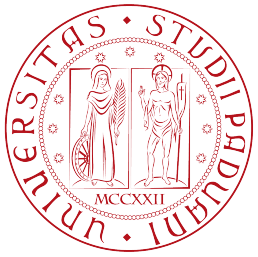


Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in
ICT for Internet and Multimedia

Robotic Operating System talks underwater: a communication framework to control underwater vehicles

Relatore:
Prof. Filippo Campagnaro

Laureando:
Davide Costa
2090108

Anno Accademico 2023/2024

Abstract

In recent years, advancements in underwater communication have enabled engineers to achieve great milestones in this domain. Presently, underwater devices such as drones and sensor stations are typically interconnected via cables; however, emerging technologies now offer the capability to realize the same connectivity using acoustic waves, similarly to the evolution of wireless networks on land.

The Robot Operating System comprises a suite of software libraries and tools utilized for building standardized robot applications in languages such as C++ and Python. Within ROS there is a specific module known as middleware, allowing the communication between different nodes. This middleware can be exploited to develop novel data exchange layers capable of utilizing various transmission methods, and specifical solutions concerning the underwater channel will be studied.

This project aims to build a middleware allowing to communicate through an underwater acustic channel using the DESERT protocols stack and the EvoLogics underwater acoustic modems, with the target to remotely control aquatic robots for research and offshore applications.

Contents

1 Introduction **1**

2 Underwater acoustic communication **3**

2.1 System model 3

2.2 Channel model 4

2.2.1 Attenuation 5

2.2.2 Noise 5

3 Acoustic modems **7**

3.1 EvoLogics modem hardware 7

3.1.1 DMAC Emulator 8

3.2 Subsea Underwater Modem 10

3.2.1 Hardware components 11

4 DESERT underwater **13**

4.1 The protocol stack used in this thesis 14

4.1.1 Application layer 14

4.1.2 Transport layer 14

4.1.3 Network layer 15

4.1.4 Data link layer 15

4.1.5 Physical layer I 16

4.1.6 Physical layer II 17

4.1.7 Protocol stack summary 17

5 Robot Operating System **19**

5.1 Framework entities 19

5.1.1 Nodes 20

5.1.2 Topics 20

5.1.3 Services 20

5.1.4 Parameter server 20
5.1.5 Graphical representation 21
5.2 Application examples 21
5.2.1 Node 22
5.2.2 Teleop key 22
5.3 Middlewares 23
5.3.1 Type supports 23

6 ROS middleware for DESERT **25**
6.1 CBOR encoding 25
6.1.1 Data structure 26
6.2 Packet structure 27
6.3 TCP daemon 28
6.4 Message serialization 28
6.5 Middleware interface 29
6.6 DESERT entities 31

7 NS simulation results **33**
7.1 UW/Physical 34
7.2 UW/UwModem/EvoLogicsS2C 36
7.3 EvoLogicsS2C and TDMA 38

8 Conclusions **41**

References **43**

Chapter 1

Introduction

Throughout history, humans have been striving to enhance their communication abilities using the waves. From the early use of simple sound signals like whistles and bells to the development of underwater telegraph cables in the 19th century, the evolution of underwater communication has been truly remarkable [1].

A significant milestone in this field was the invention of sonar during World War I, which revolutionized the ability of submarines to detect and track enemy vessels underwater. This breakthrough led to the development of more sophisticated underwater communication systems used in scientific research, marine exploration, and even in marine mammal studies. Fast-forward to today, where we have advanced underwater acoustic modems capable of high-speed data transmission over long distances [3], opening up new horizons for underwater communication in various fields such as offshore drilling, oceanography, and underwater robotics.

Among the renowned hardware for underwater communication are the EvoLogics underwater modems, showcasing cutting-edge technology. EvoLogics has left a significant mark in the industry with their innovative modems enabling reliable and high-speed data transmission through acoustic waves. These devices are designed to withstand the challenges of the underwater environment, providing researchers, engineers, and marine professionals with a potent tool for real-time data exchange. However, their high cost may not be suitable for all applications where such performance is unnecessary.

The SIGNET laboratory at the University of Padova's Department of Information Engineering developed a more cost-effective modem based on a Raspberry Pi [5]. While its performance may be lower than EvoLogics, it is ideal for short-distance underwater transmissions performed consuming minimal power.

One concrete application of the underwater communications is for controlling submarine wireless drones using common development frameworks. In this work, starting from the Robot Operating System (ROS) set of software libraries and tools used to build robot applications, it is presented a middleware extension which enables those veichles to be controlled using acoustic underwater signals through the DESERT underwater protocols stack [9] [7].

Called rmw_desert, short for ROS MiddleWare for DESERT, serves as a bridge between the physical underwater data exchange and the higher programming levels utilized by ROS developers. It enables them to seamlessly send and receive information from robots using standard and custom structures constructed from fundamental C or C++ data types.

Chapter 2

Underwater acoustic communication

Sending and receiving sound messages below the water is what underwater acoustic communication consists in. It remains a hot topic in various fields like military, commercial, educational, and scientific activities [10], catching the eye of many research companies and public universities. It is crucial for every user of underwater acoustic communication systems to receive accurate and complete data promptly.

However, bridging the gap between terrestrial and underwater acoustic communication is still a major challenge due to issues like multi-path propagation, limited bandwidth, signal attenuation over long distances, and channel variations over time. The biggest hurdle in underwater communication is the water itself, as the accuracy of acoustic signals underwater depends on factors like water type, impurity levels, pressure, water composition, and temperature.

2.1 System model

In underwater acoustic communications, the system operates by transforming electrical signals into acoustic signals at the transmitting end using a transducer, and then converting those acoustic signals back into electrical signals at the receiving end. The block diagram of this underwater acoustic communication system is presented in Figure 2.1.

Specific software handles the coding and decoding of messages using efficient techniques with minimal overhead to make the most of the limited bitrate available. Later on, a practical example of this efficiency will be provided. By trimming unnecessary data and honing in on essential information, these encoding methods ensure that all messages swiftly and accurately reach their intended

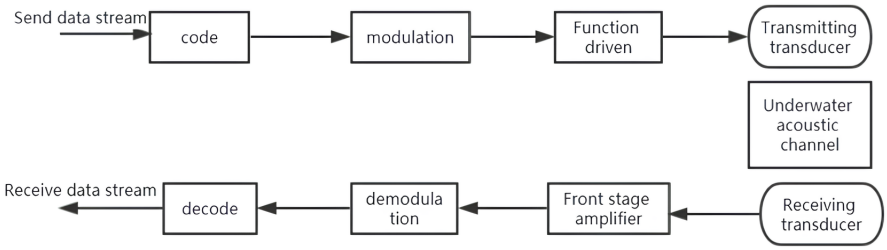


Figure 2.1: Underwater communication system block diagram.
The image has been reproduced from [2].

destination.

The subsequent steps are taken care of by underwater acoustic modems, which utilize various modulation techniques to encode information into acoustic signals that can navigate through the depths. These modems often employ standard modulation schemes such as Frequency Shift Keying, Phase Shift Keying, or Quadrature Amplitude Modulation and spread-spectrum techniques like Frequency-Hopping Spread Spectrum, Direct Sequence Spread Spectrum or Linear Frequency Modulation for effective underwater communication.

2.2 Channel model

Underwater communication faces challenges from factors like the seabed, water surface, noise from marine life, ship activity, and wind speed. Additionally a strong multi-path propagation is present, making it tough to send messages over long distances.

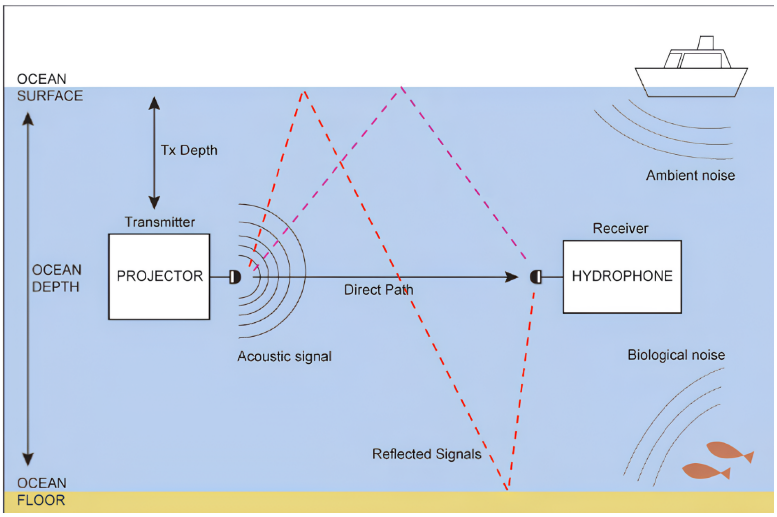


Figure 2.2: Underwater communication environment representation.

Navigating through these underwater intricacies requires a keen understanding of the channel characteristics and a strategic approach to ensure messages can travel far and wide.

2.2.1 Attenuation

Path loss or attenuation is the first issue that must be considered. In underwater environments, electromagnetic waves lose power much faster than in wireless communication on land. This happens because of spreading and absorption loss, which get worse with higher frequencies.

Spreading loss is the extra loss in wave amplitude compared to a flat wave, affected by how the transmitter and receiver are set up and the environment's properties. Absorption loss is the gradual decrease in wave amplitude as it moves through water, caused by heat and energy loss in the fluid.

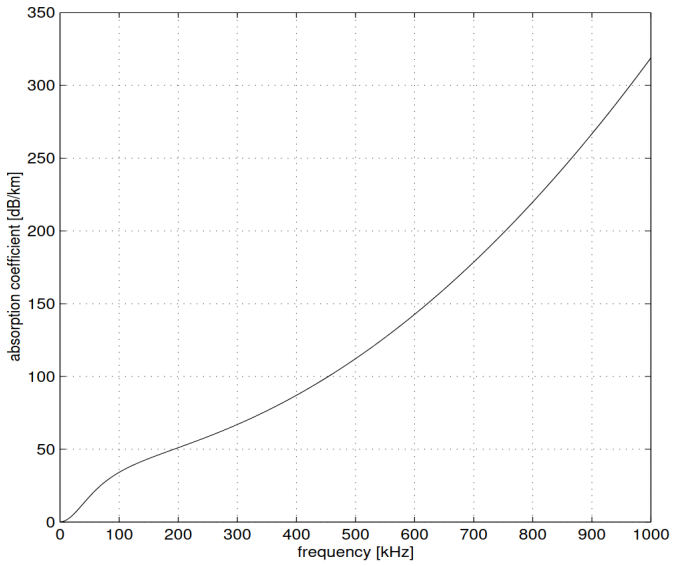


Figure 2.3: Underwater channel absorption coefficient. The image has been reproduced from [10].

2.2.2 Noise

The influence of the environment on underwater noise levels, as depicted in Figure 2.2, is a crucial factor to consider. Four distinct sources of noise have been identified and various scenarios were examined to determine the optimal carrier frequency for each circumstance.

The first source is turbulence, characterized by random and chaotic three-dimensional vorticity, impacting the lower frequency spectrum exclusively. Shipping activity, generated by the movement of distant vessels, dominates the 10 Hz-100 Hz frequency range and is quantified through the shipping activity factor.

Surface motion, induced by wind-driven waves, significantly contributes to noise levels within the 100 Hz - 100 kHz frequency range. Lastly, thermal noise, arising from random thermal electron motion, becomes prevalent for frequencies exceeding 100 kHz.

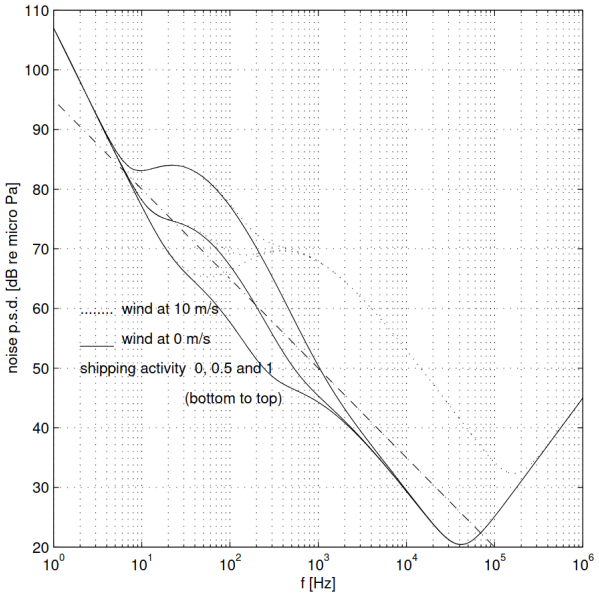


Figure 2.4: Underwater channel power spectral density of the ambient noise. The image has been reproduced from [10].

Chapter 3

Acoustic modems

Underwater acoustic modems use sound waves to transmit data between devices submerged underwater. These modems convert digital signals into sound waves that travel through water more efficiently than radio waves. By modulating the frequency, amplitude, or phase of the sound waves, information can be transmitted and received by other modems in the water.

In this chapter two different solutions are presented, the first one employing hardware devices produced by Evologics, and the second one with software defined modems developed at the University of Padova. Both are valid but it is important to choose the correct one depending on the environmental conditions and the type of application.

3.1 EvoLogics modem hardware

At the beginning of this century a company named EvoLogics emerged, as a result of the work made by their scientists and research experts. Their primary objective was the development of pioneering technologies tailored for the maritime and offshore industries. A strategic alliance that encouraged interdisciplinary collaboration between teams of engineers and life scientists developed sophisticated modems, specifically designed to enable reliable data exchange in the depths of the sea and surmounting the obstacles posed by underwater conditions.

With cutting-edge technology like acoustic signal processing and robust networking capabilities, these modems cater to the needs of various marine researchers, scientists, and oceanographers, facilitating communication and data transfer beneath the waves. They are the peak of innovation in underwater communication, offering a refined solution for transmitting information across aquatic

environments.

Figure 3.1: An underwater acoustic modem from EvoLogics.

EvoLogics has developed a variety of modem types with features tailored to suit different environments. Some provide higher bitrates and shorter ranges, while others are designed for the opposite. In Figure 3.2, a graph is provided with the operating range on the horizontal axis and the bitrate on the vertical axis to illustrate how these parameters interact across different modem models.

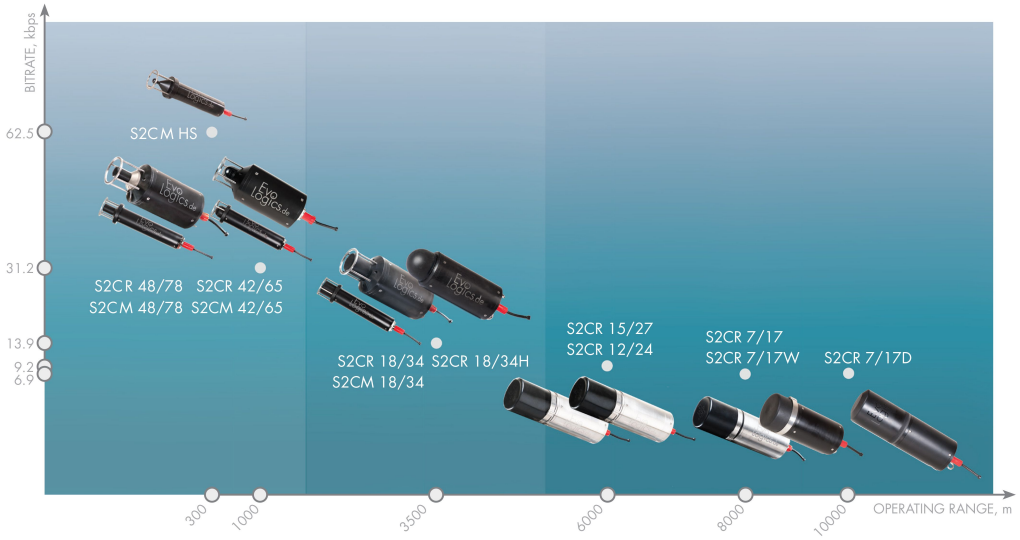


Figure 3.2: Underwater modems positioned based on operating range and bitrate capabilities. The image has been reproduced from [3].

3.1.1 DMAC Emulator

The Dual Media Access Control represents EvoLogics' pioneering data-link layer protocol. Its emulator serves as a software tool that enhances the versatility available to underwater network protocol developers of EvoLogics' underwater

acoustic modems. The primary goal is to streamline the development of underwater network protocols by eliminating the need for costly modem hardware during initial testing phases. The emulator replicates all aspects of the modem's data-link protocol layer and incorporates a phenomenological simulator of the physical protocol layer.

Any code that is written and executed on the modem emulator can subsequently be executed on the physical modem hardware without necessitating any alterations. This feature provides a time-efficient solution that reduces development expenses associated with upper layer network protocols and streamlines the process of integrating acoustic modems into underwater infrastructure.
Simulated variables are:

- signal propagation delays caused by finite speed of sound in the water;
- packet collision detection generated by multiple simultaneous transmissions;
- packet synchronization and demodulation errors of the receiver;
- movement and rotation of the virtual modems decided by the user;
- attenuation of the signal produced by spreading and absorption loss.

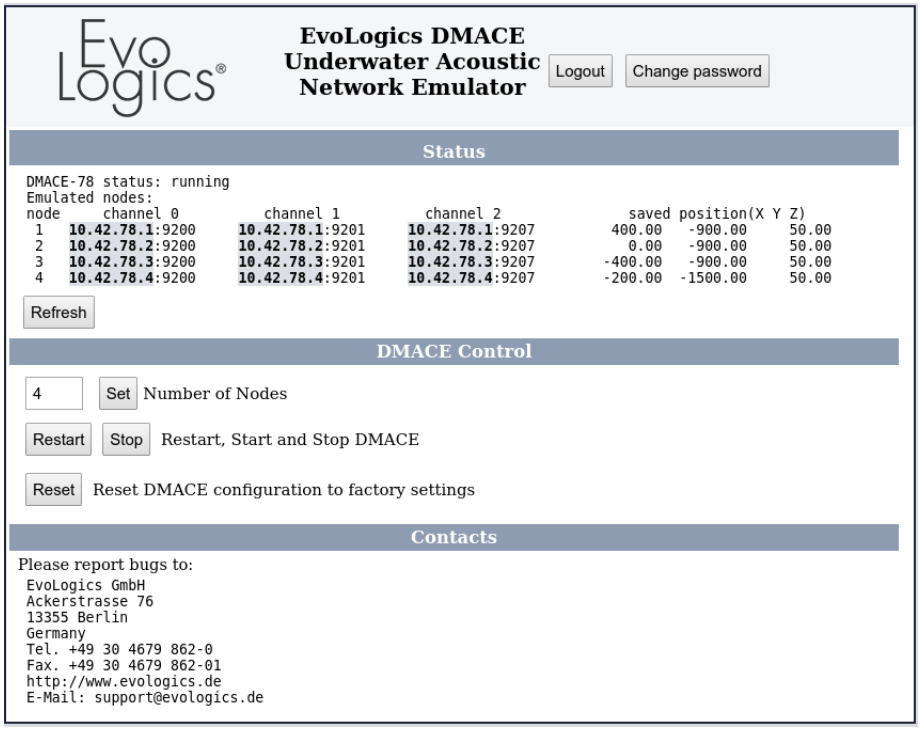


Figure 3.3: Online interface used to control simulated modems.

An online tool is available for the graphical management of virtual modems, allowing users to interact with them through control commands like position, orientation, or frequency range. Each modem has a unique IP address within EvoLogics' Virtual Private Network, enabling communication via netcat from the developer's Linux shell or Windows prompt for testing various scenarios.

3.2 Subsea Underwater Modem

The sophisticated acoustic modems produced by EvoLogics are extremely good solutions, but they are also very expensive and some applications may necessitate lower performance benchmarks and reduced power consumption than those offered by these high-end devices. For instance, the deployment of Autonomous Underwater Vehicles operating collectively in coordinated fleets requires compact, lightweight, and energy-efficient acoustic modems for integration onto this size-constrained and battery-operated drones. These challenges could be faced by employing cheap modems built with low-power and low-depth rated components.

Moreover using the Software Defined Modem architecture provides the system with improved adaptability and reconfigurability, enabling the customization of various communication stack attributes to suit the specific channel conditions, thereby enhancing reliability, minimizing energy usage, and reducing hardware requirements to a few processing units [4]. Their flexibility allows for efficient utilization of the limited resources within underwater nodes, while the general-purpose processing units they utilize prove more cost-effective than specialized hardware like digital signal processors or field-programmable gate array circuits.

The SIGNET lab, in collaboration with the Italian General Directorate of Naval Armaments, the Italian National Research Council and the startup company SubSeaPulse Srl, has engineered a software-defined modem encompassing two layers of the communication stack: the physical layer and the Medium Access Control layer (MAC).

Afterwards, SubSeaPulse enhanched this research idea creating a modem prototype now commercialized under the name SuM, ´Subsea underwater Modem´, that is capable of executing real-time operations without necessitating post-processing, relying only on cost-effective software and hardware tools. Its integration potential appeals to professionists like students, researchers and practitioners, bridging the gap between underwater communications and civilian applications.

3.2.1 Hardware components

The structure of the modem is divided into three distinct parts detailed below.

1. A Raspberry Pi, serving as the processing unit, undertakes all protocol stack functions, including routing, channel access, forward error correction, and digital signal processing tasks such as filtering, preamble synchronization, modulation, and demodulation.

2. A HiFiBerry DAC+ADC Pro HAT performing a 192 kHz 24-bit analog-to-digital and digital-to-analog conversion process.

3. An amplification and switch module is employed to transition the modem between reception and transmission modes, enhance the received signal through pre-amplification, and amplify the outgoing signal for transmission.

All analog frontend components are consolidated on a single Printed Circuit Board denoted as the SuM HAT, which has the dimensions of a standard Raspberry Pi HAT. The complete modem configuration comprises three layers: the Raspberry Pi, HiFiBerry, and SuM HAT.

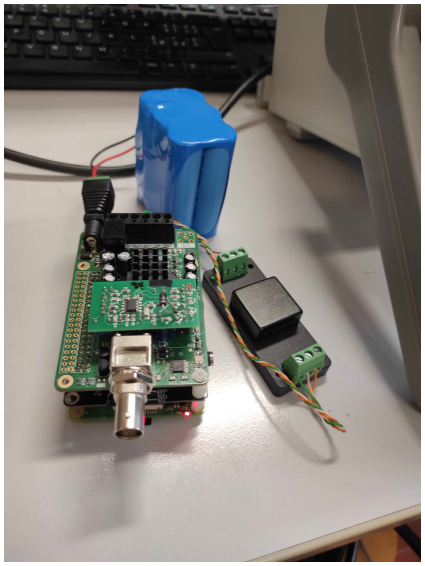


Figure 3.4: A Subsea underwater Modem.

SuM is compatible with any Raspberry Pi model: for energy-restricted applications, Raspberry Pi 0 suffices, while Raspberry Pi 4B is recommended for high-power demands, with Raspberry Pi 3B offering a balanced compromise between power consumption and processing capabilities.

The modem supports various underwater acoustic transducers, with its frequency band limitation determined by the HiFiBerry, enabling transmission from 2 Hz up to a maximum frequency of 70 kHz.

The maximum transmission power is contingent upon the impedance and Transmitting Voltage Response of the utilized transducers. For instance, with the Aquarian AS1 hydrophone, the output amplifier's 30 dB gain allows transmission of approximately 155 dB re 1 μPa at 40 kHz. Conversely, employing the Btech-2RCL transducer permits transmission of up to 180 dB re 1 μPa at 28 kHz owing to its higher TVR [5].

Chapter 4

DESERT underwater

DESERT Underwater represents a comprehensive suite of publicly available C++ libraries that enhance the NS-MIRACLE simulator to facilitate the development and deployment of underwater network protocols [6] [7]. The motivation behind its development lies in expanding the realm of underwater networking studies beyond mere simulations. The crucial significance of implementing research solutions on real devices cannot be understated in achieving a robust communication and networking framework that enables diverse nodes to communicate effectively in the underwater domain.

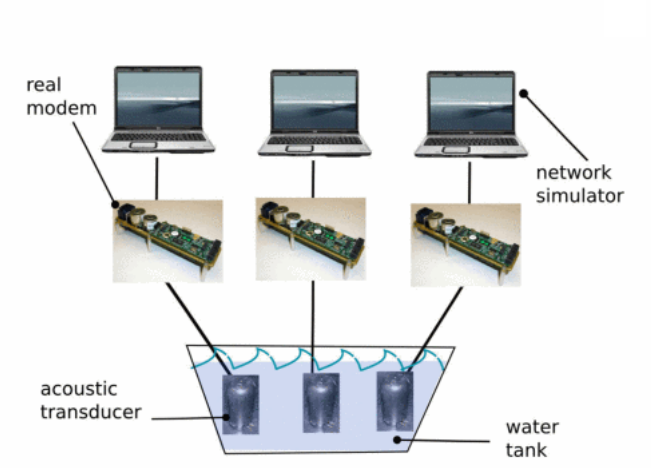


Figure 4.1: Network infrastructure used for DESERT underwater. The image has been reproduced from [7].

NS-Miracle enriches the ns2 network simulator by introducing a mechanism for managing cross-layer messages and supporting the coexistence of multiple modules within each layer of the protocol stack [8]. For instance, you can specify and utilize multiple IP addresses, link layers, MACs, or PHYs within a single

node. This feature underscores the high modularity of NS-Miracle, tailored to simulate nodes with a logical architecture closely resembling that of actual devices.

The utilization of NS-Miracle for designing protocol solutions for underwater networks presents developers with the opportunity to leverage existing code from ns2 with minimal alterations, and capitalize on NS-Miracle's modularity for enhancing the organization of their design endeavors. Furthermore, the performance evaluation of the developed protocol stack can be conducted through simulations utilizing specific channel model tools.

4.1 The protocol stack used in this thesis

Within the DESERT framework, all ISO/OSI network layers, spanning from the lowest to the highest, are fully implemented. This section provides insights into the specific layers utilized in this project.

To differentiate between NS-Miracle and DESERT modules, all those within DESERT are designated with a prefix uw-. It is worth noting that this prefix does not necessarily indicate that a module's protocol solution is tailored exclusively for underwater networking.

4.1.1 Application layer

Amnong the various application layer of DESERT, in this thesis we use UW/APPLICATION, a module that allows the transmission of adjustable size payloads between nodes. Additionally, genuine data can be sent through either a TCP or UDP socket on a specified port, enabling the connection of a sensor or device to transmit its data to UW/APPLICATION through it. This module encloses the payload with control headers and sends the packet to the lower layers.

4.1.2 Transport layer

For the Transport layer, the UW/UDP module is utilized to perform flow multiplexing and demultiplexing between the upper and the lower layers. However, it is important to note that this module lacks support for link reliability, error detection, or flow control.

4.1.3 Network layer

The Network layer is responsible of furnishing tools for the network interfaces and establishing mechanisms for data routing. In this specific setup, the UW/STATICROUTING module was employed to facilitate the simulation and testing of data traffic that must adhere to pre-defined routes.

Each network node has the option to designate a default gateway and/or populate a static routing table, with a maximum capacity rigidly set to 100 entries. This data is subsequently utilized within each node to autonomously direct network packets along the specified paths, hop by hop.

```
# Setup routing table
for {set id1 0} {$id1 < [expr $opt(nn) - 1]} {incr id1}  {
    set id2 [expr $id1 + 1]
    $ipr($id1) addRoute [$ipif_sink addr] [$ipif($id2) addr]
    $ipr_sink addRoute [$ipif($id1) addr] [$ipif($id1) addr]
}
set last_id [expr int($opt(nn) - 1)]
$ipr($last_id) addRoute [$ipif_sink addr] [$ipif_sink addr]
$ipr_sink addRoute [$ipif($last_id) addr] [$ipif($last_id) addr]
```

Listing 4.1: ns2 routing table population using TCL.

In addition, the UW/IP module is employed to allocate addresses to nodes within a designated network based on the standard IPv4 addressing scheme. This allocation includes the implementation of the Time-To-Live feature; however, it does not incorporate any routing mechanism within its functionality.

4.1.4 Data link layer

At the heart of the Data link layer lies the Medium Access Control component, which governs access to the acoustic communication channel.

ALOHA represents a random access scheme, denoting a protocol that enables nodes to transmit data packets directly without the need for any prior channel reservation procedure. Within this context, the UW/CSMA_ALOHA implementation is employed, serving as an upgraded variant of the initial ALOHA protocol. This enhanced version integrates a carrier sensing mechanism into the fundamental ALOHA framework to mitigate collision incidents.

Additionally, since node-to-node interactions at the link layer rely on MAC addresses, while those at the upper layers utilize IP addresses, there is a need for a method to correlate the latter with the former.

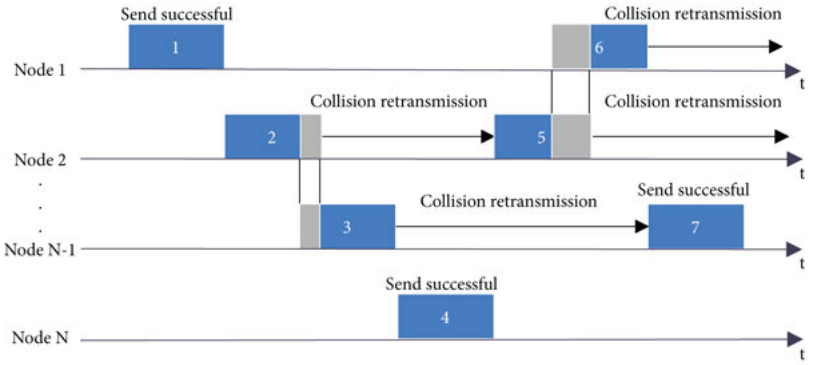


Figure 4.2: Aloha scheme of multiple channel access.

Through the use of UW/MLL, an association is established between IP and MAC addresses in advance, achieved by populating an ARP table for each network node. Alternatively, ARP tables can be automatically populated using the Address Resolution Protocol.

4.1.5 Physical layer I

This layer employs the UW/PHYSICAL module, which emulates an acoustic underwater channel utilizing a Signal-to-Noise Ratio threshold calculation model [10].

Factors such as shipping activity, wind velocity, inter-node spacing, signal power, and carrier frequency are predetermined by the user and employed to calculate attenuation and subsequent noise levels, culminating in the comparison of the obtained SNR against a preset threshold.

If the computed SNR surpasses the threshold, most packets will be successfully received; otherwise, a significant packet loss may occur due to the high Bit Error Rate. The formula is expressed in 4.1, where P is the transmitted acoustic power, A is the absorption, l the distance between transmitter and receiver, f the frequency, N the noise power spectral density, and Δf the signal bandwidth.

$$SNR(l, f) = \frac{P/A(l, f)}{N(f) \cdot \Delta f} \tag{4.1}$$

Thus, a local simulation is performed through mathematical computations; however, this does not ensure the software's readiness for integration with physical hardware. To achieve this, an alternative module was subsequently trialed in place of the existing one.

4.1.6 Physical layer II

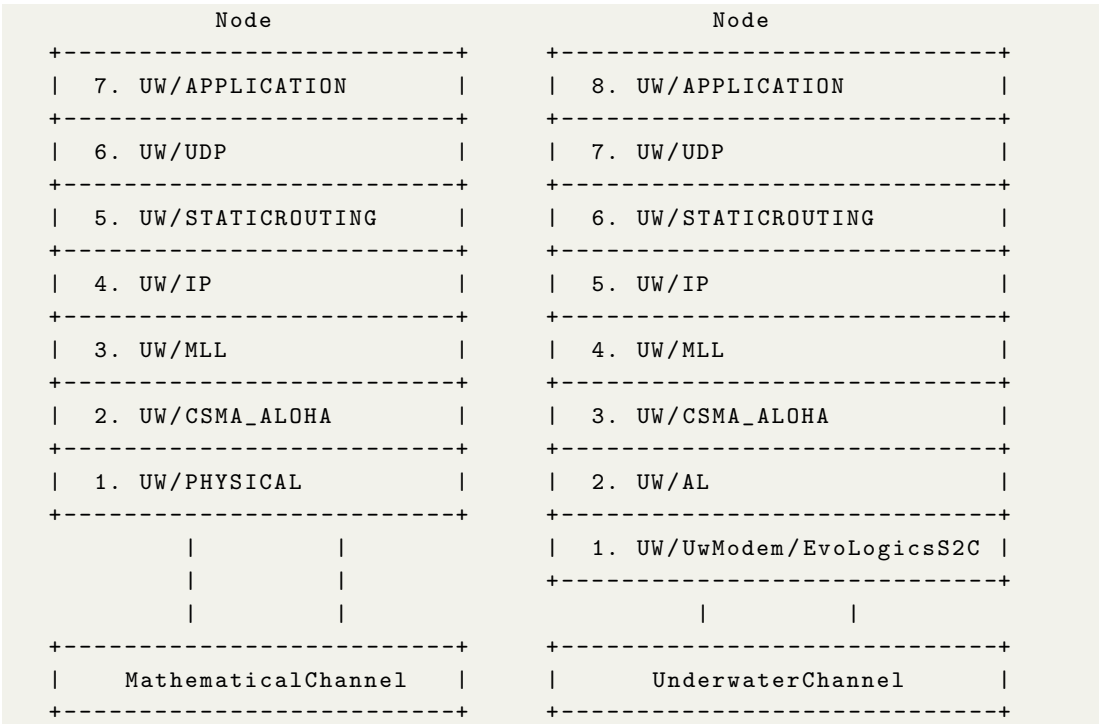
For the purpose of conducting experiments with physical hardware, a trial was conducted using underwater modem emulators from EvoLogics as detailed in preceding sections. The UW/UwModem/EvoLogicsS2C module was employed to establish a connection, defining and executing the interface between ns2 and authentic acoustic modems.

This module is responsible for managing all requisite messages for NS-Miracle like cross-layer communications between MAC and PHY layers, encompassing user-adjustable simulation parameters and associated methods for modification.

The benefit lies in the fact that whether utilizing emulators or real modems, the distinction is negligible. Therefore, validating a program in this virtualized setting suffices to guarantee its functionality on tangible devices deployed in the field.

Essentially, this module establishes a connection with the real or virtualized underwater modem via a Virtual Private Network utilizing the conventional network protocol. Subsequently, it sends directives to the device to allow the transmission and reception of data through the acoustic channel, effectively leveraging the full spectrum of functionalities of a physical modem.

4.1.7 Protocol stack summary



This section provides a summary of the aforementioned protocols. Specifically, on the left, the UW/PHYSICAL channel was utilized, whereas on the right, it was substituted with the EvoLogics interface. Furthermore, for the integration with real network stacks and their transmission/reception of packets, a real-time scheduler was employed to synchronize the simulation clock with the hardware clock. The objective of the real-time scheduler is to ensure that the simulation clock progresses in sync with the external time reference.

It is important to note that the latter configuration includes an additional layer known as the Adaptation Layer, which is essential for packet translation for real modems by interfacing with packers, performing the conversion of the headers into a bit stream and vice-versa.

Chapter 5

Robot Operating System

ROS, an acronym for Robot Operating System, serves as an open-source framework dedicated to the construction of robotic systems [9]. It furnishes software developers with libraries and tools essential for the creation of robot applications. ROS finds extensive application in research, academia, and industry for the advancement of state-of-the-art robotic technology.

Within the diverse tasks that a robot executes, ROS implements the functions typically offered by an operating system. These encompass hardware abstraction, driver-controlled device management, inter-process communication, application oversight with packages, and other frequently employed operations.

In ROS, a series of processes can be visualized as nodes interconnected in a graph structure. These nodes possess the capability to transmit, receive, and multiplex messages between themselves, sensors and actuators.

Despite the critical nature of reactivity and low latency in robotic control procedures, ROS does not function as a real-time operating system. However, it remains feasible to integrate ROS with real-time modules for enhanced performance.

5.1 Framework entities

Within the ROS framework, processes are depicted as nodes arranged in a graph structure interconnected by edges known as topics. These nodes in ROS have the capability to exchange messages via topics, initiate service requests to other nodes, offer services to other nodes, or access and modify shared data from a common repository known as the parameter server.

This decentralized design is particularly well-suited for robotic systems, which

often comprise a network of interconnected hardware components and may necessitate communication with external computers for intensive computation or command transmission.

5.1.1 Nodes

Nodes in ROS serve as individual processes within the ROS graph, each uniquely identified by a name registered with the ROS master. These nodes can exist under different namespaces with distinct names, or opt to be anonymous, in which case they receive an additional randomly generated identifier.

The foundation of ROS programming lies in nodes, as most ROS client code is structured in the form of ROS nodes that interact based on exchanged information, issuing commands, and responding to requests from other nodes.

5.1.2 Topics

Topics in ROS function as named communication channels through which nodes exchange messages. Each topic must have a unique name within its namespace. For a node to transmit messages to a topic, it must publish to that specific topic; conversely, to receive messages, it must subscribe to the topic.

This publish/subscribe model maintains anonymity, with nodes unaware of the specific senders or receivers on a given topic. The messages shared over topics can encompass a wide array of data types, including user-defined content such as sensor readings, control commands, system states, or actuator directives.

5.1.3 Services

Services are typically employed for actions with well-defined start and end points, such as capturing a single image frame, as opposed to continuous processes like motor control commands or sensor data processing. Nodes can advertise services and invoke services from other nodes within the ROS ecosystem.

5.1.4 Parameter server

The parameter server in ROS serves as a shared database accessible to all nodes, facilitating centralized access to static or semi-static information. Data that remains relatively static and infrequently accessed, such as environmental distances

or robot weight, are commonly stored in the parameter server for collective reference and retrieval.

5.1.5 Graphical representation

This section offers a graphical depiction of the publisher-subscriber and client-service architectures to enhance comprehension of the underlying mechanisms governing these components.

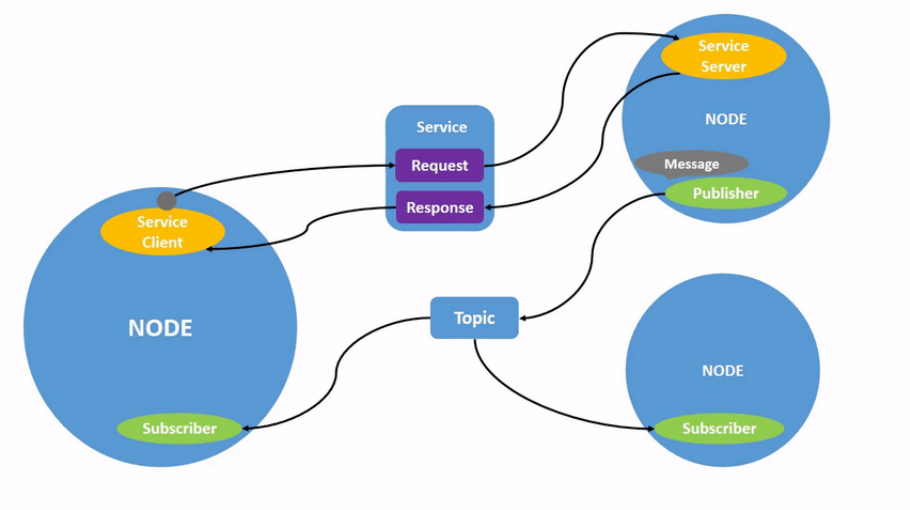


Figure 5.1: A service, a publisher and two subscribers.

5.2 Application examples

A valuable tool utilized for introducing the framework is Turtlesim, a lightweight simulator designed for educational purposes within ROS 2. Turtlesim serves as a fundamental demonstration of ROS 2 operations, offering insights into the activities one may engage in with an actual robot or a simulated robot in subsequent stages.

Essentially, the turtlesim_node showcases a graphical interface featuring a turtle that periodically transmits data regarding its position, color, and other attributes, while concurrently awaiting commands dictating its movement via topics and services.

By simulating the functionalities of an abstract robot, the turtle assumes the role of the target device necessitating control. Conversely, a program responsible for dispatching control commands, known as turtle_teleop_key, is also imperative for the interactive process.

5.2.1 Node

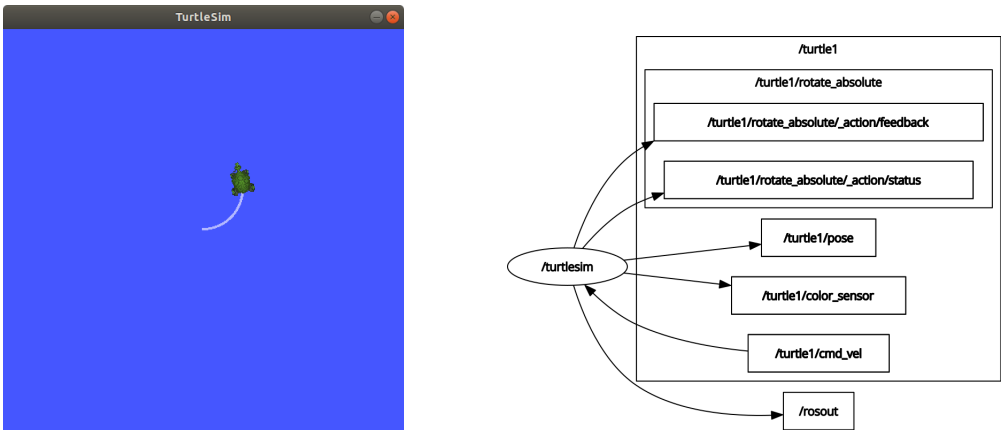


Figure 5.2: Turtlesim node topics and services.

Figure 17 represents the topic hierarchy of the node turtle1, engaging in message publication within the pose and color_sensor topics to relay data pertaining to its current attributes. Concurrently, it subscribes to the cmd_vel topic to receive directives for control.

Furthermore, a service named rotate_absolute is available for client utilization, enabling manipulation of turtle1 to adopt a predetermined absolute orientation in the plane, with feedback confirming the successful execution of the task.

5.2.2 Teleop key

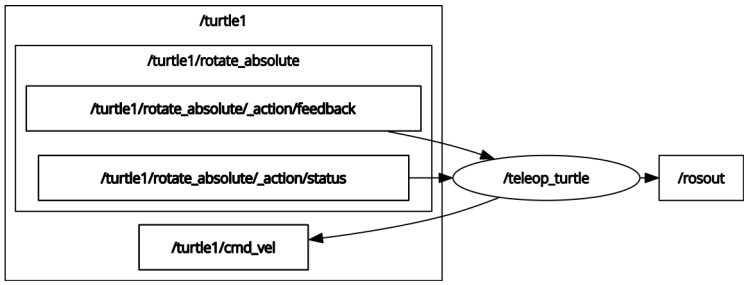


Figure 5.3: Teleop key topics and services.

The Teleop key module within Turtlesim operates in a terminal environment, capturing keyboard inputs that are translated into movement instructions. This module functions by publishing control commands derived from arrow key inputs to the cmd_vel topic, while also acting as a client to the rotate_absolute service for executing rotational maneuvers on the turtle.

5.3 Middlewares

Middlewares represent modules within ROS designed to act as drivers allowing the interaction between high-level data exchange and the physical transmission of data across a network. One primary objective of the middleware interface is to shield the user land code from any system-specific data distribution implementation.

As a result, the ROS client library situated ´above´ the middleware interface exclusively operates on ROS data structures. ROS 2 adheres to employing ROS message files to outline the composition of these data entities and generate corresponding data structures in each supported programming language.

Below the middleware interface, the middleware implementation is responsible for converting ROS data objects from the client library into a customized data format before transmission through the network protocol stack. Conversely, incoming custom data objects from the network necessitate conversion into ROS data entities before they are relayed back to the ROS client library.

The specification of middleware-specific data types is derived from the details outlined in the ROS message files. A predefined mapping between the primitive data types of ROS messages and middleware-specific data types ensures seamless bidirectional conversion. The conversion functionality between ROS types and the implementation-specific types or API is encapsulated within the type support mechanism.

```
+----------------------+
|      user land       |   1) Create a ROS message
+----------------------+      v
|  ROS client library  |   2) Publish the ROS message
+----------------------+      v
| middleware interface |   3) Convert the message into a bytestream
+----------------------+      v
|   data distribution  |   4) Send it to the network
+----------------------+      v
|        network       |
+----------------------+
```

5.3.1 Type supports

In ROS, messages are conveyed to the middleware in the form of raw void pointers pointing to a memory location without any specific type specification. The challenge arises from the diversity of data that can potentially populate this memory location, encompassing various types such as integers, floats, strings, and more.

To enable the interpretation of this diverse data, ROS includes a special variable known as type support, passed during the registration of topics or services. This type support variable is a structured entity that encapsulates crucial details regarding the type, dimensions, and field quantities present within the message. It is important to note that type supports solely encompass the structural information of the exchanged data, which remains consistent across all topics sharing the same name.

```
/// Structure used to describe all fields of a single interface type.
typedef struct ROSIDL_TYPESUPPORT_INTROSPECTION_CPP_PUBLIC MessageMembers_s
{
  /// The namespace in which the interface resides, e.g. for
  /// example_messages/msg the namespaces generated would be
  /// example_message::msg".
  const char * message_namespace_;
  /// The name of the interface, e.g. "Int16"
  const char * message_name_;
  /// The number of fields in the interface
  uint32_t member_count_;
  /// The size of the interface structure in memory
  size_t size_of_;
  /// A boolean value indicating if there are any members annotated as ‘@key‘ in the structure.
  bool has_any_key_member_;
  /// A pointer to the array that describes each field of the interface
  const MessageMember * members_;
  /// The function used to initialise the interface’s in-memory representation
  void (* init_function)(void *, rosidl_runtime_cpp::MessageInitialization);
  /// The function used to clean up the interface’s in-memory representation
  void (* fini_function)(void *);
} MessageMembers;
```

Listing 5.1: Struct containing type support informations in ROS.

Chapter 6

ROS middleware for DESERT

The central focus of this project lay in the development of RMW desert, which stands for ROS MiddleWare for DESERT. This creation enables Robot Operating System applications to engage in communication via acoustic wireless underwater means sans any alterations to their original code.

Thanks to the modularity of ROS, this objective was successfully reached through the creation of a package housing a fresh implementation of the rmw interface. That interface resides within the C++ header files, delineating all essential functions for a middleware layer.

It intercepts messages from the user land and dissects them into a series of fundamental data types, which are later serialized using a specific encoding method. In this particular project, the Coincise Binary Object Representation was the chosen encoding due to its high efficiency and minimal overhead.

Tasks are executed within the instances of diverse classes embodying publishers, subscribers, clients, and services, denoted in this context with the Desert- prefix, such as DesertClient.

The code can be found on github [11].

6.1 CBOR encoding

Coincise Binary Object Representation is a compact binary data format that boasts a succinct and efficient way to serialize and deserialize data structures. It is particularly beloved in the field of computer science for its ability to represent complex data in a concise manner, making it ideal for scenarios where bandwidth or storage space is limited like underwater channels.

CBOR follows a standardized format, allowing data to be encoded and de-

coded consistently across different systems and programming languages. It is a data format whose design goals include the possibility of extremely small code size, fairly small message size, and extensibility without the need for version negotiation [12].
The attributes of this encoding method are as follows:

1. the encoding is adept at unambiguously representing the majority of common data formats utilized in Internet standards;

2. the implementation of an encoder or decoder can be concise to cater to systems harboring minimal memory, processing capabilities, and instruction sets;

3. data can be decoded sans the necessity of a schema description; therefore, encoded data must possess self-descriptive qualities so that a generic decoder can be crafted;

4. the serialization process is reasonably compact, although the compactness of code takes precedence over the compactness of data for both the encoder and decoder;

5. the format is expansible, allowing for the incorporation of additional data that can be deciphered by earlier decoders;

6. the format is crafted with longevity in mind, intended to remain relevant and functional for decades to come.

6.1.1 Data structure

Encoded data presents itself as a sequence of data items. Each individual data item comprises a header byte housing a 3-bit type designation and a 5-bit short count. Subsequently, there may be an optional extended count, which comes into play if the short count is too small to contain the data dimension, alongside an optional payload. For data types unsigned integer, negative integer, and floating point number, no payload is present; the count itself represents the value. In the case of data types byte string and text string, the count signifies the length of the payload.

CBOR	**Data item**			
Byte count	1 byte (data item header)		Variable	Variable
Structure	Major type	Short count	Extended count	Data payload
Bit count	3 bits	5 bits	8 bits × variable	8 bits × variable

The interesting detail about this encoding is that all the data are sent only with the strictly neccessary number of bytes nedded for a certain information. Integer types do not contain any zero-padding, and for numerical values incorporating decimal points, three tiers of precision have been established: IEEE 754 half-precision, single-precision, and double-precision floating-point representations.

The minimalistic implementation for CBOR from Kyunghwan Kwon was used in rmw_desert, choosen for its lightness and efficency [13].

6.2 Packet structure

Packets are composed by variable length payloads containing CBOR-encoded informations, preceded by a header and closed by a tail. A predefined bit sequence marks the initiation of the packet, followed by another sequence denoting its termination. After the two fixed starting bytes, a field with the payload dimension is included, succeeded by the payload itself and by a single byte signifying the conclusion of the packet. Within the payload, the initial two fields consistently represent unsigned integers denoting the transmitting node type – be it a publisher, client, or service – and the respective topic or service identifier.

While ROS employs strings to manage topic names, a mechanism was implemented to mitigate the transmission of superfluous and bandwidth-intensive string data. This entailed the integration of a configuration file containing static mappings between names and integers. To ensure seamless communication, this configuration must be shared across all underwater network devices.

Starting Sequence	Starting Sequence	Payload Dimension	Stream Type	Stream Identifier	Remaining Payload	Ending Sequence
1 byte	1 byte	1 byte	1 byte	1 byte	n bytes	1 byte

6.3 TCP daemon

A class offering essential functionalities is TcpDaemon, utilized for establishing communication with the DESERT application layer socket, serving as the conduit between userland software and the network protocol stack.

An initialization procedure begins the middleware connection via a TCP stream to DESERT, subsequently creating threads for incoming and outgoing messages and detaching them prior to the return statement.

These threads operate continuously to guarantee comprehensive data management, thus justifying the classification as a daemon. Specifically, the receiving thread deposits received packets into a static packet queue, whereas the transmitting thread retrieves content from another static packet queue and transmits it to the socket.

Transmitting data is a straightforward process, requiring the insertion of three bytes as a header and an additional byte as a tail. Conversely, receiving data necessitates monitoring the stream until a valid header is encountered, verifying that the payload aligns with the specified dimension, and upon reaching the ending sequence, storing the packet in the queue.

At this point other classes can access those payloads as a sequence of bytes contained in static variables encapsulating CBOR-encoded fields. To streamline the conversion process between C++ data types and encoded fields, TxStream and RxStream classes were implemented.

6.4 Message serialization

Before sending data types down to the stream the message structure coming from nodes must be interpreted, in order to correctly locate the memory address of each field. The pointer passed through a function argument indicates the starting location of the entire message, and the middleware is responsible for splitting it correctly.

In the MessageSerialization namespace, there are methods that implement these functionalities based on the type support of each topic or service. Initially, the outer structure is scanned and can contain basic types or more complex ones, like strings. Each field can be single, an array, or a sequence of uniform data, and it needs to be handled using C or C++ structures depending on the implementation.

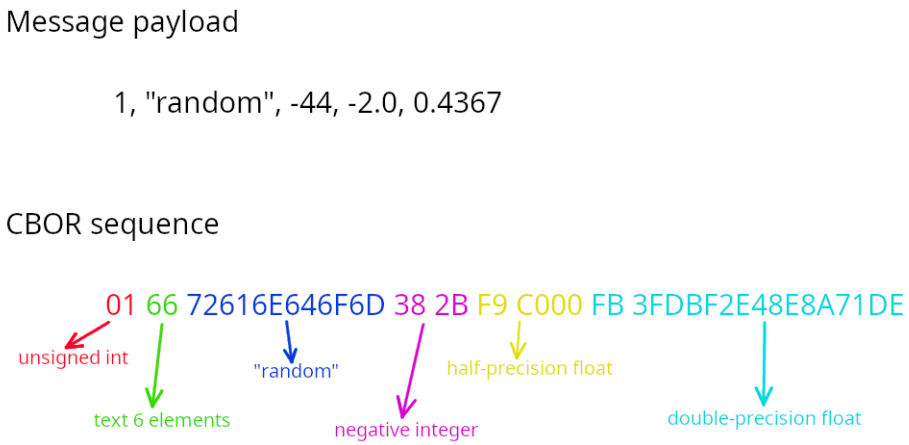


Figure 6.1: Example of message payload being encoded.

Figure 6.1 illustrates a serialized message with various data types. Small integers occupy one byte because the short count can contain the entire number. Larger integers necessitate more space, exemplified by the violet number in the provided example. Strings incorporate a payload field, denoted in blue, representing a character sequence with a length equivalent to the final five bits in the green data item header.

Additionally, floating point numbers adhere to a comparable structure. The F9 header signifies a 16-bit float, FA represents a 32-bit float, and FB designates a 64-bit float, also recognized as double precision floating point. The sequence is essentially a series of bytes, underscoring the significance of the header. It enables the encoding algorithm to discern the commencement and ending of each field, preventing confusion between them.

Certain data types such as booleans and UTF-16 strings are absent from the serialization standard, therefore the middleware transforms them into alternative types. Booleans are encapsulated within integers, adopting values of zero and one. On the other paw, u16strings, defined in the type support, are transmuted into plain strings by truncating characters outside the Unicode Transformation Format 8 bits encoding.

6.5 Middleware interface

During developement the initial stride involved extracting informations concerning the functions inside the headers. Each method had to be implemented, otherwise a runtime error would be thrown during the execution of programs on the top of rmw_desert.

Initially, all these functions were implemented in a skeletal form, devoid of any operational logic. Subsequently, those responsible for allowing data exchange were located and constructed using classes.

For instance, the method rmw_create_publisher goes about allocating memory to store a pointer to a new instance of the DesertPublisher class, complete with all the requisite details to identify and manage the topic. This process ensures that later, when a node endeavors to dispatch data to the topic, it forwards to the rmw_publish function the identical pointer containing the class instance along with all its members.

```
rmw_publisher_t * rmw_create_publisher(const rmw_node_t * node,
    const rosidl_message_type_support_t * type_supports, const
    char * topic_name, const rmw_qos_profile_t * qos_profile,
    const rmw_publisher_options_t * publisher_options)
{
  DEBUG("rmw_create_publisher" "\n");

  rmw_publisher_t * ret = rmw_publisher_allocate();
  ret->implementation_identifier =
   rmw_get_implementation_identifier();
  ret->topic_name = topic_name;

  DesertPublisher * pub = new DesertPublisher(topic_name,
   type_supports);
  ret->data = (void *)pub;

  return ret;
}
```

Listing 6.1: Function declared in the ROS interface implemented to create a publisher.

```
rmw_ret_t rmw_publish(const rmw_publisher_t * publisher, const
    void * ros_message, rmw_publisher_allocation_t * allocation)
{
  DEBUG("rmw_publish" "\n");

  DesertPublisher * pub = static_cast<DesertPublisher *>(
   publisher->data);
  pub->push(ros_message);

  return RMW_RET_OK;
}
```

Listing 6.2: Function declared in the ROS interface implemented to publish a message.

As you can see the publish method just executes a member procedure inside the class, where the logic that coverts and sends the message to the channel is present. Note also that during the instance creation also type supports are stored, in order to correctly interpret messages coming from the application.

6.6 DESERT entities

Within the middleware, a class is designated for each entity, encompassing all requisite functions for task execution and member variables for information storage. Each class features a get_type_support function, crucial for identifying whether the application employs C or C++ data types. This function initiates the extraction of members, a structure housing comprehensive details about the data formats, stored within a private member variable.

DesertPublisher only comprises the ´push´ public function, utilized to publish a message in the specific topic stored in the instance through the constructor, employing the serialization methods previously elucidated.

DesertSubscriber implements a ´has_data´ public function that enables the interpretation of packets stored in TcpDaemon, verifying the availability of data in the topic. If confirmed, the ´read_data´ method triggers the deserialize procedure, enabling the transfer of content to the appropriate memory location.

DesertClient and DesertService implement all functions previously outlined, as the communication in this scenario is bidirectional. The client initiates a request, which the service reads and generates a response to be sent back to the client. Additionally, two categories of members exist within these entities: one for the request message structure and another for the response.

Chapter 7

NS simulation results

The final phase of this project consisted in the integration of the DESERT protocol stack with the Robot Operating System through the middleware proposed in this study to assess its functionalities and suitability for real-world applications.

This integration was made possible by ns2, a discrete event simulator tailored for networking research, offering extensive support for simulating TCP, routing, and multicast protocols across networks. Essentially, all protocols detailed in Section 4.1 were run within this local simulation and subsequently linked at the application layer with rmw_desert.

The simulations employed Turtlesim as the ROS application operating on the top of rmw_desert, exchanging data through DESERT. The node, represented as a graphical interface featuring a turtle, serves in this context as an analogy to an underwater drone. Instead, the teleop key functions as a control interface, maneuvered by an external human operator to receive real-time robot position data and send navigational directives.

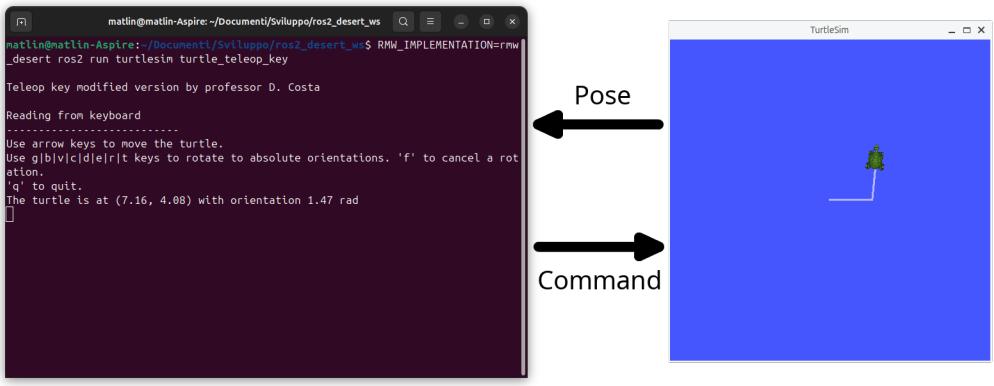


Figure 7.1: Simulator windows with the controlling interface and the node.

Two distinct testing approaches were employed by changing the physical layer: one involving mathematical simulations based on Signal-to-Noise ratio threshold computation, and the other utilizing EvoLogics underwater modem emulators. Diverse results were achieved, primarily highlighting protocol issues unrelated to ROS.

7.1 UW/Physical

The initial simulation leveraged UW/PHYSICAL module parameters to manipulate the node distances and environmental variables like wind speed. Various configurations were tested, including scenarios with and without acknowledgments and wind speeds ranging from zero to fifteen meters per second, to elucidate the significant impact of this variable.

The study focused on the Packet Delivery Ratio of ROS data at different distances while keeping wind speed constant, resulting in the graphical representations below.

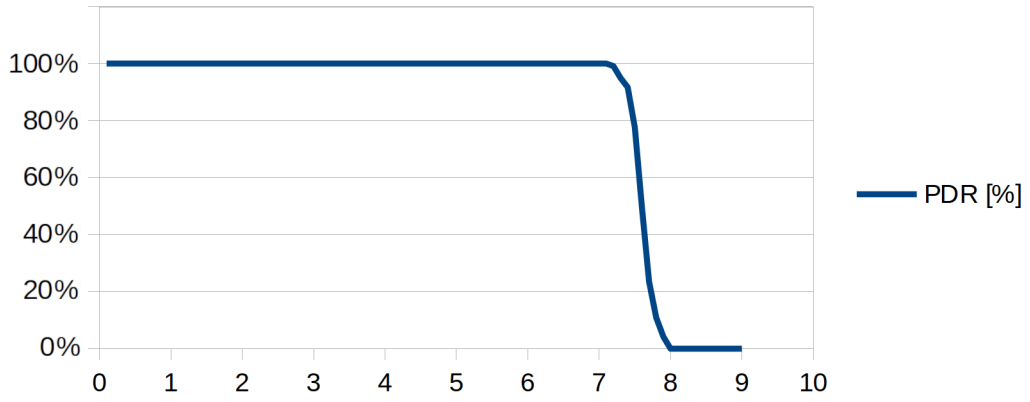


Figure 7.2: Simulation with no wind.

Under ideal conditions with no wind, all packets were successfully received until nodes were separated by less than 7 kilometers. Subsequently, the PDR rapidly declined, leading to complete packet loss after one kilometer. It must be noted that in realistic settings, sustaining a constant maximum PDR indefinitely is highly improbable due to the interferences of the environment that are not considered by this model, preventing such ideal performance.

Introducing a wind speed of five meters per second caused the PDR curve to decline earlier by two kilometers, while other characteristics remained stable. Further testing revealed a consistent behavior across varying conditions. A more

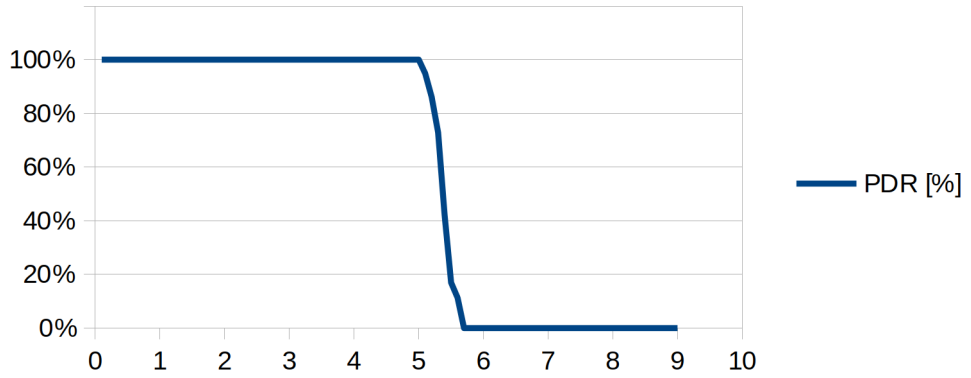


Figure 7.3: Simulation with a wind of five meters per second.

realistic scenario was observed with a wind speed of ten meters per second, where a threshold of four kilometers appeared plausible, with no packet loss evident before this distance.

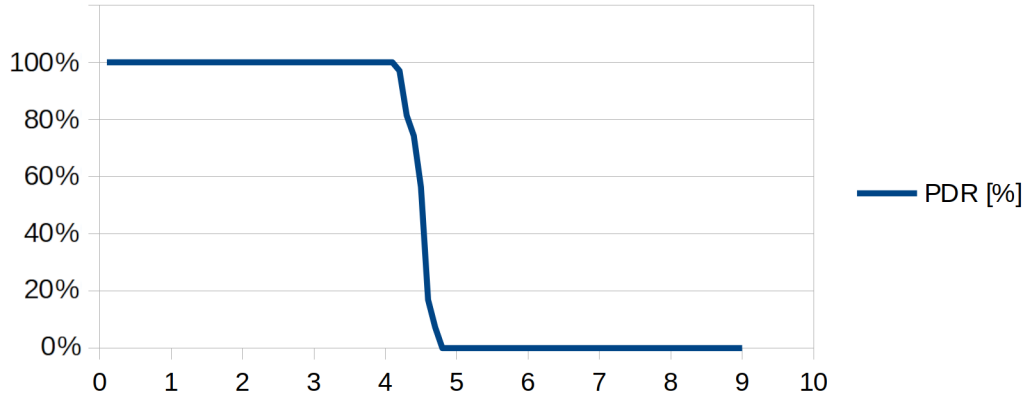


Figure 7.4: Simulation with a wind of ten meters per second.

The conclusion of the testing evidences that in all greather-than-zero PDR situations the ROS application tutorial Turtlesim worked as expected, successfully executing received command packets and remaining idle in cases of packet loss.

It is noteworthy that the blocking of unnecessary topics at the middleware level played a crucial role in mitigating the substantial volume of traffic generated by the application. This underscores the importance of developing software tailored for underwater communications taking into account the limited bitrate capabilities. Hence, it is imperative to ensure that any program interfacing with DESERT generates an appropriate quantity of packets suitable for transmission through the acoustic channel.

Another identified inefficiency arose when activating the stop-and-wait ac-

knowledgment mode, attributable to the extended latencies stemming from the reduced speed of sound propagation in water. The practice of waiting for individual ACKs before transmitting subsequent packets significantly impeded the overall data transfer rate, rendering this mode impractical for real-world applications.

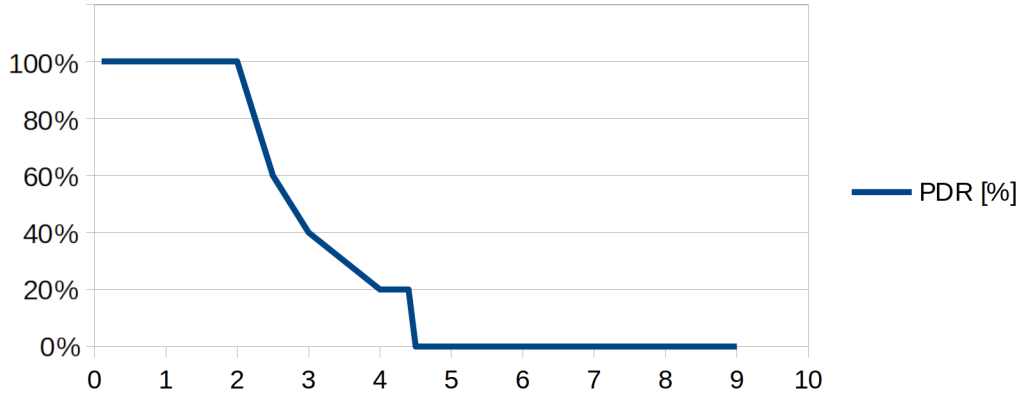


Figure 7.5: Simulation with acknowledgements enabled.

This approach not only led to a rapid decline in PDR but also resulted in an unacceptably low volume of transmitted packets. A potential workaround could involve bundling multiple packets and awaiting a collective ACK; however, it is advisable to steer clear of this mode for optimal performance in underwater communication settings.

7.2 UW/UwModem/EvoLogicsS2C

In order to simulate the behavior of a real modem, the UW/UwModem/EvoLogicsS2C module was utilized as the physical layer to connect with EvoLogics modem emulators, while keeping the other layers unchanged.

No changes were made to the middleware or the ROS application, except for adjusting the NS configuration to accommodate the functions required by the new module. One notable addition was the implementation of a special library called ´packer´, which includes instructions for compressing and converting the header of a specific protocol into a bit stream and vice versa [14].

In the initial test, only the topic /turtle1/cmd_vel was activated to enable teleoperation control over the turtle's movement through one-way communication. The outcomes were positive: the first node successfully transmitted the packet through the channel, and the second node received it accurately.

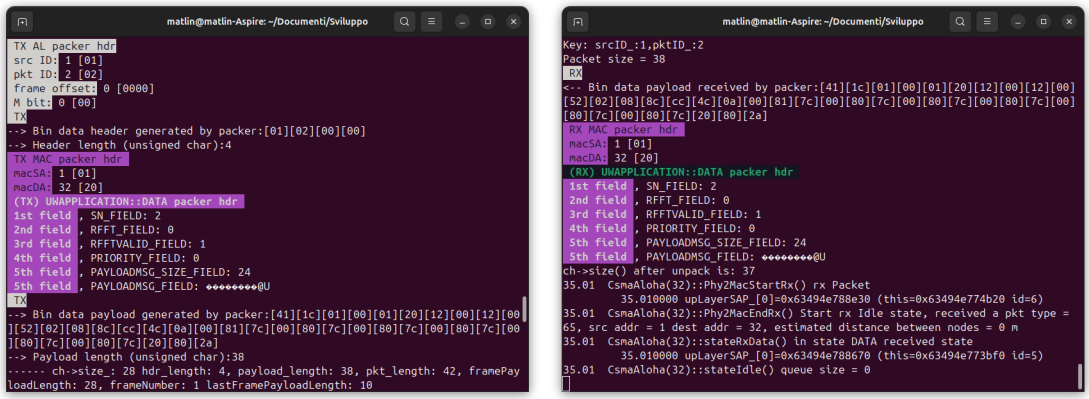


Figure 7.6: Output of the Network Simulator.

Within Figure 7.6 's left section lies the diagnostic output from NS linked to the teleop key application, showcasing the binary header produced by the 'packer' alongside the payload containing control data. Furthermore, numerous additional fields are included to allow proper packet routing throughout the protocol stack to the intended destination.

On the right terminal, the displayed output is connected to Turtlesim, which receives messages from the EvoLogics emulated modem and transfers them to higher layers via the CSMA ALOHA module. In this scenario no collisions occurred due to the unidirectional communication.

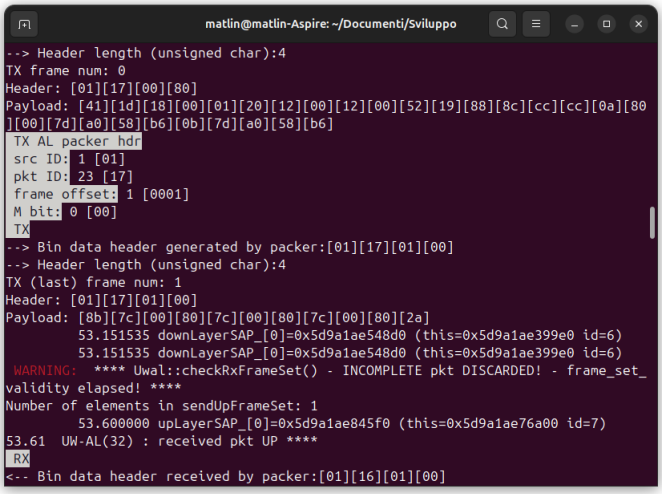


Figure 7.7: The Adaptation Layer received an incomplete packet and discarded it.

Regrettably, upon attempting to activate the position topic that transmits data in the opposing direction, packets originating from the teleop key arrived incomplete and were consequently discarded. This led to the inability to control

the turtle, a challenge that perturbed other developers grappling with the same protocols.

High frequency of packet collisions in the underwater channel likely contributes to the incomplete data reception, as evidenced by the warning in Figure 7.7. To investigate this phenomenon, an alternative Medium Access Control protocol was tested in combination with the EvoLogics physical layer, whose results are detailed in the subsequent section.

7.3 EvoLogicsS2C and TDMA

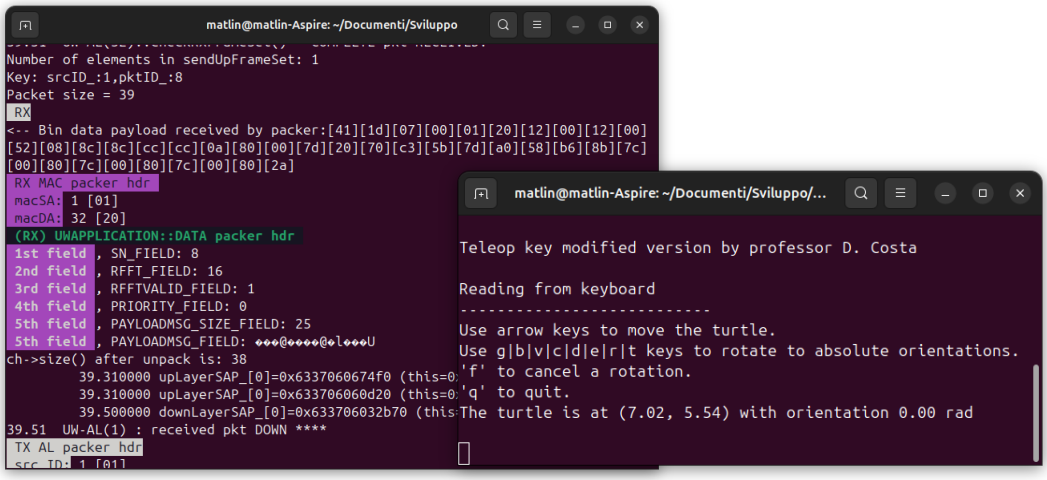
This last NS configuration uses a module which implements the Time Division Multiple Access multiplexing technique, that allows multiple users to share the same frequency channel by dividing the signal into different time slots. Each node transmit its data at a specific time slot, ensuring that it does not interfere with other nodes.

The UW/TDMA module was set to create two slots with a frame period of one second, so that each node waits untils the whole channel is reserved for it, preventing collisions. This adjustment successfully resolved the issues encountered with CSMA ALOHA and the bidirectional communication worked as expected.

```
            Node 1                          Node 32
  +-----------------------------+     +-----------------------------+
  |  8. UW/APPLICATION          |     |  8. UW/APPLICATION          |
  +-----------------------------+     +-----------------------------+
  |  7. UW/UDP                  |     |  7. UW/UDP                  |
  +-----------------------------+     +-----------------------------+
  |  6. UW/STATICROUTING        |     |  6. UW/STATICROUTING        |
  +-----------------------------+     +-----------------------------+
  |  5. UW/IP                   |     |  5. UW/IP                   |
  +-----------------------------+     +-----------------------------+
  |  4. UW/MLL                  |     |  4. UW/MLL                  |
  +-----------------------------+     +-----------------------------+
  |  3. UW/TDMA                 |     |  3. UW/TDMA                 |
  +-----------------------------+     +-----------------------------+
  |  2. UW/AL                   |     |  2. UW/AL                   |
  +-----------------------------+     +-----------------------------+
  |  1. UW/UwModem/EvoLogicsS2C |     |  1. UW/UwModem/EvoLogicsS2C |
  +-----------------------------+     +-----------------------------+
           |         |                       |         |
  +-------------------------------------------------------------------+
  |                         UnderwaterChannel                         |
  +-------------------------------------------------------------------+
```

Above the complete protocol stack employed in this test, with the first node operating turtlesim assigned to address 1, and the subsequent node running the

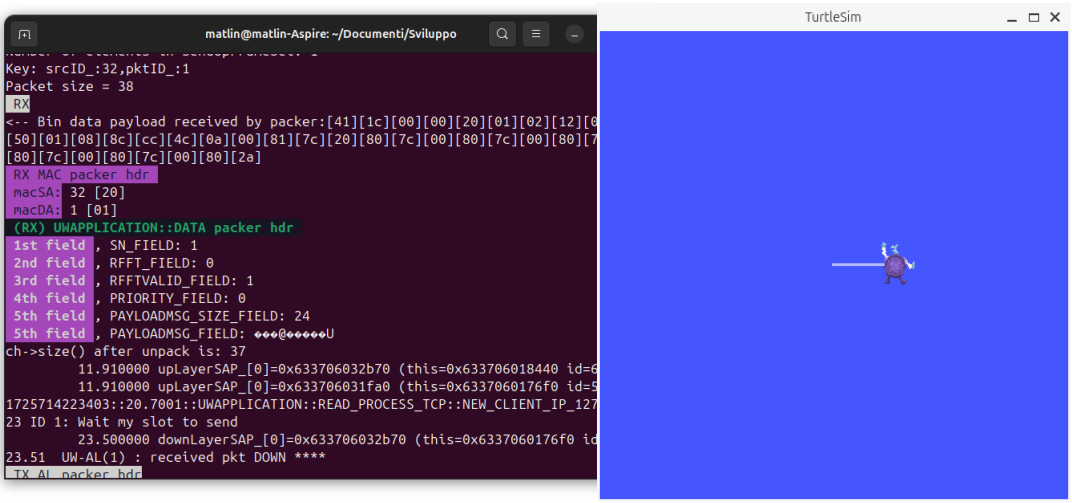
teleop key designated to address 32. It is important to note that no alterations were made to any layers beyond TDMA in this setup.



(a) (b)

Figure 7.8: Teleop key received a packet.

In Figure 7.8a, we observe a snapshot representing a packet transmitted by turtlesim via rmw_desert on the ´pose´ topic and successfully received by teleop key, with pertinent details stored in the PAYLOADMSG field. The presence of random characters is a result of the binary encoded stream being directly converted to a char sequence, which deviates from the expected CBOR encoding. However, Figure 7.8b the correct interpretation of the pose is displayed by the application.



(a) (b)

Figure 7.9: Turtlesim node received a packet.

Conversely, Figure 7.9 showcases a control packet originating from teleop key and received by turtlesim, effectively directing the turtle to the intended orientation. This outcome serves as validation for the accuracy and functionality of the bidirectional communication provided by the current modules configuration in real world scenarios.

Chapter 8

Conclusions

This thesis introduces a middleware application designed for the Robot Operating System, enabling the remote operation of automated systems via an underwater communication channel. The application serves as a bridge, allowing the integration of such systems with the DESERT protocol stack, that autonomously manages the network infrastructure without necessitating any code modifications. Tests conducted with the EvoLogics underwater modem simulator validate the readiness of rmw_desert, the middleware component, for practical deployment in real-world scenarios, contingent upon the establishment of a valid network configuration. In fact the assessments revealed inherent structural issues within certain protocols that impede efficient data exchange.

Specifically, observations indicate that the tutorial application Turtlesim successfully operated using the TDMA module at the MAC layer in conjunction with the EvoLogics physical interface, whereas protocols like CSMA ALOHA encountered challenges due to collision-related issues. This underscores the efficacy of the final simulation proposed in the thesis for underwater environments interfacing with ROS-controlled systems. Future enhancements entail the implementation of some empty methods in the rmw interface used to get informations about events, quality of service, count of nodes and topics and service availability. While these functionalities were not indispensable for basic communication, their incorporation is vital for the operation of specific features, so a continuation of the development and refinement of the system architecture is suggested.

Another aspect requiring further development pertains to simulating the system using position-varying nodes, enabling the software to dynamically adjust the coordinates of devices to replicate the authentic movement of an underwater drone. This enhancement is essential for assessing the propagation-related effects

on target control, a factor of significance given the slower velocity of acoustic waves in comparison to conventional radio waves. Additionally, extended trials must be conducted to ensure the system's reliability over prolonged operational durations, mitigating the risk of unintended communication disruptions. Finally, a practical evaluation with hardware modems deployed in the sea would serve as the ultimate validation of the system's functionality.

References

[1] M. Schwartz, J. F. Hayes, *A history of transatlantic cables*, October 2008.

[2] N. Tang et al 2020 J. Phys.: Conf. Ser. 1617 012036, *Research on Development and Application of Underwater Acoustic Communication System.*

[3] EvoLogics GmbH, *Modems using acoustic waves to perform underwater communications*, `https://www.evologics.com/acoustic-modems`.

[4] E. Coccolo, R. Francescon, F. Campagnaro, M. Zorzi, *Field Tests of the Software Defined Modem Prototype for the MODA Project.*

[5] A. Montanari, F. Marin, V. Cimino, D. Spinosa, F. Donegà, D. Cosimo, L. Bazzarello, D. Natale, F. Campagnaro, M. Zorzi, *Experimenting Various JANUS Frequency Bands with the Subsea Software-Defined Acoustic Modem.*

[6] N. Baldo, F. Maguolo, M. Miozzo, M. Rossi, M. Zorzi, *ns2-MIRACLE: a modular framework for multi-technology and cross-layer support in network simulator 2*, October 2007.

[7] R. Masiero, S. Azad, F. Favaro, M. Petrani, G. Toso, F. Guerra, P. Casari, M. Zorzi, *DESERT Underwater: an NS-Miracle-based framework to Design, Simulate, Emulate and Realize Test-beds for Underwater network protocols*, May 2012.

[8] NS2 soft solution, *A discrete event network simulator based on TCL scripting languages*, `https://networksimulator2.com/`.

[9] Open robotics, *A set of software libraries and tools that help you build robot applications*, `https://www.ros.org/`.

[10] M. Stojanovic, *On the Relationship Between Capacity and Distance in an Underwater Acoustic Communication Channel*, November 2007.

[11] D. Costa, *ROS middleware implementation for the DESERT underwater communication protocol*, `https://github.com/dcostan/rmw_desert`.

[12] Internet Engineering Task Force, *RFC 8949 - Concise Binary Object Representation (CBOR)*, December 2020.

[13] K. Kwon, *Minimalistic implementation for CBOR, the Concise Binary Object Representation*, `https://github.com/libmcu/cbor`.

[14] G. Toso, I. Calabrese, F. Favaro, L. Brolo, Pa. Casari, M. Zorzi, *Testing Network Protocols via the DESERT Underwater Framework: the CommsNet'13 Experience*.