

MAES 2.0: A ROS Compatible Simulation Tool for Multi Robot Exploration and Coverage

Malte Z. Andreassen¹^a, Philip I. Holler¹^b and Magnus K. Jensen¹^c

¹*Department of Computer Science, Aalborg University, Selma Lagerlöfs Vej 300, 9220 Aalborg, Denmark
malte@mza.dk, philipholler94@gmail.com, magnjensen@gmail.com*

Keywords: Distributed Systems, ROS, Robot Operating System, ROS 2, Distributed Exploration, Area Exploration, Continuous Space, Autonomous Robots, Online Terrain Coverage, Exploration, Coverage, Simulation, Swarm Robotics, Multi Agent, Frontier-Based Exploration, MAES, Nav2, SLAM

Abstract: Multi Agent Exploration Simulator (MAES) is an open-source physics based discrete step multi robot simulator created using Unity (Andreassen et al., 2022a). We present MAES 2.0 as an extension to MAES 1.0, which introduces improved features in the form of heatmap visualization, custom signal degradation, custom user-provided maps, and an extended GUI for debugging. Furthermore, MAES 2.0 allows for interfacing with the simulator through ROS (Robot Operating System), which results in easier portability of developed algorithms to real world robots. MAES 2.0 includes both unit tests and system tests to ensure intended behavior of the code. Furthermore, performance tests were conducted with modest hardware, which showed that MAES 2.0 is able to simulate up to 5 robots in ROSMode (using the ROS integration) and up to 120 robots in UnityMode. A usability test was conducted which showed that the target audience of robotics researchers and developers were able to quickly install, setup, and use MAES for implementing simple robot logic.

1 INTRODUCTION

Testing implementations of swarm robotics can be an expensive and difficult task. For this reason, many developers look to simulations as they offer a cheaper and easier solution for testing an algorithm or swarm behavior. Popular simulation solutions include Argos (ARGoS, 2022) and Gazebo (Open Robotics, 2022e). Even simulations, however, can be difficult to set up and configure. This difficulty is usually caused by the simulation tools having a heavy emphasis on modularity and customizability, which often comes at the cost of increased complexity for the user. Additionally, some simulation software have heavy requirements in terms of CPU power on the host machine, which can necessitate using multiple computers, or even clusters.


In this paper we introduce a new version (2.0) of the Multi Agent Exploration Simulator (MAES) tool. MAES 1.0 was initially developed for simulating swarm behavior for achieving online exploration and/or coverage of an unknown space. Developing


algorithms was done directly in the C# code in the Unity Editor (Andreassen et al., 2022a). While this approach allows for easy development, it is not compatible with the workflow of many robotics researchers and developers. In addition, the algorithms developed in MAES 1.0 were not easily ported to real robots, because the interface had very little overlap with existing approaches used in robotics.


For this reason, MAES 2.0 introduces a new interface for controlling the robots using ROS 2 (Robot Operating System) (Open Robotics, 2022h), which is a commonly used tool-set for developing robot software for both simulation and real life applications. This allows robotics researchers and developers to use many existing libraries and solutions when developing control algorithms in MAES. The ROS-based algorithms can also be more easily ported to real robots.

MAES 2.0 thus has two modes: *UnityMode* with support for C# development and *ROSMode* which enables the new ROS integration.

This aims at documenting and presenting the work done to create MAES 2.0. In Section 2 we will discuss ROS in further detail, as well as provide a quick summary of the features of MAES 1.0. In Section 3 we introduce MAES 2.0 including the new architecture and adaption for ROS compatibility. In Section 4

^a <https://orcid.org/0000-0002-2338-265X>

^b <https://orcid.org/0000-0001-7587-531X>

^c <https://orcid.org/0000-0001-9594-0297>

we showcase how MAES 2.0 scales when increasing the number of robots and the size of the map. Section 5 provides a description of, and the results from, a usability test conducted with robotics researchers and students. In Section 6 we conclude on the results of MAES 2.0, the results of the performance test, and results of the usability test. Section 7 touches on related work before Section 8 describes the future work in stall for MAES.

The main contributions of this paper include

- Integrating ROS 2 into MAES
- A usability test proving that robotics researchers and developers are able to use MAES with ROS 2
- Performance test showing that MAES scales well with the number of robots, even on low-power platforms
- Integrating additional visualization features into MAES, e.g. a heatmap and environment tags

2 BACKGROUND INFORMATION

This section introduces important concepts necessary for understanding the rest of the paper. These include a quick summary of ROS, other simulator tools and their ROS support, as well as a quick overview of MAES 1.0.

2.1 ROS and Simulators

Multiple simulation tools exists for robotics, such as Argos (ARGoS, 2022), Gazebo (Open Robotics, 2022e), PlayerStage (Player/Stage, 2022), NVIDIA’s Isaac (NVIDIA, 2022), and more. All of the aforementioned simulators also include some kind of ROS support, either for ROS1 or ROS2.

ROS is a set of libraries and tools for building robotics systems (Open Robotics, 2022h). ROS is currently developed and maintained by Open Robotics (Open Robotics, 2022g). One of the main ideas behind ROS is that the software can be used in both real world robots and simulated environments.

As of writing, the most popular simulator for use with ROS appears to be Gazebo, as it is a first party simulator, that is also developed by Open Robotics. Gazebo is highly modular and customizable. For example the user can specify the physical structure of the robots through URDF (Unified Robot Description Format) files. This type of flexibility allows it to support most use cases but comes at the cost of increased structural complexity, a cumbersome setup process and a high computational overhead.

2.2 ROS Definitions

Throughout this article we will use several ROS specific terms. In this Section these terms will be defined. As no formal definition is provided in the ROS 2 documentation, the following definitions are summaries of the explanations from the ROS2 documentation (Open Robotics, 2022h). There are many other domain specific terms when working with ROS, but in this section we focus on the ones used in this paper.

Definition 1. A **node** is a fundamental ROS 2 element that serves a single, modular purpose in a robotics system (Open Robotics, 2022o).

Definition 2. Nodes publish messages over **topics**, which allows any number of other nodes to subscribe to and access that information (Open Robotics, 2022p).

Definition 3. A **message** is data formatted to follow a specification defined in a ROS workspace and can be sent through a topic. ROS2 contains many built-in message definitions, e.g. `geometry_msgs/Pose` (Open Robotics, 2022a).

Definition 4. ROS 2 relies on the notion of combining **workspaces** using the shell environment. *Workspace* is a ROS term for a location in the file system that contains a number of ROS related development files. The core ROS 2 workspace is called the *underlay*. Subsequent local workspaces are called *overlays*. A workspace can contain any number of packages (Open Robotics, 2022c).

Definition 5. A **package** can be considered a container for ROS 2 code. To be able to install code or share it with others, it is required to have it organized in a package. With packages, it is possible release ROS 2 work and allow others to build and use it easily (Open Robotics, 2022d).

Definition 6. A **launch file** is a file with a definition of how to one or more nodes with a specific parameter configuration. A launch file can be called from the ROS 2 CLI (Command Line Interface) (Open Robotics, 2022f).

Definition 7. A **transform tree** defines the relations between different coordinate frames, in terms of translation, rotation, and relative motion (Open Robotics, 2022b)(Open Robotics, 2022k).

Definition 8. **Actions** are one of the communication types in ROS 2 and are intended for long running tasks. They consist of three parts: a goal, feedback, and a result (Open Robotics, 2022l)

Definition 9. **Services** are another method of communication for nodes in the ROS graph. Services are

based on a call-and-response model, versus topics' publisher-subscriber model (Open Robotics, 2022n).

Definition 10. Nodes have **parameters** to define their default configuration values. You can get and set parameter values from the ROS 2 CLI. You can also save the parameter settings to a file to reload them in a future session (Open Robotics, 2022m).

Definition 11. A **costmap** is grid-like structure of cells (an occupancy grid), representing the surrounding environment in terms of the cost of moving from the current location to the location represented by the cell. A low/zero value means a cell is free and a high value means a cell is occupied (Open Robotics, 2022j).

2.3 MAES 1.0

MAES 1.0 is a discrete-step simulator created in Unity for comparing online exploration/coverage algorithms in continuous space (Andreasen et al., 2022a). MAES 1.0 allows for implementing custom algorithms with custom robot hardware capabilities using C#. Included in the MAES 1.0 tool are implementations of several state-of-the-art exploration and coverage algorithms including Local Voronoi Decomposition (LVD) (Fu et al., 2009), The Next Frontier (TNF) (Colares and Chaimowicz, 2016), and Selective Spiraling and Backtracking (SSB) (Gautam et al., 2018). Additionally, an implementation of Random Ballistic Walk (Kegeleirs et al., 2019) is provided as a baseline for comparison. The algorithms all have different requirements in terms of hardware capabilities for the robots, which allowed MAES 1.0 to provide insight into the benefit of integrating more extensive hardware into the robot. MAES 1.0 has built-in simulated SLAM, broadcast communication, environment tagging, and a robot controller for moving the robot. The tool also features a map generator for dynamically generating cave type and building type maps given a random seed and some parameters. This allows users to test a given algorithm in the general case and not just a predefined map. Furthermore, MAES 1.0 includes functionality for exporting statistics for both exploration and coverage. Using MAES 1.0, it was shown that SSB provides the best coverage and exploration performance, while also having the strictest assumptions about robot hardware and inaccuracies. TNF performed nearly as well in terms of exploration, but with fewer assumptions.

3 MAES 2.0

This section describes the new development featured in MAES 2.0. In Section 3.1 we provide detailed insights into how ROS support is integrated into MAES. In Section 3.2 additional new features in MAES 2.0 are discussed.

3.1 Integration of ROS

This Section describes how ROS is integrated into MAES 2.0. Section 3.1.1 discusses which version of ROS is used and why. In Section 3.1.2 we describe the adaptations made from MAES 1.0 to MAES 2.0 to allow for ROS integration. Section 3.1.3, 3.1.4 and 3.1.5 describe in detail the architecture and communication of MAES with ROS. In Section 3.1.7 an example of an algorithm for controlling the robots is shown. Finally, Section 3.1.8 provides details on how MAES can be deployed both with and without Docker.

3.1.1 ROS1 or ROS2?

As of writing, ROS exists in two major versions, ROS1 and ROS2, where ROS2 is a complete remake created based on observations and lessons learned from ROS1. The latest release for ROS1 is ROS Noetic Ninjemys, which came out in May 2020 and has end-of-life in May 2025. No further releases for ROS1 are planned. The first version of ROS2 was released as alpha1 in August 2015 and the first official release came out in 2017. The currently newest version of ROS2 is Galactic Geochelone, which came out in May 2021 (Open Robotics, 2022i). Galactic is, however, not a long term support (LTS) version. The first LTS version of ROS2, Humble Hawksbill, came out on the 23rd of May 2022 and will have support until May 2027. As of the time of writing, Humble Hawksbill does not yet support some important packages used for navigation purposes, thus the descriptions in this document refer to the non-LTS ROS2 (Galactic Geochelone) release.

The downside of using ROS2 is that a large amount of existing libraries are written for ROS1 and many of them do not have a version compatible with ROS2. However, due to the end of life of ROS1 in 2025 and the continued support for ROS2, we chose to target ROS2 for the MAES simulator. Additionally, we assume it to be easier to upgrade from ROS 2 Galactic Geochelone to Humble Hawksbill, compared to upgrading from one of the ROS 1 LTS versions.

3.1.2 Adapting MAES for ROS2

Unity, on which MAES is based, already supports both ROS1 and ROS2 communication through the 'ROS TCP Connector' and 'ROS TCP Endpoint' plugins. These plugins work in tandem to facilitate communication between Unity and an external ROS system. The TCP Endpoint is a ROS node that relays communication from Unity to the ROS system and vice versa. The TCP Connector is responsible for establishing a connection between between Unity and the TCP Endpoint node.

As described in (Andreasen et al., 2022a), in MAES 1.0 all algorithms had to implement the `IExplorationAlgorithm` interface in order to be injected into the controller of the robot. This interface contains an update function called every logic tick of the simulation as well as a start method for bootstrapping. In adapting MAES 1.0 for ROS we want to maintain compatibility with the old algorithms by using the same interface. For this purpose we created the `Ros2Algorithm` class that implements the `IExplorationAlgorithm` interface. During the logic update phase, the `Ros2Algorithm` evaluates messages received from the ROS system and translates them to the corresponding action in MAES as described in 3.1.5. This algorithm also publishes the current robot state every logic tick as described in 3.1.4.

Unlike the C# algorithm implementations described in (Andreasen et al., 2022a), the `ROS2Algorithm` is nondeterministic because it is dependent on the timing of network messages. For this reason the MAES simulator is non-deterministic when used in `ROSMODE` and deterministic when in `UnityMode`.

MAES 2.0 continues to allow usage of controller algorithms that are written in C# and compiled along side the simulator itself. To avoid the computational overhead introduced by the ROS TCP Endpoint and TCP Connector plugins, the global settings configuration contain a boolean value that toggles these plugins, such that they no dot consume resources when they are actually not needed.

3.1.3 Architecture

The architecture of MAES 1.0 was designed for the personal use of the developers themselves. As a results of this design, any addition of new algorithms and scenarios required editing the existing MAES code. As MAES 2.0 is intended for use as an importable library - the architecture is remodeled to support a framework-like structure where the implementation details of the simulator are encapsulated and hidden from the user. MAES 2.0 exposes an interface

that allows full configuration without needing to manipulate MAES code. This interface exposes methods for simulator instantiation, injection of algorithms and scenarios, and allows for extraction of performance metrics. MAES 2.0 now also contains a Unity Package definition, which allows it to be used in the official Unity Package Manager tool. This means that users with access to a Unity Editor also can access the MAES 2.0 library by pasting the Github repository url into the Unity Package Manager. Code snippet 1 shows an example of how a simulation can be set up using the MAES 2.0 framework. This code can be attached to an empty Unity *GameObject* (the base-objects used in the Unity Editor) which will cause the simulator to be instantiated and run.

The configuration classes (`CaveMapConfig`, `BuildingMapConfig`, `RobotConstraints`, `SimulationScenario`) all have a large set of optional parameters that can be used to tweak the simulation. All of these parameters are described in the documentation in the public repository. For example the user can provide custom robot control algorithm by providing the scenario class with an algorithm factory. This custom algorithm must, however, still extend the `IRobotAlgorithm` interface as described in the original MAES paper (Andreasen et al., 2022a).

Listing 1: Example of MAES 2.0 usage in a unity project.

```
1 void Start(){
2     // Get/instantiate simulation
3     var simulator =
4         ↪ Maes.Simulator.GetInstance();
5
6     // Configure the scenario
7     var caveConfig = new CaveMapConfig(123,
8         ↪ widthInTiles: 75, heightInTiles: 75);
9     var scenario = new
10         ↪ SimulationScenario(123, mapSpawner:
11         ↪ generator =>
12         ↪ generator.GenerateCaveMap(caveConfig));
13     simulator.EnqueueScenario(scenario);
14
15     simulator.StartSimulation();
16 }
```

The size and complexity of the MAES architecture increases significantly when in `ROSMODE` due to the large amount of ROS nodes running alongside MAES. When using MAES in `ROSMODE`, the user must first launch the ROS components, and then subsequently launch MAES. All ROS nodes for all robots are launched from a single custom ROS Launch file called `maes_ros2_multi_robot_launch.py`. An abstract overview of the ROS nodes launched from this launch

file can be seen in Figure 1. The overview is abstract in the sense, that the internal sub nodes created by the Nav2 and Slam.toolbox packages are excluded from the graph for the sake of readability. A full picture containing all nodes and topics can be seen in Appendix C.

In our configuration the script `maes_ros2_multi_robot.launch.py` initially launches the `default_server_endpoint` node from the `ROS_TCP_Endpoint`, which is used to communicate with MAES, as described in Section 3.1.4. This node is not namespaced, and only a single `default_server_endpoint` node is launched for all robots. After launching the `default_server_endpoint`, `maes_ros2_multi_robot.launch.py` starts launching namespaced nodes. Namespace refers to a prefix, e.g. `/robot0/`, which is prepended to all topic names used by the node, as well as the node name itself

In order to know how many robots, i.e. namespaces, to create, `maes_ros2_multi_robot.launch.py` reads from the same configuration file as MAES, which ensures that they are synchronized. In addition to the number of robots, `maes_ros2_multi_robot.launch.py` also reads parameters such as the raytrace range from the configuration file and injects it into the parameter file of each robot. This approach allows for a single parameter file to be modified and used for all robots, which enables easier development and debugging. A disadvantage of this approach is that all robots have the exact same configuration, which can reduce realism and flexibility.

After successfully reading and injecting the parameters, `maes_ros2_multi_robot.launch.py` starts launching the robot specific nodes inside their respective namespaces. For each robot a `maes_robot_controller` node is launched, which is the node containing the robot logic written in Python. Additionally, for each robot, a namespaced launch file called `maes_bringup.launch` is called. Each instance of the `maes_bringup.launch` further launches the `slam_toolbox` and `Nav2` nodes inside the given namespace. These nodes are used for navigation and mapping purposes respectively.

Finally, for each robot a namespaced `rviz` (ROS' integrated visualization tool) instance can be launched depending on the value of a command line argument.

3.1.4 Publishing Robot State through ROS

Many nodes in the MAES ROS workspace depend on input from MAES to function. To supply the needed input MAES publishes data to the `/tf`, `/scan` and `/maes_state` topics. The `/tf` topic contains infor-

mation about the transforms, i.e. the position and rotation of a given robot including the relative positions of sub-components of the robot. As of writing, the transforms published by MAES do not contain any inaccuracies. This reduces realism, as a position derived through odometry would likely be imprecise due to sensor inaccuracies. However, in the future this could be adapted to use the positional inaccuracy variable that already exists for SLAM in UnityMode algorithms (Andreasen et al., 2022a).

The `/scan` topic data is created by performing a series of ray casts from the current position of the robot. Every message sent to the `/scan` topic consists of distance information for 180 ray casts performed at a 2 degree interval around the robot. The scan is performed and published ten times per second for each robot.

The `/maes_state` topic data is constructed from the robot state inside MAES and contains information specific to the simulated robot in MAES. The scope of information is very similar to what is provided to the UnityMode algorithms, which are implemented in C#.

The `/maes_state` topic uses a custom ROS2 message called `StateMsg`. `StateMsg` contains information regarding the current simulation tick, the current status of the given robot (e.g. rotating or moving), whether the robot is colliding with something, incoming broadcast messages, environment tags nearby as well as information about other nearby robots.

As mentioned in Section 3.1.2 we use the ROS TCP Connector package from Unity in order to publish the messages to the ROS topics (Unity-Technologies, 2022a). The ROS TCP Connector allows for generating serializable C# classes from message definitions found in a ROS workspace. The generated classes can then be instantiated as objects in the C# code, populated with values, serialized, and then published to the corresponding topic. The messages are sent through the ROS TCP Connector and then directly to the ROS TCP Endpoint(Unity-Technologies, 2022b) node through a direct TCP connection. The ROS TCP Endpoint relays all messages as if they came from a regular ROS node, e.g. a component of a real robot publishing to the ROS network. A visualization of this system can be seen in Figure 2.

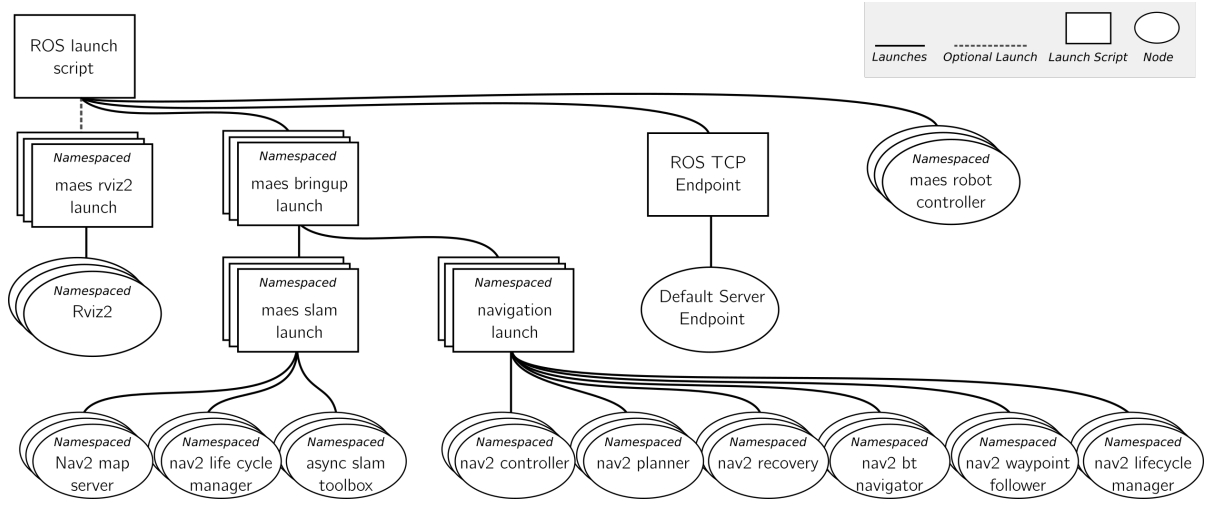


Figure 1: Visualization of how nodes are launched from our ROS launch script

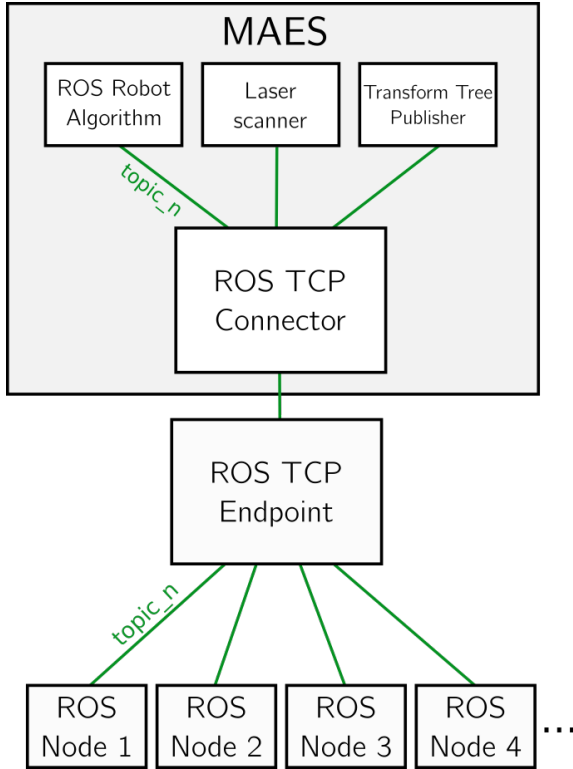


Figure 2: A visualization of how MAES publishes to ROS nodes on ROS topics

The `/maes_state` topic is used by the `maes_robot_controller` ROS node (described in Section 3.1.7). Information received from this topic can then be used in the logic controller for the robot to decide what action to perform next.

Nav2 is the ROS2 package used for navigation (Macenski et al., 2020). Nav2 contains several nodes that depend on the `/tf` topic for information

regarding current position. Additionally, Nav2 constructs costmaps from the `/scan` topic used for navigation.

Slam_toolbox (Macenski and Jambrecic, 2021) also uses both the `/tf` and `/scan` topics for SLAM. `/scan` is used for creating the map with the `/tf` used for positioning the robot within the map. Slam_toolbox, however, differs from Nav2 by having a higher resolution and the ability to export the map for later use. The costmap does not necessarily need as high a resolution as the map generated for export.

For MAES 2.0 we initially wanted to be able to merge maps for both the `slam_toolbox` as well as the Nav2 costmap, in order to mirror the map merging feature that is available to UnityMode algorithms and allow cooperative navigation in Nav2. This is, however, not yet possible using the official `slam_toolbox` and `nav2` packages. In ROS1, the de facto standard for slam is the `slam_gmapping` package (ros-perception, 2022). Gmapping supports multiple map merging, but has not yet been ported to ROS 2. Some unofficial forks using the binaries from `gmapping` for map merge using `slam_toolbox` have been made available, e.g. (robo-friends, 2022), but for MAES we choose to wait for the official release.

3.1.5 Mapping Nav2 Commands to MAES

Whenever a robot is instructed to navigate to some point using Nav2, the Nav2 nodes publish movement instructions on the `'cmd_vel'` topic. These instructions consist of a desired force application for each wheel of robot. The standard Nav2 configuration assumes that differential steering is possible, i.e. that the robot can rotate while moving. This was not possible in MAES 1.0, but has been introduced to the

MAES 2 controller interface. In addition, the Nav2 stack in the MAES 2.0 ROS workspace uses a 'Rotation Shim Controller' plugin. This plugin ensures, that the Nav2 nodes map the movement instructions such that the robot will stand still while performing large rotations and only do small rotational corrections while moving forward.

3.1.6 Configuring ROS 2 and MAES

One of the goals of MAES is to allow for easier development and less setup and configuration compared to using ROS 2 with other simulators. This is achieved partly by having a single configuration file `maes.config.yaml`, where all customization is done. In addition, most of the parameters have default values, allowing the user to quickly configure a standard simulation. Both MAES, and the ROS2 launch file (see more in Section 3.1.3) read from this same file, and configure themselves depending on the values found.

3.1.7 Controlling the Robot in MAES using ROS

In MAES UnityMode it is possible for the user to program their own algorithm (i.e. robot logic), using C#. We introduce the `maes_robot_controller` node to allow for similar functionality when running in ROSMode.

When the `maes_robot_controller` is initialized, all of the subscriptions and clients for the topics, services, and action servers are created in the given namespace, including callback functions. The services and action servers include functionality for navigation, broadcasting messages, and depositing environment tags in MAES. This allows the user to start implementing logic immediately instead of also having to configure the controller. The logic controlling the behavior of the robot is written in the main function of the `maes_robot_controller`. The user can then utilize the services, information received on topics, and action servers to express the intended behavior. A more detailed description of the data published to ROS from MAES can be found in Section 3.1.4.

An example of a very simple frontier algorithm implemented using this interface can be seen in Code Snippet 2, where it is shown how `maes_robot_controller` exposes easy to use interfaces for logging, getting the state of the robot, and using the costmap for navigation.

Additionally, `maes_robot_controller` exposes methods for using the asynchronous navigation server such as `nav.to_pos`. `nav.to_pos` makes the robot navigate to an (x,y) coordinate asynchronously. Since actions servers in ROS can be canceled, we allow for checking the status of the current goal with

a `goal_handle`, but we also implement easy to use functions such as `cancel_nav`, which simply cancels the current goal using the navigation action client.

Listing 2: Example of a Frontier Algorithm implemented in the `maes_robot_controller` in Python

```

1 def main(args=None):
2     rclpy.init(args=args)
3
4     # Initialise controller
5     robot = RobotController()
6     robot.wait_for_maes_to_start_simulation()
7
8     # Declare of logic variables
9     next_goal: Coord2D = None
10    next_goal_costmap_index: int = None
11
12    # This method returns true if the tile is not itself
13    # unknown, but has 2 neighbors, that are unknown
14    def is_frontier(map_index: int, costmap: MaesCostmap):
15        # -1 = unknown, 0 = certain to be open, 100 =
16        # certain to be obstacle
17        # It is itself unknown
18        if costmap.costmap.data[map_index] == -1:
19            return False
20        # It is itself a wall
21        if costmap.costmap.data[map_index] >= 65:
22            return False
23
24        return
25        # costmap.has_at_least_n_unknown_neighbors(index=map_index,
26        # n=2)
27
28    # While ROS is running
29    while rclpy.ok():
30        rclpy.spin_once(robot) # Allow for callback execution
31
32        # If no goal found or current nav complete
33        if next_goal is None or robot.is_nav_complete():
34            # Find index of first tile in costmap that is a
35            # frontier
36            goal_index = next((index for index, value in
37                               enumerate(robot.global_costmap.costmap.data) if
38                               is_frontier(index, robot.global_costmap)), None)
39
40            # No more frontiers found, just continue
41            if goal_index is None:
42                robot.logger.log_info("Robot {0} has no more
43                # frontiers".format(robot.topic_namespace_prefix))
44                continue
45
46            next_goal =
47            # robot.global_costmap.costmap_index_to_pos(goal_index)
48            next_goal_costmap_index = goal_index
49            # Deposit tag every time a new target/goal is
50            # found
51            robot.deposit_tag("From tick
52            # {0}".format(robot.state.tick))
53            robot.nav_to_pos(next_goal.x, next_goal.y)
54            # If goal present but not yet reached, i.e. it is
55            # still a frontier
56            elif is_frontier(next_goal_costmap_index,
57                             robot.global_costmap):
58                # This section allows for logging feedback from
59                # action server
60                continue
61            # If goal is explored
62            else:
63                next_goal_costmap_index = None
64                next_goal = None
65                robot.cancel_nav()

```

3.1.8 Setup & Deployment

In the Github Repository(Andreasen et al., 2022b) we include release packages and a readme with documentation to guide the user. This readme was also used in the usability test found in Section 5. While it is possible to install all ROS 2 and Nav2 dependencies required for the MAES ROS workspace natively, we also include Docker support to make it easier to install. The docker image has everything preinstalled, which allows for a much easier installation process, while not being dependent on running a specific version of Ubuntu as the host OS. Additionally, we include the dependencies in the correct version in the Docker image, which ensures that no dependency upgrade from a third party breaks the code.

A Dockerfile, i.e. a script for building our docker image from the basis of public docker repositories (MAES, 2022a), is included in the release alongside the MAES executable. The Docker image depends on using a shared directory with the release package found on Github, where the MAES ROS workspace is found. Sharing said directory with docker is also explained in the readme. This allows the user to open the workspace in their favorite IDE (Integrated Development Environment), instead of having to use terminal editors like Vim or Nano directly within a running container. Additionally, instructions for allowing a docker container to display GUI elements (e.g. the window from rviz) through to the host system can also be found in the readme. This means, that the user can still use rviz visualization, even though the ROS environment is running in a docker container. Piping the rviz window from the Docker container to the host system is, however, only tested to work on Ubuntu (22.04, 21.10, 21.04 and 20.04), but we expect it to be doable on most fairly recent Linux distributions. Some limited functionality can be achieved in Windows 10 as well, albeit at a reduced performance. Windows 11 has not been tested at all, and the piping is not supported on macOS.

In the future, whenever a new version of MAES or the MAES ROS workspace is completed, a new release package should be posted to Github with the newest version of the docker image and the MAES executable.

3.2 Additional features in MAES 2.0

This Section describes new features of MAES 2.0 in addition to ROS support. These include a new Heatmap, signal degradation, custom maps, expanded statistics gathering and more.

3.2.1 Signal Degradation

In MAES 1.0 it was possible specify parameters that determined the maximum communication range and whether robots can communicate through walls. In MAES 2.0 we expand upon this by allowing the user to provide a custom function for calculating communication success. This function takes as arguments the total distance traveled for the signal, and the distance traveled through solid walls. This function then must return a boolean value indicating whether the signal can be received.

The advantage of this approach is that the user can control the model used for signal degradation. If a stochastic model is desired, the user can factor random number generation into the signal success function. If no signal success function is supplied, MAES will use a default function that always yields success, which results in lossless communication at any distance.

3.2.2 Custom Maps

As detailed in (Andreasen et al., 2022a), MAES 1.0 provided automated map generation for building and cave map types with configurable parameters. In MAES 2.0 we introduce an additional option of specifying a completely custom map by providing an image in the 'Portable Gray Map' format (.pgm). PGM is also the format exported by the Nav2 Map Saver node, meaning that ROS generated SLAM maps can now be imported into MAES. When provided with an image, the MAES 2.0 map generator creates wall tiles for every completely black pixel and open tiles for every other pixel. Figure 3 shows an example of an image that has been converted into a map.

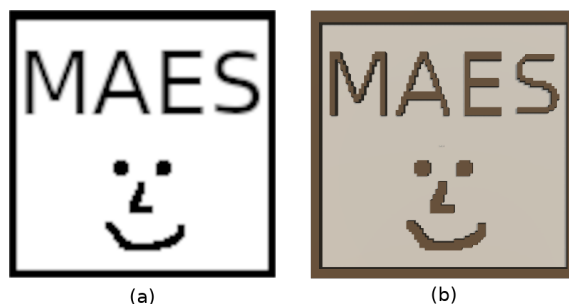


Figure 3: This figure shows an example of a custom map of size 50x50 tiles, i.e. image has a resolution of 50x50 pixels. The map generator uses the input image (a) and generates a map (b) with walls for each black pixels.

3.2.3 Heatmaps

To allow better analysis of algorithm behavior, MAES 2.0 features two new heatmap visualization modes: one for the exploration measure and one for coverage. This could for example be useful for testing surveillance algorithms. The heatmaps display a color at each tile of the map, indicating how recently it has been explored/covered by an agent. Tiles that have been explored/covered recently have a red tint. The tiles progressively change to a blue tint as time passes without a robot exploring/covering the tiles.

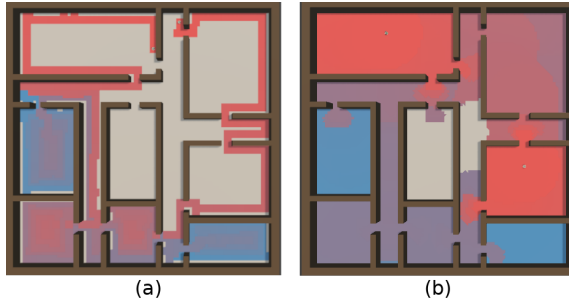


Figure 4: (a) Shows the heatmap visualisation for coverage and (b) shows the heatmap for exploration. Red areas have been explored/covered recently. Blue areas have been explored/covered earlier. The beige areas have not yet been explored at all.

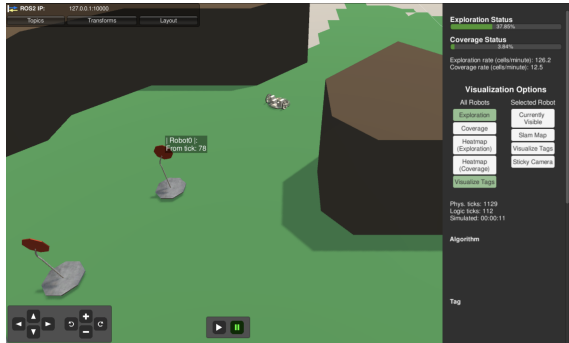


Figure 5: A screenshot from MAES 2.0 with an environment tag visualized with the hover menu containing sender and content of the tag.

3.2.4 Environment Tags

In MAES 1.0 it was difficult as a user to see and debug environment tags, since they were only displayed as Unity Gizmos (debugging elements only visible in the Unity Editor). In MAES 2.0, environment tags are now rendered the same way as the robots, which means they are visible outside of the Unity Editor. Additionally, MAES 2.0 includes functionality for showing the content of the environment tag by hover-

ing over the tag with the cursor. A tag can be selected by clicking on the tag, after which the content will permanently (or until another tag is selected) be displayed in the debug info menu on the right side of the screen. In Figure 5 the new environment tag model can be seen.

3.2.5 MAES 2.0 Graphical User Interface

As shown in Figure 5, the GUI of MAES 2.0 includes several panels for controlling the simulation. In MAES 2.0, the GUI adapts to whether it is run in ROSMode or UnityMode. For example, the visualisation of the ROS connection in the top left corner is not present when running in UnityMode. Additionally, the fast forward buttons are hidden in ROSMode, due to timing issues in the nodes when fast forwarding with ROS - some Nav2 nodes have some assumptions about the robot speed, which can be violated if MAES is sped up.

The settings menu on the right of the screen is mostly identical to MAES 1.0. However, the visualization options for all robots and a given selected robot have been added.

The GUI panel for controlling the camera in the bottom left corner of the screen allows for zoom, tilt and rotation of the camera and is identical to the one found in MAES 1.0.

3.3 MAES 2.0 Tests

MAES 2.0 is intended as an open source project where new features may be implemented by many different contributors who may not have in-depth experience with the MAES framework. To assist in verifying the correctness of new code, we introduce tests that assert the correctness of existing functionality and as such function as regression tests. These tests can help detect when new changes break existing code. MAES 2.0 provides automated unit tests that verify the behavior of selected individual components. The components are chosen for testing based on how likely we deem it that future changes may affect them. Moreover, we also specify a test plan for a manually executed system test that can help verify the correctness of the system as a whole. The system test also verifies correct behavior of the ROS components which can be difficult to achieve with unit tests.

3.3.1 Unit Tests

Unity provides a framework for unit testing that is split into two modes: edit mode tests and play mode tests. Edit mode tests are akin to traditional unit tests where each component is tested in complete isolation

independent from the Unity engine. In contrast, play mode tests are executed while the unity engine runs the simulator in the background. This allows us to evaluate the behavior of components that are dependent on the state of the simulator or whose behavior is otherwise tied to the Unity engine.

For example, we perform play mode tests for the *2DRobotController*. The controller is responsible for moving and rotating the robot which is dependent on the *Unity Physics2d Engine*. Code snippet 3 shows an example of a play mode test for the *2DRobotController*. This test verifies that the distance provided to the *Move()* function of the controller approximately corresponds to the distance that the robot actually moves. This is a parameterized test which takes the target movement distance as input. A separate unit test is executed for each input specified by the *[TestCase]* entries in lines 2-4.

Listing 3: Pseudo code example of a play mode test for the *RobotController* that verifies the distance traveled when issuing a move command.

```

1  [UnityTest]
2  [TestCase(1.0f)]
3  [TestCase(5.0f)]
4  [TestCase(20.0f)]
5  void CorrectDistanceTest(targetDistance):
6      startPos = robot.currentPosition()
7      simulation.start()
8
9      robot.controller.move(targetDistance)
10     while (robot is moving)
11         waitTillNextFrame()
12
13     endingPos = robot.currentPosition()
14     movedDistance = endingPos - startPos
15     Assert.Equals(targetDistance,
        ↪ movedDistance, delta: 0.1f)

```

In addition to parameterizing individual tests we also parameterize the entire *2DRobotController* test suite through *TestFixturees*, which allows us to test all of these behaviors on different simulation configurations, for example scenarios where the movement speed of the robots is altered.

As of writing, the play mode tests and edit mode tests provide a total of 30% code coverage. The reason for this relatively low coverage percentage is that we chose to down prioritize testing in the early stages of development. This was done because we found the requirements and functionality to be changing so rapidly that tests would quickly become obsolete or even serve as a hindrance for further development. However, as the work progresses and the tool

is now becoming more clearly defined, it could be argued that there is a need for more extensive regression testing to properly facilitate open source development. While the coverage percentage is relatively low, it should be mentioned that we prioritize testing the components that we assess to be the most critical. This includes the robot controller, communications manager and the performance metrics handlers.

None of the unit tests cover any functionality related to external ROS components as these are exceedingly difficult to properly setup in an isolated and reproducible testing environment. Instead testing of the ROS environment is done through system tests.

3.3.2 System Test in ROSMode

Running MAES 2.0 in ROSMode is inherently non-deterministic due to the timing differences of running many ros nodes (processes) at the same time. Even external factors like the scheduler of the host operating system in use can impact the results slightly. However, running a system test in ROSMode may still yield useful results, that can show whether all of the components can work together. For this reason we design a system test with ROSMode. The test is not automated, but this would be a useful addition in the future, e.g. through a script running and closing MAES and ROS automatically. The test consists of a yaml configuration file for MAES and ROS, where we know that the configuration produces fairly consistent results in terms of exploration rate. A contributor can then use this configuration and run the example to check if some new feature has significantly impacted the compatibility between the components of the system in some way. The configuration file is called *maes_config_ros_system_test.yaml* and can be found in the *maes_ros2_interface* package in the MAES ROS workspace.

The test yaml file creates a small 30x30 map with a single robot, and should be run with the example frontier algorithm included in the *maes_robot_controller.py* file.

As can be seen in Table 1, the exploration results are quite similar between runs. Additionally, the system test only takes about 2-3 minutes to complete, which is also an advantage when trying to make contributors use it, compared to requiring a daunting 30-40 minute test. This test can thus be used before creating a merge request to the repository in order to test functionality of the system as a whole. As long as the robot achieves ~99,9% explored within about 2,5 minutes, the code contribution is unlikely to have broken any critical parts of the system.

A downside of this test is that a host-system slowdown could impact the exploration performance and

thus the reliability of the test. For example, if the computer running the test does not have the needed resources, or is busy with a lot of other background tasks, and therefore cannot execute commands in all processes fast enough, the robot may receive outdated commands or even no commands at all.

The test has been shown to work on a laptop with modest hardware, and should thus work on most computers. The test results found in Table 1 were achieved on a laptop with Ubuntu 20.04, 4 cores / 8 threads @3.0GHz, 8GB ram laptop with integrated Intel graphics.

Table 1: ROSMode system test results

	Time spent until 99,9% explored
Run 1	1450 ticks (2:25 minutes)
Run 2	1500 ticks (2:30 minutes)
Run 3	1480 ticks (2:28 minutes)

4 PERFORMANCE TESTS

As mentioned in Section 1, existing simulating tools have heavy requirements in terms of computing power, which is an obstacle that might be encountered even when simulating just a single robot. As robots in real life are more and more likely to act as a part of a swarm rather than as a single isolated agent, the simulating tools must be able to simulate more than one robot in the same environment.

To accommodate this, we orchestrated a performance test, which both gives insight in the current state of performance, and can act as a baseline for benchmarking future performance-upgrades to MAES.

The tests were conducted on a thin and light laptop with an AMD 4500U CPU (6 cores @2.3GHz) and 8 GB of DDR4 memory running Ubuntu 20.04 LTS as its operating system.

A logging-script was constructed which enabled us to log the total CPU utilization, the total memory usage, and the network traffic to/from a specified networking interface - all data points being logged once per second. Each log entry would be labeled with timestamp, making alignment of the data easier. While the logging-script was running, it was possible to type in events in the terminal, which would also be labeled with a timestamp, making it easier to determine which event triggered some particularly interesting-looking data point.

Tests were run in both ROSMode and UnityMode, each with the goal of seeing performance impact of a varied amount of robots as well as varied map

sizes. The robots in the ROSMode tests were running the simple frontier algorithm example found in the MAES ROS workspace, and the robots in the UnityMode tests were running with the same baseline random-walk algorithm used in (Andreasen et al., 2022a). Tests in UnityMode had no network activity to log, since it does not communicate with other processes. During each test, we logged when MAES was started, when and how ROS was started (when applicable), and when simulation was started and halted. Tests were run once for each configuration.

4.1 Map sizes

Both ROSMode and UnityMode was tested with map sizes 30x30, 40x40, and 50x50, but the map size did not notably impact the network use or CPU utilization. When running in ROSMode, there was a significant increase in network traffic, when enabling the rviz plugin. This difference is seen in Figures 7 and 9 (the red lines on the right half of each figure). This increase is due to the entire map of every robot being sent from MAES to rviz, and a larger map means more data to send.

4.2 Number of robots

Both ROSMode and UnityMode was tested with one, three, and five robots, each on maps of sizes 30x30, 40x40, and 50x50. The impacts of the map sizes were covered in Section 4.1. Varying between these numbers of robots produced no significant performance impacts in UnityMode. In ROSMode, however, there were several increases in resource usage.

Figures 6, 7, and 8 all show a run on the same 50x50 cave-map, with their respective number of robots. The results show that using MAES in ROSMode is highly dependent on the available CPU resources (green solid line), as this is the only resource reaching its limit during the run with five robots on a 50x50 size map with rviz enabled (i.e. the heaviest run throughout all tests).

If rviz is not enabled, the tests show that running with up to five robots is possible given test system. If rviz is needed, the test system was unable to run more than three robots without slow downs, that could impact the performance of the robots.

These limits are, of course, highly dependent on machine running MAES in ROSMode, and we purposely chose not to run these tests on a high performing computer in order to cement the fact that MAES is a relatively lightweight simulator compared to other ROS 2 compatible simulators. Figure 10 further backs up the claim of MAES being lightweight.

Here, MAES is run with the same number of robots, with the same size cave-map as the test in Figure 8, but now in UnityMode. The resource demands are thus significantly lower when running in UnityMode, and this test demonstrates the overhead of running ROS 2.

As a final set of tests we also tried pushing the limits of MAES in UnityMode with respect to the number of robots. Figure 11 shows the results of 120 robots being simulated on a cave-map of size 75x75, where MAES is still was able to simulate in real-time. It should be noted that the frame rate was reduced to 4-5 frames per second when trying to simulate this many robots, but the underlying physical simulation was not affected by this. As Figure 11 also shows, the computer still had a surplus of CPU resources, but increasing the number of robots from 120 resulted in simulations not keeping up with real-time. This is due to MAES not being able to fully utilize multiple CPU cores, which is an optimization that could be investigated in the future, which we will expand further upon in Section 8.

5 USABILITY TEST

The ROS MAES integration was made in cooperation with the Robotics Lab at AAU, which provided us with continuous feedback during development. Regardless, we still choose to design and perform usability tests to confirm the usability of MAES 2.0 in ROSMode for the target audience, i.e. robotics researchers and students. The design and execution of this usability test is described in Section 5.1. In Section 5.2 we analyze the results of the usability test. Finally, in Section 5.3 we discuss the changes made to MAES following the feedback from the usability test.

5.1 Test Description

The goal of the usability test is to test whether the target audience can setup and use MAES in ROSMode only using the guides and documentation from the readme included in the GitHub repository. In order to test that claim, we must first define the target audience. Since the primary function of MAES is to simulate multi robot behavior, we determine the target audience to be robotics developers and researchers. We recruit the subjects for the usability test directly from the target audience, meaning we can make some assumptions about their domain knowledge. For example, we assume that they have at least some knowledge of ROS.

The test design consists of a short interview, followed by three phases of tasks, and then finally a small closing interview with room for open discussions. The entire test takes at most 60 minutes for each subject. The complete list of tasks, questions, interviewer guidelines etc. can be seen in Appendix B. The interviewer guidelines helps ensure a similar experience for all subjects. Additionally, we define the ways in which the interviewer is allowed to help the subject, since we are interested in testing whether the subject can complete the setup using only the documentation in the readme from the repository. This is important, as MAES is intended as an Open Source project, where everyone can use it and contribute to it.

The initial interview is meant to get an understanding of the subject's current knowledge of ROS, MAES, Gazebo, and Python. This is important to know, since some of the tasks require some knowledge of Python. If a subject struggles with a task requiring Python skills, we cannot necessarily assume that something is wrong with MAES or ROS, if the subject has never worked with Python.

The tasks are divided into the *Setup Phase*, the *Configure Phase*, and the *Use Phase*. This division was done, since each phase depends on the previous phase. A subject can struggle with setup and still have a good experience using it, once it is set up and configured. For this reason we have included a limit of 20 minutes for each phase, after which the interviewer helps the subject finish the current task and move on to the next phase.

The Setup Phase includes tasks for setting up Docker and initially running MAES. The Configuration Phase revolves around changing the system parameters in MAES. Phase 3, the Use Phase, includes tasks for testing if the subject can alter the `maes_robot_controller` python script to control the behavior of the robots, as well as tasks for using the ROS services exposed by MAES, such as depositing environment tags.

The closing interview is meant as an open discussion. This is included, since MAES 2.0 is still in development, and we are still open to new ideas.

5.2 Results of Usability Test

The usability test was completed by five subjects, which we will refer to as S1 through S5. The subjects are anonymized except for their title and experience using ROS, Python, and Gazebo. The full, anonymized information about all of the subjects can be found in Appendix D.

Table 12 shows the time spent on each task for all

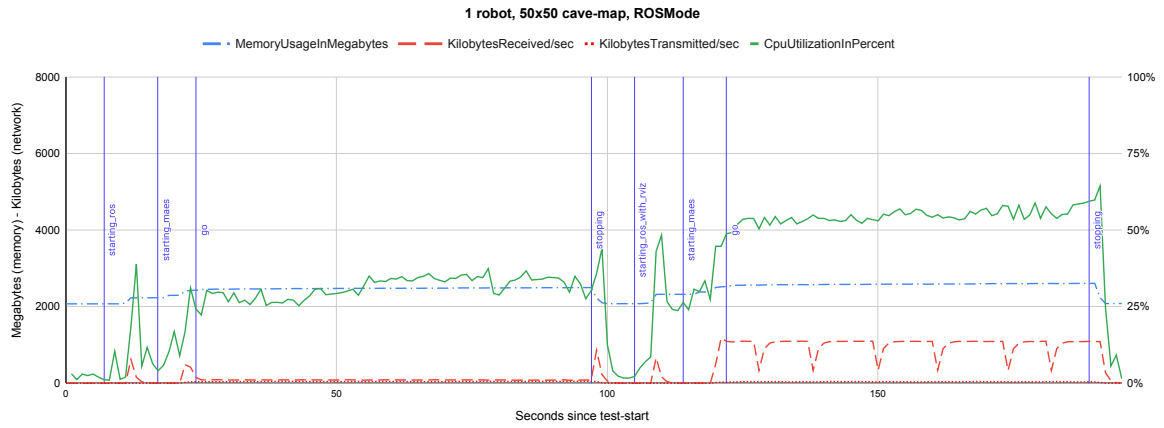


Figure 6: Performance metrics of one robot in a 50x50 cave-type map in ROSMode



Figure 7: Performance metrics of three robots in a 50x50 cave-type map in ROSMode

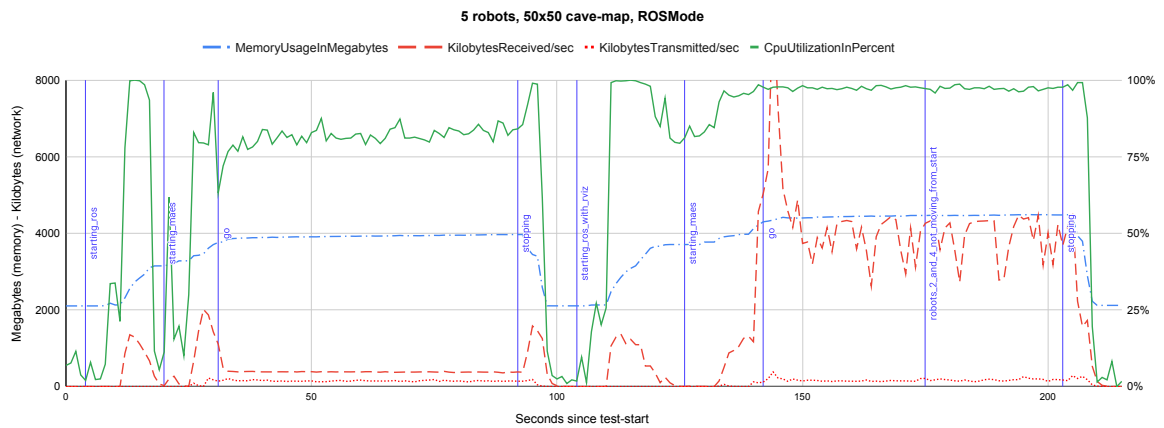


Figure 8: Performance metrics of five robots in a 50x50 cave-type map in ROSMode

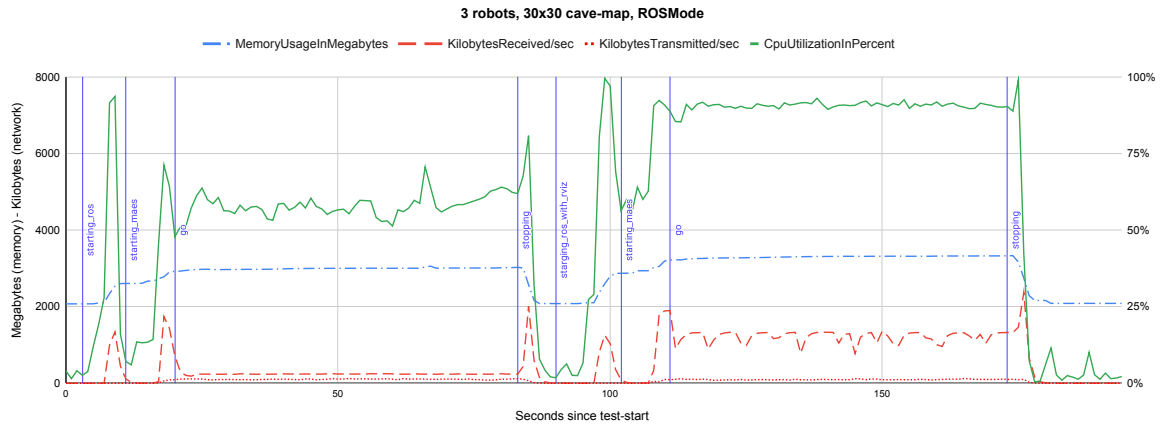


Figure 9: Performance metrics of three robots in a 30x30 cave-type map in ROSMode

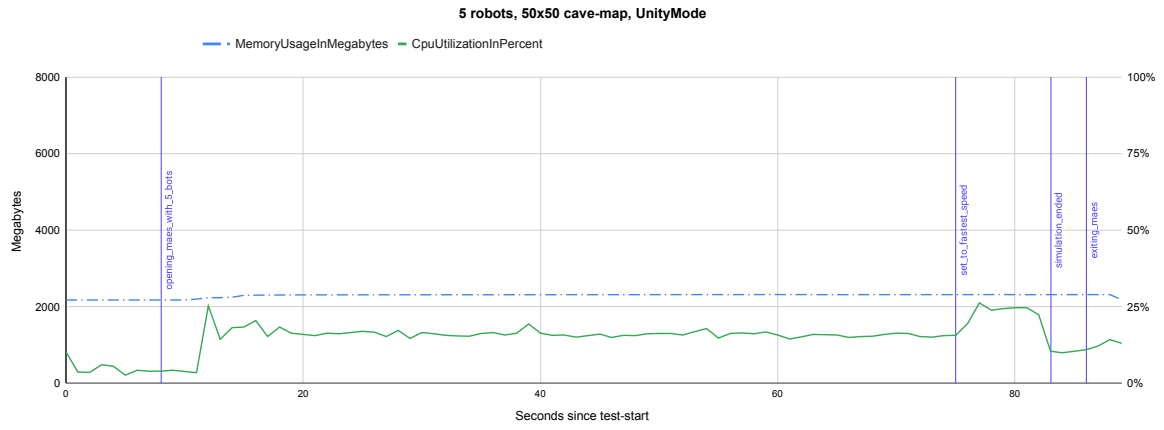


Figure 10: Performance metrics of five robots in a 50x50 cave-type map in UnityMode

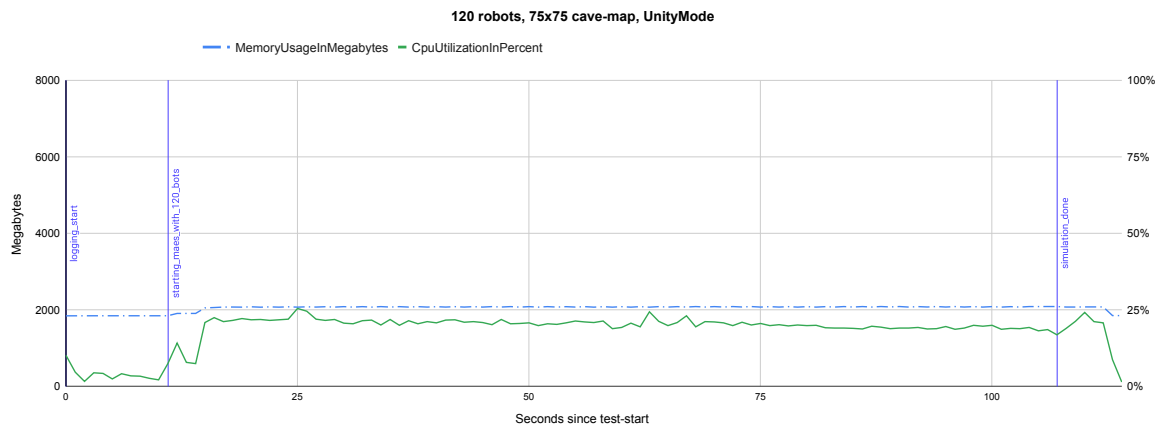


Figure 11: Performance metrics of 120 robots in a 75x75 cave-type map in UnityMode

subjects. The exact data can be found in Appendix D, and audio recordings from the usability test can be found in (MAES, 2022b). Note that these audio recordings include both the interviews before and after the test, as well as the introduction and some unexpected interruptions during the interview. For this reason the timestamps do not exactly match the results of Figure 12, as we only include the time actually spent on the test in this Figure.

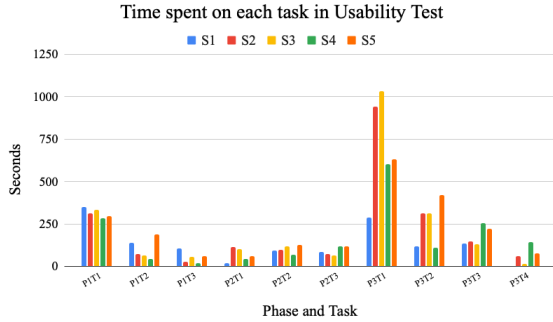


Figure 12: Comparison of time used for each task for all subjects. X axis is phase and task and y axis is time spent in seconds.

The subjects spent time in the range of 22 to 37 minutes completing all the tasks. Phase 1 Task 1 (P1T1) includes building the docker image and the ROS workspace, and for this reason a lot of the time spent was waiting for the computer to complete the build. This is probably also the reason why, the subjects were so close in terms of time spent completing P1T1. All tasks in Phase 1 were completed by all subjects. The target audience thus appear to be able to set up MAES in ROSMode.

In Phase 2, four out of five subjects finished all tasks. S3, however, accidentally changed the map type to custom map and not cave map, and thus did not finish P2T2. Regarding P2T3, S1 tried build the ROS workspace from the wrong directory once, but managed to build it correctly later. Configuration of MAES in ROSMode, i.e. Phase 2, appears to be possible for the target audience.

In Phase 3, i.e. the Use Phase, the subjects had some difficulties. P3T1 is to implement some very simple logic into the controller of the robot. While S1 finished this task without difficulties, both S2, S3 and S4 made the same mistake. It was unclear to S2, S3 and S4 where to put the code, and they all tried to put it into the main function of the robot controller Python script. This made sense to the subjects, since many ROS nodes are controlled this way. For the version of MAES used for testing, however, the logic of the robot had to be written in the `logic_loop_callback`

function, which is continuously called whenever the controller receives a state update from MAES, which happens every 0.1 second with default settings. (Note: This has since been changed as a result of this usability test. See Section 5.3) Information regarding the `logic_loop_callback` function was included in the readme directly below the steps for configuration. Additionally, some instructions were left in the `logic_loop_callback` function itself. The subjects did, however, not read the rest of the readme nor the aforementioned instructions. Subject S2, S3, S4 eventually did either read the readme or find the instructions, but it took significantly longer to complete P3T1 in this way.

S5 *did* read the readme, but misunderstood how the Nav2 ActionClient is implemented into the controller. ActionClients can be used with either asynchronous or synchronous calls. S5 assumed synchronous calls and tried to wait for a result, but this blocked the execution of the rest of the code. Eventually, S5 figured out, that the call used in our controller is asynchronous, and that S5 should use the `goal_handle` to ask for the status of the action call, after which S5 finished P3T1.

P3T2 was finished by S1, S3, S4 and S5 without major issues. S2, however, did not complete P3T2. S5 created logic, that continuously sent navigation requests and then canceled them immediately afterwards. This way, the robot enters an infinite loop oscillating between canceling and requesting navigation.

P3T3 asked the subjects to implement logic for continuously depositing environment tags. P3T3 was finished without major issues by all subjects.

P3T4 regarding opening the topic menu was already finished by S1 during P3T1, and S1 thus did not try again. S5 had trouble finding the topic menu, since S5 assumed it to be a menu in the top menu bar of the Ubuntu operating system. S5 eventually found the topic menu.

5.3 Changes made after usability test

As described in Section 5.2, most subjects of the usability test were able to complete almost all tasks. Most of the subjects, however, struggled with Task 1 in Phase 3 (P1T3). Specifically S2, S3, and S4 were unable to easily figure out where to put the logic code inside the controller. Most subjects initially tried the main function, even though we already had a description in the readme telling how to do it, as well as instructions in the correct function inside the robot controller. In the version used for the usability test, the logic had to be expressed inside of a `logic_loop` call-

back function called every time the state of the robot was published to ROS. While this solution provided easy coupling of MAES and ROS, it was not intuitive to use for the target audience. For this reason the logic is now moved to the main function.

S1 and S5 did not understand the notion of a tick (i.e. the smallest unit of progression in the MAES simulation). In order to help the user understand this more easily, we now include a small explanation in the robot controller node, which is where the user is supposed to utilize it.

S2 suggested having a grid to show the size of one unit of length in MAES, since it was difficult to grasp from the visualization numerically how far two robots were from each other. The suggestion from S2 with a grid in the middle of the map would mimic the behaviour of RVIZ. We, however, do not consider this a good solution, since the map can easily be larger than the grid, and it can then eventually again be difficult to see the size of a unit. We solve the problem in a different way. Instead of the grid, MAES now shows the coordinate of the point in the map which the cursors points to. This eases debugging of robot movement, since it is possible to find target positions visually using the cursor. An screenshot showing the coordinate GUI can be seen in 13.



Figure 13: New UI element showing the coordinates of the position pointed to with cursor

S5 suggested having a list of all robots in the UI to allow for easily selected and finding a robot with a given name. While we like the suggestion, we do not consider it essential for version 2.0. A list of robots would be especially useful for swarm systems with many robots. We include this as an 'Issue' for an enhancement on the GitHub code repository (Andreasen et al., 2022b).

S2 was unsure of how to evaluate the physical dynamics of a robot in MAES, e.g. the interactions between different parts. However, this is expected, as the design of MAES prioritized reduced complexity over customizable robots. We thus do not consider doing more about this suggestion.

S5 suggested making it clearer, that the topic menu in the top left corner of the MAES GUI is in fact a menu. However, 4 out of 5 of the subjects found the menu quickly and S5 also did within a minute, so

we do not consider this critical and will thus not act on it.

6 CONCLUSION

MAES 2.0 can be used for simulating, and developing, the logic behind swarm robots in a 2 dimensional environment. MAES 2.0 includes both UnityMode with C# development as well as the new ROSMode. ROSMode allows for developing the logic of the robot directly as a ROSNode, and thus bridge the gap to real world robot programming. MAES 2.0 also includes several improvements from 1.0, e.g. heatmaps, custom maps, improved environment tags, and a better graphical user interface. MAES 2.0 is tested using both system tests and unit tests. Furthermore, performance tests showed, that MAES 2.0 can be run in real time on modest hardware with up to 5 robots in ROSMode. UnityMode has less overhead than ROSMode, and MAES 2.0 can run with up to 120 robots in UnityMode given the same hardware. A usability test was conducted, that shows that MAES can be setup, configured, and used by target audience of robotics researchers and developers within 60 minutes.

7 RELATED WORK

MAES is intended as an alternative simulator to use with ROS with an emphasis on ease of use. Many other simulation tools for robots already exist which emphasize different aspects of robotics in different ways. ARGoS (ARGoS, 2022), for example, is another popular simulation tool with ROS support is with a main focus on multi robot systems. Furthermore, Gazebo (Open Robotics, 2022e) is very popular due to its vast modularity and compatibility with ROS. Gazebo is part of the Open Robotics Foundation, which ROS also belongs to. There exist other simulation tools like NVIDIA's Isaac (NVIDIA, 2022), Player/Stage (Player/Stage, 2022) and more.

8 FUTURE WORK

MAES is intended to continue as an open source project. In order for an open source project to succeed, contributors and a community are needed. We want to make it as easy as possible to join and start contributing to MAES by making sure the tools used are modern and have support for many years.

As mentioned in Section 3.1.1, on the 23rd of

May 2022 the first long-term support (LTS) version of ROS2, called Humble Hawksbill, is released. Humble Hawksbill will have support until May 2027. Additionally, Humble Hawksbill will have Ubuntu 22.04 as its first tier operating system. Ubuntu 22.04 is also an LTS version with support until 2027. The Unity editor used for creating MAES is the LTS version 2021.3.2f1 with support until at least mid 2023. If we can upgrade the ROS workspace to Humble Hawksbill, we thus significantly expand the possible lifetime of MAES given the current code base. For this reason we consider upgrading to Humble Hawksbill the most important future work.

In addition, we also want to create a community around the GitHub code repository. We already know, that a few groups at Aalborg University will continue work on MAES in 2022, and as such we will leave them a bit of guidance on the GitHub page. This includes community rules for merge requests, ideas for future features, as well as a list of known issues in the current version 2.0, which all in all should allow contributors to jump right in and help. MAES will be released under the GPL-3 license, which allows everyone to copy, distribute and modify the code as long as they also release it under the same license. Some of the future work features, which we did not have time to implement, include headless runs in ROSMode, position inaccuracies when running in ROSMode, as well as a small bug with the UI where scrolling on the TCP connector visualisation also zooms in/out in the simulation. We hope that implementing these features will allow the contributors to get a better understanding of the code, and thus make it easier to implement new ideas and features.

Furthermore, we want to support ROS map merging between robots to allow for the robots to co-operate in mapping an environment. However, as mentioned in Section 2, the current version of slam_toolbox does not support this, and as such this feature would have to wait for a newer version of slam_toolbox or be implemented manually.

Another interesting feature for the future is that of multiple robot types. Currently MAES only supports a single robot type. This would allow for simulating and testing system with multiple robots of different types.

Unity, which is used to construct MAES, has built-in systems to support safe, high performing multi-threaded code and very efficient memory management called Data-Oriented Technology Stack (DOTS) (Unity-Technologies, 2022c). Upgrading MAES with Unity DOTS would likely increase MAES' performance greatly, but we predict it will be a significant feat to accomplish as it would require redesigning and

rewriting a lot of MAES' internal workings. However, this would likely only notably affect algorithms running in UnityMode, as the ROS system appears to be the main bottleneck when in ROSMode.

Making performance optimizations would also warrant a more systemized performance test. As mentioned in Section 4, we have supplied a script for logging data and events, which could be expanded with event triggering automation (or a manuscript) to make sure certain use cases have their performance tested in a consistent manner.

9 ACKNOWLEDGMENTS

We thank Simon Bøgh and Casper Schou from the Robotics and Automation department at AAU for their support in the development of the ROS 2 interfaces for MAES. Additionally, we want to thank all of who participated in the usability test.

REFERENCES

- Andreasen, M., Holler, P., Jensen, M., and Albano, M. (2022a). Comparison of online exploration and coverage algorithms in continuous space. In *Proceedings of the 14th International Conference on Agents and Artificial Intelligence - Volume 1: SDMIS*, pages 527–537. INSTICC, SciTePress.
- Andreasen, M. Z., Jensen, M. K., and Holler, P. I. (2022b). Maes. <https://github.com/MalteZA/MAES>.
- ARGoS (2022). Argos - large-scale robot simulations. <https://www.argos-sim.info>.
- Colares, R. G. and Chaimowicz, L. (2016). The next frontier: Combining information gain and distance cost for decentralized multi-robot exploration. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC '16*, page 268–274, New York, NY, USA. Association for Computing Machinery.
- Fu, J. G. M., Bandyopadhyay, T., and Ang, M. H. (2009). Local voronoi decomposition for multi-agent task allocation. In *2009 IEEE International Conference on Robotics and Automation*, pages 1935–1940.
- Gautam, A., Richhariya, A., Shekhawat, V. S., and Mohan, S. (2018). Experimental evaluation of multi-robot online terrain coverage approach. In *2018 IEEE International Conference on Robotics and Biomimetics (RO-BIO)*, pages 1183–1189.
- Kegeleirs, M., Garzón Ramos, D., and Birattari, M. (2019). Random walk exploration for swarm mapping. In Althoefer, K., Konstantinova, J., and Zhang, K., editors, *Towards Autonomous Robotic Systems*, pages 211–222, Cham. Springer International Publishing.
- Macenski, S. and Jambrecic, I. (2021). Slam toolbox: Slam for the dynamic world. *Journal of Open Source Software*, 6(61):2783.

Macenski, S., Martín, F., White, R., and Ginés Clavero, J. (2020). The marathon 2: A navigation system. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

MAES (2022a). aaumaes/ros4maes. <https://hub.docker.com/r/aaumaes/ros4maes>.

MAES (2022b). Maes article raw data file. https://drive.google.com/drive/folders/1a_YjI1w0TLiakSW6m0-3XuDf7yt0vZfQ.

NVIDIA (2022). Nvidia isaac sim. <https://developer.nvidia.com/isaac-sim>.

Open Robotics (2022a). About ros 2 interfaces. <https://docs.ros.org/en/galactic/Concepts/About-ROS-Interfaces.html>.

Open Robotics (2022b). About tf2. <https://docs.ros.org/en/galactic/Concepts/About-Tf2.html>.

Open Robotics (2022c). Configuring your ros 2 environment. <https://docs.ros.org/en/galactic/Tutorials/Configuring-ROS2-Environment.html>.

Open Robotics (2022d). Creating your first ros 2 package. <https://docs.ros.org/en/galactic/Tutorials/Creating-Your-First-ROS2-Package.html>.

Open Robotics (2022e). Gazebo - simulate before you build. <https://gazebo.sim.org/home>.

Open Robotics (2022f). Introducing ros 2 launch. <https://docs.ros.org/en/galactic/Tutorials/Launch/CLI-Intro.html>.

Open Robotics (2022g). Powering the world's robots. <https://www.openrobotics.org>.

Open Robotics (2022h). Ros - robot operating system. <https://www.ros.org>.

Open Robotics (2022i). Ros - robot operating system. <https://docs.ros.org/en/galactic/Releases.html>.

Open Robotics (2022j). Setting up sensors. https://navigation.ros.org/setup_guides/sensors/setup_sensors.html#costmap-2d.

Open Robotics (2022k). Setting up transformations. https://navigation.ros.org/setup_guides/transformation/setup_transforms.html.

Open Robotics (2022l). Understanding ros 2 actions. <https://docs.ros.org/en/galactic/Tutorials/Understanding-ROS2-Actions.html>.

Open Robotics (2022m). Understanding ros 2 parameters. <https://docs.ros.org/en/galactic/Tutorials/Parameters/Understanding-ROS2-Parameters.html>.

Open Robotics (2022n). Understanding ros 2 services. <https://docs.ros.org/en/galactic/Tutorials/Services/Understanding-ROS2-Services.html>.

Open Robotics (2022o). Understanding ros2 nodes. <https://docs.ros.org/en/galactic/Tutorials/Understanding-ROS2-Nodes.html>.

Open Robotics (2022p). Understanding ros2 topics. <https://docs.ros.org/en/galactic/Tutorials/Topics/Understanding-ROS2-Topics.html>.

Player/Stage (2022). The player project. <http://playerstage.sourceforge.net>.

robo-friends (2022). m-explore ros2 port. <https://github.com/robo-friends/m-explore-ros2#multirobot-map-merge>.

ros-perception (2022). slam_gmapping. https://github.com/ros-perception/slam_gmapping.

Unity-Technologies (2022a). Ros tcp connector. <https://github.com/Unity-Technologies/ROS-TCP-Connector>.

Unity-Technologies (2022b). Ros tcp endpoint. <https://github.com/Unity-Technologies/ROS-TCP-Endpoint>.

Unity-Technologies (2022c). Unity dots. <https://unity.com/dots/packages>.

APPENDIX

A Links to code repository

<https://github.com/MalteZA/MAES>

B Usability Test

Goal: Can MAES with ROS be set up and used by the target audience? (Robotics developers and Researchers)

Duration: At most 60 minutes

Allowed aid: Readme from Github

B.1 Interviewer Guidelines

1. Inform the subject that their names will be anonymized, but we will record their voices and the screen.
2. Remember to start screen recording
3. Record audio with a phone as backup
4. Do not interrupt, unless the fault is clearly in MAES / ROS / our docker-image and not the subject.
5. If any phase is not finished within 20 min, the interviewer steps in and finishes the phase to allow for the next phase to start
6. If any phase is completed incorrectly and then next depends on it completed correctly, the interviewer can correct the error.
7. Do not elaborate on the guide, but it is allowed to explain terms (We are not testing their english skills)
8. In case of crash it is allowed to help get them to the point before the crash
9. SPEAK English!

B.2 The test

Initial Interview

1. Which semester are you on? (If working: What is your title at work)
2. What is your experience using Python?
3. What is your experience using ROS / ROS2?
4. What is your experience using Gazebo?
5. What is your current knowledge of MAES?

Phase 1 - Setup

Task 1: Build and start ROS2 with RVIZ running in Docker container

Task 2: Start MAES.x86_64 from MAES package

Task 3: Stop ROS2 and close MAES

Phase 2 - Configure

Task 1: Set number of robots to 1 (current is 2) using `maes_config_file` found in the `maes_ros2_interface` ROS package

Task 2: Change map type from building to cave using `maes_config_file` (and remember to 'colcon build' after changes to make them take effect)

Task 3: Launch first ROS and then MAES and confirm new number of robots and map type from the GUI. Close ROS and MAES when done.

Phase 3 - Use

Task 1: Open `maes_robot_controller.py` found in the `maes_robot_controller` ros package. Change the behaviour to make the robot move to coordinate (0, 0). Colcon build and start first ROS and then MAES. Click play in MAES and click the robot to follow and show debug information for the robot. Close ROS and MAES.

Task 2: Now change behaviour so that the robot cancels its navigation to (0, 0) if the tick is higher than 30. Colcon build and start first ROS and then MAES. Confirm behaviour, and then close ROS and MAES.

Task 3: Now change behavior to also make the robot deposit an environment tag with a message saying "Hello world!" continuously. Colcon build, open ROS and then MAES and confirm behaviour. Click "Visualize Tags" under "All Robots" in the side menu to enable visualization of tags.

Task 4: Click the topic menu in top left corner of MAES and click the `/robot0/maes_state` topic to see what is published on this topic. Now close ROS and MAES.

Closing Interview

1. What are your thoughts on the ROS integration with MAES?
2. Do you have any further remarks, comments or feedback?

C ROS2 System Overview

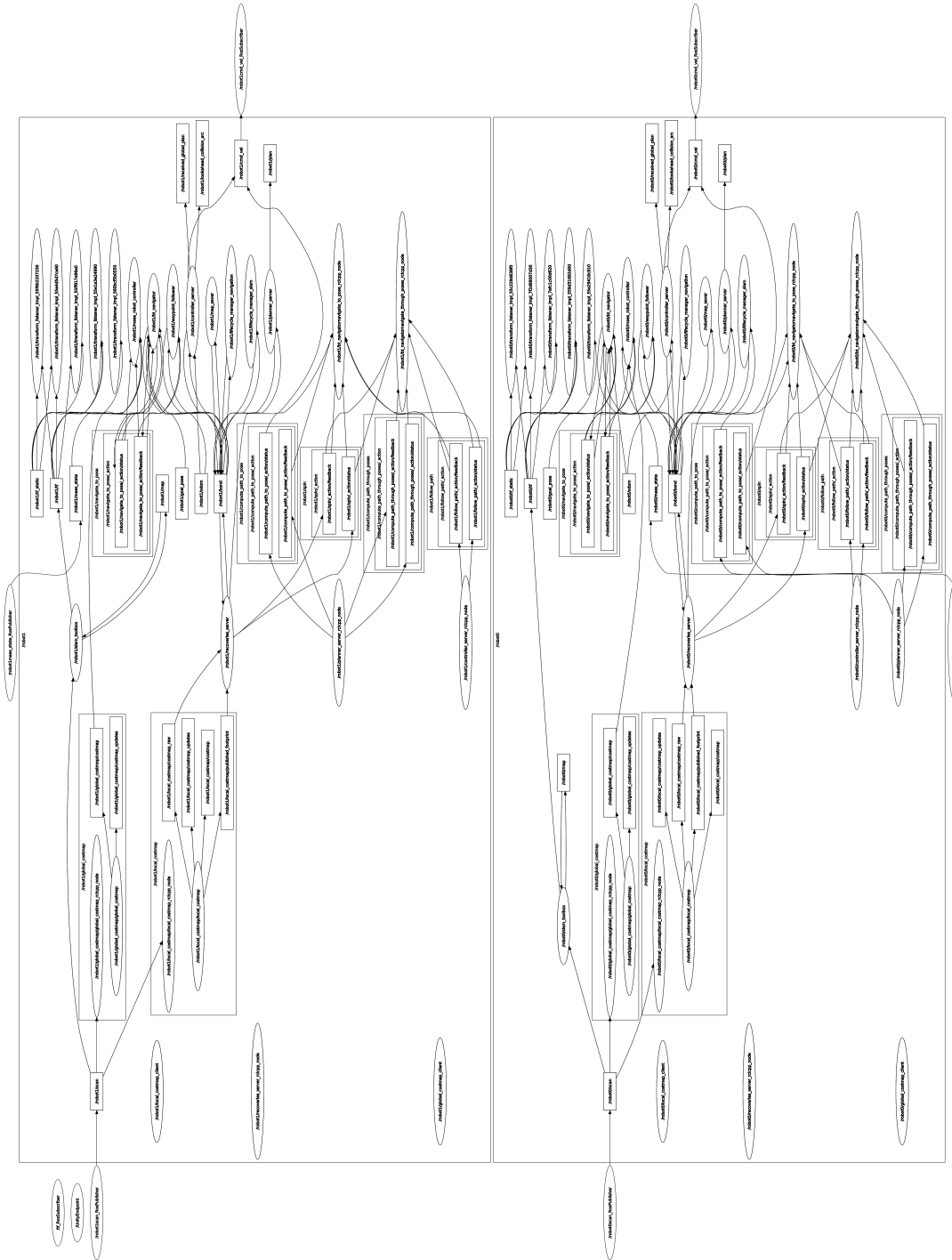


Figure 14: Overview of R OS2 system with two robots launched with `maes ros2_multi_robot.launch.py`. The graph is generated using the `rqt_graph` tool. Showing only Nodes/Topics (active). Dead sinks, leaf topics, debug, tf, unreachable and parameters are hidden to make it more readable. Oval shapes are nodes and squares are topics.

D Usability Test Results & Interviews

Table 2: Time used for each task for subject 1 (S1) to 5 measured in seconds

		S1	S2	S3	S4	S5
Phase 1 Set up	Task 1	349	311	333	284	295
	Task 2	137	72	64	42	190
	Task 3	107	25	56	18	58
Phase 2 Config	Task 1	17	114	103	42	60
	Task 2	95	96	120 ¹	68	128
	Task 3	84 ²	72	65	120	120
Phase 3 Use	Task 1	287	943 ³	1031 ³	601 ³	633 ⁴
	Task 2	119	314 ⁵	312	111	421
	Task 3	136	148	132	254	222
	Task 4	0 ⁶	61	15	141 ⁷	77 ⁸
Total		1331	2156	2231	1681	2204

D.1 Subject 1 Answers

Title: 10th semester student at Robotics

Experience with Python: 1 year

Experience with ROS/ROS2: 4.5 years on and off (1 year for ROS2)

Experience with Gazebo: Used as main simulation for ROS/ROS2 systems

Knowledge of MAES: Heard two 5 minute presentations, but not looked into code

Feedback:

Seems nice with MAES and Rviz visualization. Liked the topic list visualization. Liked the documentation. Did not initially understand the notion of a 'tick'.

D.2 Subject 2 Answers

Title: 10th semester student at Robotics

Experience with Python: 1 year

Experience with ROS/ROS2: 5 years ROS1, 1 year

¹Did not finish. Changed the map type to custom map instead of cave map

²Used colcon build from wrong directory

³Initially added logic to main function instead of logic loop

⁴Misunderstood the asynchronous call to navigation and tried waiting for return value

⁵Did not finish. Was unable to make the robot cancel, since the subject kept sending navigation requests to the robot every tick. S2 did, however, explain the correct solution

⁶S1 already finished this during Phase 3 Task 1

⁷S4 already finished this earlier, but did not realize it

⁸S5 spent a long time looking for a system menu called topic, instead of a menu in MAES

ROS2

Experience with Gazebo: Used for 5 years along ROS1 and ROS2

Knowledge of MAES: Heard two 5 minute presentations, but not looked into code

Feedback:

Hard to evaluate from this short usability test. MAES seems much more user friendly than Gazebo to get going. Easy to set up, which is important to new ROS users. MAES' graphical interface is much prettier than Gazebo's, but also lacks some of the features of Gazebo. Was unsure of how to see the TF links of the robot in MAES. Subject is unsure of how to evaluate the physical dynamics of the robot in MAES, i.e. the movement of individual parts. A good stepping stone for learning ROS. Suggests a grid to show the size of 1 unit in the coordinate system as well as a measurement system. It is hard to see, "what is a meter?"

D.3 Subject 3 Answers

Title: 10th semester student at Robotics

Experience with Python: 5-6 years

Experience with ROS/ROS2: 3 years ROS1, 5 days ROS2

Experience with Gazebo: Single course at University and two weeks of debugging

Knowledge of MAES: Heard one 5 minute presentation, but not looked into code

Feedback:

Seems cool. Nice that it works with RVIZ. It was unclear where to put navigation logic in the controller. Seems to be built well.

D.4 Subject 4 Answers

Title: Assistant Professor

Experience with Python: Not a lot

Experience with ROS/ROS2: A lot with ROS1, but none with ROS2

Experience with Gazebo: Minimal to none

Knowledge of MAES: Heard two 5 minute presentations, but not looked into code

Feedback:

It's great. It seems like almost a direct replacement for Gazebo. Although Gazebo is built for any kind of robot (MAES is only for mobile robots), Gazebo is also significantly more difficult to set up, configure and use. MAES seems tailored for mobile robots. MAES is a lot easier to set up, especially with Docker. Nice integration with RVIZ. MAES interface

is easier to comprehend than RVIZ. Suggestion:
Spawn each robot at a predefined position.

D.5 Subject 5 Answers

Title: 6th semester student at Robotics

Experience with Python: 1 year

Experience with ROS/ROS2: ROS1 6 months, ROS2
4 months. Prefers ROS2

Experience with Gazebo: Almost a year

Knowledge of MAES: Heard two 5 minute presenta-
tions, but not looked into code

Feedback:

Looks very promising. Likes the UI and 3D render-
ings of MAES. Unsure of what a 'tick' is. It was not
clear why one could not click a robot before running
the simulation. Suggests to make it more clear that
the topic menu is in fact a menu. Suggests a UI
element with a list of all robots, where a given robot
could be selected more easily.