

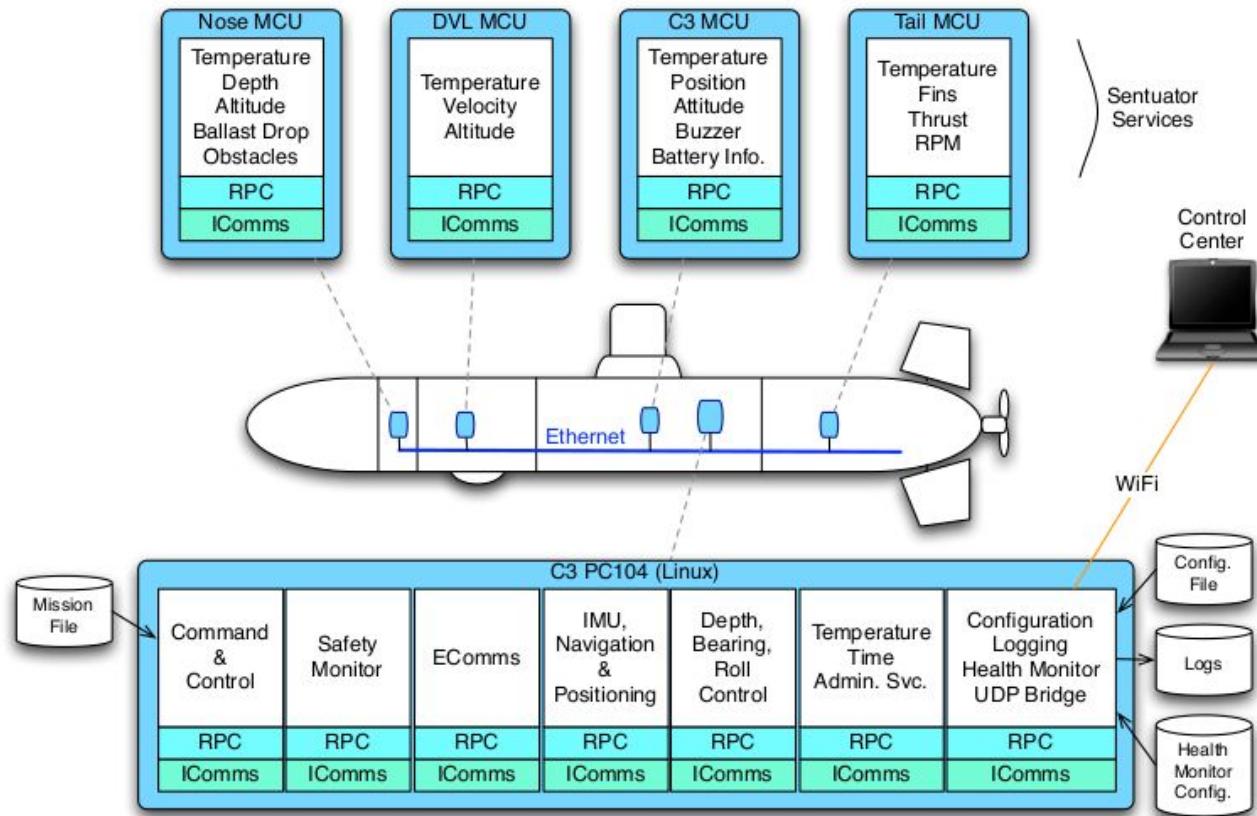
Project Presentation

Topic: Exploring the Possibilities of
using micro-ROS and ROS2 for
underwater and surface vehicles

By: Manan Gupta
2017A3TS0488H

Background

- Underwater Robots like the STARFISH developed at ARL use in-house developed frameworks like DSAAV and JC2
- These custom frameworks require a high maintenance cost
- Distract from robotics research



STARFISH - Typical Autonomous Underwater Vehicle Configuration

Background

- Different subsystems on AUVs deploy their own microcontrollers which offer limited computational power
- ROS and ROS2 too “heavy” to run directly on MCUs

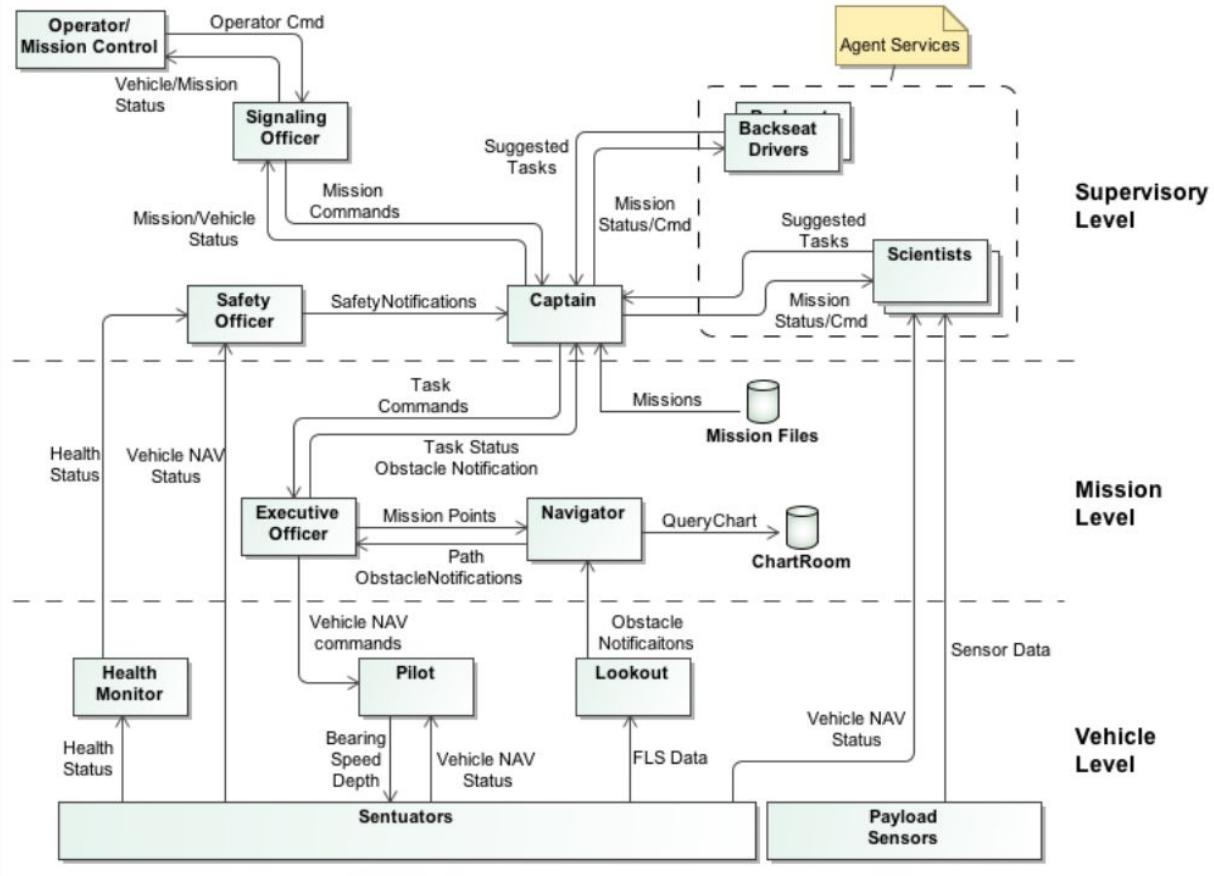


The STARFISH AUV

Background

- STARFISH - Hierarchical multi-agent command and control (C2) system for AUVs
 - Inspired by command structure of manned navy submarine
 - C2 allocates tasks to individual “self-contained agents”
 - Agents distributed over different levels of hierarchies
- Advantages:
 - Complex mission objectives divided into simpler sub-tasks handled by different agents
 - New mission scenarios can be handled by introducing new agents

Free Form Diagram [C2_SystemArchitecture]



Overview of the C2 system

Proposed Idea

- While keeping in mind the problems discussed earlier and the advantages preferred, we look at micro-ROS
- micro-ROS is derived from ROS2 (Robot Operating System 2) and can be deployed on microcontrollers
- We explore using micro-ROS applications deployed on MCUs on different subsystems/agents in the AUVs like STARFISH



ROS2

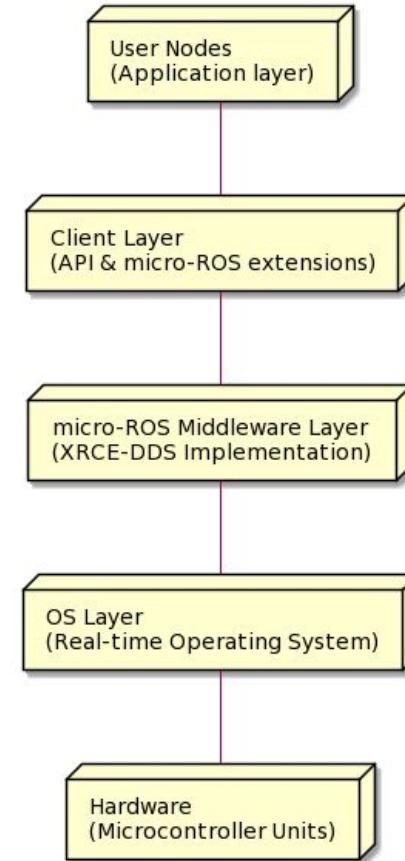
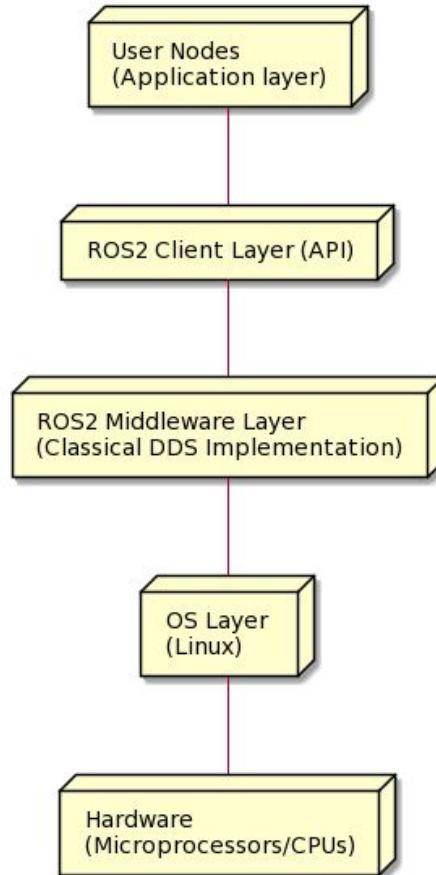
- Robot Operating System is a set of tools and libraries for building robot applications
- It is open-source and well maintained by the community
- Uses publish/subscribe model as the communication protocol
- Promotes “loosely-coupled” systems, implying hardware modularity and increases flexibility

micro -ROS

- micro-ROS derives from ROS2 and brings in the following major changes:
 - ROS2 runs on linux-based systems, whereas micro-ROS uses Real-Time Operating System (RTOS)
 - Instead of the classical DDS (Data Distribution Service), micro-ROS uses DDS for eXtremely Resource Constrained Environments (DDS-XRCE)

micro-ROS

- Explore using micro-ROS applications for the different agents in the AUVs and communicate with the central microprocessor which runs a ROS2 stack
- Allows increased modularity and flexibility
- One subsystem doesn't need to know of the existence of another, and doesn't depend on it directly
- Can make the system fault-tolerant; even if a subsystem fails, the vehicle can continue to function



ROS2 vs micro-ROS structure

Setting up ROS2 environment

- We setup the environment to test applications and the practicality of our idea
- micro-ROS requires ROS2 installation as a prerequisite
- ROS2 installed for Ubuntu 20.04 LTS computer system
- A basic code was run to check for the functioning of the ROS2 system setup
- It consisted of a “talker-listener” application
- Shows that different “nodes” are independent of each other

The image shows two terminal windows side-by-side. Both have a dark background and light-colored text. The left terminal window has a title bar 'manan@PC: ~'. It contains the following text:
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'

The right terminal window also has a title bar 'manan@PC: ~'. It contains the following text:
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]

The node running on the left is a “publisher”, written in C++. It is publishing “Hello World” messages

The node running on the right is a “subscriber”, written in Python. It listens to the “Hello World” messages

Note how the two nodes are written in different languages and still communicating, exhibiting loosely coupled systems

Setting up micro-ROS environment

- micro-ROS can be setup either on linux-based systems or RTOS-based systems.
- This provides freedom to work on computer systems without microcontrollers
- I first setup micro-ROS for Ubuntu 20.04 LTS computer
- We tested a simple “ping-pong” application run on this, similar to publish-subscribe

Setting up micro-ROS environment

- Our goal is to test applications on hardware-constrained microcontrollers using micro-ROS and hence we needed to test them on RTOS
- To do this, we proceeded with setting up the “Zephyr emulator” on the computer
- Zephyr is an open-source RTOS that can be setup on MCUs
- Zephyr also provides an emulator which can be setup on a computer itself to test applications on RTOS without having to actually work with an MCU everytime

Zephyr Emulator

- I setup the Zephyr emulator on my computer
- However, since I had already setup the environment meant for linux systems, there were some problems with the setup due to clashes during the setup
- To workaround this, a Docker container can be setup to run the environment independently of other applications on the system

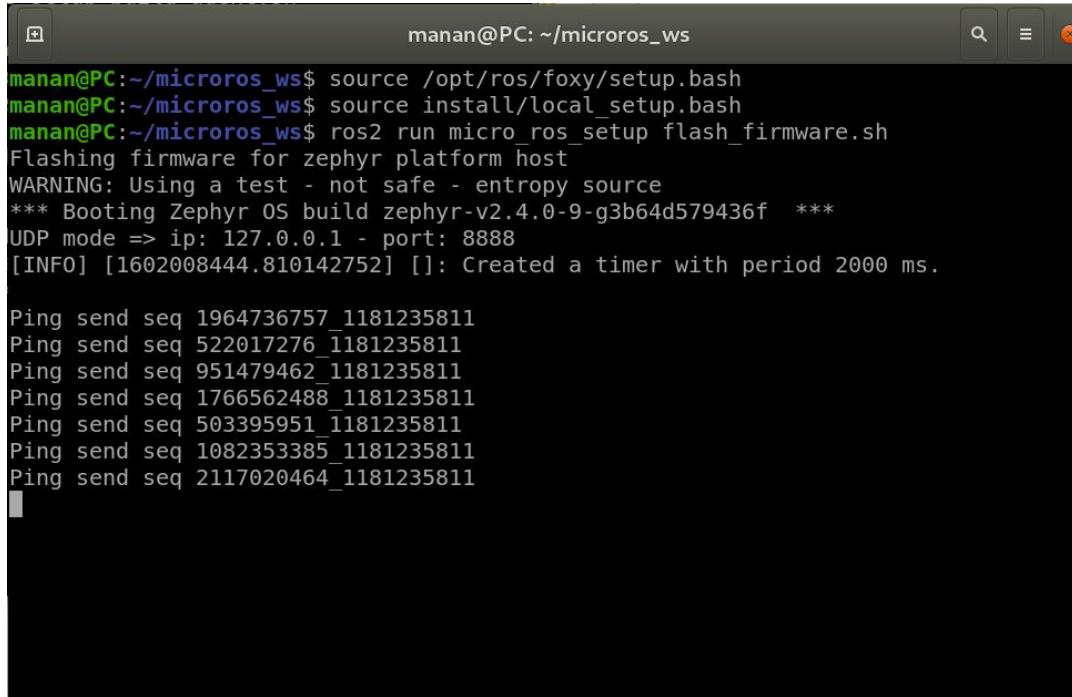
Ping-Pong Test

- After setting up the Zephyr emulator, a simple ping-pong application was again used to test the environment
- One node publishes “ping”, whereas the other reads them. It then publishes a “pong”.

First step:

micro-ROS User node
(Firmware flashed using ros2 command)

- To run micro-ROS node inside Zephyr RTOS emulator, we flash the firmware as follows:



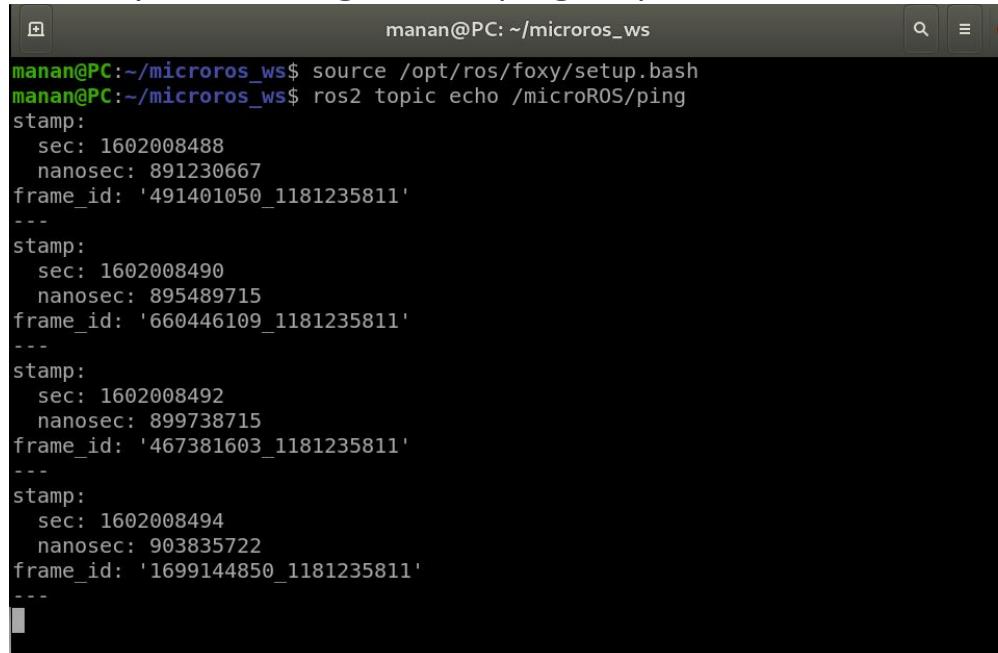
```
manan@PC: ~/microros_ws
manan@PC:~/microros_ws$ source /opt/ros/foxy/setup.bash
manan@PC:~/microros_ws$ source install/local_setup.bash
manan@PC:~/microros_ws$ ros2 run micro_ros_setup flash_firmware.sh
Flashing firmware for zephyr platform host
WARNING: Using a test - not safe - entropy source
*** 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
```

Pings are being published here

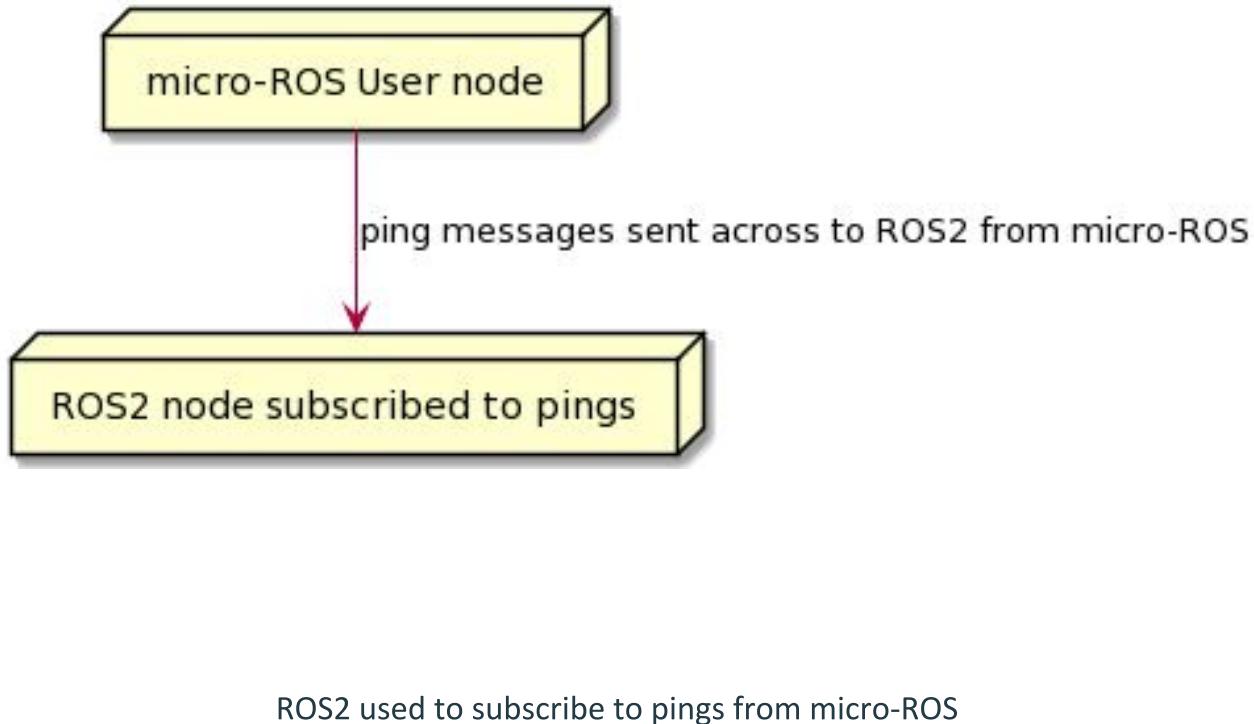
Second Step:

- Next step is to listen to the “ping” topic that is being published. In another terminal, we test this by subscribing to the “ping” topic

A screenshot of a terminal window titled "manan@PC: ~/microros_ws". The window shows the command "source /opt/ros/foxy/setup.bash" followed by "ros2 topic echo /microROS/ping". The output displays four messages from the "/microROS/ping" topic, each containing a timestamp and a frame ID.

```
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'
---
```

This shows that the pings are being published correctly in the previous step



Third step:

- We now publish a “fake_ping” message using ROS2. This also shows up in the ping subscriber

```
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')

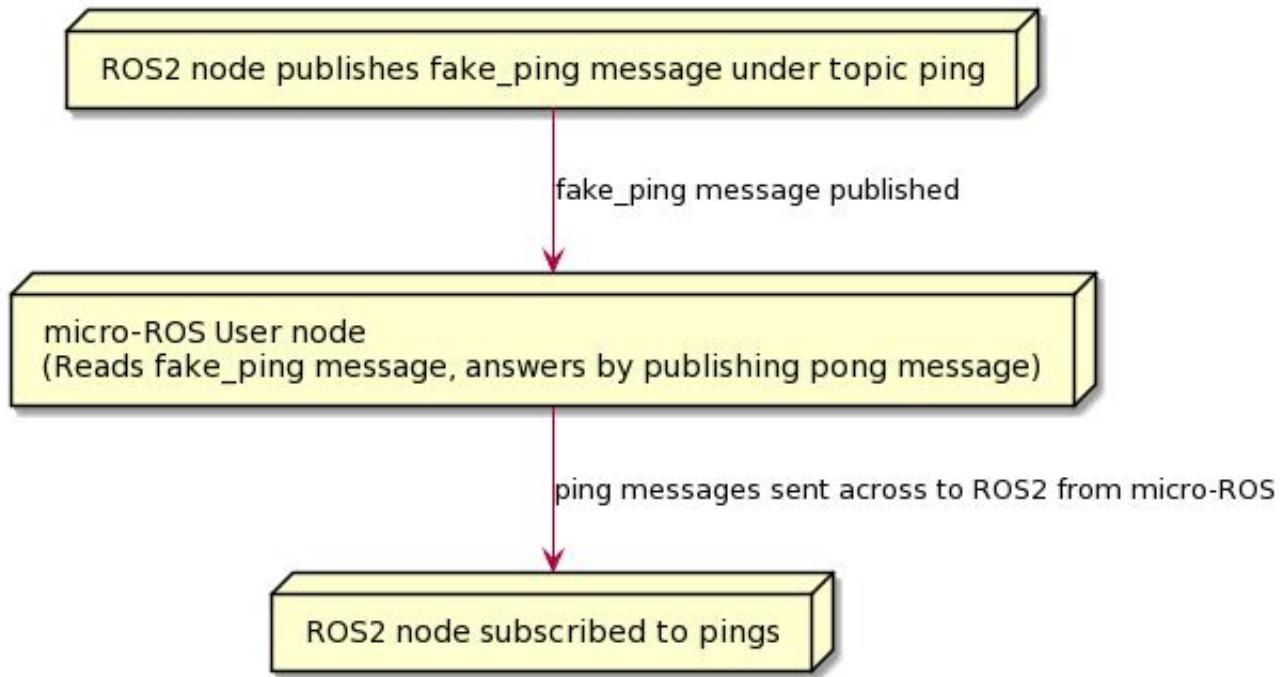
manan@PC:~/microros_ws$
```

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

Fourth Step:

- A fake_ping has been published and hence the micro-ROS node that is running answers this 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
```

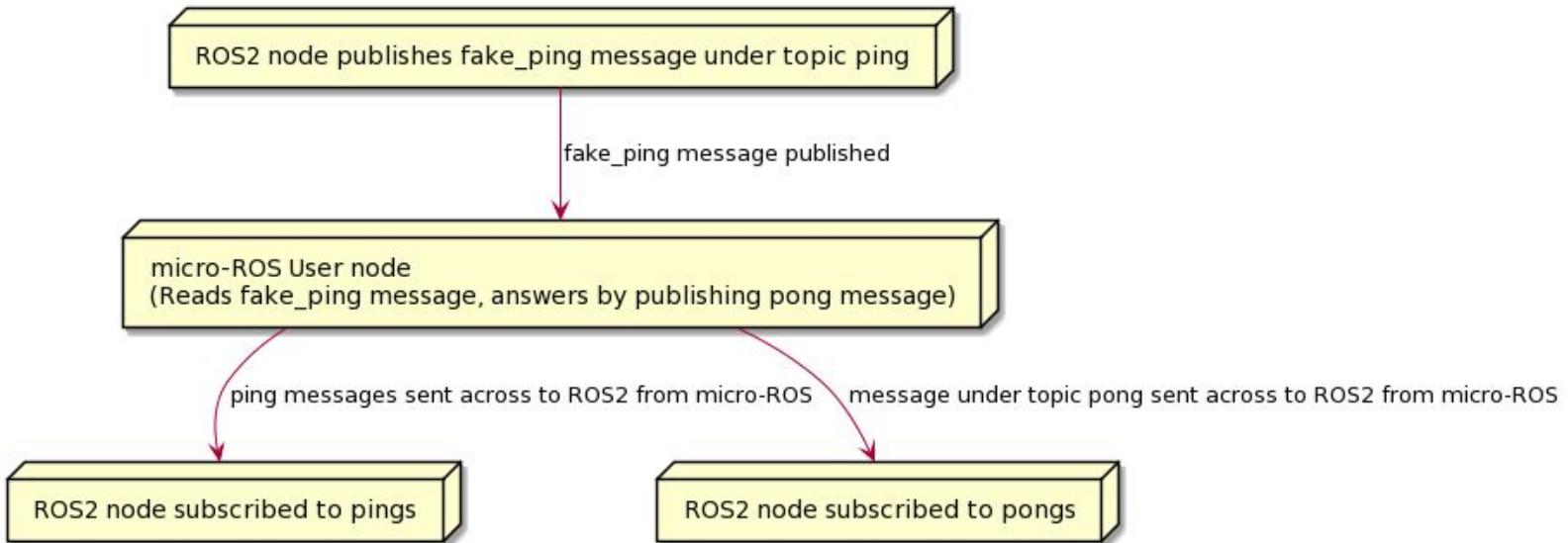


Fake_ping published using ROS2, micro-ROS receives and replies with pong

Fifth Step:

- To test whether a pong has been received as a consequence of the fake_ping we subscribe to topic “pong” as shown here:

```
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
---
```



ROS2 used to subscribe to pong from micro-ROS