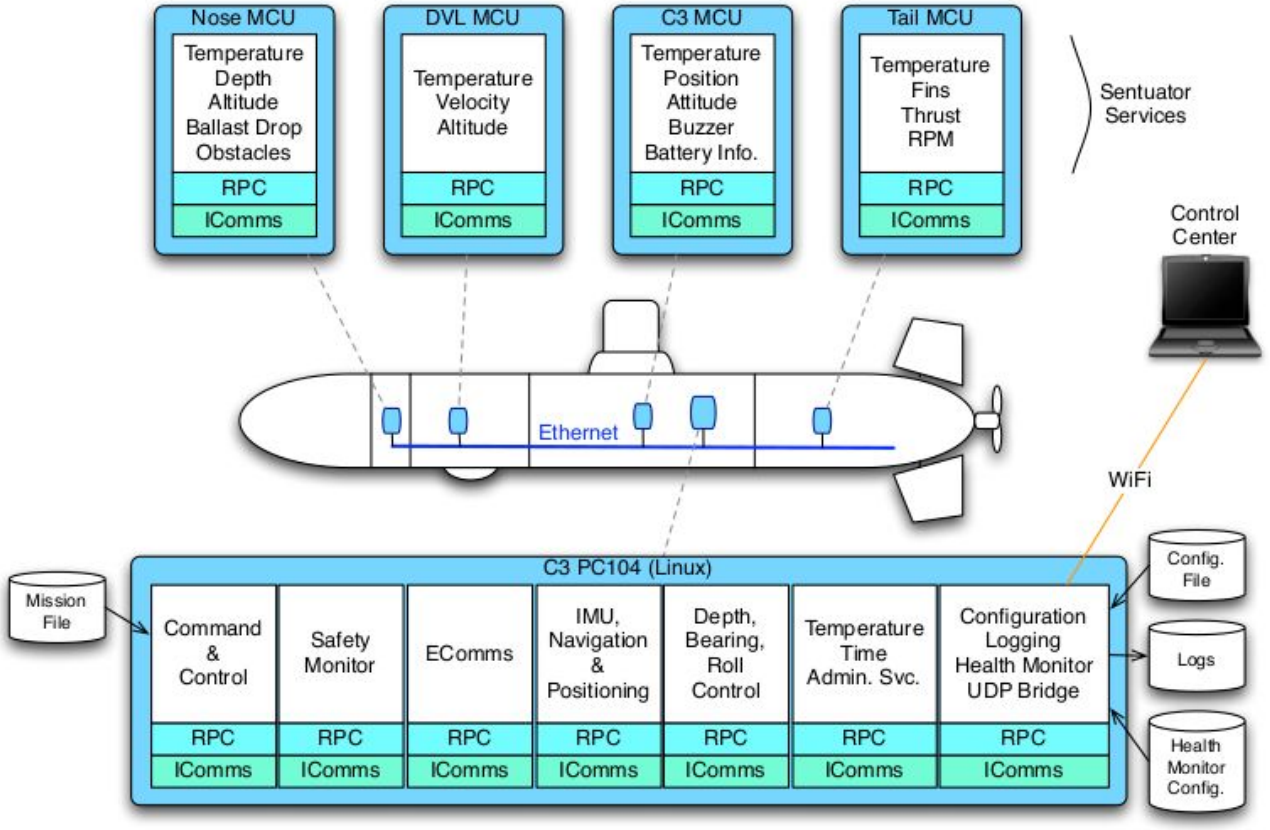
Project Presentation

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

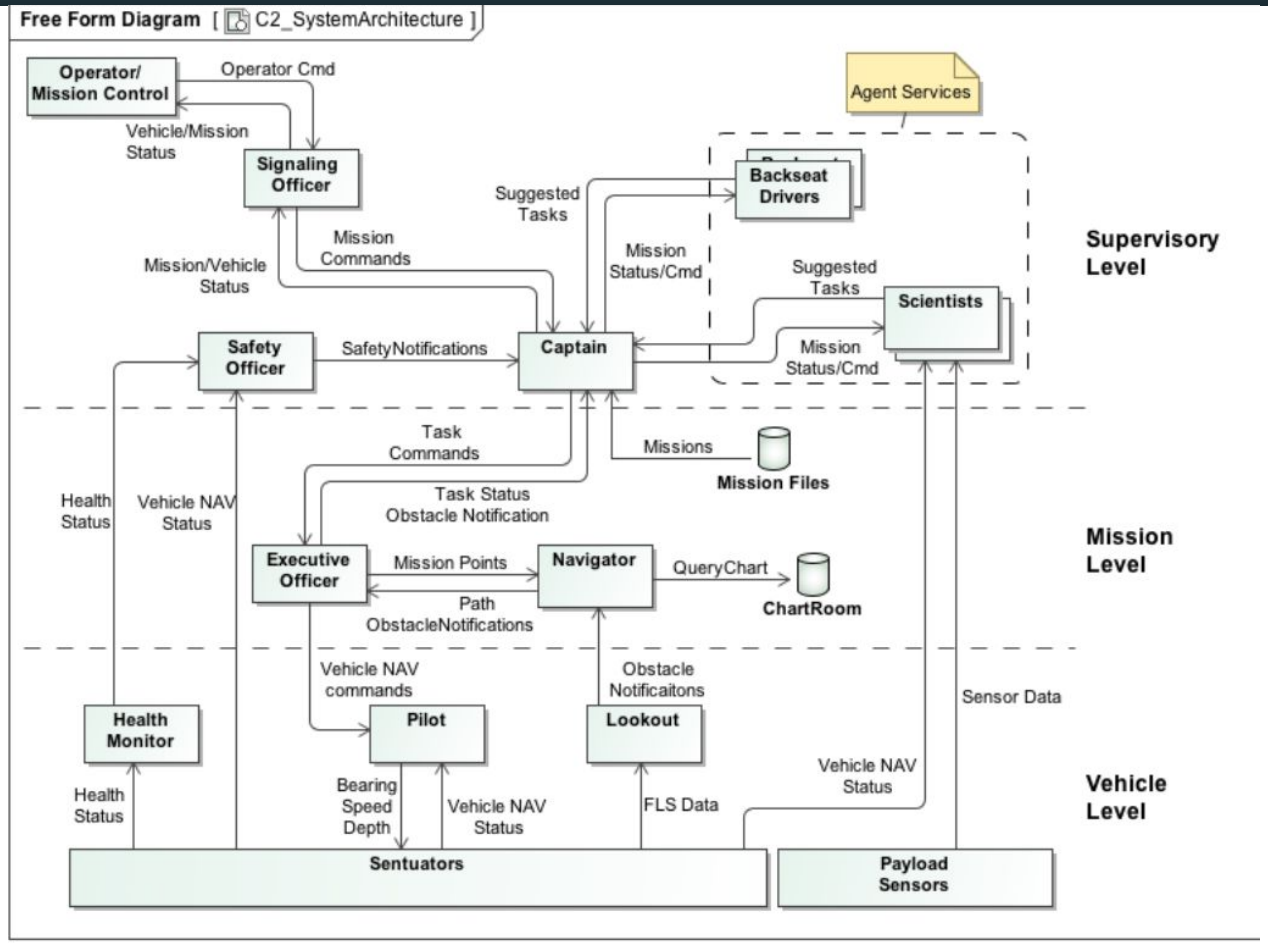
Background

- 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



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

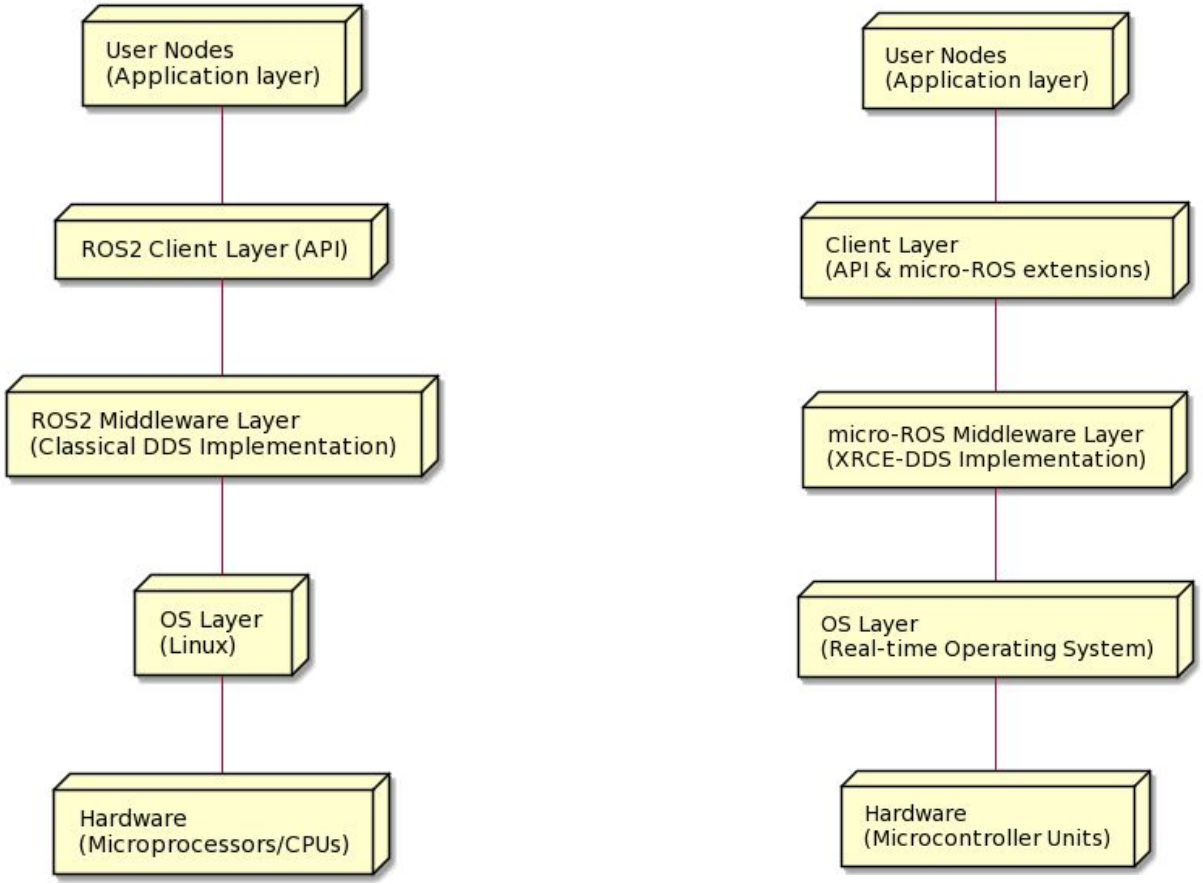
- Robot Operating System is a set of tools and libraries for building robot applications
- It is open-source and well maintained by the community
- Uses publish/subscribe model as the communication protocol
- Promotes “loosely-coupled” systems, implying hardware modularity and increases flexibility

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

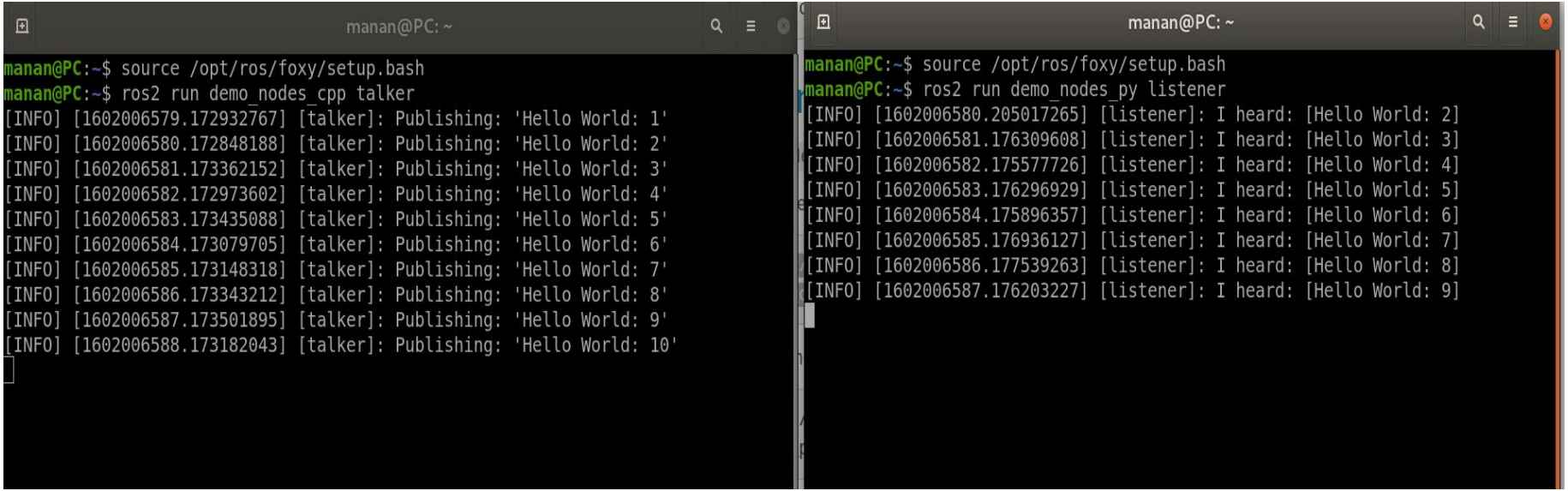
- Explore using micro-ROS applications for the different agents in the AUVs and communicate with the central microprocessor which runs a ROS2 stack
- Allows increased modularity and flexibility
- One subsystem doesn’t need to know of the existence of another, and doesn’t depend on it directly
- Can make 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 node running on the left is a “publisher”, written in C++. It is 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

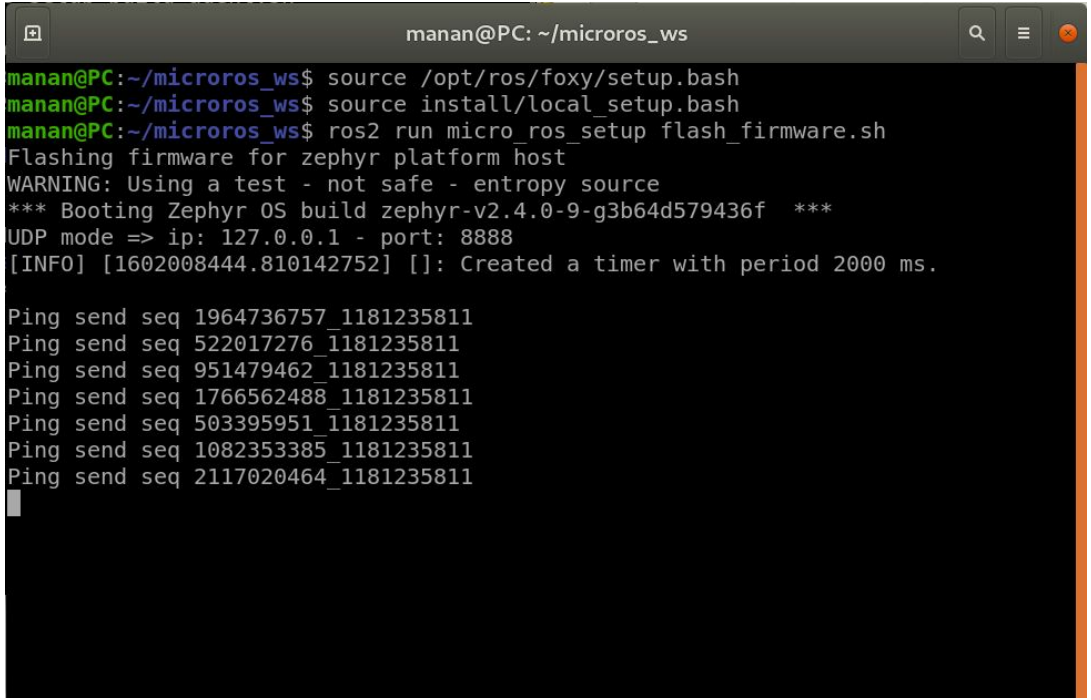
Ping-Pong Test

- 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)

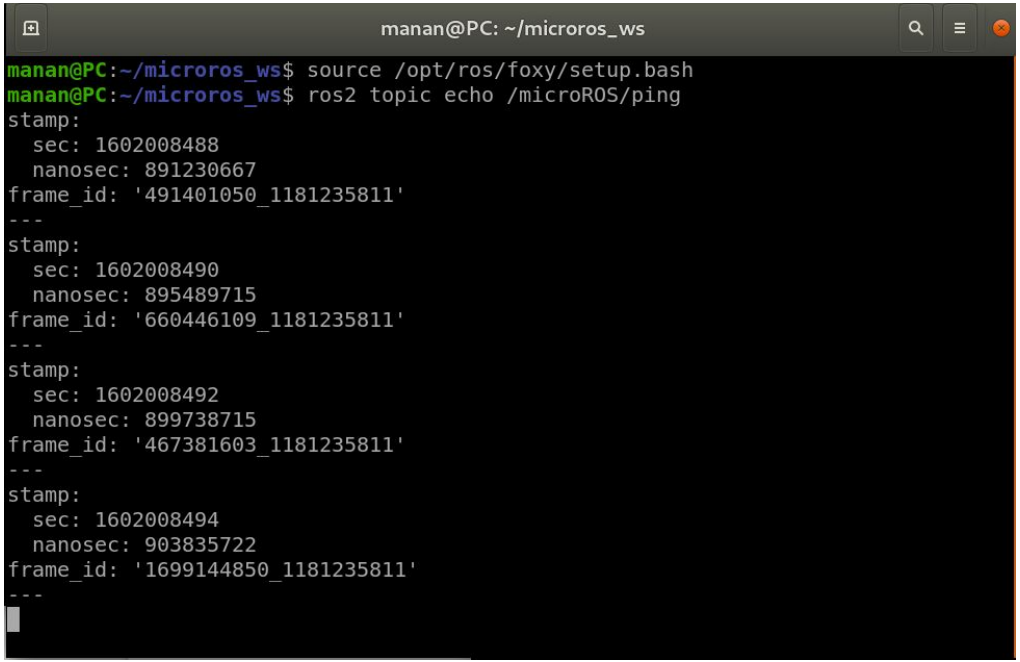
- To run micro-ROS node inside Zephyr RTOS emulator, we flash the firmware as follows:



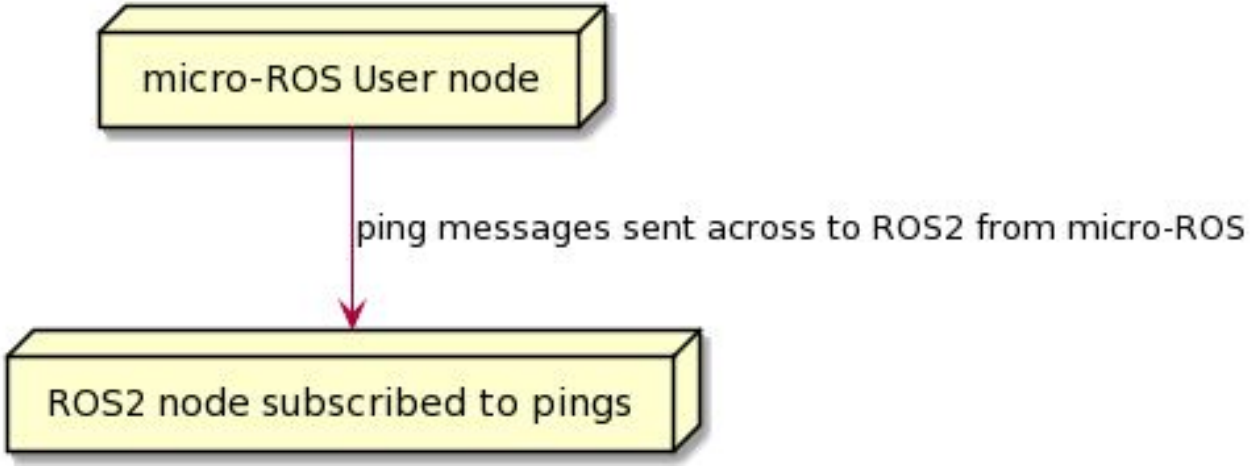
Pings are being published here

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



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



ROS2 used to subscribe to pings from micro-ROS

Third step:

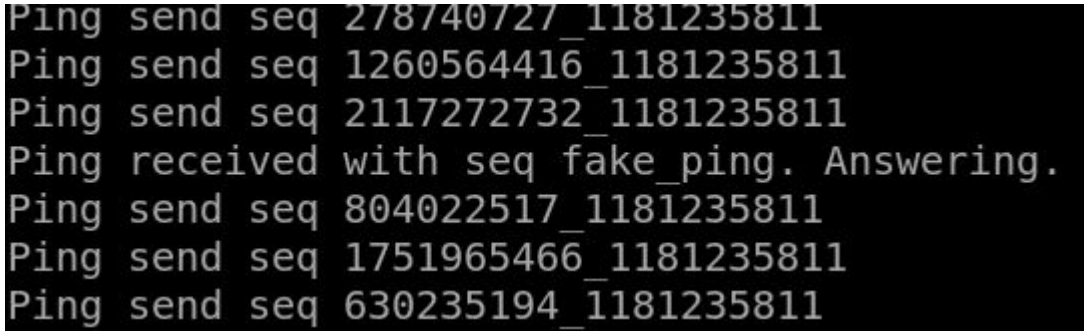
- We now publish a “fake_ping” message using ROS2. This also shows up in the ping subscriber

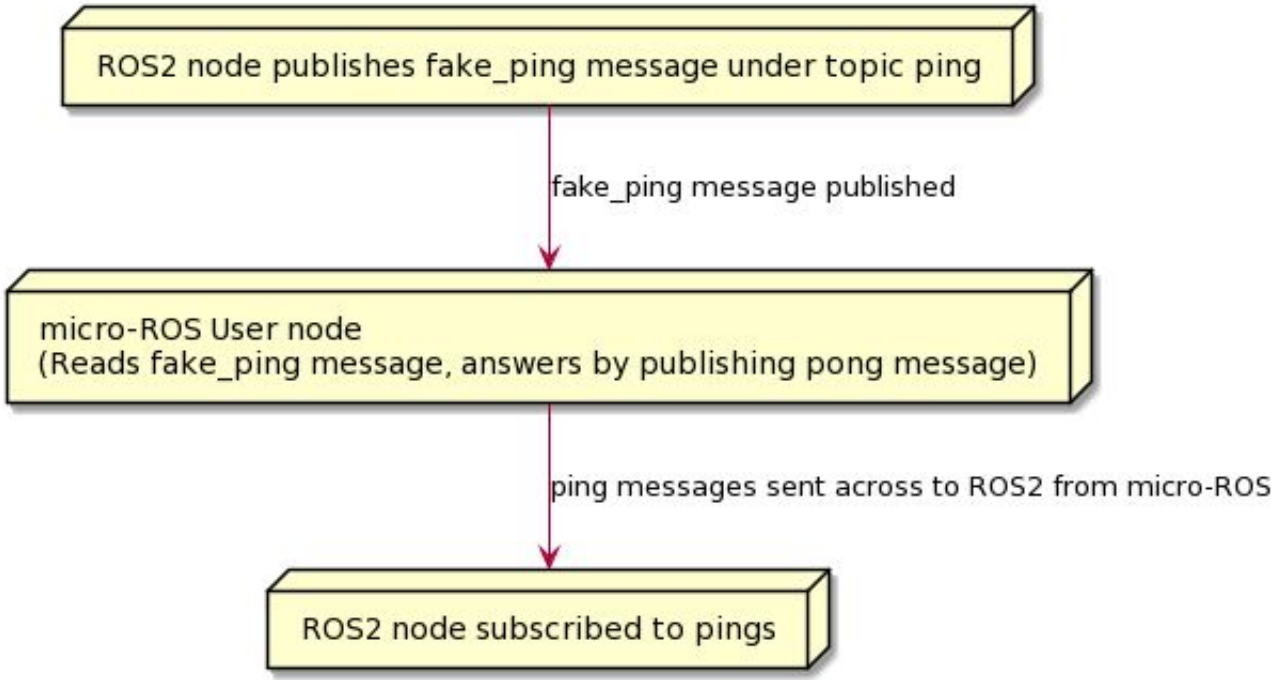
```
manan@PC:~/microros_ws$ source /opt/ros/foxy/setup.bash
manan@PC:~/microros_ws$ ros2 topic pub --once /microROS/ping std_msgs/msg/Header
 '{frame_id: "fake_ping"}'
publisher: beginning loop
publishing #1: std_msgs.msg.Header(stamp=builtin_interfaces.msg.Time(sec=0, nano
sec=0), frame_id='fake_ping')

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”

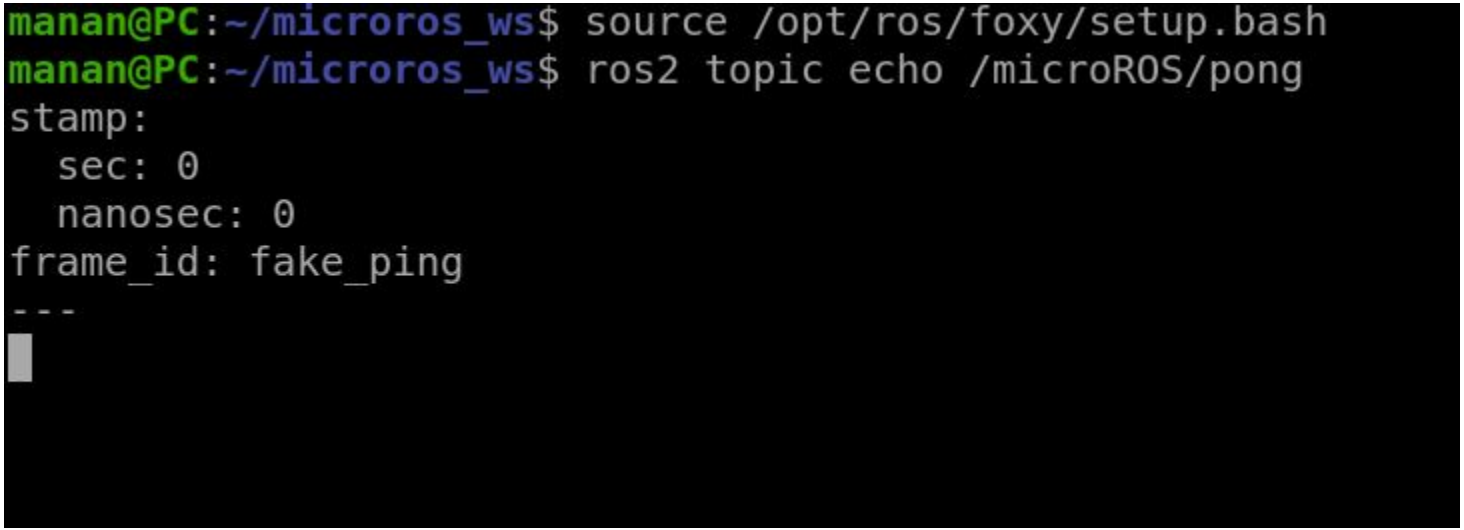


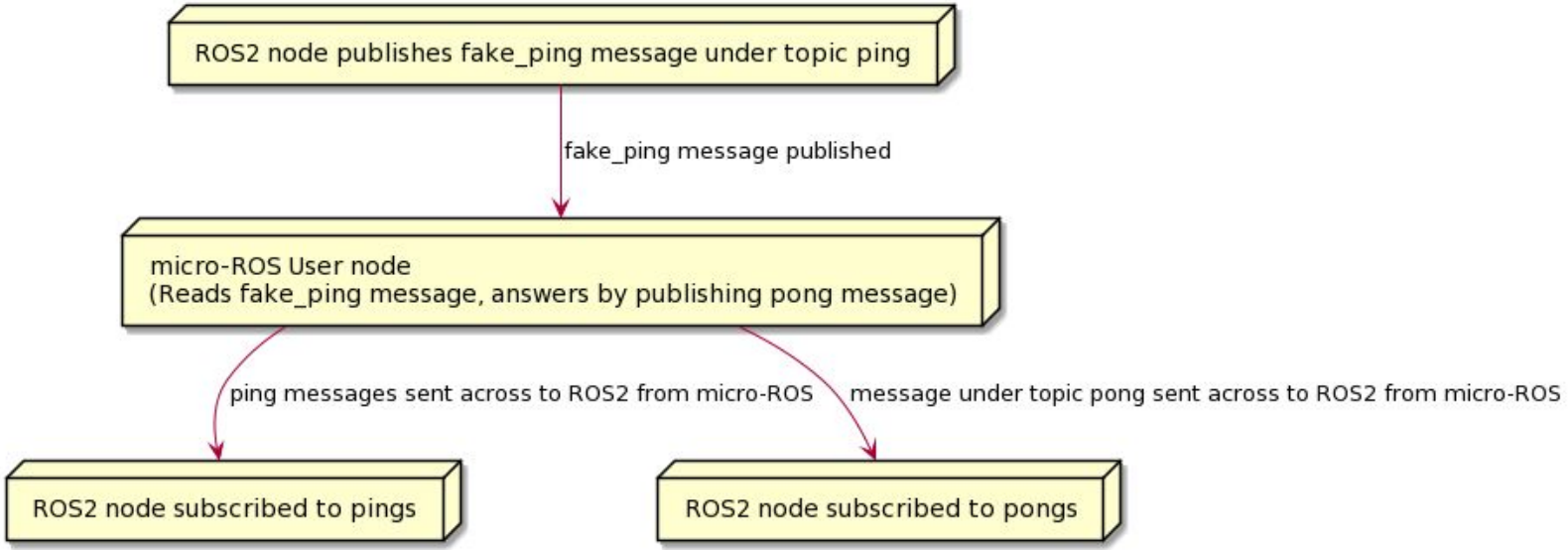


Fake_ping published using ROS2, micro-ROS receives and replies with pong

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:





ROS2 used to subscribe to pong from micro-ROS