Exploring the Possibilities of using micro-ROS and ROS2 for Underwater and Surface Vehicles

REPORT

By

Manan Gupta

2017A3TS0488H

ABSTRACT

Exploring the Possibilities of using micro-ROS and ROS2 for Underwater and Surface Vehicles

Autonomous Underwater Vehicles (AUVs) technology has seen major advancements in recent years. Modularity at the level of the software and hardware has paved the way for rapid development. STARFISH is one such AUV developed at the Acoustic Research Laboratory (ARL) at National University of Singapore (NUS). STARFISH currently deploys software components based on the DSAAV architecture. This, however, requires costly maintenance of the software, distracting us from robotics research. The different subsystems/agents on the AUVs have their own microcontroller units (MCUs) to handle their tasks. This promotes modularity at the hardware level.

This project work proposes to explore the possibilities of using ROS2 (Robot Operating System 2) and micro-ROS to be deployed on MCUs in AUVs. ROS2 and micro-ROS offer the advantages that were previously offered by DSAAV in terms of maintaining hardware modularity and interface flexibility, while also solving the problem posed since it is open-source and offers support for a wide variety of applications. micro-ROS derives from ROS2 and offers support for using packages on MCUs as it runs on Real-Time Operating System (RTOS), in contrast to ROS2 which runs on Linux-based operating systems hence requiring extensive hardware utilization. This helps bring the advantage of ROS2 to MCUs and we can interface sensors and actuators (sentuators) with them using the same. These software tools also act as the middleware in communication and use the publish/subscribe model, promoting loosely-coupled systems. micro-ROS also introduces Data Distribution Service for eXtremely Resource Constrained Environments (DDS-XRCE) as compared to the classical Data Distribution Service (DDS) used by ROS2.

TABLE OF CONTENTS

INTRODUCTION

Autonomous Underwater Vehicles (AUVs) have seen major technological advancements in recent years at all levels - software, electronics and mechanical allowing users to configure their AUVs based on mission requirements and use cases. There is the freedom to make choices in terms of software tools, electronics components and mechanical parts. Collaborative or swarm missions of multiple AUVs can also be configured. These configurations are possible due to modularity in AUVs at all the levels. Such modularity makes it easier to maintain and update AUVs. Modularity also offers a lot of other benefits such as allowing users to change modules and configure AUVs for specific mission applications by working with the necessary components. Using different sensor modules and configuring a team of heterogeneous AUVs can allow users to work with a swarm of vehicles for collaborative missions. Modular systems also provide an added advantage of independent subsystems, meaning if a non-mission-critical module fails, the vehicle can still continue to operate, providing easier maintenance than monolithic AUVs. Despite these benefits, classical modular components have some disadvantages. There can be issues related to the system compatibility and the need for a robust communication network for the different modules. Also, most of the software technologies used in AUVs, however, tend to be complicated and costly to maintain. It is also proprietary in most cases.

In recent years, we are also seeing improvements in the fields of open software-hardware interfaces which are being widely put into use. These well-documented open software-hardware interfaces promote the free and easy usage of open-architectures and open-systems.

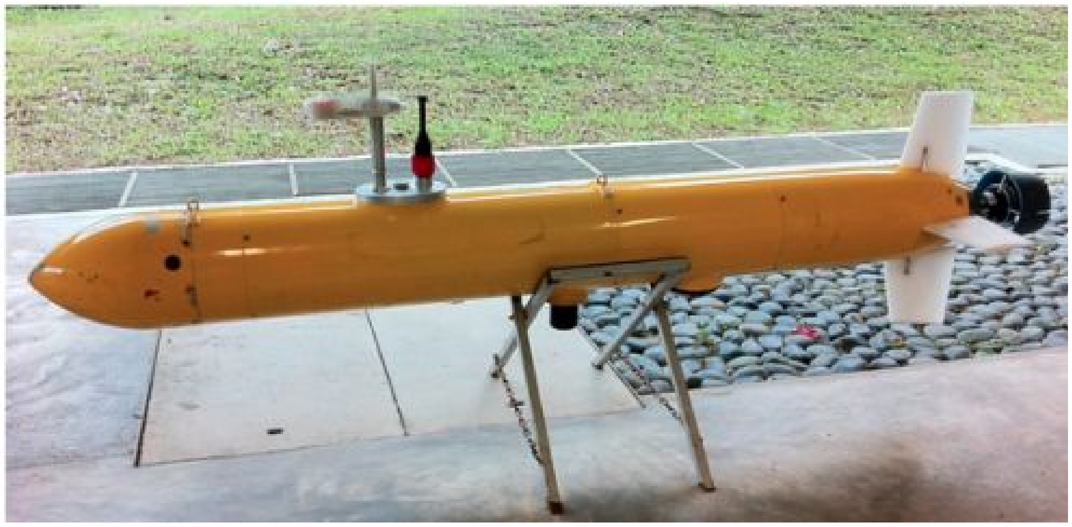


Figure 1: The STARFISH AUV
(Photo by Y. T. Tan et al. [4])

Currently, AUVs such as the STARFISH [1] developed at the Acoustic Research Laboratory (ARL) use the Distributed Software Architecture for Autonomous Vehicles (DSAAV) [2]. DSAAV was primarily designed for AUVs, but it can also be used for Unmanned Surface Vessels (USVs). Software architectures like the DSAAV usually require a huge maintenance cost associated with building modular AUVs.

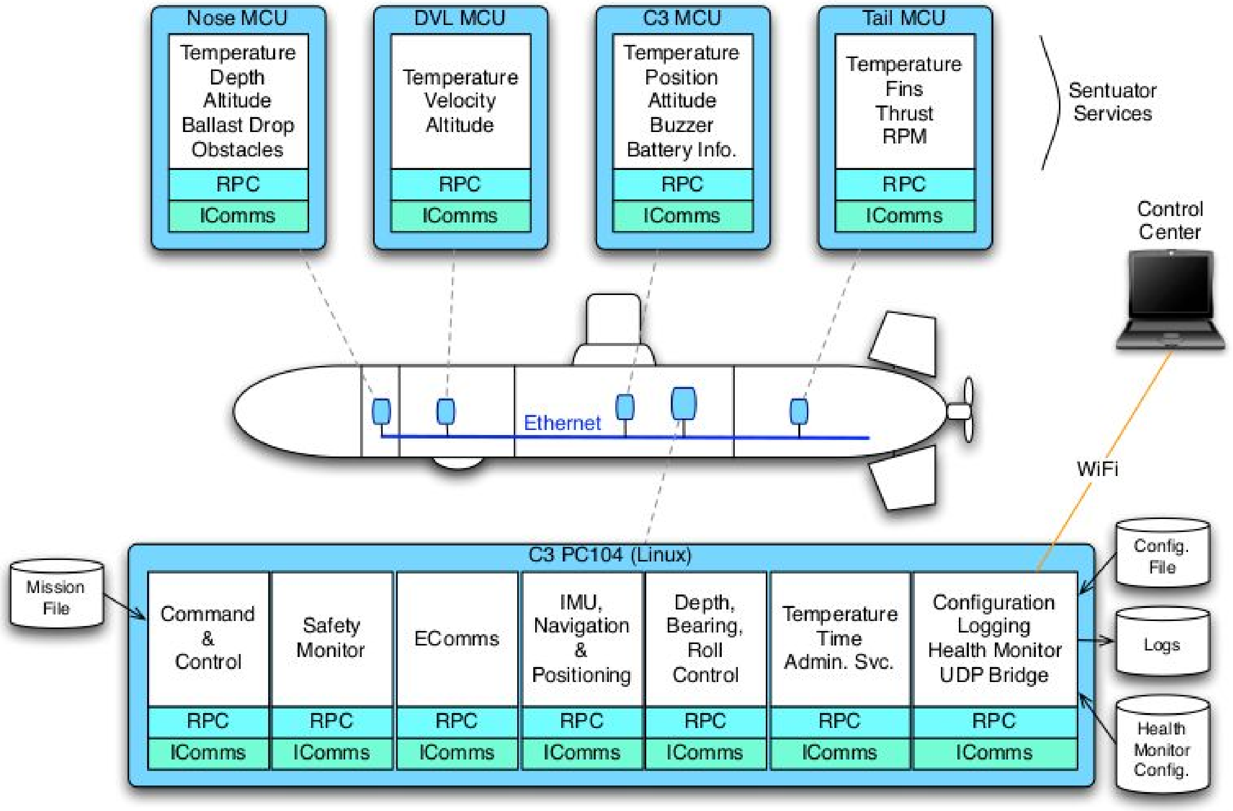


Figure 2: STARFISH - Typical Autonomous Underwater Vehicle Configuration
(Photo by M. Chitre [2])

For example, the Mission Orientated Operating Suite (MOOS) developed at MIT addresses these problems to some extent by the use of a central database with a well-defined communication protocol for client software components to access it [3]. MOOS adopts a “star” architecture with all components connecting to a central database in order to deposit or retrieve information. This can potentially lead to a single data bottleneck and a single point of failure. Such systems also pose other problems such as huge dependency and load on.

The DSAAV platform used in STARFISH adopts a peer-to-peer communication architecture, while also using a centralized configuration database and logging service. STARFISH also uses the Command and Control (C2) [4] which is developed keeping in mind the structure of an actually manned navy submarine. This provides AUVs with individual, self-contained agents that have their own set of tasks and responsibilities and are independent of each other's workings.

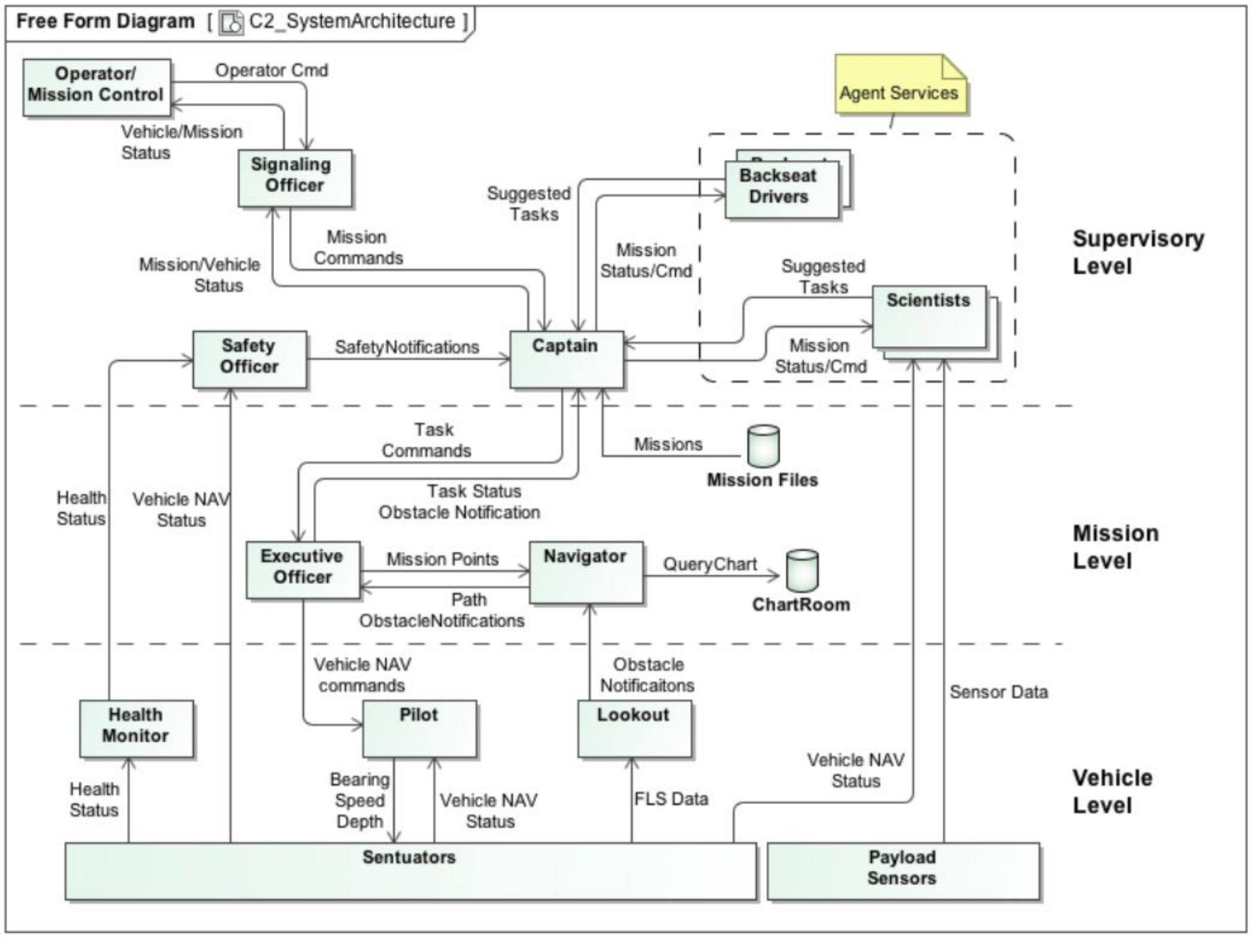


Figure 3: Overview of the Command and Control System
(Photo by Y. T. Tan et al. [4])

To introduce a workaround to the problems discussed up until now, a platform is needed that will not only offer the advantages discussed before but will also solve the problem of maintaining expensive software. Software packages should be designed such that they can run on MCUs in the different subsystems, and hence should not be very computationally extensive.

The Robot Operating System 2 (ROS2) provides a set of tools, libraries and software components for developing robots, including everything from drivers to state-of-the-art algorithms and with powerful development tools. ROS2 is an open-source platform, meaning it is maintained and updated by the community and hence benefits from continued development. ROS2 packages run on Linux-based operating systems and hence could not be run on microcontroller units (MCUs) until recently.

micro-ROS was introduced and is derived from ROS2, allowing the usage of ROS2 features on microcontrollers. In comparison to ROS2, micro-ROS can be used on microcontrollers majorly due to the following two changes:

1. micro-ROS runs on Real-Time Operating System (RTOS) instead of Linux as was the case with ROS2
2. micro-ROS utilizes Data Distribution Service for Xtremely Resource-Constrained Environments (DDS-XRCE) as compared to the classical Data Distribution Service (DDS) used by ROS2

This work presents the exploration of the possibility of using micro-ROS and ROS2 platforms to be deployed in the development of AUV systems. We explore the practicality of using such packages over the already existing software tools and present the results that are obtained.

<u>ROBOT OPERATING SYSTEM 2 (ROS2)</u>

ROS2 is a set of software tools, libraries and packages that can be used for developing robots. It is open-source software and is well maintained by the community, keeping it up-to-date with the latest technological advancements happening in the field. ROS2 uses the publisher-subscriber (Pub-Sub) model for communication between the different nodes.

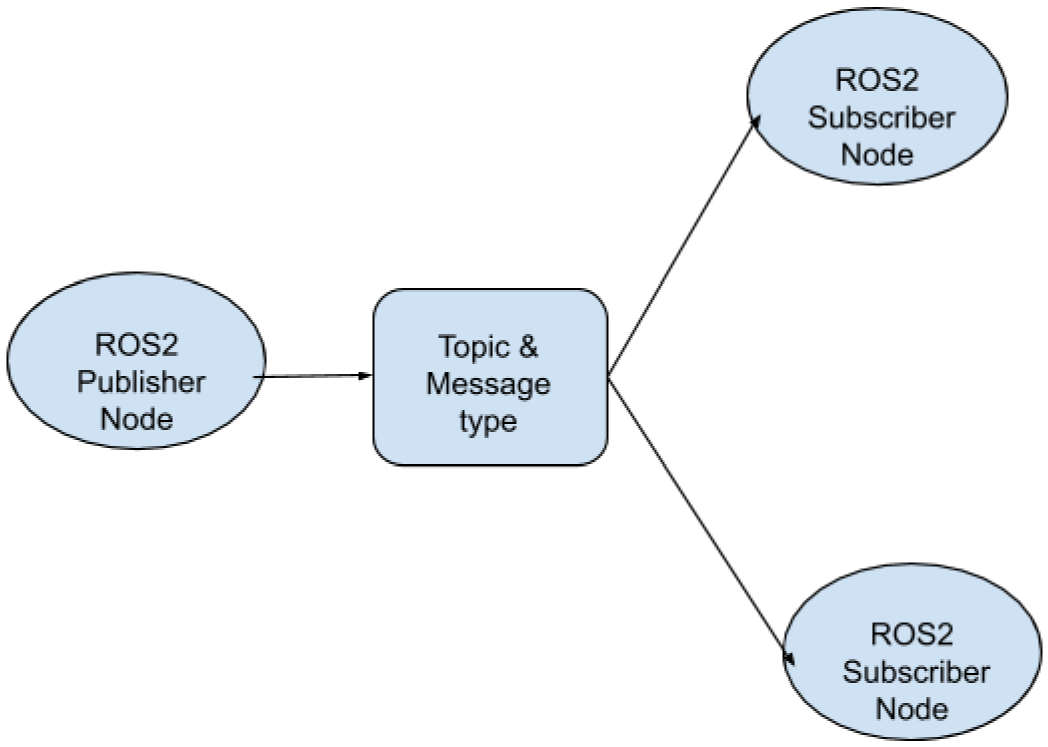


Figure 4: ROS2 Pub-Sub Model

Publish-subscribe is a protocol for communication between nodes using messages. Nodes sending messages are publishers, and those that receive any type of message is a subscriber. This is supported by the categorization of different types of data into topics. Nodes can publish and subscribe to certain topics and will be transmitting or receiving the messages under those topics. When a node subscribes to messages of a given topic, it just has direct communication with the ROS2 applications and doesn't need to directly connect to the publisher. This offers the flexibility of creating "loosely-coupled systems", in the sense that one node doesn't need to know of the existence of another. This also provides us with an added advantage of not having to bother too much about the file format and compatibility issues between different nodes.

micro-ROS

micro-ROS is derived from ROS2 and is used to bring the capabilities offered by ROS2, to microcontrollers. It does this by majorly bringing in changes necessary for usage in resource-constrained hardware platforms such as MCUs. MCUs offer lesser hardware capabilities as compared to a microprocessor board and also cannot run a full-fledged Operating System (OS) on them, which is a basic necessity for ROS2. Hence, micro-ROS runs on RTOS which is more suited for use on MCUs where the applications are mostly pre-defined and the tasks are happening in real-time. RTOSes are also lightweight as compared to the Linux-based OS that ROS2 requires.

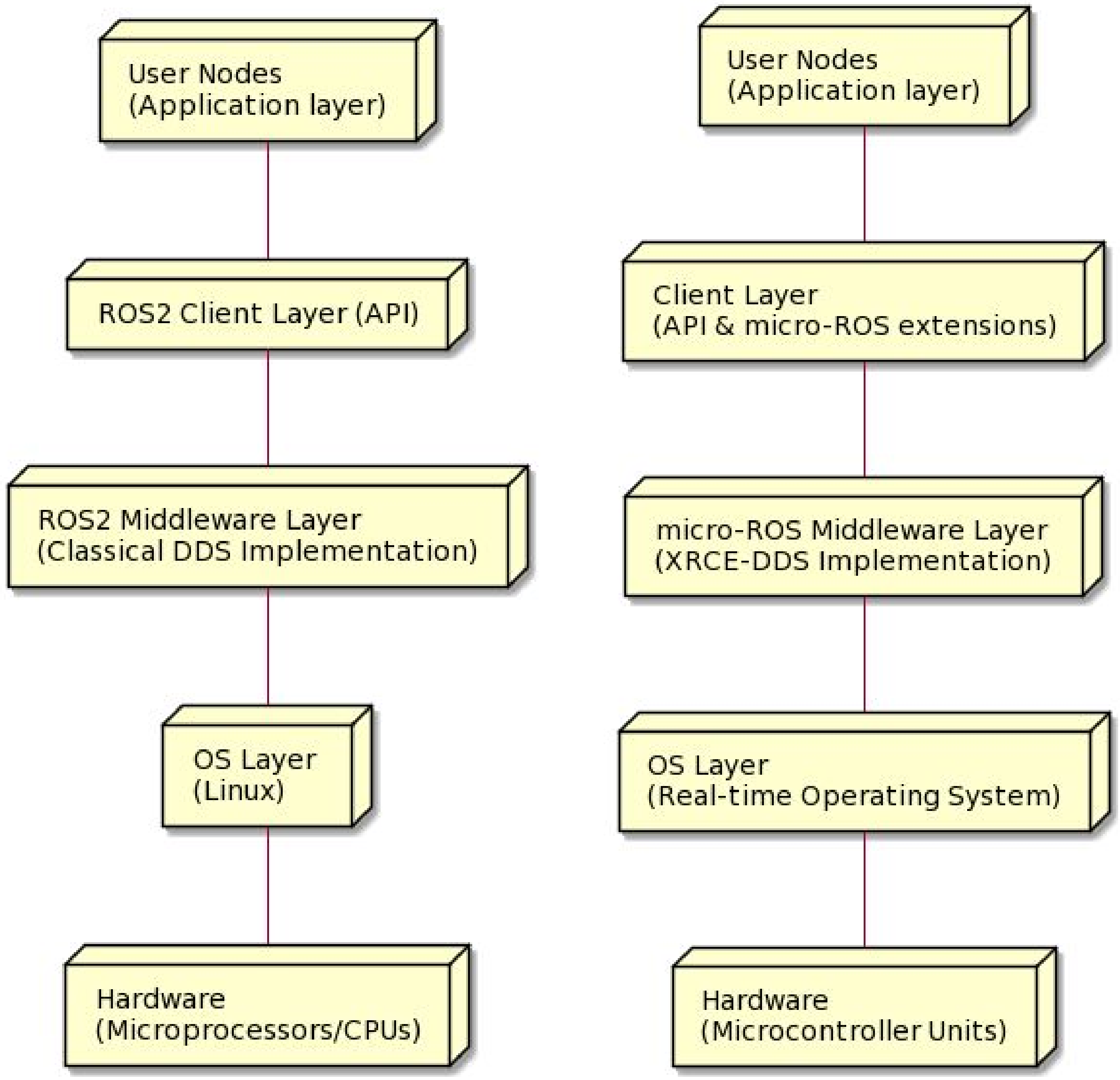
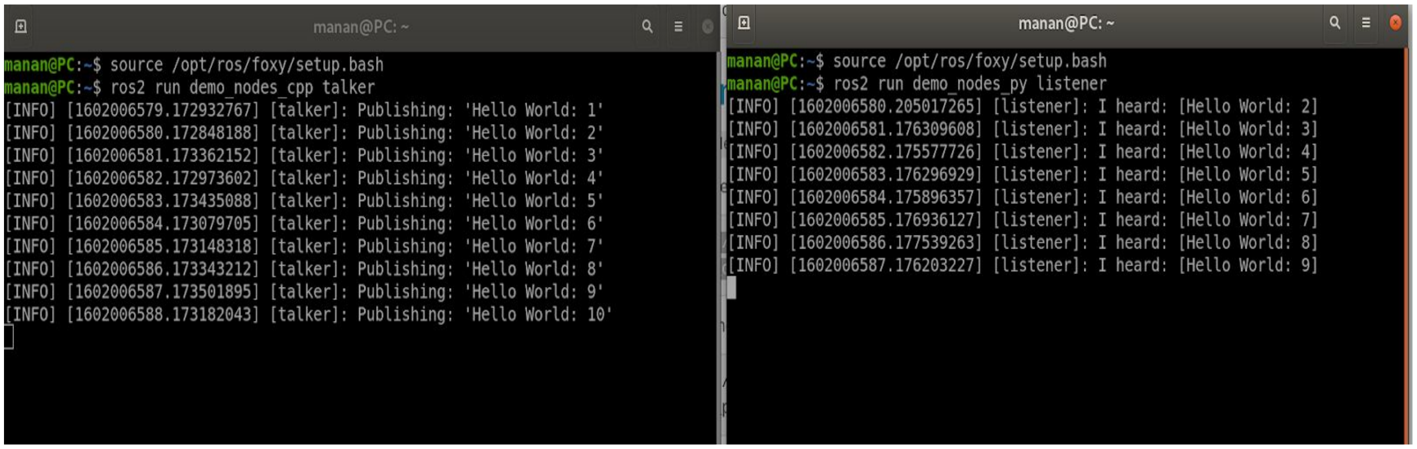


Figure 5: micro-ROS vs ROS2 system setup

To tackle the problem of the classical DDS that ROS2 used, micro-ROS introduces DDS for eXtremely Resource Constrained Environments (DDS-XRCE). The DDS-XRCE protocol stands to provide access to the DDS Global-Data-Space from resource-constrained devices. This is done by a client-server architecture. Low resource devices, as is the case with MCUs, are called XRCE Clients and are connected with a server, called an XRCE Agent. The XRCE Agent acts on behalf of its clients in the DDS Global-Data-Space.

EXPERIMENTING WITH ROS2

The first step that was carried out was the setting up of the ROS2 environment on a Linux-based computer system. For this, I performed the installation of ROS2 on a Ubuntu 20.04 LTS computer system, which the latest Ubuntu flavour of Linux. Once the setup was complete, a simple application was run to test the system. This is a simple talker-listener application, where there are two nodes, one publishing data in a given topic and the other receiving it by subscribing to that topic. The two nodes were run in independent terminals as shown below.



```
manan@PC:~$ source /opt/ros/foxy/setup.bash
manan@PC:~$ ros2 run demo_nodes_cpp talker
[INFO] [1602006579.172932767] [talker]: Publishing: 'Hello World: 1'
[INFO] [1602006580.172848188] [talker]: Publishing: 'Hello World: 2'
[INFO] [1602006581.173362152] [talker]: Publishing: 'Hello World: 3'
[INFO] [1602006582.172973602] [talker]: Publishing: 'Hello World: 4'
[INFO] [1602006583.173435088] [talker]: Publishing: 'Hello World: 5'
[INFO] [1602006584.173079705] [talker]: Publishing: 'Hello World: 6'
[INFO] [1602006585.173148318] [talker]: Publishing: 'Hello World: 7'
[INFO] [1602006586.173343212] [talker]: Publishing: 'Hello World: 8'
[INFO] [1602006587.173501895] [talker]: Publishing: 'Hello World: 9'
[INFO] [1602006588.173182043] [talker]: Publishing: 'Hello World: 10'
```

```
manan@PC:~$ source /opt/ros/foxy/setup.bash
manan@PC:~$ ros2 run demo_nodes_py listener
[INFO] [1602006580.205017265] [listener]: I heard: [Hello World: 2]
[INFO] [1602006581.176309608] [listener]: I heard: [Hello World: 3]
[INFO] [1602006582.175577726] [listener]: I heard: [Hello World: 4]
[INFO] [1602006583.176296929] [listener]: I heard: [Hello World: 5]
[INFO] [1602006584.175896357] [listener]: I heard: [Hello World: 6]
[INFO] [1602006585.176936127] [listener]: I heard: [Hello World: 7]
[INFO] [1602006586.177539263] [listener]: I heard: [Hello World: 8]
[INFO] [1602006587.176203227] [listener]: I heard: [Hello World: 9]
```

Figure 6: Talker-listener application running using ROS2

As we see in the image here, the node on the left is publishing data in the form of numbered “Hello World” messages. These are published every second. The node on the left is the listener. It has subscribed to the topic under which the “Hello World” messages are sent. As we see, the listener is able to read the specific messages as when they are published by the talker. Another observation made here is that the listener has started listening to the messages from the second “Hello World” message. This is because the listener node was run after the talker node had already published the first message. There is no buffer/FIFO storage here, and the data is only kept in the data stream for as long as it is published. This was done just for the initial tests and later on to store all messages of a topic, a simple buffer application can be created.

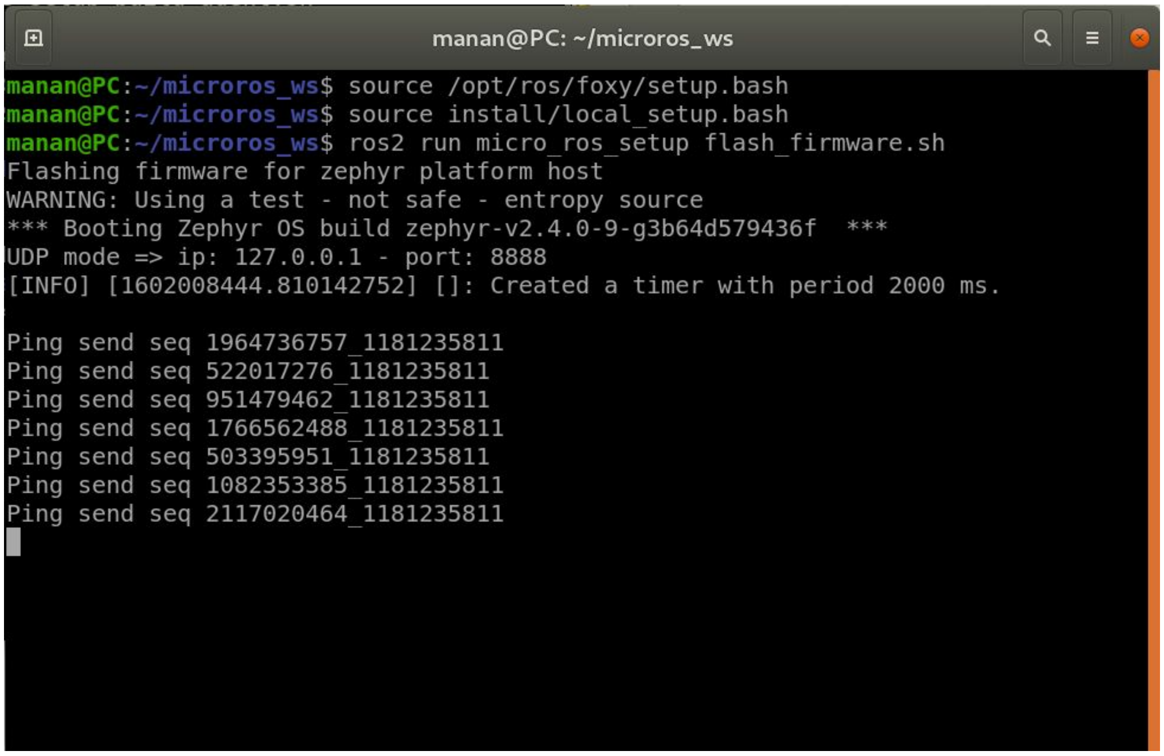
An important point to note here is that the two nodes above are written in different languages: the talker is written in C++, whereas the listener is written in Python. This proves a very important concept that two or more nodes that are seeing a transfer of data do not need to know of the file type settings or other specifications. They are linked using the ROS2 application and the different languages do not matter, given that the topic is the same. This proves that ROS2 can be used to create loosely coupled systems.

EXPERIMENTING WITH micro-ROS

The first step in working with and experimenting with micro-ROS was to set up the environment where micro-ROS applications can be run and tested. micro-ROS offers a setup environment for both Linux-based systems and RTOSes. At first, the setup was done for the Linux-based environment itself on the Ubuntu 20.04 LTS computer system. A simple ping-pong application was run to test the setup. This is similar to the talker-listener test run as explained earlier. The test was successful in transmitting and receiving "pings".

Since the aim is to experiment with the possibility of using micro-ROS on MCUs running an RTOS, the next step is to install an emulator on the computer system which can simulate the RTOS setup. For this, the Zephyr emulator was installed which provides the ability to test micro-ROS applications running on RTOS on a computer system itself without having the need to flash the software on an MCU for each test.

We then proceeded to test the emulator set up using the ping-pong test. To run the micro-ROS node inside the Zephyr RTOS emulator, we flash the firmware as follows:



```
manan@PC:~/microros_ws$ source /opt/ros/foxy/setup.bash
manan@PC:~/microros_ws$ source install/local_setup.bash
manan@PC:~/microros_ws$ ros2 run micro_ros_setup flash_firmware.sh
Flashing firmware for zephyr platform host
WARNING: Using a test - not safe - entropy source
*** Booting Zephyr OS build zephyr-v2.4.0-9-g3b64d579436f  ***
UDP mode => ip: 127.0.0.1 - port: 8888
[INFO] [1602008444.810142752] []: Created a timer with period 2000 ms.

Ping send seq 1964736757_1181235811
Ping send seq 522017276_1181235811
Ping send seq 951479462_1181235811
Ping send seq 1766562488_1181235811
Ping send seq 503395951_1181235811
Ping send seq 1082353385_1181235811
Ping send seq 2117020464_1181235811
```

Figure 7: Pings are being published here

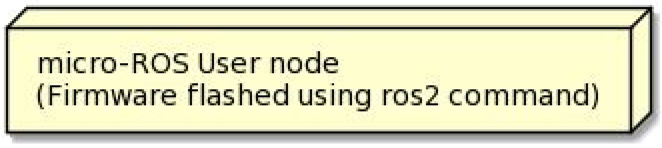
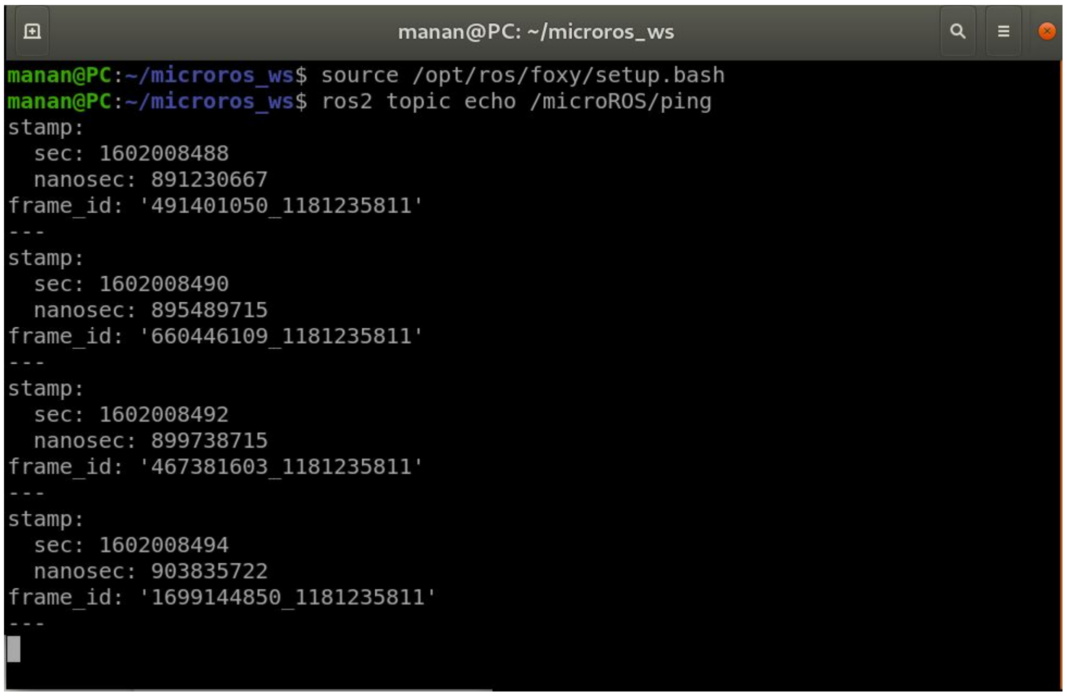


Figure 8: Ping-pong application running using micro-ROS

Once this is done, the node starts publishing “pings” at an interval of every 2 seconds. The next step is to listen to the “ping” topic that is being published. In another terminal, separate from the previous one, we subscribe to the “ping” topic using a ros2 command.



```
manan@PC:~/microros_ws$ source /opt/ros/foxy/setup.bash
manan@PC:~/microros_ws$ ros2 topic echo /microROS/ping
stamp:
  sec: 1602008488
  nanosec: 891230667
frame_id: '491401050_1181235811'
---
stamp:
  sec: 1602008490
  nanosec: 895489715
frame_id: '660446109_1181235811'
---
stamp:
  sec: 1602008492
  nanosec: 899738715
frame_id: '467381603_1181235811'
---
stamp:
  sec: 1602008494
  nanosec: 903835722
frame_id: '1699144850_1181235811'
---
```

Figure 9: This shows that pings published in the previous node are being read correctly

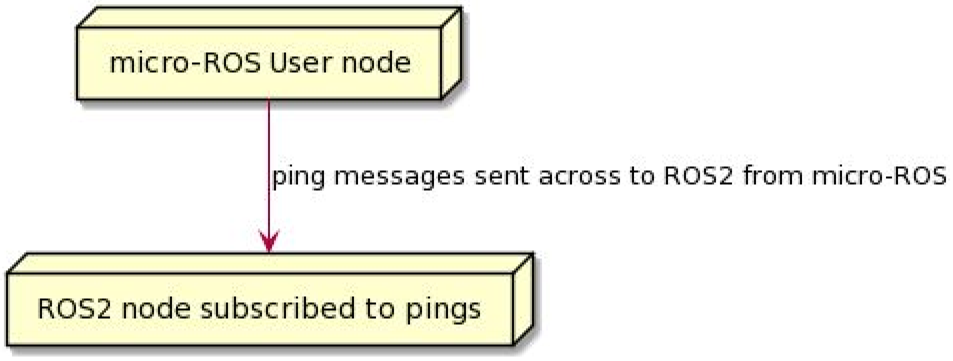
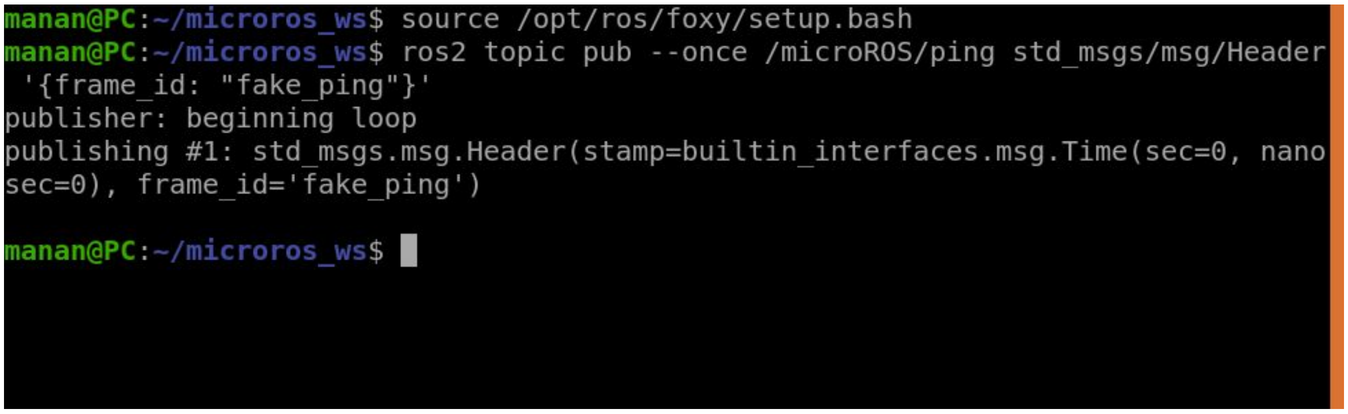

micro-ROS User node
ping messages sent across to ROS2 from micro-ROS
ROS2 node subscribed to pings

Figure 10: ROS2 used to subscribe to pings

As we see here, the pings that were published earlier are now being read by this node with the associated timestamp. To show another important feature, a "fake_ping" message is published using ROS2 under the ping topic from an altogether different terminal.

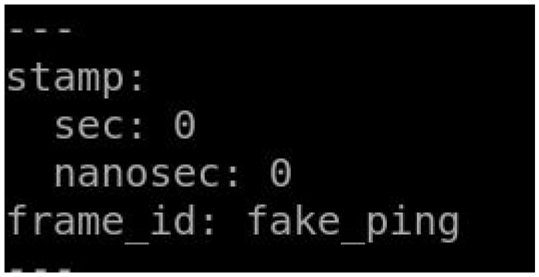


```
manan@PC:~/microros_ws$ source /opt/ros/foxy/setup.bash
manan@PC:~/microros_ws$ ros2 topic pub --once /microROS/ping std_msgs/msg/Header
 '{frame_id: "fake_ping"}'
publisher: beginning loop
publishing #1: std_msgs.msg.Header(stamp=builtin_interfaces.msg.Time(sec=0, nano
sec=0), frame_id='fake_ping')

manan@PC:~/microros_ws$ 
```

Figure 11: "fake_ping" message published

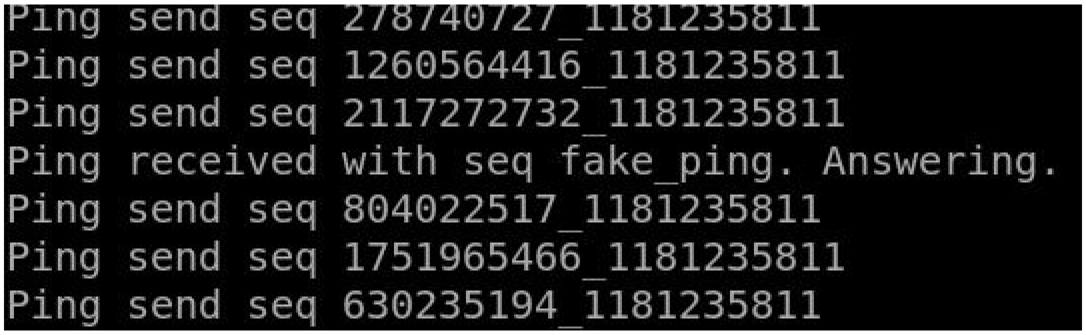
Now, since the second node has subscribed to the topic of "ping", it should also read this "fake_ping" message apart from the regular pings it is already receiving. This happens successfully as shown below:



```
---
stamp:
  sec: 0
  nanosec: 0
frame_id: fake_ping
```

Figure 12: fake_ping message received by the subscriber

The test of the "pong" topic has to be carried out. When the "fake_ping" message is published, the micro-ROS firmware that was first flashed also receives that, and answers with a "pong".



```
Ping send seq 278740727_1181235811
Ping send seq 1260564416_1181235811
Ping send seq 2117272732_1181235811
Ping received with seq fake_ping. Answering.
Ping send seq 804022517_1181235811
Ping send seq 1751965466_1181235811
Ping send seq 630235194_1181235811
```

Figure 13: First micro-ROS node receiving fake_ping and replying with a pong

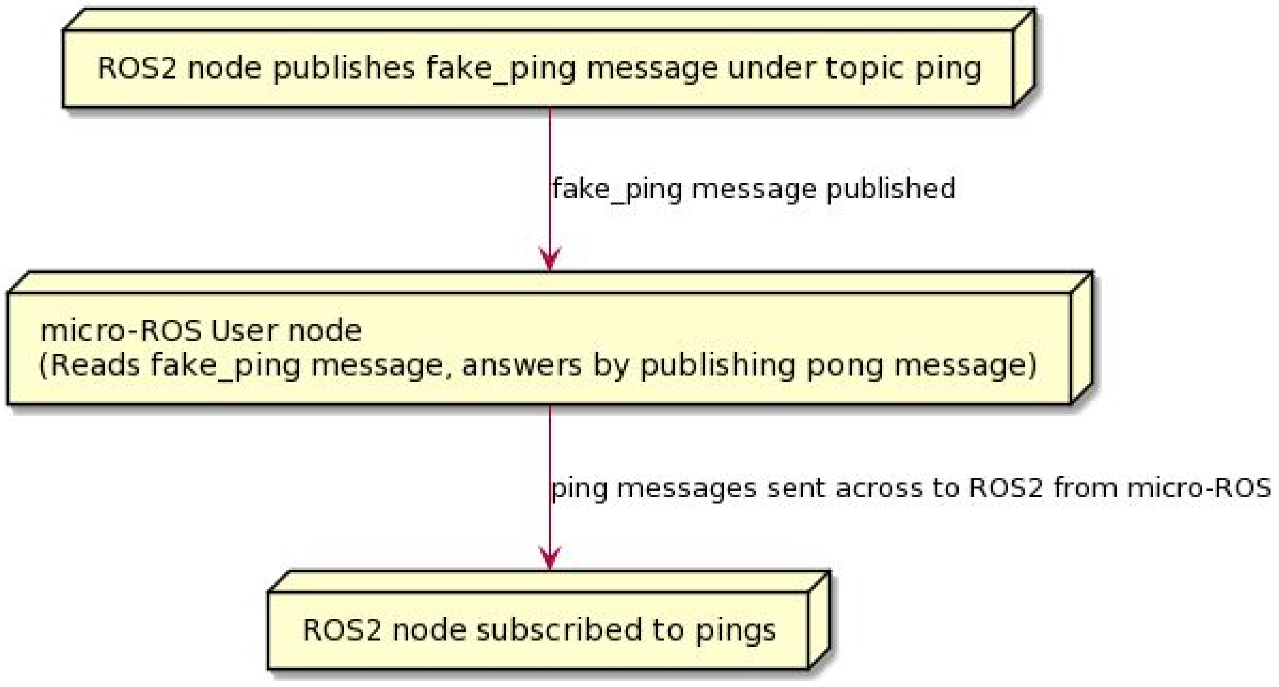
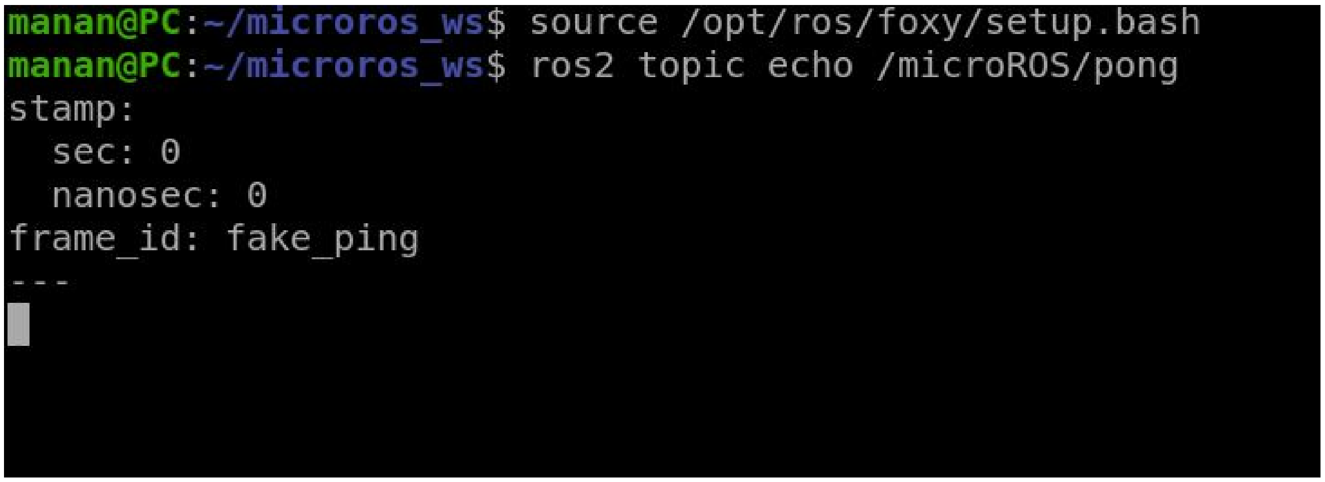

ROS2 node publishes fake_ping message under topic ping
fake_ping message published
micro-ROS User node
(Reads fake_ping message, answers by publishing pong message)
ping messages sent across to ROS2 from micro-ROS
ROS2 node subscribed to pings

Figure 14: fake_ping message published using ROS2

The node has now responded with a "pong". To be able to read this, in another terminal we subscribe to the topic "pong" using a ros2 command and should be able to receive the fake_ping message through that, which it does successfully.



```
manan@PC:~/microros_ws$ source /opt/ros/foxy/setup.bash
manan@PC:~/microros_ws$ ros2 topic echo /microROS/pong
stamp:
  sec: 0
  nanosec: 0
frame_id: fake_ping
---
```

Figure 15: Pong subscriber has received the message

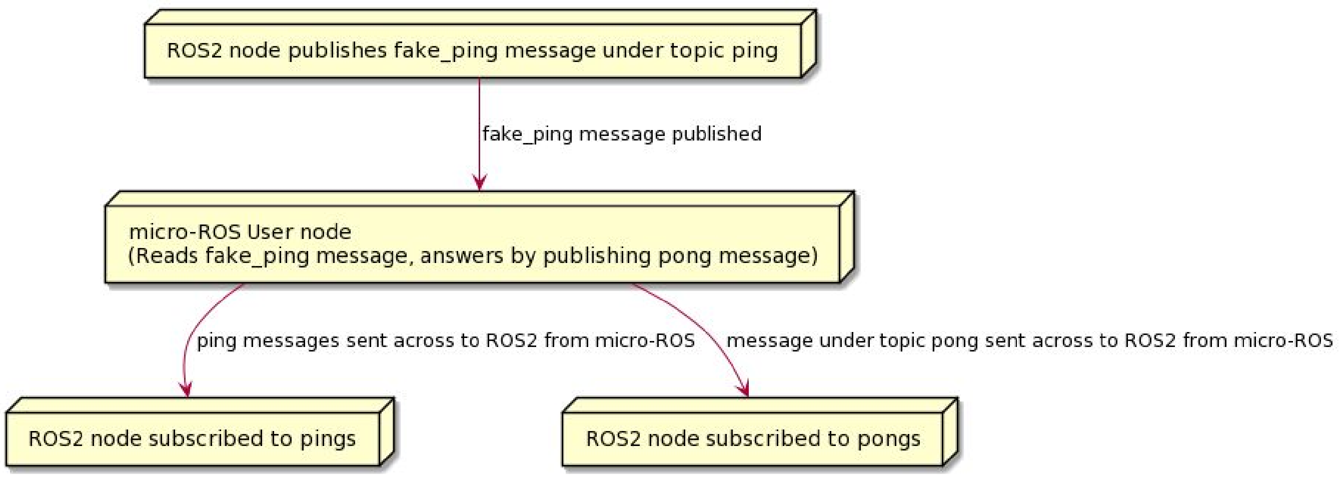

ROS2 node publishes fake_ping message under topic ping
fake_ping message published
micro-ROS User node
(Reads fake_ping message, answers by publishing pong message)
ping messages sent across to ROS2 from micro-ROS
message under topic pong sent across to ROS2 from micro-ROS
ROS2 node subscribed to pings
ROS2 node subscribed to pongs

Figure 16: pong message published by micro-ROS received by ROS2

CONCLUSION

The initial problems that were described can possibly be overcome by the use of micro-ROS and ROS2 in AUVs. The results that are shown above clearly show how ROS2 applications can run on Linux-based systems, such as in the case of microprocessors on the vehicle. The different nodes can transmit and receive data from each other without knowing about another's existence, paving the way for creating loosely-coupled systems and increased modularity at all levels.

The micro-ROS test that was run on the Zephyr RTOS emulator clearly shows how these applications can be run in resource-constrained environments such as the microcontrollers that are used in AUVs. The tests also showed how even micro-ROS nodes can show the same capabilities as expected from ROS2 and system modularity among the different subsystems can be maintained. This is because the micro-ROS firmware was being used to publish and subscribe to pings and pongs, and this was tested by using a few ROS2 commands successfully.

BIBLIOGRAPHY

[1] M. Sangekar, M. Chitre, and T. Koay, "Hardware architecture for a modular autonomous underwater vehicle STARFISH," in *OCEANS 2008 MTS/IEEE*, (Quebec City, Canada), September 2008

[2] Chitre, M. (2008). DSAAV - A Distributed Software Architecture for Autonomous Vehicles. *OCEANS 2008 MTS/IEEE*. DSAAV - A distributed software architecture for autonomous vehicles - IEEE Conference Publication

[3] Newman P.M., "MOOS - Mission Orientated Operating Suite," available at: MOOS : Main - Home Page browse. Accessed 16 July 2008

[4] Y. T. Tan and M. Chitre, "Hierarchical multi-agent command and control system for autonomous underwater vehicles," in *IEEE AUV 2012*, (Southampton, UK), September 2012