Language (HTML) document, as illustrated in Figure 4.1. In terms of ROS packages, for each executable, for example a talker.cpp containing a publisher node, Emscripten will output three files per executable: talker.wasm, talker.js and talker.html.

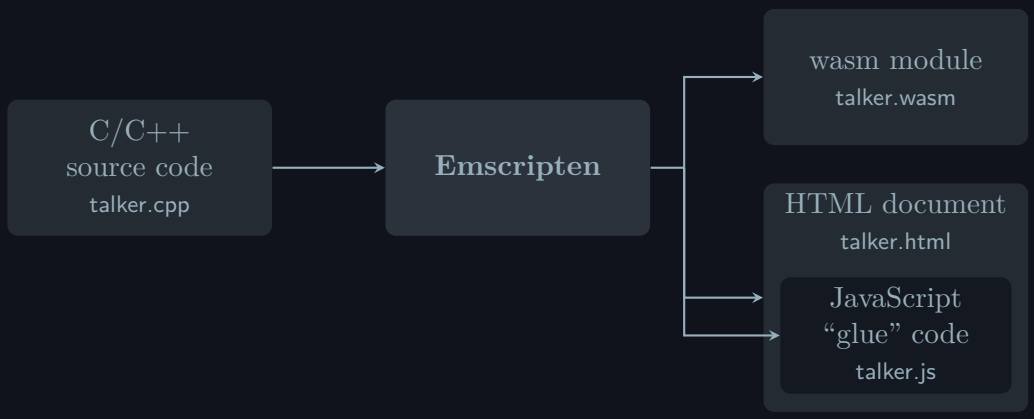


Figure 4.1. Transformation of C/C++ code to WebAssembly through Emscripten [Ste23].

In order to cross-compile the 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).

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 4.2.

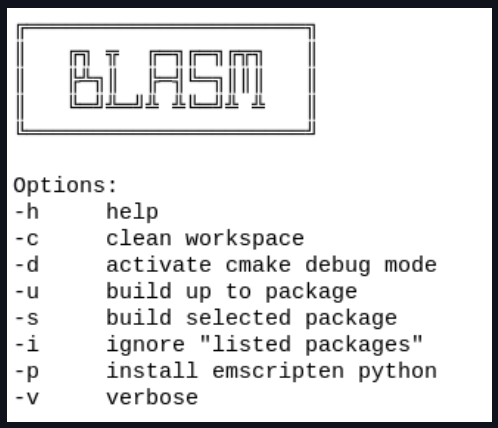


Figure 4.2. Blasm script options.

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.3. (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.

4.6.1. Package Management and Distribution

TODO: - Automating package building - Pipelines - robostack?

5. ROS 2 Middleware

A significant change from ROS 1 to ROS 2 is the shift from a custom transport layer consisting of Transmission Control Protocol ROS (TCPROS) to Data-Distribution Service (DDS). DDS is a publish-subscribe communication standard defined by Object Management Group (OMG). DDS uses Interface Description Language (IDL) for defining and serializing messages [Woo19]. In contrast to ROS 1, which requires a ROS master in order for nodes to discover and communicate with each other, ROS 2 discovery system is handled by DDS and each of the DDS vendors provides different options for customizing the communication layer.

One notable advantage of moving away from a custom transport protocol is that the ROS client libraries are now agnostic to the middleware interface; this means that the complexities of the DDS implementation are not exposed to the end user [Tho14]. As a consequence, multiple middleware interfaces can be implemented as long as they fulfill the following requirements:

- publishing and subscribing
- message serialization
- discovery

The interaction between the ROS user, the ROS client libraries, and the middleware layers is shown in Figure 5.1.

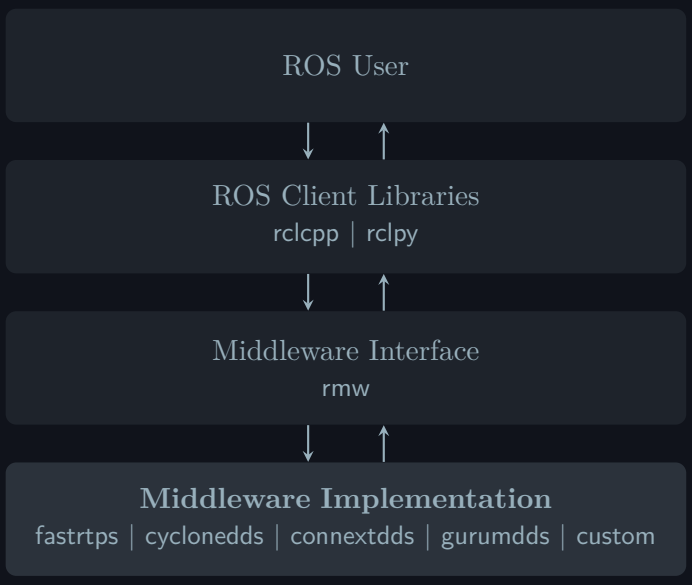


Figure 5.1. Relations between the user, the ROS client libraries and the middleware packages [Tho14].

5.1. Supported Implementations

Currently, ROS 2 releases provide full support for three middleware implementations: eProsima Fast DDS, Eclipse Cyclone DDS, and Real-Time Innovations (RTI) Connext DDS. The binaries also support Gurum DDS, but the implementation requires a separate installation [Lor22].

5.1.1. eProsima Fast DDS

eProsima Fast DDS, also known as Fast Real-Time Publish Subscribe Protocol (RTPS), is the default middleware implementation for ROS 2 packages. Some of the main advantages of Fast DDS is that it is free, open source, and it is developed for most platforms including Linux, Windows, Mac OS, and QNX. A rich set of Quality of Service (QoS) parameters is also available for tuning the communication protocols to any particular system. Fast DDS follows a Data-Centric Publish Subscribe (DCPS) model, which consists four elements: publishers, subscribers, topics, and domains [Pon20a]. This model introduces the concept of Data Writers and Data Readers which, as the names imply, have read and write permissions to the "Global Data Space" as specified by the DDS standard [Pon20b]. Figure 5.2 displays an example of the Fast DDS architecture and demonstrates how the different elements interact with each other.

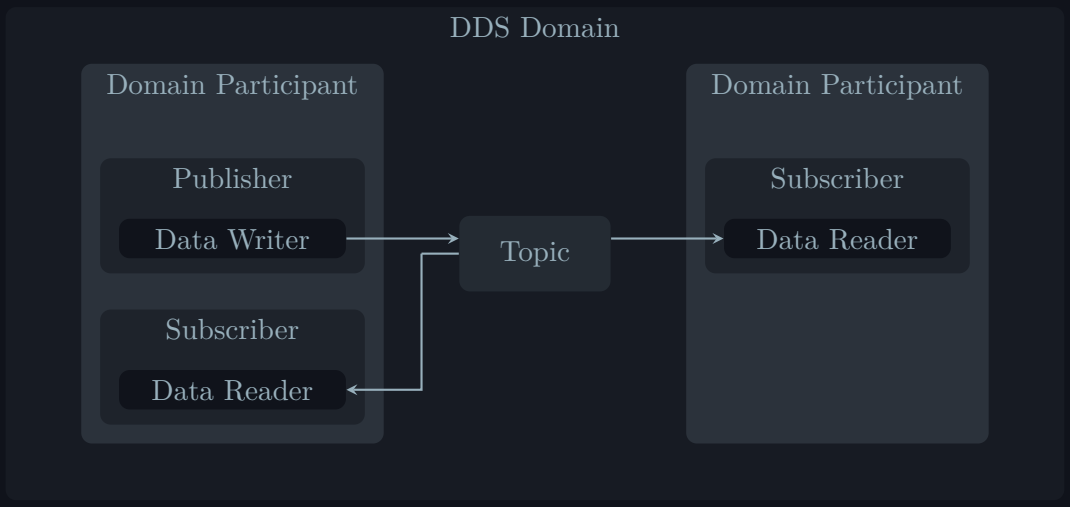


Figure 5.2. Instance of a typical Fast DDS domain model.

5.1.2. Eclipse Cyclone DDS

Similar to Fast DDS, Cyclone DDS is free and open source and supports the three major platforms, Linux, Windows, and Mac OS. Cyclone DDS offers a "data-centric" architecture with space- and time-decoupling with a zero configuration discovery system [Fou23]. Additionally, this implementation includes Python bindings to simplify the definition of data types.

5.1.3. RTI Connext DDS

Unlike the previously mentioned DDS implementations, RTI Connext DDS requires a separate installation plus the purchase of a license, however, free licenses are available for researchers and academics [Weo23]. RTI also offers a variety of tools to ROS users including admin consoles, system monitors, and recording and routing services [Put18].

5.1.4. Gurum Networks Gurum DDS

The last implementation which is supported by the ROS 2 binaries is GurumDDS. And as in the case of Connext DDS, GurumDDS also requires its own installation after the purchase of a license, although, free trials are available [Yun23].

5.2. Custom Middleware

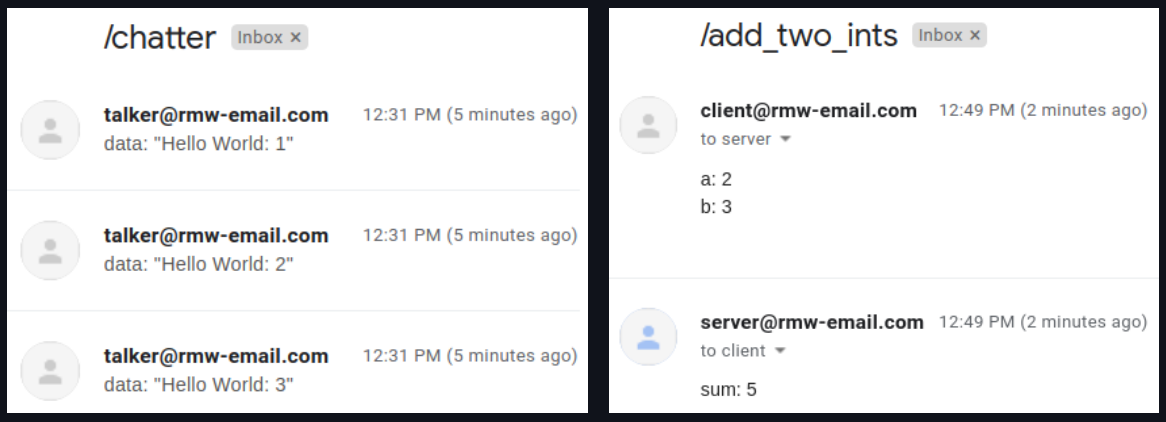
Out of the aforementioned middleware implementations, none of them have targeted the web browser as the basis for the DDS Global Data Space. There exists a Web-Enabled DDS (WebDDS) standard, also specified by OMG, which dictates how a DDS system can be exposed to web clients via a WebDDS service [16]. However, this setup would still require a native DDS system. As an alternative to DDS, custom middleware packages can be implemented as long as they meet the basic requirements listed in Section 5.

5.2.1. Email

An excellent example of a middleware implementation which is not based on the DDS standard is rmw-email[1]. With this implementation, all of the ROS 2 communications for publishing and subscribing to topics and to call and respond to services are handled by sending emails.

The implementation consists of two parts: email for handling the communications and rmw-email which acts as an adapter to interface with rmw [Béd21]. Figure 5.3 shows a ROS publisher, a service client, and a service server using this email middleware. Although rmw-email does not reach the level of performance of the standard DDS implementations, it showcases the flexibility of the additional abstraction layer integrated in ROS 2 to support various middleware designs.

[1] For readability, package names with underscores, e.g. rmw_email, are written with dashes.



(a) Email publisher **(b)** Email service client and server

Figure 5.3. Email ROS Middleware (RMW) implementation.

5.2.2. Zenoh

Continuing the trend of non-DDS middleware implementations, another approach involves the use of Eclipse Zenoh. Zenoh is a publication, subscription and query protocol to unify data which can be used in embedded micro-controllers or even data centers [Cor23]. Open Robotics has tested the potential of using Zenoh as a middleware for ROS 2 applications and concluded that it could alleviate some of the common problems encountered with the default DDS implementations by providing the users with hassle-free experience which does not require configuration and tuning unlike its DDS counterparts [Big22].

Similar to rmw-email, the interface between Zenoh and ROS 2 is handled by rmw-zenoh. As of 2023, rmw-zenoh still remains in the experimental phase. One notable aspect of this implementation, is the adaptation of Fast CDR to provide type support for Zenoh.

5.2.3. Minimal Middleware Implementation

The development of a custom middleware implementations consist of two major parts: the creation of a middleware "adapter" to communicate with the ROS 2 middleware interface, and the formation of the middleware implementation itself. An overview of a custom middleware architecture is demonstrated in Figure 5.4.

There are no standardized guidelines for how the middleware implementation should be formatted or what platforms it should support. The design of the implementation is highly dependent on the use case of the ROS application. The only design requirements are that the implementation should have the means to publish and subscribe data, perform message serialization from and to ROS message types, and have a discovery system to detect other participants in the specified domain. For this project, all of the communications are handled by the web browser and thus, the middleware is implemented in JavaScript.

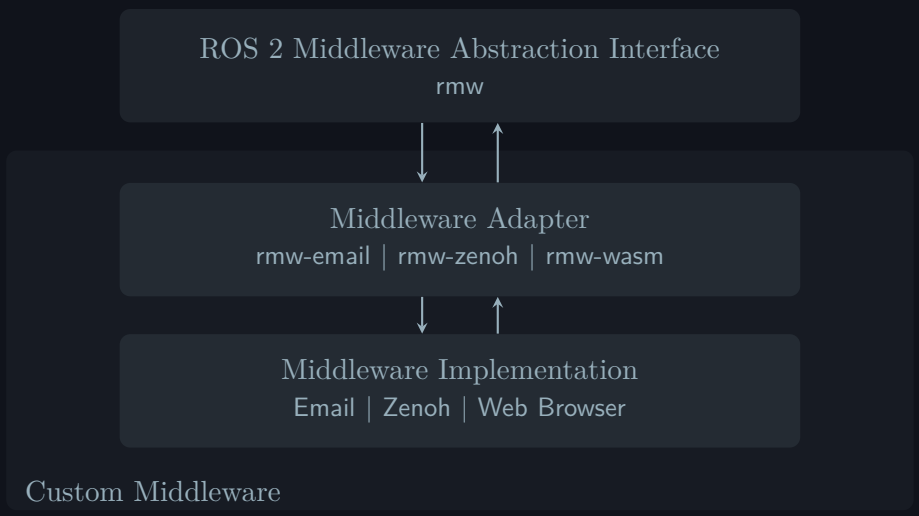


Figure 5.4. Architecture overview of a ROS 2 custom middleware implementation.

For the creation of a middleware "adapter," rmw provides a set of primitives which are needed for higher level ROS APIs. Generating the adapter requires implementing all of the relevant interface functions defined by rmw. Figures 5.5 through 5.10 highlight a few of the necessary interface functions.

```
// Return a zero initialized context structure
rmw_context_t   rmw_get_zero_initialized_context (void) {...}

// Initialize the middleware with the given options and yield a context
rmw_ret_t   rmw_init (
    const rmw_init_options_t *options,
    rmw_context_t *context) {...}

// Shutdown the middleware for a given context
rmw_ret_t   rmw_shutdown (rmw_context_t *context) {...}

// Finalize a context
rmw_ret_t   rmw_context_fini (rmw_context_t *context) {...}

// Return a zero initialized init options structure
rmw_init_options_t  rmw_get_zero_initialized_init_options (void) {...}

// Initialize given init_options with the default values and implementation
    specific values
rmw_ret_t   rmw_init_options_init (
    rmw_init_options_t *init_options,
    rcutils_allocator_t allocator) {...}

// Copy the given source init options to the destination init options
rmw_ret_t   rmw_init_options_copy (
    const rmw_init_options_t *src,
    rmw_init_options_t *dst) {...}

// Finalize the given init_options
rmw_ret_t   rmw_init_options_fini (rmw_init_options_t *init_options) {...}
```

Figure 5.5. Functions for initialization and shutdown.

```
// Create a node and return a handle to that node
rmw_node_t *  rmw_create_node (
    rmw_context_t *context,
    const char *name,
    const char *namespace_,
    size_t domain_id,
    bool localhost_only) {...}

// Finalize a given node handle, reclaim the resources, and deallocate the node
    handle
rmw_ret_t   rmw_destroy_node (rmw_node_t *node) {...}

// Return a guard condition which is triggered when the ROS graph changes
const rmw_guard_condition_t *   rmw_node_get_graph_guard_condition (
    const rmw_node_t *node) {...}
```

Figure 5.6. Node functions.

```
// Create and return an rmw publisher
rmw_publisher_t *   rmw_create_publisher (
    const rmw_node_t *node,
    const rosidl_message_type_support_t *type_support,
    const char *topic_name,
    const rmw_qos_profile_t *qos_policies,
    const rmw_publisher_options_t *publisher_options) {...}

// Destroy publisher
rmw_ret_t   rmw_destroy_publisher (
    rmw_node_t *node,
    rmw_publisher_t *publisher) {...}

// Initialize a publisher allocation to be used with later publications
rmw_ret_t   rmw_init_publisher_allocation (
    const rosidl_message_type_support_t *type_support,
    const rosidl_runtime_c__Sequence__bound *message_bounds,
    rmw_publisher_allocation_t *allocation) {...}

// Destroy a publisher allocation object
rmw_ret_t   rmw_fini_publisher_allocation (
    rmw_publisher_allocation_t *allocation) {...}

// Return a rmw_publisher_options_t initialized with default values
rmw_publisher_options_t   rmw_get_default_publisher_options (void) {...}

//  Publish a given ros_message
rmw_ret_t   rmw_publish (
    const rmw_publisher_t *publisher,
    const void *ros_message,
    rmw_publisher_allocation_t *allocation) {...}
```

Figure 5.7. Publisher functions.

```
// Create and return an rmw subscription
rmw_subscription_t *  rmw_create_subscription (
    const rmw_node_t *node,
    const rosidl_message_type_support_t *type_support,
    const char *topic_name,
    const rmw_qos_profile_t *qos_policies,
    const rmw_subscription_options_t *subscription_options) {...}

// Destroy subscription
rmw_ret_t   rmw_destroy_subscription (
    rmw_node_t *node,
    rmw_subscription_t *subscription) {...}

// Retrieve the number of matched publishers to a subscription
rmw_ret_t   rmw_subscription_count_matched_publishers (
    const rmw_subscription_t *subscription,
    size_t *publisher_count) {...}

// Retrieve the actual qos settings of the subscription
rmw_ret_t   rmw_subscription_get_actual_qos (
    const rmw_subscription_t *subscription,
    rmw_qos_profile_t *qos) {...}

// Take an incoming message from a subscription
rmw_ret_t   rmw_take (
    const rmw_subscription_t *subscription,
    void *ros_message,
    bool *taken,
    rmw_subscription_allocation_t *allocation) {...}
```

Figure 5.8. Subscriber functions.

```
// Create an rmw service server that responds to requests
rmw_service_t *   rmw_create_service (
    const rmw_node_t *node,
    const rosidl_service_type_support_t *type_support,
    const char *service_name,
    const rmw_qos_profile_t *qos_policies) {...}

//  Destroy and unregister the service from this node
rmw_ret_t   rmw_destroy_service (
    rmw_node_t *node,
    rmw_service_t *service) {...}

// Attempt to take a request from this service's request buffer
rmw_ret_t   rmw_take_request (
    const rmw_service_t *service,
    rmw_service_info_t *request_header,
    void *ros_request,
    bool *taken) {...}

//  Send response to a client's request
rmw_ret_t   rmw_send_response (
    const rmw_service_t *service,
    rmw_request_id_t *request_header,
    void *ros_response) {...}
```

Figure 5.9. Service server functions.

Besides the essential functions for creating and destroying nodes, publishers, subscribers, and services, there exists additional functions to handle events, wait sets, guard conditions, QoS policies, and allocators. Additionally, there are utility functions and macros for handling errors, customizing log outputs and name validation. A full list of all 83 function headers derived from the rmw API documentation can be found in Appendix C.4 [Woo23].

5.3. Substituting ROS 2 Middleware

There are two methods of using a particular middleware implementations when launching ROS applications other than the default implementation. The first method requires setting the environment variable `RMW_IMPLEMENTATION` to the implementation identifier of choice. An example of how to launch a publisher node with this method is shown in Figure 5.11.

Nonetheless, the caveat of this first method is that the ROS 2 binaries installed must have support for the specified implementation [Sco22]. Alternatively, the second method involves rebuilding all of the desired ROS 2 packages from source and specify the `RMW_IMPLEMENTATION` as a CMake argument. When building from source, if the packages for multiple RMW implementations are available, all of them will be built starting with Fast DDS as the default. If Fast DDS is not available, then the default implementation will be selected by the next available

```
// Create an rmw client to communicate with the specified service
rmw_client_t *  rmw_create_client (
    const rmw_node_t *node,
    const rosidl_service_type_support_t *type_support,
    const char *service_name,
    const rmw_qos_profile_t *qos_policies) {...}

// Destroy and unregister a service client
rmw_ret_t   rmw_destroy_client (
    rmw_node_t *node,
    rmw_client_t *client) {...}

// Send a service request to the rmw server
rmw_ret_t   rmw_send_request (
    const rmw_client_t *client,
    const void *ros_request,
    int64_t *sequence_id) {...}

// Attempt to get the response from a service request
rmw_ret_t   rmw_take_response (
    const rmw_client_t *client,
    rmw_service_info_t *request_header,
    void *ros_response,
    bool *taken) {...}

// Check if a service server is available for the given service client
rmw_ret_t   rmw_service_server_is_available (
    const rmw_node_t *node,
    const rmw_client_t *client,
    bool *is_available) {...}
```

Figure 5.10. Service client functions.

```
RMW_IMPLEMENTATION=rmw_connextdds ros2 run demo_nodes_cpp talker
```

Figure 5.11. Launching a node with Connext DDS.

implementation identifier in alphabetical order [Lor22]. To ensure that only a single implementation is available for use, the simplest solution is to explicitly ignore or remove the packages belonging to undesired implementations from the workspace.

6. ROS 2 Middleware Implementation for WebAssembly

The design of a custom middleware implementation, rms-wasm, that can be cross-compiled to WebAssembly modules is divided into three distinct packages as observed in Figure 6.1. For this project, these packages were developed in the order shown, starting with rmw-wasm-cpp.

Figure 6.1. Architecture of custom middleware implementation to target WebAssembly.

6.1. rmw-wasm-cpp

The first package, rmw-wasm-cpp, works as the "adapter" between ROS 2 and the designed middleware. This package implements all of the functions required for rmw as described in Section 5.2.3. The source code for this package is entirely written in C++ and can be found: TODO:

TODO: add diagram (maybe)

In addition to the rmw function, one main objective of this package is to convert the ROS messages to YAML Ain't Markup Language (YAML) strings. This is accomplished with the help of a dynamic message introspection (dynmsg) package developed by Open Robotics. These YAML strings are send to the middleware implementation, and when a message is returned, it is then converted back from a YAML string into a ROS message.

6.2. wasm-cpp

The role of wasm-cpp is to implement the middleware and to function as a bridge to JavaScript modules. This package, along with wasm-js constitute the middleware implementation without the adapter, and thus, they can function independently of rmw-wasm-cpp.

The ROS elements in this package build on top of each other, where the smallest unit is a `Participant`. Any ROS subscriber, publisher, service server, service client, action server, or action client is simply a `Participant` with a specific role. Each `Participant` must have a valid name which follows the ROS guidelines (TODO: add ref for valid names). For publishers and subscribers, this name is their respective topic name. And for servers and clients, the name corresponds to the service or action name. Upon initialization, each `Participant` also receives a unique ID or `gid` which is used to keep track of individual participants. A class diagram illustrating the relations between the different elements is shown in Figure 6.2.

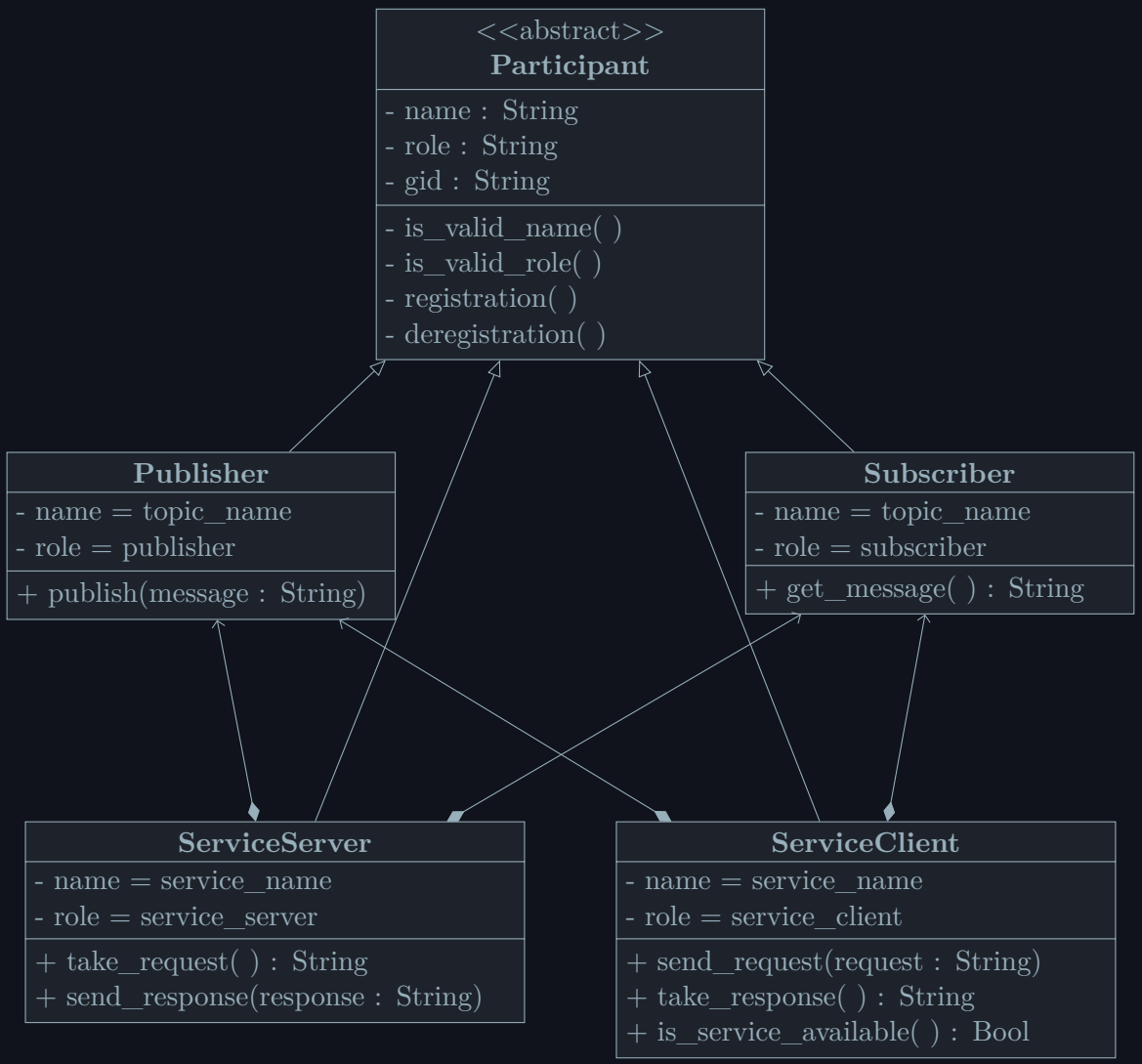


Figure 6.2. A class diagram of wasm-cpp.

Subscribers and publishers are the simplest type of participants. And each of them only has one function; a publisher needs to be able to *publish* a message, and a subscriber must be able to *retrieve* the message from a specific topic. The messages which the publisher handles are yaml strings. These messages are then passed to wasm-js by using `emscripten::val` which transliterates JavaScript code into C++ (TODO: add ref). Figure 6.3 shows how these functions are implemented to create a *bridge* between the two packages.

```cpp
// wasm_cpp/src/publisher.cpp
#include <emscripten/emscripten.h>
#include <emscripten/val.h>
...
emscripten::val js_publish = emscripten::val::module_property("publishMessage");
bool is_published = js_publish(yaml_message, topic_name).as<bool>();
```

```cpp
// wasm_js/src/pre.js
Module["publishMessage"] = function publishMessage(message, topic_name)
{
    if (message.startsWith("data:")) {
    self.postMessage({
        command: "publish",
        topic:    topic_name,
        message: message
    });
    }

    return true;
}
```

Figure 6.3. Publisher sending a message to JavaScript for publishing.

Conversely, the subscriber *waits* for a response from wasm-js which will indicate that a new message has been published to the topic in question. The response message type is coerced into a yaml string to send back to rmw-wasm-cpp. This process is illustrated in Figure 6.4.

```cpp
// wasm_cpp/src/subscriber.cpp
emscripten::val js_retrieve = emscripten::val::module_property("retrieveMessage");
emscripten::val js_response = js_retrieve(topic_name).await();
std::string yaml_message = js_response.as<std::string>();
```

Figure 6.4. Subscriber retrieving a message from JavaScript.

With the configuration described, subscribers and publishers do not necessarily need to be aware of the existence of other participants within the same topic name. That is to say, a publisher can send messages even when there are no subscribers, and a subscriber can listen to a topic even when there are no messages published. Additionally, this allows multiple publishers to publish to the same topic simultaneously with different ROS message types since all messages

are handled as strings.

Services are constructed from publishers and subscribers. A service server and a server client each have one publisher and one subscriber. Before a service client can call a service, it must verify that the service server is available to process a request. This is accomplished by making a query to wasm-js which keeps track of all available entities. If the service server is available, then the service client can *publish* a request to the service request topic. Meanwhile, the service server is *subscribed* to the service request topic to receive the request. Once the service server processes the request, the response tis *published* to the service response topic. And lastly, the service client *subscribed* to the response topic will retrieve the response. This process is summarized in Figure 6.5.

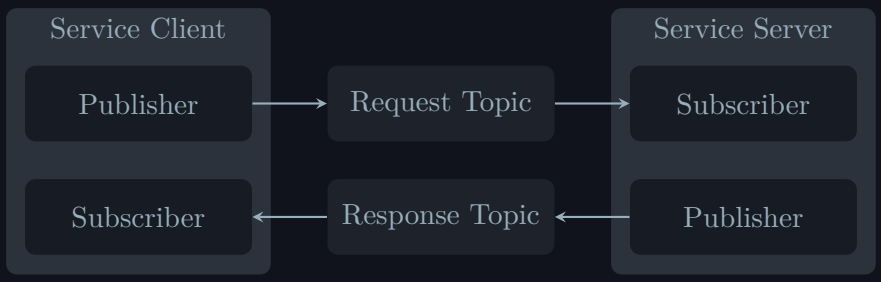


Figure 6.5. Data flow between ROS service client and server.

Analogously to services, action servers and clients are handled in the same manner with the addition that action servers will continuously publish feedback to the action response topic until the request being processed is completed.

6.3. wasm-js

The last piece of the puzzle is wasm-js. This package is written in JavaScript and its role is to launch and track all the participating entities, and to store and distribute all ROS messages accordingly.

Given that processes on a browser run on a single "main thread", wasm-js uses *web workers* to launch the requested ROS nodes. Web workers run in background threads and thus, they do not interfere with the user interface while performing tasks [23]. wasm-js spawns a new web worker for each ROS node. The ROS nodes can pass information to and from the main thread by posting messages (`postMessage()`). Each of the elements can also be configured to perform a task when a message is received by declaring an `onmessage()` function. Additionally, the workers can communicate between one another by creating message channels between worker pairs [Rau17]. These communication paths are displayed in Figure 6.6.

When the main thread receives messages from the web workers, it must not treat all messages equally. To accomplish this, each message contains a *command* which tells the main thread how to process the information within. The commands are limited to the following options:

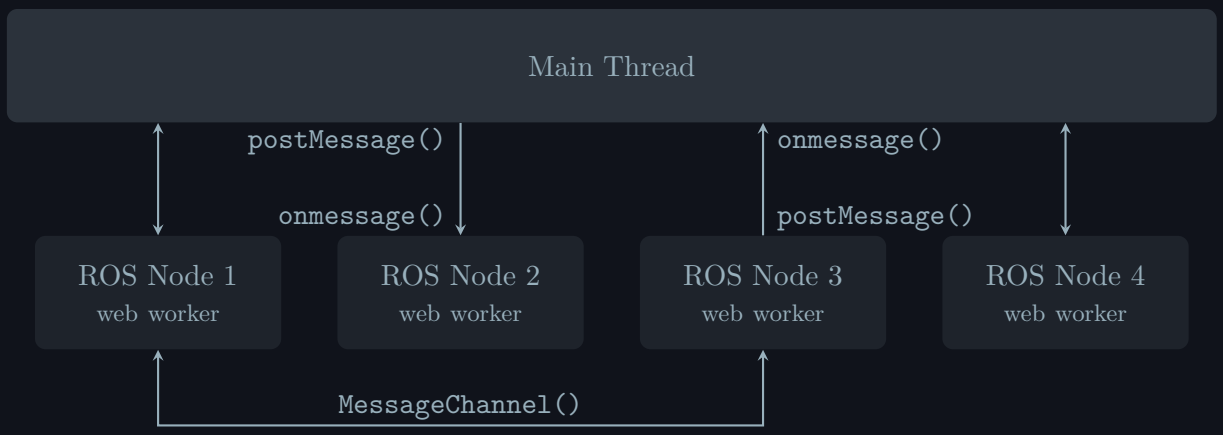


Figure 6.6. Communication between ROS nodes as handled by wasm-js.

- **`register`**. A message with this command will also contain the name, the role, and the unique ID of a new participant to be added to the registration map.
- **`deregister`**. This command type will provide the `gid` of the participant to be removed from the registration list; and in the case where this participant is also the last member of a topic or service, the message stacks for the particular topic or service will also be eliminated.
- **`publish`**. With this command, the message that the main thread receives will contain the ROS message as a string and the name of the topic to be able to store the ROS message in the correct message stack.
- **`retrieve`**. This command perform the opposite of publishing. The message data will contain the name of the topic where the most recent message should be retrieved to be able to select the message stack accordingly. And once the last ROS message has been taken back and removed from the stack, the main thread will send this ROS message back to the web worker which requested its retrieval.
- **`console`**. The message data sent with this command will include any errors, warnings, or simply output messages which would normally appear on a terminal. This helps make the information more accessible to the user.

To expand on how participants are added to the registration map; this map consists of multiple objects, where each object is responsible for storing all of the information relevant to a unique topic. Each object contains a list of ID's of topic members and a message stack. For services, two topic objects are created, one to store the information about service *requests* and the other to store service *responses*. Additionally, each node automatically creates the following subtopics for parameters:

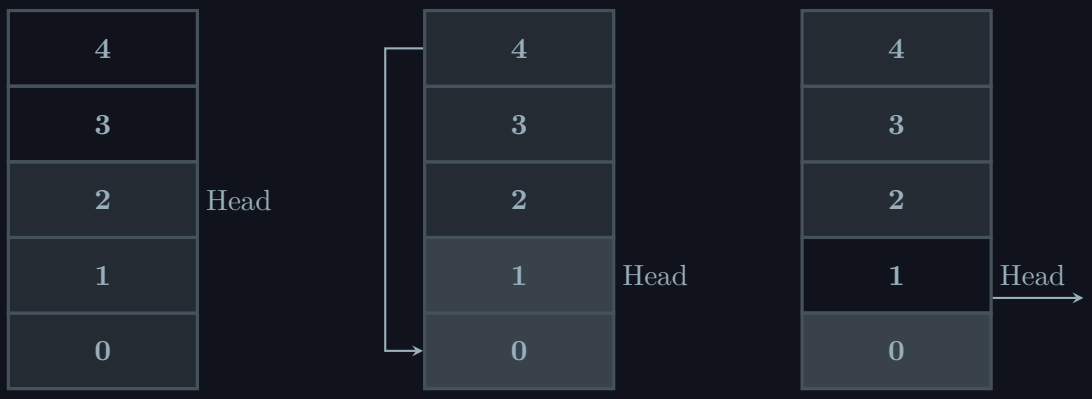
- /get_parameters

- /get_parameter_types
- /set_parameters
- /set_parameters_automatically
- /describe_parameters
- /list_parameters

The messages for each topic or service are stored in stacks similar to the ones shown in Figure 6.7. The default values of the QoS setting are the following:

- History → keep last. The stack will store the last n messages as opposed to storing *all* messages.
- Depth → 10. The stack is able to store a total of 10 elements. When more than 10 elements are stacked, the oldest element will be overwritten.
- Reliability → best effort. The delivery of messages is attempted but there is no guarantee that the messages are delivered. However, this setting is not affected by a poorly performing network, given the lack of network.
- Durability → volatile. There is no attempt to persists samples, only the most recent message is delivered.
- Deadline → ∞. There is no time limit between subsequent messages being published to a topic.
- Lifespan → 0. Once a message is published and there are subscribers to the topic, the message will be delivered to the subscribers immediately. Otherwise, the message will be considered stale and will be overwritten once the queue has filled.
- Liveliness → automatic. All node publishers are considered alive.
- Lease Duration → ∞. Publishers are considered alive without any indication from the publishers themselves.

The messages which wasm-js receives from wasm-cpp are YAML strings. These strings are stripped of new line characters ("\n") to prevent truncation when the messages are retrieved. By using js-yaml, the YAML strings can optionally be converted to JSON objects which enable simpler manipulation of the message data.



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

Figure 6.7. Modified circular stack, Last In, First Out (LIFO)

6.4. Data Flow

After conception at the origin node, a ROS message undergoes four type conversions before it is delivered to the target node. Initially, rmw-wasm-cpp will convert the ROS message to a YAML string and send it to wasm-cpp for processing. Once wasm-cpp has processed the YAML string, it sends it through the `emscripten::val` bridge to wasm-js. The YAML string is stored in one of the message stacks when it reaches wasm-js. Optionally, the string is converted to a JSON object and sent to externally interfacing packages.

In order to deliver the message to the target node, the YAML string is retrieved from a message stack and sent back to wasm-cpp. This package passes along the string to rmw-wasm-cpp which then converts the YAML string back to a ROS message. The flow between packages of the message type conversions is illustrated in Figure 6.8.

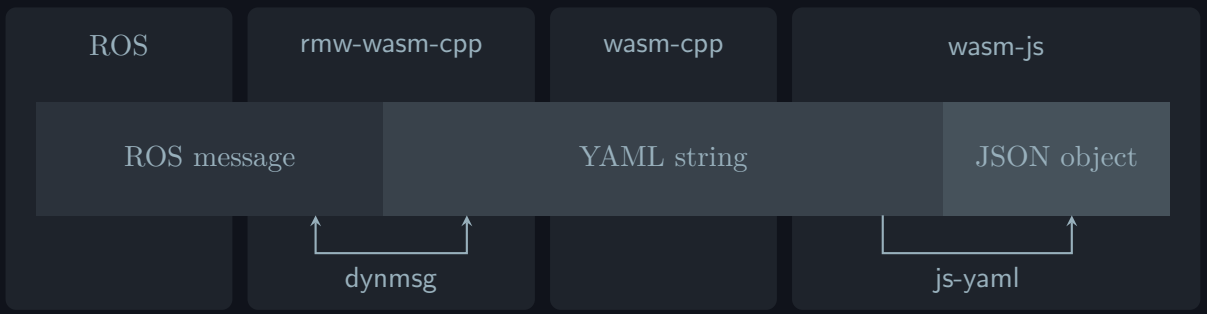


Figure 6.8. Message conversion flow.

7. Concept Assessment

The assessment of this project is calculated based on the number of interaction and complexity levels achieved as described in Tables 3.2 and 3.3. Illustrated in Table 7.1 is the relation between the level of user interaction and the technical complexity. The blue dots indicate the complexity level required to achieve a specific level of interaction. For example, interaction level $\mathcal{I}4$ (intermediate) requires a complexity of $\mathcal{C}5$ in order to be implemented. The highlighted section denotes the completed objectives for this project.

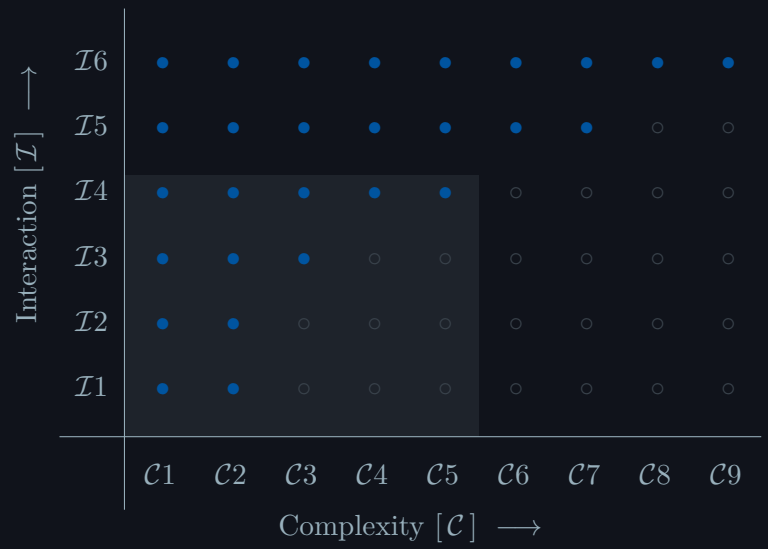


Table 7.1. Relation between user interaction and increasing technical complexity.

It is assumed that users will vary depending on the application. A beginner user, $\mathcal{U}1$, will benefit from an intermediate interface, $\mathcal{I}4$, when learning ROS concepts. However, a student with programming experience, $\mathcal{U}2$, will demand a more interactive ROS playground, $\mathcal{I}5$ and above. Similarly, experienced users, $\mathcal{U}3$ and $\mathcal{U}4$ will require all ROS tools to be available, $\mathcal{I}6$, to justify a migration to a web browser platform.

7.1. Layer Progression

The following sections describe the chronological progression of layers to achieve the results highlighted with gray in Table 7.1. As observed in the table, the progression through the levels is not a linear path, but a series of zigzag steps in multiple directions.

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.

7.1.1. Complexity Level 1

The groundwork for achieving any sort of user interface with ROS running on the browser, lies on the development of a compatible middleware implementation ($\mathcal{C}1$). The design of this middleware implementation has been described in Chapter 6. The three packages (rmw-wasm-cpp, wasm-cpp, and wasm-js) on their own, do not aid the user in interacting with ROS.

Example 1

The source code for the custom middleware implementation packages can be found in the following repository:
https://github.com/ihuicatl/rmw_wasm/

7.1.2. Complexity Level 2

The next step in complexity, $\mathcal{C}2$, involves creating a ROS package for testing and cross-compiling such package along with its dependencies. For simplicity, the test package follows the official ROS tutorials for how to create a publisher and a subscriber that communicate with each other in C++ [Lam23]. By having this standard baseline, this ensures that the behavior of the middleware implementation packages can be adjusted until it matches the expected behavior for a typical ROS publisher and subscriber. The CMake instructions were modified for each executable in order to have a successful cross-compilation with Emscripten.

Example 2

The testing package, test-wasm, used for this project can be found in the following repository:
https://github.com/ihuicatl/rmw_wasm/tree/main/test_wasm

To replicate the process locally, a build script (blasm) is provided in Appendix C.1. As pre-requisites, it is necessary to have an active installation of Emscripten and a copy of the ROS 2 source code in a workspace. Figure 7.1 shows the recommended procedure for cloning all ROS 2 packages by using a Version Control System (VCS) tool [Bro23].

An even simpler solution for new users to build packages is to use a GitHub workflow. An example of such workflow is given in Appendix C.2. By adding this script to a repository which

```
cd ~/workspace/src
vcs import --input https://raw.githubusercontent.com/ros2/ros2/humble/ros2.repos .
```

Figure 7.1. Cloning all ROS 2 packages with vcstool.

includes the desired ROS package to be built, the user can manually trigger this workflow to build the target package without the need to have a local build set up. Figure 7.2 displays the interface for running the build-package workflow from GitHub.

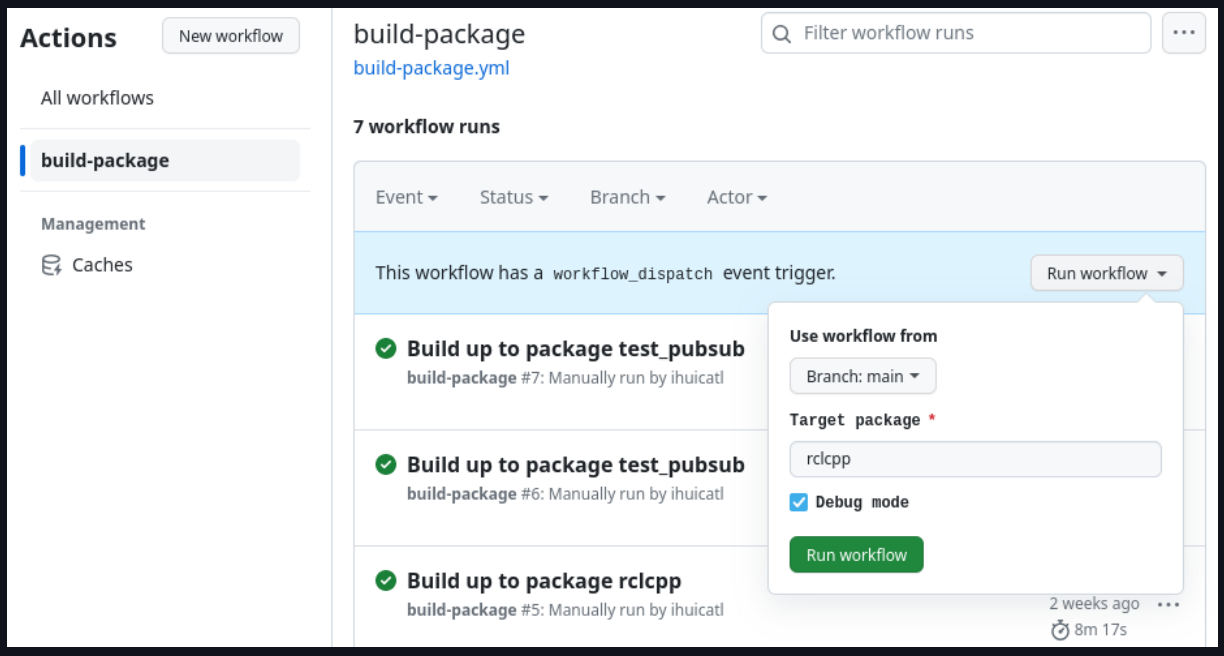


Figure 7.2. Triggering build-package workflow to build rclcpp.

The workflow runs through the following steps:

1. Install and activate emsdk.
2. Create a ROS workspace.
3. Install vcstool.
4. Clone all ROS 2 packages to the workspace.
5. Removes unsupported packages such as the default middleware implementations, e.g. FastRTPS.
6. Applies a patch to rcutils package.
7. Creates a conda environment with the colcon packages installed.
8. Runs the blasm script to build up to the target package.
9. And uploads the artifacts.

It is important to note that the artifacts are available for the user to download once the workflow has completed successfully. The contents of these artifacts will include any WASM, JavaScript, and HTML files generated by the target package during the build. However, the target package must contain executables in order for these files to be generated in the first place.

This build-package workflow provides an overview of the procedure required to build packages locally or on an external server. Although the initial setup may seem laborious, automation tools can greatly simplify the process.

Example 3

The source code for the described workflow can be found in the following repository:
https://github.com/ihuicatl/ros2wasm/blob/main/.github/workflows/build-package.yml
The reader is encouraged to fork the repository to test the workflow.

7.1.3. User Interaction Level 1: Non-Interactive

Once $\mathcal{C}1$ and $\mathcal{C}2$ levels are completed, this opens the door for minimal user interaction. With the testing package described above containing a publisher derived from the ROS tutorials, a WASM and a JavaScript are generated when cross-compiling the publisher executable. By creating a bare bones HTML page, the publisher (*talker*) can be run on the browser.

As indicated in Figure 7.3, the *talker* depends on wasm-js for handling messages and to display outputs on the page. Alternatively, these outputs can also be observed from the console. It is important to note that at this complexity level, $\mathcal{C}2$, wasm-js does not need to be fully equipped; it

```html
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Non-Interactive Publisher</title>
        <link rel="stylesheet" href="style.css">
        <script src="wasm_js/rosMain.js"></script>
        <script src="talker.js"></script>
    </head>

    <body>
        <textarea id="talkerOutput"  rows="10"></textarea>
    </body>
</html>
```

Figure 7.3. Minimal HTML page to run a publisher node on the browser.

suffices to receive a YAML string message from the publisher node and to display the information on the page. Since there are no subscriber nodes, messages can be discarded automatically.

This type of setup constitutes $\mathcal{I}1$ or non-interactive. The user has no control of the publisher node other than reloading the page to restart the publisher. The output from a publisher in this level is seen in Figure 7.4.

```
Publisher initializing.
[INFO] [1682069093.833000000] [wasm_cpp]: Context initializing.
[INFO] [1682069094.864000000] [wasm_publisher]: Publishing: 'Hello there! 0'
[INFO] [1682069095.877000000] [wasm_publisher]: Publishing: 'Hello there! 1'
[INFO] [1682069096.894000000] [wasm_publisher]: Publishing: 'Hello there! 2'
[INFO] [1682069097.906000000] [wasm_publisher]: Publishing: 'Hello there! 3'
[INFO] [1682069098.922000000] [wasm_publisher]: Publishing: 'Hello there! 4'
[INFO] [1682069099.940000000] [wasm_publisher]: Publishing: 'Hello there! 5'
```

Figure 7.4. Output from non-interactive $\mathcal{I}1$.

Example 4

A demonstration of a *non-interactive* user interface ($\mathcal{I}1$) can be found at ttps://ros2wasm.dev/pages/demo01/index.html

NOTE: The page must be reloaded to restart the node.

7.1.4. User Interaction Level 2: Minimal

With the addition of buttons and the introduction of web workers to the wasm-js middleware package, the user gains the bare minimum control over the publisher node. At this point, the

user can only start and stop the node. And as stated previously, without subscribers there is no need for persistent message storage. Figure 7.5 illustrates this *minimal* user interface, $\mathcal{I}2$.

Figure 7.5. Interactive buttons to start and stop the publisher node.

Example 5

A demonstration of a *minimal* user interface ($\mathcal{I}2$) can be found at https://ros2wasm.dev/pages/demo02

7.1.5. Complexity Level 3

Achieving $\mathcal{C}3$ requires two components. The first is the formation of a message storing system in wasm-js in order to allow publishers and subscribers to communicate with each other; this systems is composed of message stacks depicted in Figure 6.7.

The second component is developing a mechanism to retrieve the messages stored in the stacks. This functionality is not as trivial as it might seem. The message stacks live in the main thread, and the web workers can only communicate with the main thread via `postMessage( )`; however, the `postMessage( )` function does not offer any return values. Thus, in order to solve this the web worker will post a message to the main thread making a *request* to retrieve the most recent message for a particular topic. When the main thread receives this request, it will pop the relevant message from one of the message stacks and post the contents to *all* web workers available. Although this is not the most efficient strategy, it ensures that subscribers to the same topic receive the most recent message. Once the web worker receives a new message from the main thread, it will temporarily store this message so that whenever the next retrieve function is called, it can return the message contents via wasm-cpp. Figure 7.6 shows how the main thread handles requests to retrieve messages from web workers; in this case, the web worker containing a subscriber is called the *listener*.

```
// wasm_js/rosMain.js

// Receive messages from workers
let onMessageFromWorker = function( event ) {
    switch( event.data.command )
    {
        case "retrieve":
            let msgPopped = topicMap[event.data.topic].messages.pop();

            if (msgPopped !== null) {
                // Broadcast to all subscribers
                if (listener !== null) {
                    listener.postMessage({
                        topic: event.data.topic,
                        message: msgPopped
                    });
                };
                break;
            }
    }
}
```

Figure 7.6. Handling of requests by the web workers to retrieve messages from the stacks in the main thread.

7.1.6. User Interface Level 3: Basic

After $\mathcal{C}3$ is up and running, this permits the creation of a basic user interface, $\mathcal{I}3$. This consists of a publisher and a subscriber node which can both be started and stopped by the user. The messages from the publisher are sent to the main thread to be stored in a stack so that the subscriber can retrieve the messages as needed. This process follows the message flow shown in Figure 6.8. A demonstration of $\mathcal{I}3$ with the *talker* and *listener* running concurrently on the same web page is shown in Figure 7.7.

Example 6

A demonstration of a *basic* user interface ($\mathcal{I}3$) can be found at
https://ros2wasm.dev/pages/demo03/index.html

Publisher

START STOP CLEAR

```
[INFO] [1682068596.347000000] [wasm_publisher]: Publishing: 'Hello there! 16'
[INFO] [1682068597.361000000] [wasm_publisher]: Publishing: 'Hello there! 17'
[INFO] [1682068598.375000000] [wasm_publisher]: Publishing: 'Hello there! 18'
[INFO] [1682068599.394000000] [wasm_publisher]: Publishing: 'Hello there! 19'
[INFO] [1682068600.407000000] [wasm_publisher]: Publishing: 'Hello there! 20'
[INFO] [1682068601.321000000] [wasm_publisher]: Publishing: 'Hello there! 21'
[INFO] [1682068602.342000000] [wasm_publisher]: Publishing: 'Hello there! 22'
```

Subscriber

START STOP CLEAR

```
Subscriber initializing.
[INFO] [1682068598.495000000] [wasm_cpp]: Context initializing.
[INFO] [1682068598.812000000] [wasm_subscriber]: I heard: 'Hello there! 18'
[INFO] [1682068599.632000000] [wasm_subscriber]: I heard: 'Hello there! 19'
[INFO] [1682068600.675000000] [wasm_subscriber]: I heard: 'Hello there! 20'
[INFO] [1682068601.495000000] [wasm_subscriber]: I heard: 'Hello there! 21'
[INFO] [1682068602.513000000] [wasm_subscriber]: I heard: 'Hello there! 22'
Subscriber terminated.
```

Figure 7.7. Publisher and subscriber nodes running simultaneously.

7.1.7. Complexity Level 4

To establish $\mathcal{C}4$, changes were only required in the wasm-js packages. Compared to all the previous levels where only one message stack was required to store incoming messages regardless of the topic, $\mathcal{C}4$ makes a distinction between the topics. A topic *map* is created to store a message stack for each unique topic. The structure and information contained in the topic map is shown in Figure 7.8.

```
topicMap:
    topic_1:
        messages:    [stack]
        publishers:  [gid_1, gid_2, ..., gid_n]
        subscribers: [gid_1, gid_2, ..., gid_n]
    topic_2:
        messages:    [stack]
        publishers:  [gid_1, gid_2, ..., gid_n]
        subscribers: [gid_1, gid_2, ..., gid_n]

    ...

    topic_N:
        messages:    [stack]
        publishers:  [gid_1, gid_2, ..., gid_n]
        subscribers: [gid_1, gid_2, ..., gid_n]
```

Figure 7.8. Topic map to store message stacks for each topic.

Apart from storing the messages, the topic map stores information about the members for each topic. The members are subdivided into publishers and subscribers but these can also be service participants. As an example, a service server will subscriber member to a service request topic, as well as a publisher member to a service response topic. The information stored about these topic members is simply their unique identifier or `gid`. This `gid` is randomly generated when the participant is first created.

The message stacks stored for each topic are not (yet) configurable. However, they are initialized with default values to match the QoS settings described in Section 6.3. This means that all message stacks have a length of 10, and the oldest message is overwritten when it reaches capacity. Popping a message from the stack will remove that message from the *head* to provide the most recent message. A visualization is shown in Figure 6.7.

Another feature of $\mathcal{C}4$ is that there is no limit imposed on the number of members a topic can have or the number of topics in general. A topic branch is eliminated when there are no publishers or subscribers left. Similarly, the `gid` of a participant is removed from the relevant topic whenever the participant requests to be deregistered, this can happen when the node is terminated.

7.1.8. Complexity Level 5

Manipulation of a physical robot with $\mathcal{C}5$ marks the beginning of practical applications for this project. Data can be sent and received from a robot with the publishers and subscribers developed in the previous steps as long as there is a communication bridge between the robot and the web browser. There are two primary ways to communicate with a robot. The most widely used solution is configure a robot which to launch a `rosbridge_server` to send information to the browser. The constraint of this method is that the robot must already be running ROS to create a `rosbridge`.

An alternative for robots which do not run ROS natively is to create an interface between the browser and the robot's system. The bridge can be established via bluetooth, WiFi, or a wired connection. For this project, the LEGO BOOST Vernie robot was tested. Vernie can be connected to any computer via bluetooth. A pre-existing library, lego-boost-browser was used to established the interface between ROS and Vernie. This library uses Web Bluetooth API to control the LEGO BOOST robot [Tuh20]. Additional upgrades for this external lego-boost-browser package were performed by Thorsten Beier [Bei23].

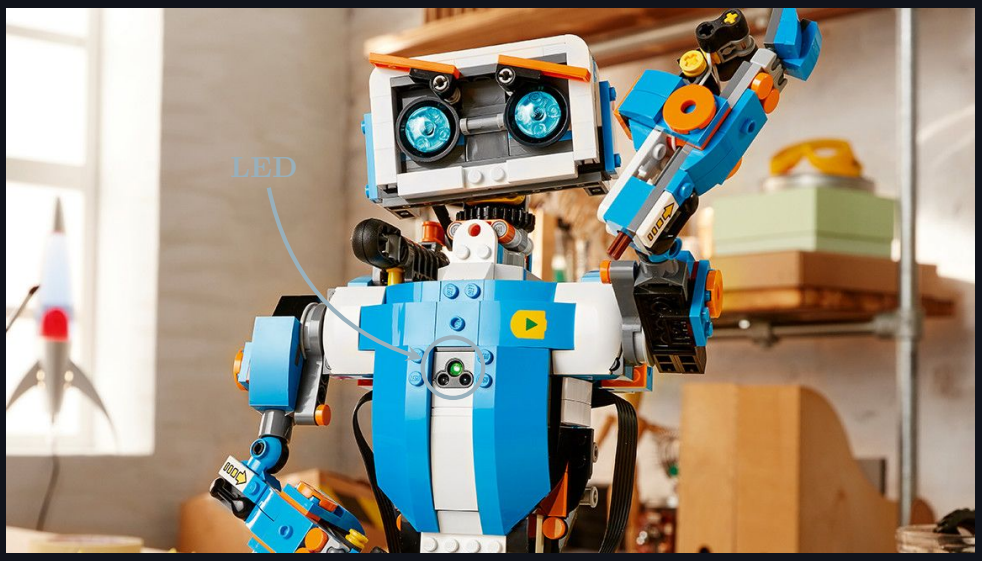


Figure 7.9. Position of Light Emitting Diode (LED) on the Vernie robot.

The lego-boost-browser library can send commands to Vernie to change the LED color (Figure 7.9) or activate the two motors for locomotion. A publisher is created which simply publishes every second a different LED color as a string from the acceptable list of colors by Vernie. When these messages reach the message stacks of the respective topic (e.g. `rainbow_topic`), they are forwarded to the robot through lego-boost-browser. Figure 7.10 displays what a web page running an LED color publisher appears like. The user would have to first connect to Vernie via bluetooth before being able to send the commands to change LED color to the one that matches the last message published.

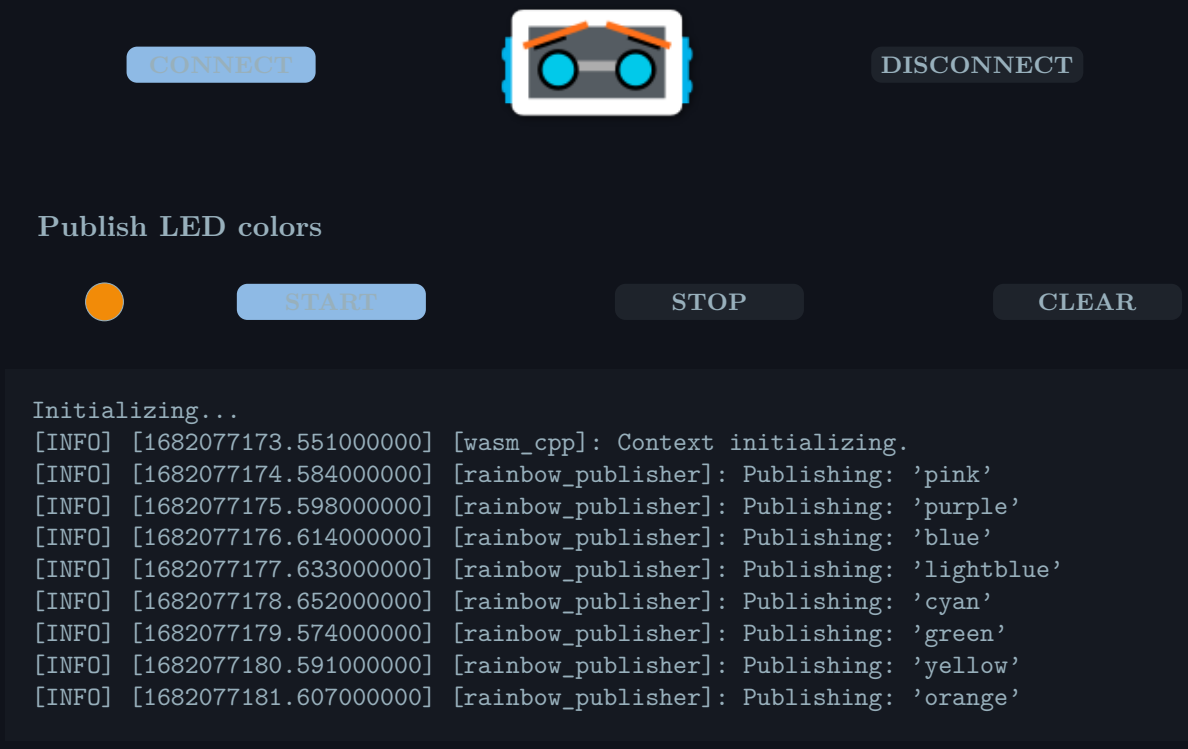


Figure 7.10. Publisher sending LED colors to the Vernie robot.

To control the position of the Vernie robot, a publisher which sends `geometry_msgs/Twist` messages was created. These messages contain a linear and an angular vector to express the velocity in free space. To ensure compatibility with lego-boost-browser, a few simplifications were made. First, the robot can only move forward or backwards along the x-axis and rotate on the same plane, z-axis; thus, the only relevant values from the `Twist` message are the linear velocity $\dot{x}_{lin}$ and the angular velocity $\dot{z}_{rot}$. Since the commands sent to Vernie with lego-boost-browser can only be expressed in terms of motor power (P), from 0 to 100%, the linear and angular command velocities are converted with the following formula:

$$\begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_{lin} \\ z_{rot} \end{bmatrix} = \begin{bmatrix} P_A \\ P_B \end{bmatrix} \tag{7.1}$$

Afterwards, the motor power is normalized to ensure that it does not exceed the 100% limit.

With this implementation, a ROS publisher can send command velocities by publishing a `Twist` message, then these messages are converted and forwarded to the Vernie robot to activate the motors with the desired velocities.

Example 7

A demonstration of $\mathcal{C}4$ with Vernie can be found at:
https://ros2wasm.dev/pages/demo04/index.html
NOTE: The Web Bluetooth API is experimental and must be enabled manually in some browsers. Chrome is recommended.

7.1.9. User Interaction Level 4: Intermediate

services and info about environment

A depiction of Level 4 is shown in Figure 7.11

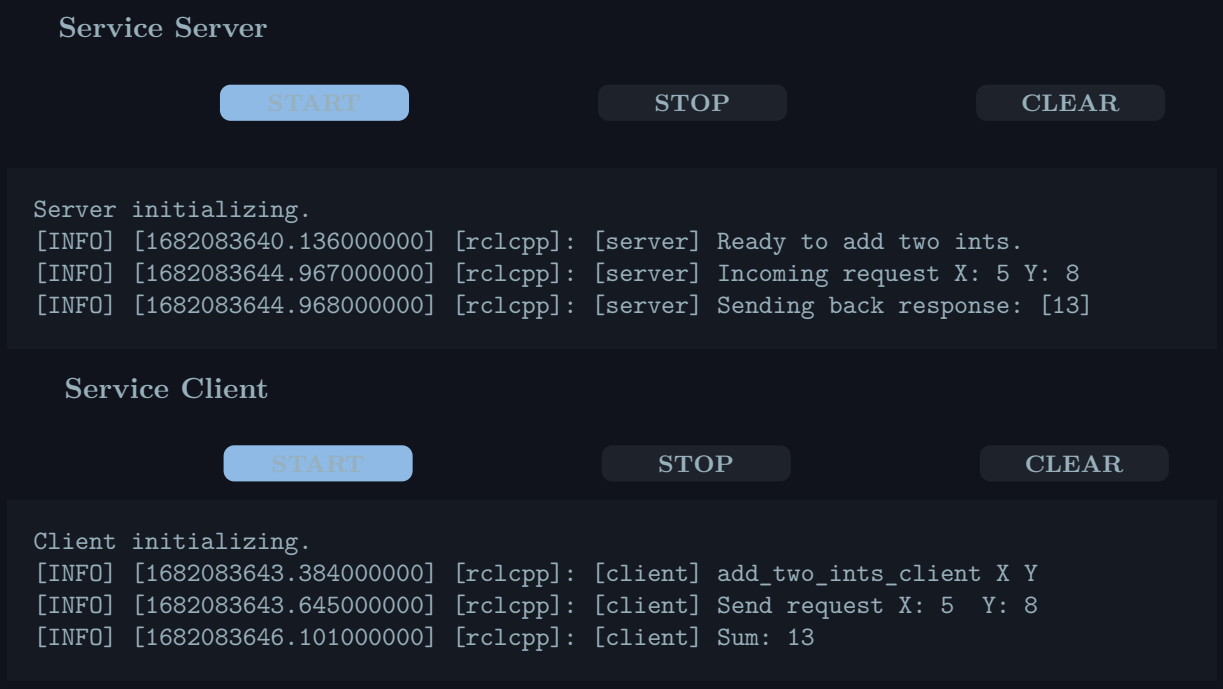


Figure 7.11. Service server and client interacting on the browser.

Example 8

A demonstration of an *intermediate* user interface ($\mathcal{I}4$) can be found at
https://ros2wasm.dev/pages/demo05/index.html

8. Conclusion

8.1. Outlook

- JupyterLite - client library - Zethus and visualizations - Message conversion - Compiling on the browser - Packaging Gazebo - Optimization

Bibliography

[16] *Web-Enabled DDS*, formal/2016-03-01, Version 1.0, Object Management Group, June 2016.

[23] *Using Web Workers*, MDN Web Docs, 2023, https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers (visited on 2023).

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

[Arr22] Arruda, M. A.
ROSWeb
https://eesc-labrom.github.io/roswebpage/index.html (visited on 2023).

[Ban] Bandes-Storch, J. (ed.)
Robot Web Tools
http://robotwebtools.org/ (visited on 2023).

[Béd21] Bédard, C.
ROS 2 Over Email: rmw_email, an Actual Working RMW Implementation
https://christophebedard.com/ros-2-over-email/ (visited on 2023).

[Bei23] Beier, T. (ed.)
Lego Boost in Browser
https://github.com/DerThorsten/lego-boost-browser (visited on 2023).

[Ben20] Bendejacq, D.
Pong! by Jackbenfu
https://jackbenfu.itch.io/pong (visited on 2023).

[Ber18] Berscheid
ROS Control Center
https://github.com/pantor/ros-control-center (visited on 2023).

[Big22] Biggs, G.
Improving the Communications Layer of Robot Applications with ROS 2 and Zenoh
https://www.youtube.com/watch?v=1NE8cU72frk (visited on 2023).

[Bor] Bore, N.
roswasm_suite: Libraries for compiling C++ ROS nodes to Webassembly using Emscripten
https://github.com/nilsbore/roswasm_suite/tree/master (visited on 2023).

[Bro23] Brown, C. (ed.)
Get ROS 2 Code
https://docs.ros.org/en/humble/Installation/Alternatives/Ubuntu-Development-Setup.html (visited on 2023).

[Cor23] Corsaro, A. (ed.)
What is Zenoh?
https://zenoh.io/docs/overview/what-is-zenoh/ (visited on 2023).

[Cuv22] Cuvillier, G.
D3WASM: A Port of ID Tech 4/DOOM 3 Engine to WebAssembly
http://www.continuation-labs.com/projects/d3wasm/ (visited on 2023).

[Ech16] Echterhoff, J.
Unity WebGL Player - AngryBots WebGL Demo
https://files.unity3d.com/jonas/AngryBots/ (visited on 2023).

[Fou23] Foundation, E. (ed.)
Eclipse Cyclone DDS
https://cyclonedds.io/docs/cyclonedds/latest/about_dds/eclipse_cyclone_dds.html (visited on 2023).

[Kri17] Krill, P.
WebAssembly is now ready for browsers to use
In: InfoWorld (, Mar. 2017), https://www.infoworld.com/article/3176681/webassembly-is-now-ready-for-browsers-to-use.html.

[Lam23] Lamprianidis, N. (ed.)
Writing a Simple Publisher and Subscriber (C++)
https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Cpp-Publisher-And-Subscriber.html (visited on 2023).

[Lor22] Loretz, S. (ed.)
About Different ROS 2 DDS/RTPS Vendors
https://docs.ros.org/en/humble/Concepts/About-Different-Middleware-Vendors.html (visited on 2023).

[Mil23a] Milbert, R.
Atmos
https://github.com/Razakhel/Atmos (visited on 2023).

[Mil23b] Millán, J.
Publishing and Visualizing ROS 2 Transforms - Foxglove
https://foxglove.dev/blog/publishing-and-visualizing-ros2-transforms (visited on 2023).

[Pon20a] Ponz Segrelles, E. (ed.)
Fast DDS - The DCPS Conceptial Model
https://fast-dds.docs.eprosima.com/en/latest/fastdds/getting_started/definitions.html (visited on 2023).

[Pon20b] Ponz Segrelles, E. (ed.)
Introduction to DDS
https://www.eprosima.com/index.php/resources-all/whitepapers/dds (visited on 2023).

[Put18] Puthuff, N.
ROS2 + DDS Integration: When Ecosystems Merge | RTI
https://www.rti.com/blog/ros2-dds-when-ecosystems-merge (visited on 2023).

[Rau17] Rauschmayer, A.
Communicating Between Web Workers va MessageChannel
https://2ality.com/2017/01/messagechannel.html (visited on 2023).

[Ros23] Rossberg, A. (ed.)
WebAssembly Specification
https://webassembly.github.io/spec/core/ (visited on 2023).

[Sch22] Schultz, C. (ed.)
ROS Wiki - rosbridge_suite
https://wiki.ros.org/rosbridge_suite (visited on 2023).

[Sco22] Scott, K. (ed.)
Working with Multiple ROS 2 Middleware Implementations
https://docs.ros.org/en/humble/How-To-Guides/Working-with-multiple-RMW-implementations.html (visited on 2023).

[Sht] Shtylman, R. (ed.)
Foxglove Studio Docs
https://foxglove.dev/docs/studio (visited on 2023).

[Ste23] Steiner, T. (ed.)
WebAssembly Concepts
https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts (visited on 2023).

[Tho14] Thomas, D.
ROS 2 Middleware Interface
https://design.ros2.org/articles/ros_middleware_interface.html (visited on 2023).

[Tho23] Thomson, S. (ed.)
WebAssembly website
https://webassembly.org (visited on 2023).

[Tuh20] Tuhkanen, T.
Lego Boost in Browser
https://github.com/ttu/lego-boost-browser (visited on 2023).

[Ven21] Venkatraman, D.
ROSboard
https://github.com/dheera/rosboard (visited on 2023).

[Weo23] Weon, E. (ed.)
RTI Connext DDS
https://docs.ros.org/en/humble/Installation/DDS-Implementations/Install-Connext-University-Eval.html (visited on 2023).

[Woo19] Woodall, W.
ROS on DDS
https://design.ros2.org/articles/ros_on_dds.html (visited on 2023).

[Woo23] Woodall, W. (ed.)
rmw: ROS Middleware Abstraction Interface
https://docs.ros2.org/latest/api/rmw/rmw_8h.html (visited on 2023).

[Yun23] Yun, Y. (ed.)
GurumNetworks GurumDDS
https://docs.ros.org/en/humble/Installation/DDS-Implementations/Working-with-GurumNetworks-GurumDDS.html (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

export RMW_IMPLEMENTATION="rmw_wasm_cpp"

#-----------------------------------------------------------------
# OPTIONS
#-----------------------------------------------------------------

while getopts "hcvpu:s:i:" option; do
    case $option in
        h) # Display help
            Help
            exit;;

        c) # Clean workspace
            [[ -d "${PWD}/install" ]] && rm -rf "${PWD}/install"
            [[ -d "${PWD}/build" ]]   && rm -rf "${PWD}/build"
            [[ -d "${PWD}/log" ]]     && rm -rf "${PWD}/log"
            exit;;

        d) # Activate cmake debug
            debug_mode=ON
            echo "[BLASM]: CMake debug mode activated.";;

        v) # Make verbose
            VERBOSE=1
            EMSDK_VERBOSE=1
            verbose_args="--event-handlers console_direct+"
            echo "[BLASM]: Verbose activated.";;

        p) # Install emscripten python
            echo "[BLASM]: Installing python."
            package_ignore=""
            InstallPython;;

        u) # Build up to package
            package_args="--packages-up-to ${OPTARG}"
            echo "[BLASM]: Build up to ${OPTARG}.";;

        s) # Build selected package
            [[ -d "${PWD}/build/${OPTARG}" ]] && rm -rf "${PWD}/build/${OPTARG}"
            package_args="--packages-select ${OPTARG}"
            echo "[BLASM]: Build only ${OPTARG}.";;

        i) # Ignore "given packages"
            package_ignore="--packages-ignore ${OPTARG}"
            echo "[BLASM]: Ignore packages ${OPTARG}.";;
```

```
        \?) # Invalid option
            echo "Error: Invalid option."
            Help
            exit;;
    esac
done

#------------------------------------------------------------
# MAIN
#------------------------------------------------------------
[[ -z "${package_args}" ]] && { echo "No args given."; exit 1; }
[[ -d "${PWD}/src" ]] || { echo "Not a workspace directory"; exit 1; }

echo "[BLASM]: Commencing build."

colcon build \
    ${package_args} ${package_ignore} ${verbose_args} \
    --packages-skip-build-finished \
    --merge-install \
    --cmake-args \
        -DCMAKE_TOOLCHAIN_FILE="${EMSDK_DIR}/upstream/emscripten/cmake/Modules/
    Platform/Emscripten.cmake" \
        -DBUILD_TESTING=OFF \
        -DBUILD_SHARED_LIBS=ON \
        -DCMAKE_VERBOSE_MAKEFILE=${debug_mode} \
        -DCMAKE_FIND_ROOT_PATH_MODE_PACKAGE=ON \
        -DCMAKE_CROSSCOMPILING=TRUE \
        -DCMAKE_FIND_DEBUG_MODE=${debug_mode} \
        -DFORCE_BUILD_VENDOR_PKG=ON \
        -DPYBIND11_PYTHONLIBS_OVERWRITE=OFF
```

C.2. Build Workflow

```
name: build-package
on:
  workflow_dispatch:
    inputs:
      package:
        description: 'Target package'
        type: string
        default: 'test_pubsub'
        required: true
      debug-mode:
        description: 'Debug mode'
        type: boolean
        default: false
run-name: Build up to package ${{ inputs.package }}


jobs:
  setup:
    runs-on: ubuntu-latest
    env:
      TARGET_PLATFORM: emscripten-32
      RMW_IMPLEMENTATION: rmw_wasm
      ROS_VERSION: 2
      ROS_DISTRO: humble

    steps:

    - name: Setup emsdk
      uses: mymindstorm/setup-emsdk@v11
      with:
        version: 3.1.27

    - name: Verify emsdk
      run: |
        emcc -v
        find $EMSDK -iname "upstream"
        echo "Find upstream"

    - name: Create ROS2 workspace
      run: mkdir -p ros_workspace/src

    - name: Install rmw_wasm
      uses: actions/checkout@v3
      with:
        path: ros_workspace/src/myrepo
```

```
- name: Copy ROS2 packages
  run:  |
    curl -s https://packagecloud.io/install/repositories/dirk-thomas/vcstool/
script.deb.sh | sudo bash
    sudo apt-get update
    sudo apt-get install python3-vcstool
    vcs import --input https://raw.githubusercontent.com/ros2/ros2/humble/ros2
.repos ros_workspace/src
    vcs import --input ros_workspace/src/myrepo/src/repos.yaml ros_workspace/
src

- name: Remove unsupported packages
  run: |
    cd ros_workspace/src
    while read F; do rm -rf $F; done < myrepo/src/unsupported.txt

- name: Apply patches
  run: |
    cd ros_workspace/src/ros2/rcutils
    git apply ${GITHUB_WORKSPACE}/ros_workspace/src/myrepo/src/rcutils.patch

- name: Create environment
  uses: mamba-org/provision-with-micromamba@main
  with:
    environment-file: ros_workspace/src/myrepo/src/env.yaml
    environment-name: ros-env
    micromamba-version: '1.4.1'

- name: Activate environment and build package
  run: |
    eval "$(micromamba shell hook --shell=bash)"
    micromamba activate ros-env
    cd ros_workspace
    if [${{ inputs.debug }}]; then ./src/myrepo/src/blasm.sh -d -v -u ${{
inputs.package }}; else ./src/myrepo/src/blasm.sh -u ${{ inputs.package }}; fi

- name: Upload artifacts
  uses: actions/upload-artifact@v3
  with:
    name: ${{ inputs.package }}-artifacts
    path: |
      ros_workspace/build/${{ inputs.package }}/*.js
      ros_workspace/build/${{ inputs.package }}/*.wasm
      ros_workspace/build/${{ inputs.package }}/*.html
```

C.3. JavaScript Functions

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

C.4. RMW Adapter Function Headers

C.4.1. Events

```
// Initialize an rmw publisher event
rmw_ret_t rmw_publisher_event_init(
    rmw_event_t * rmw_event,
    const rmw_publisher_t * publisher,
    rmw_event_type_t event_type) {}

// Initialize an rmw subscription event
rmw_ret_t rmw_subscription_event_init(
    rmw_event_t * rmw_event,
    const rmw_subscription_t * subscription,
    rmw_event_type_t event_type) {}

//
rmw_ret_t rmw_event_set_callback(
    rmw_event_t * rmw_event,
        rmw_event_callback_t callback,
        const void * user_data) {}

// Take an event from the event handle
rmw_ret_t rmw_take_event(
    const rmw_event_t * event_handle,
    void * event_info,
    bool * taken) {}
```

C.4.2. Implementation Information

```
// Get the name of the rmw implementation being used
const char * rmw_get_implementation_identifier() {}
```

```
// Retrieve the information for all publishers to a given topic.
rmw_ret_t rmw_get_publishers_info_by_topic(
    const rmw_node_t * node,
    rcutils_allocator_t * allocator,
    const char * topic_name,
    bool no_mangle,
    rmw_topic_endpoint_info_array_t * subscriptions_info) {}

// Retrieve the information for all subscriptions to a given topic
rmw_ret_t rmw_get_subscriptions_info_by_topic(
```

```
    const rmw_node_t * node,
    rcutils_allocator_t * allocator,
    const char * topic_name,
    bool no_mangle,
    rmw_topic_endpoint_info_array_t * subscriptions_info) {}
```

```
// Return a list of published topic names and their types.
rmw_ret_t rmw_get_publisher_names_and_types_by_node(
    const rmw_node_t * node,
    rcutils_allocator_t * allocator,
    const char * node_name,
    const char * node_namespace,
    bool no_demangle,
    rmw_names_and_types_t * names_and_types) {}

// Return a list of subscribed topic names and their types.
rmw_ret_t rmw_get_subscriber_names_and_types_by_node(
    const rmw_node_t * node,
    rcutils_allocator_t * allocator,
    const char * node_name,
    const char * node_namespace,
    bool no_demangle,
    rmw_names_and_types_t * names_and_types) {}

// Return a list of service topic names and their types.
rmw_ret_t rmw_get_service_names_and_types_by_node(
    const rmw_node_t * node,
    rcutils_allocator_t * allocator,
    const char * node_name,
    const char * node_namespace,
    rmw_names_and_types_t * names_and_types) {}

// Return a list of service client topic names and their types.
rmw_ret_t rmw_get_client_names_and_types_by_node(
    const rmw_node_t * node,
    rcutils_allocator_t * allocator,
    const char * node_name,
    const char * node_namespace,
    rmw_names_and_types_t * names_and_types) {}

// Return a list of topic names and their types.
rmw_ret_t rmw_get_topic_names_and_types(
    const rmw_node_t * node,
    rcutils_allocator_t * allocator,
    bool no_demangle,
    rmw_names_and_types_t * names_and_types) {}
```

```
// Return a list of service names and their types.
// This function returns a list of service names in the ROS graph and
// their types.
rmw_ret_t rmw_get_service_names_and_types(
    const rmw_node_t * node,
    rcutils_allocator_t * allocator,
    rmw_names_and_types_t * service_names_and_types) {}

// Return a list of node name and namespaces discovered via a node.
// This function will return a list of node names and a list of node
// namespaces that are discovered via the middleware. The two lists represent
// pairs of namespace and name for each discovered node. The lists will be
// the same length and the same position will refer to the same node across
// lists.
// The node parameter must not be NULL, and must point to a valid node.
// The node_names parameter must not be NULL, and must point to a valid
// string array.
// The node_namespaces parameter must not be NULL, and must point to a valid
// string array.
// This function does manipulate heap memory. This function is not
// thread-safe. This function is lock-free.
rmw_ret_t rmw_get_node_names(
    const rmw_node_t * node,
    rcutils_string_array_t * node_names,
    rcutils_string_array_t * node_namespaces) {}

// Return a list of node name and namespaces discovered via a node with
// its enclave.
// Similar to rmw_get_node_names, but it also provides the enclave name.
rmw_ret_t rmw_get_node_names_with_enclaves(
    const rmw_node_t * node,
    rcutils_string_array_t * node_names,
    rcutils_string_array_t * node_namespaces,
    rcutils_string_array_t * enclaves) {}
```

```
// Get the unique identifier of the publisher.
rmw_ret_t rmw_get_gid_for_publisher(
    const rmw_publisher_t * publisher,
    rmw_gid_t * gid) {}

// Check if two gid objects are the same.
rmw_ret_t rmw_compare_gids_equal(
    const rmw_gid_t * gid1,
    const rmw_gid_t * gid2,
    bool * result) {}
```

```
// Set the current log severity
rmw_ret_t rmw_set_log_severity(
        rmw_log_severity_t severity) {}
```

C.4.3. Guard Conditions

```
// Create a guard condition and return a handle to that guard condition.
// This function can fail, and therefore return NULL, if:
//     context is NULL
//     context is invalid
//     memory allocation fails during guard condition creation
//     an unspecified error occurs
// The context must be non-null and valid, i.e. it has been initialized
// by rmw_init() and has not been finalized by rmw_shutdown().
rmw_guard_condition_t * rmw_create_guard_condition(rmw_context_t * context) {}

// Finalize a given guard condition handle, reclaim the resources,
// and deallocate the handle.
rmw_ret_t rmw_destroy_guard_condition(
    rmw_guard_condition_t * rmw_guard_condition) {}

// Trigger guard condition
rmw_ret_t rmw_trigger_guard_condition(
    const rmw_guard_condition_t * rmw_guard_condition) {}
```

C.4.4. Initialization and Shutdown

```
// Initialize given init_options with the default values and implementation
// specific values.
// The given allocator is used, if required, during setup of the init
// options, but is also used during initialization.
// In either case the given allocator is stored in the returned init options.
// The impl pointer should not be changed manually.
rmw_ret_t rmw_init_options_init(
    rmw_init_options_t * init_options,
    rcutils_allocator_t allocator) {}

// Copy the given source init options to the destination init options.
// The allocator from the source is used for any allocations and stored
// in the destination.
// The destination should either be zero initialized with
// rmw_get_zero_initialized_init_options() or should have had
// rmw_init_options_fini() called on it. Giving an already initialized init
```

```
// options for the destination will result in a failure with return code
// RMW_RET_INVALID_ARGUMENT.
rmw_ret_t rmw_init_options_copy(
    const rmw_init_options_t * src,
    rmw_init_options_t * dst) {}

// Finalize the given init_options.
// The given init_options must be non-NULL and valid, i.e. had
// rmw_init_options_init() called on it but not this function yet.
rmw_ret_t rmw_init_options_fini(rmw_init_options_t * init_options) {}

// Initialize the middleware with the given options, and yielding an context.
// The given context must be zero initialized, and is filled with middleware
// specific data upon success of this function. The context is used when
// initializing some entities like nodes and guard conditions, and is also
// required to properly call rmw_shutdown().
rmw_ret_t rmw_init(
    const rmw_init_options_t * options,
    rmw_context_t * context) {}

// Shutdown the middleware for a given context.
// The given context must be a valid context which has been initialized
// with rmw_init().
rmw_ret_t rmw_shutdown(rmw_context_t * context) {}

// Finalize a context.
// The context to be finalized must have been previously initialized with
// rmw_init(), and then later invalidated with rmw_shutdown(). If context is
// NULL, then RMW_RET_INVALID_ARGUMENT is returned. If context is
// zero-initialized, then RMW_RET_INVALID_ARGUMENT is returned.
// If context is initialized and valid (rmw_shutdown() was not called on it),
// then RMW_RET_INVALID_ARGUMENT is returned.
rmw_ret_t rmw_context_fini(rmw_context_t * context) {}
```

C.4.5. Nodes

```
// Create a node and return a handle to that node.
// This function can fail, and therefore return NULL, if:
//     context, name, namespace_, or security_options is NULL
//     context, security_options is invalid
//     memory allocation fails during node creation
//     an unspecified error occurs
// The context must be non-null and valid, i.e. it has been initialized by
// rmw_init() and has not been finalized by rmw_shutdown().
// The name and namespace_ should be valid node name and namespace, and this
```

```
// should be asserted by the caller (e.g. rcl).
// The domain ID should be used to physically separate nodes at the
// communication graph level by the middleware. For RTPS/DDS this maps
// naturally to their concept of domain id.
// The security options should always be non-null and encapsulate the
// essential security configurations for the node and its entities.
rmw_node_t * rmw_create_node(
    rmw_context_t * context,
    const char * name,
    const char * namespace_) {}

// Finalize a given node handle, reclaim the resources, and deallocate the
// node handle.
// The method may assume - but should verify - that all publishers,
// subscribers, services, and clients created from this node have already
// been destroyed. If the rmw implementation chooses to verify instead of
// assume, it should return RMW_RET_ERROR and set a human readable error
// message if any entity created from this node has not yet been destroyed.
rmw_ret_t rmw_destroy_node(
    rmw_node_t * rmw_node) {}

// Return a guard condition which is triggered when the ROS graph changes.
// The handle returned is a pointer to an internally held rmw guard condition.
// This function can fail, and therefore return NULL, if:
//     node is NULL
//     node is invalid
// The returned handle is made invalid if the node is destroyed or if
// rmw_shutdown() is called.
// The guard condition will be triggered anytime a change to the ROS graph
// occurs. A ROS graph change includes things like (but not limited to) a
// new publisher advertises, a new subscription is created, a new service
// becomes available, a subscription is canceled, etc.
const rmw_guard_condition_t * rmw_node_get_graph_guard_condition(
    const rmw_node_t * rmw_node) {}
```

C.4.6. Quality of Service

```
// Retrieve the actual qos settings of the publisher.
rmw_ret_t rmw_publisher_get_actual_qos(
    const rmw_publisher_t * publisher,
    rmw_qos_profile_t * qos) {}

// Retrieve the actual qos settings of the subscriber.
rmw_ret_t rmw_subscription_get_actual_qos(
    const rmw_subscription_t * subscription,
```

```
    rmw_qos_profile_t * qos) {}

// Retrieve qos settings for service response
rmw_ret_t rmw_service_response_publisher_get_actual_qos(
    const rmw_service_t * service,
    rmw_qos_profile_t * qos) {}

// Retrieve qos settings for client response
rmw_ret_t rmw_client_response_subscription_get_actual_qos(
    const rmw_client_t * client,
    rmw_qos_profile_t * qos) {}

// Retrieve qos settings for client request
rmw_ret_t rmw_client_request_publisher_get_actual_qos(
    const rmw_client_t * client,
    rmw_qos_profile_t * qos) {}

// Retrieve qos settings for service request
rmw_ret_t rmw_service_request_subscription_get_actual_qos(
    const rmw_service_t * service,
    rmw_qos_profile_t * qos) {}

// Check if two qos profiles are compatible
rmw_ret_t rmw_qos_profile_check_compatible(
    const rmw_qos_profile_t publisher_profile,
    const rmw_qos_profile_t subscription_profile,
    rmw_qos_compatibility_type_t * compatibility,
    char * reason,
    size_t reason_size) {}
```

C.4.7. Serialization

```
// Get the unique serialization format for this middleware.
// Return the format in which binary data is serialized. One middleware can
// only have one encoding. In contrast to the implementation identifier, the
// serialization format can be equal between multiple RMW implementations.
// This means, that the same binary messages can be deserialized by RMW
// implementations with the same format.
const char * rmw_get_serialization_format() {}

// Compute the size of a serialized message.
// Given a message definition and bounds, compute the serialized size.
rmw_ret_t rmw_get_serialized_message_size(
        const rosidl_message_type_support_t * type_support,
        const rosidl_runtime_c__Sequence__bound * message_bounds,
```

```
        size_t * size) {}

// Serialize a ROS message into a rmw_serialized_message_t.
// The ROS message is serialized into a byte stream contained within the
// rmw_serialized_message_t structure. The serialization format depends on
// the underlying middleware.
rmw_ret_t rmw_serialize(
        const void * ros_message,
        const rosidl_message_type_support_t * type_support,
        rmw_serialized_message_t * serialized_message) {}

// Deserialize a ROS message.
// The given rmw_serialized_message_t's internal byte stream buffer is
// deserialized into the given ROS message. The ROS message must already be
// allocated and initialized, and must match the given typesupport structure.
// The serialization format expected in the rmw_serialized_message_t depends
// on the underlying middleware.
rmw_ret_t rmw_deserialize(
        const rmw_serialized_message_t * serialized_message,
        const rosidl_message_type_support_t * type_support,
        void * ros_message) {}
```

C.4.8. Service Clients

```
// Create an rmw client to communicate with the specified service.
rmw_client_t * rmw_create_client(
    const rmw_node_t * node,
    const rosidl_service_type_support_t * type_support,
    const char * service_name,
    const rmw_qos_profile_t * qos_profile) {}

// Destroy and unregister a service client.
rmw_ret_t rmw_destroy_client(
    rmw_node_t * node,
    rmw_client_t * client) {}

//
rmw_ret_t rmw_client_set_on_new_response_callback(
    rmw_client_t * client,
        rmw_event_callback_t callback,
        const void * user_data) {}

// Check if a service server is available for the given service client.
// This function will return true for is_available if there is a service
// server available for the given client.
```

```
// The node parameter must not be NULL, and must point to a valid node.
// The client parameter must not be NULL, and must point to a valid client.
// The given client and node must match, i.e. the client must have been
// created using the given node.
// The is_available parameter must not be NULL, and must point to a bool
// variable. The result of the check will be stored in the is_available
// parameter.
// This function does manipulate heap memory.
// This function is not thread-safe. This function is lock-free.
rmw_ret_t rmw_service_server_is_available(
    const rmw_node_t * node,
    const rmw_client_t * client,
    bool * is_available) {}

// Send a service request to the rmw server.
rmw_ret_t rmw_send_request(
    const rmw_client_t * client,
    const void * ros_request,
    int64_t * sequence_id) {}

// Attempt to get the response from a service request.
rmw_ret_t rmw_take_response(
    const rmw_client_t * client,
    rmw_service_info_t * request_header,
    void * ros_response,
    bool * taken) {}
```

C.4.9. Service Servers

```
// Create an rmw service server that responds to requests.
rmw_service_t * rmw_create_service(
    const rmw_node_t * node,
    const rosidl_service_type_support_t * type_support,
    const char * service_name,
    const rmw_qos_profile_t * qos_profile) {}

// Destroy and unregister the service from this node.
rmw_ret_t rmw_destroy_service(
    rmw_node_t * node,
    rmw_service_t * service) {}

rmw_ret_t rmw_service_set_on_new_request_callback(
    rmw_service_t * rmw_service,
        rmw_event_callback_t callback,
        const void * user_data) {}
```

```
// Attempt to take a request from this service's request buffer.
rmw_ret_t rmw_take_request(
    const rmw_service_t * service,
    rmw_service_info_t * service_info,
    void * ros_request,
    bool * taken) {}

// Send response to a client's request.
rmw_ret_t rmw_send_response(
    const rmw_service_t * service,
    rmw_request_id_t * request_id,
    void * ros_response) {}
```

C.4.10. Publishers

```
// Publish a given ROS message via a publisher.
rmw_ret_t rmw_publish(
    const rmw_publisher_t * publisher,
    const void * ros_message,
        rmw_publisher_allocation_t * allocation) {}

// Publish an already serialized message.
// The publisher must already be registered with the correct message type
// support so that it can send serialized data corresponding to that type.
// This function sends the serialized byte stream directly over the wire without
// having to serialize the message first. A ROS message can be serialized
// manually using the rmw_serialize() function.
rmw_ret_t rmw_publish_serialized_message(
    const rmw_publisher_t * publisher,
    const rmw_serialized_message_t * serialized_message,
        rmw_publisher_allocation_t * allocation) {}

// Publish a loaned ROS message via a publisher and return ownership of the
// loaned message back to the middleware.
rmw_ret_t rmw_publish_loaned_message(
        const rmw_publisher_t * publisher,
        void * ros_message,
        rmw_publisher_allocation_t * allocation) {}

// Initialize a publisher allocation to be used with later publications.
// This creates an allocation object that can be used in conjunction with
// the rmw_publish method to perform more carefully control memory allocations.
// This will allow the middleware to preallocate the correct amount of
// memory for a given message type and message bounds. As allocation is
```

```
// performed in this method, it will not be necessary to allocate in the
// rmw_publish method.
rmw_ret_t rmw_init_publisher_allocation(
        const rosidl_message_type_support_t * type_support,
        const rosidl_runtime_c__Sequence__bound * message_bounds,
        rmw_publisher_allocation_t * allocation) {}

// Destroy a publisher allocation object.
// This deallocates any memory allocated by rmw_init_publisher_allocation.
rmw_ret_t rmw_fini_publisher_allocation(
        rmw_publisher_allocation_t * allocation) {}

// Create and return an rmw publisher.
// The argument publisher_options must not be nullptr.
rmw_publisher_t * rmw_create_publisher(
    const rmw_node_t * node,
    const rosidl_message_type_support_t * type_support,
    const char * topic_name,
    const rmw_qos_profile_t * qos_profile,
    const rmw_publisher_options_t * publisher_options) {}

// Destroy publisher.
rmw_ret_t rmw_destroy_publisher(
    rmw_node_t * node,
    rmw_publisher_t * publisher) {}

// Retrieve the number of matched subscriptions to a publisher.
// Query the underlying middleware to determine how many subscriptions are matched
     to a given publisher.
rmw_ret_t rmw_publisher_count_matched_subscriptions(
    const rmw_publisher_t * publisher,
    size_t * subscription_count) {}

// Manually assert that this Publisher is alive
// (for RMW_QOS_POLICY_LIVELINESS_MANUAL_BY_TOPIC)
// If the rmw Liveliness policy is set to
    RMW_QOS_POLICY_LIVELINESS_MANUAL_BY_TOPIC,
// the creator of this publisher may manually call assert_liveliness at some
// point in time to signal to the rest of the system that this Node is still alive
    .
rmw_ret_t rmw_publisher_assert_liveliness(const rmw_publisher_t * publisher) {}

// Borrow a loaned message.
// The memory allocated for the ros message belongs to the middleware
// and must not be deallocated.
rmw_ret_t rmw_borrow_loaned_message(
        const rmw_publisher_t * publisher,
```

```
        const rosidl_message_type_support_t * type_support,
        void ** ros_message) {}

// Return a loaned message previously borrowed from a publisher.
// The ownership of the passed in ros message will be transferred back to the
// middleware. The middleware might deallocate and destroy the message so that
// the pointer is no longer guaranteed to be valid after this call.
rmw_ret_t rmw_return_loaned_message_from_publisher(
        const rmw_publisher_t * publisher,
        void * loaned_message) {}

// Count the number of publishers matching a topic name
rmw_ret_t rmw_count_publishers(
    const rmw_node_t * node,
    const char * topic_name,
    size_t * count) {}
```

C.4.11. Subscribers

```
// Initialize a subscription allocation to be used with later takes.
// This creates an allocation object that can be used in conjunction with
// the rmw_take method to perform more carefully control memory allocations.
// This will allow the middleware to preallocate the correct amount of
// memory for a given message type and message bounds. As allocation is
// performed in this method, it will not be necessary to allocate in the
// rmw_take method.
rmw_ret_t rmw_init_subscription_allocation(
        const rosidl_message_type_support_t * type_support,
        const rosidl_runtime_c__Sequence__bound * message_bounds,
        rmw_subscription_allocation_t * allocation) {}

// Destroy a publisher allocation object.
// This deallocates memory allocated by rmw_init_subscription_allocation.
rmw_ret_t rmw_fini_subscription_allocation(
        rmw_subscription_allocation_t * allocation) {}

// Create and return an rmw subscription.
// The argument subscription_options must not be nullptr.
rmw_subscription_t * rmw_create_subscription(
    const rmw_node_t * node,
    const rosidl_message_type_support_t * type_support,
    const char * topic_name,
    const rmw_qos_profile_t * qos_policies,
    const rmw_subscription_options_t * subscription_options) {}
```

```
// Destroy subscription.
rmw_ret_t rmw_destroy_subscription(
    rmw_node_t * node,
    rmw_subscription_t * subscription) {}

// Retrieve the number of matched publishers to a subscription.
// Query the underlying middleware to determine how many publishers are matched to
     a given subscription.
rmw_ret_t rmw_subscription_count_matched_publishers(
    const rmw_subscription_t * subscription,
    size_t * publisher_count) {}

// Count the number of subscribers matching a topic name
rmw_ret_t rmw_count_subscribers(
    const rmw_node_t * node,
    const char * topic_name,
    size_t * count) {}

rmw_ret_t rmw_subscription_set_on_new_message_callback(
    rmw_subscription_t * subscription,
        rmw_event_callback_t callback,
        const void * user_data) {}
```

```
// Take an incoming ROS message from a given subscription.
rmw_ret_t rmw_take(
    const rmw_subscription_t * subscription,
    void * ros_message,
    bool * taken,
    rmw_subscription_allocation_t * allocation) {}

// Take an incoming ROS message from a given subscription with
// additional metadata.
rmw_ret_t rmw_take_with_info(
        const rmw_subscription_t * subscription,
        void * ros_message,
        bool * taken,
        rmw_message_info_t * message_info,
        rmw_subscription_allocation_t * allocation) {}

// Take a message without deserializing it.
// The message is taken in its serialized form. In contrast to rmw_take,
// the message is not deserialized in its ROS type but rather returned as a
// byte stream. The subscriber has to be registered for a specific type.
// But instead of receiving the message as its corresponding message type,
// it is taken as a byte stream. If needed, this byte stream can then be
// deserialized in a ROS message with a call to rmw_deserialize.
```

```
rmw_ret_t rmw_take_serialized_message(
        const rmw_subscription_t * subscription,
        rmw_serialized_message_t * serialized_message,
        bool * taken,
        rmw_subscription_allocation_t * allocation) {}

// Take a message without deserializing it and with its additional
// message information.
// The same as rmw_take_serialized_message(), except it also includes the
// rmw_message_info_t.
rmw_ret_t rmw_take_serialized_message_with_info(
        const rmw_subscription_t * subscription,
        rmw_serialized_message_t * serialized_message,
        bool * taken,
        rmw_message_info_t * message_info,
        rmw_subscription_allocation_t * allocation) {}

// Take a loaned message.
// If capable, the middleware can loan messages containing incoming messages.
// The message is owned by the middleware and thus has to be returned with a
// call to rmw_return_loaned_message_from_subscription.
rmw_ret_t rmw_take_loaned_message(
        const rmw_subscription_t * subscription,
        void ** loaned_message,
        bool * taken,
        rmw_subscription_allocation_t * allocation) {}

// Take a loaned message with its additional message information.
rmw_ret_t rmw_take_loaned_message_with_info(
        const rmw_subscription_t * subscription,
        void ** loaned_message,
        bool * taken,
        rmw_message_info_t * message_info,
        rmw_subscription_allocation_t * allocation) {}

// Return a loaned message previously taken from a subscription.
// After the taking a loaned message from the middleware, the middleware has
// to keep the memory for the loaned message alive and valid as long as the
// user is working with that loan. In order to indicate that the loaned
// message is no longer needed, the call to
// rmw_return_loaned_message_from_subscription tells the middleware that
// memory can be deallocated/destroyed.
rmw_ret_t rmw_return_loaned_message_from_subscription(
        const rmw_subscription_t * subscription,
        void * loaned_message) {}

// Take multiple incoming messages from a subscription with additional metadata.
```

```
// Take a sequence of ROS messgages from a given subscription.
// While count messages may be requested, fewer messages may be available
// on the subscription. In this case, only the currently available messages
// will be returned. The taken flag indicates the number of messages actually
// taken. The method will return RMW_RET_OK even in the case that fewer
// (or zero) messages were retrieved. from the subscription, and will
// RMW_RET_ERROR in the case of unexpected errors. In the case that count is
// zero, the function will return RMW_RET_INVALID_ARGUMENT.
// message_sequence and message_info_sequence should be initialized and have
// sufficient capacity. It is not critical that the sequence sizes match,
// and they may be reused from previous calls. Both must be valid (not NULL)
// for the method to run successfully.
rmw_ret_t rmw_take_sequence(
    const rmw_subscription_t * subscription,
    size_t count,
    rmw_message_sequence_t * message_sequence,
    rmw_message_info_sequence_t * message_info_sequence,
    size_t * taken,
    rmw_subscription_allocation_t * allocation) {}
```

C.4.12. Wait Sets

```
// Create a wait set to store conditions that the middleware will block on.
// This function can fail, and therefore return NULL, if:
//     context is NULL
//     context is invalid
//     memory allocation fails during wait set creation
//     an unspecified error occurs
// If max_conditions is 0, the wait set can store an unbounded number of
// conditions to wait on. If max_conditions is greater than 0, the number
// of conditions that can be attached to the wait set is bounded at
// max_conditions.
rmw_wait_set_t * rmw_create_wait_set(
    rmw_context_t * context,
        size_t max_conditions) {}

// Destroy and free memory of this wait_set.
rmw_ret_t rmw_destroy_wait_set(rmw_wait_set_t * wait_set) {}

// Waits on sets of different waitable entities and returns when one is ready.
// Add conditions to the wait set and wait until a response comes in, or
// until the timeout is reached. The arrays contain type-erased
// representations of waitable entities. This function casts the pointers to
```

```
// middleware-specific conditions and adds them to the wait set.
// The count variables in the arrays represents the number of valid pointers
// in the array. NULL pointers are in the array considered invalid. If they
// are encountered, an error is returned.
// The array structs are allocated and deallocated outside of this function.
// They do not have any information about how much memory is allocated for
// the arrays.
// After the wait wakes up, the entries in each array that correspond to
// conditions that were not triggered are set to NULL.
rmw_ret_t rmw_wait(
    rmw_subscriptions_t * subscriptions,
    rmw_guard_conditions_t * guard_conditions,
    rmw_services_t * services,
    rmw_clients_t * clients,
    rmw_events_t * events,
    rmw_wait_set_t * wait_set,
    const rmw_time_t * wait_timeout) {}
```