Università degli Studi di Padova

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

University of Padua, Department of Information Engineering
Bachelor's degree in Computer Engineering

Designing, implementing and testing a ROS 2 operating system for underwater wildlife monitoring systems

Pasetto Niccolò

ID 2066695

Supervisor

Prof. Varagnolo Damiano

University of Padua

Stage Supervisor

Fusari Riccardo

University of Padua

Academic Year 2024/2025

"error: invalid cast from type
'custom_msgs_<std::allocator<void> >::_box_type'
aka 'std::array<float, 4>'
to type 'std::array<float, 4>'"
I guess this is the kind of error message one has to overcome in order to complete his/her thesis.

Contents

Abstract

This thesis is part of a smaller project called "*Sensing Rigs*", a subgroup of the student association "*Nautilus*". The goal of this project is to design, implement and test an underwater photo trap for monitoring the underwater wildlife. To develop such a complex system we divided it into components, and every one of them was assigned to a different student, who developed it as his thesis for a total of 6 students working on the "Sensing Rigs" project.

However, the sub-components we identified to develop were three: Computer Vision (CV) algorithms (for detecting the underwater wildlife) composed of three students, Robotic Operating System (ROS) 2 operating system (for integrating the algorithms with the hardware) composed of two students, and communication management (to make the system communicate with the exterior) composed of one student.

I choose the task of implementing an operating system using ROS 2 as framework, in cooperation with another student. I made this choice because I prefer the topic over the others, and I also like the idea of creating something that can be useful to others while learning something new. We had the duty to integrate the works of the others into a fully functional operating system, for this reason we had to interact a lot with the other sub-groups in order to satisfy their requirements.

1 Introduction

1.1 What's ROS2

To build the source code for our project, we used Robotic Operating System (ROS) 2, which is not an operating system itself, as the name suggests, but an open-source robotics middleware suite. Basically, it is a set of libraries and tools for the development of robot software.

Its first version, ROS 1, came out in 2007 and since then a lot has changed; from new features of programming languages to new standards in the robotics community. For this reason, under the maintenance of Open Source Robotics Foundation (OSRF), a new major version was announced at *ROScon 2014* [1] with the aim of renovating existing Application Programming Interface (API) and introducing new technologies, such as *real-time programming*, Data Distribution Service (DDS), *WebSockets*, to name a few...[1]

Since then, OSRF continued to publish new releases of ROS 2, integrating new changes and extending its support to older versions and newer hardware.[2]

[1] https://roscon.ros.org/2014/program/

[2] https://github.com/ros2/ros2/releases

1.1.1 Versions

Currently, the latest long term support version is named *Jazzy Jalisco*, released on 23 May 2024 with end-of-life support until May 2027 (ignoring the rolling release, which is a development one).
However, the version used by Nautilus for their project is the previous, named *Humble Hawkshell* released on 23 May 2022 with end-of-life support until May 2025, because at the time that was the main version. In the first moment, we planned to use *Humble* for our new project, to follow the Nautilus standards.

However, we decided to move to *Jazzy* not only for the extended support, but also because in our freshly purchased hardware *Raspberry Pi 5* the version of the operating system required by ROS 2 is no longer supported. To be more precise, Jazzy requires *Ubuntu Noble 24.04*, while ROS 2 Humble requires *Ubuntu Jammy 22.04*. Unfortunately, the latter cannot run on a Raspberry Pi 5 without adding unnecessary complications to the system. In addition, this new version of ROS 2 will slowly become the new standard over the next three years.

For these reasons, we decided to move to the latest stable release, which means that the older projects of Nautilus should be updated to meet the new requirements. Fortunately, a part from the operating system version required, Jazzy did not overturn the code too much. In fact, the versions used for *Python* changed from *3.10* to *3.12* while for *C++* they started to add support for the new standard *C++ 23*. To make the code compilable by Jazzy, it is sufficient to revolve the dependencies and fix the new warnings,

which derive from the newer programming language versions.

In addition to this, they basically introduced new features, making some tools easier to use. Here is a brief list of some changes that have proven useful for our project (for a more detailed one[3]):

- It is now possible to specify custom settings Quality of Service (QoS) for the nodes.
- Added a new class **ParameterEventHandler** to monitor / respond to changes to parameters via parameter events.
- It is now possible to record and play service data with the **ros2bag** command line interface.

1.1.2 ROS2 main concepts

Before explaining the design and development process of the operating system, I will explain some key concepts of ROS 2.

Nodes

Nodes are the main components of a ROS system and typically the unit of computation. Each should perform a specific task so that a modular system is created. To retrieve the needed inputs and share the outputs usually they are a complex combination of publishers, subscribers, service servers, service clients, action servers, and action clients; all at the same

[3]https://docs.ros.org/en/jazzy/Releases/Release-Jazzy-Jalisco.html#new-features-in-this-ros-2-release

time. In our case, we had to handle and respond to a series of events, like sending a pair of images after a pre-processing phase, or sending the results of the computer vision algorithms. For this reasons we decided to use a publisher / subscriber model, meaning that one node acts as a publisher, sending out data to a *topic* when ready, while another one acts as a subscriber, retrieving the data published on the *topic*. One topic can have multiple subscribers, but only one publisher can publish to a given topic. The discovery of nodes occurs automatically through ROS 2 middleware, where each node sends messages to the underlying network informing the others of their status and parameters to establish a connection.

Topics

Topics can be seen as the "channels" where information / messages are shared between nodes. They act as a continuous data source, and each one should carry a specific message (like sensor data, connection state, etc.). They don't need to have point-to-point communication, it can be one-to-many or many-to-many. In our system, every topic corresponds to a specific output type, such as raw results from the computer vision nodes, formatted data in the format JavaScript Object Notation (JSON), or raw data from the cameras.

Actions

Actions are a long-running remote procedure call, as there is overhead in setting up and monitoring the connection, with feedback and the ability

to cancel or preempt the goal. However, we did not utilize this component in our project because there is only one main action to perform: analyze the input frames. The system does this in a loop by default, so specifying it as an action would be useless.

ROS bags

A ROS *bag* is a file format, with extension *'.bag'*, for recording ROS messages on a selected topic. There are a lot of tools to store, process, analyze and even visualize them; to simulate a real-time data flow, there is even an option to add a *simulated clock* that corresponds to the time the data were recorded in the file. The bags can be played back on the same topic they were recorded on, but can also be remapped to others, making it easier to share data across different systems. In our project, we used these features to store relevant messages (such as when something shows up on the cameras) for future analysis.

1.2 Development environment

Before testing the whole system on physical hardware, the group needed a uniform development platform. With the help of a member of the ROS 2 subgroup (TODO: should I name him?), we created a custom *Docker* container for this purpose.

Docker is an open-source virtualization platform that allows developers to build, deploy, and manage applications in portable containers. Instead of virtualizing the whole operating system as a virtual machine would

do, it virtualizes all the components needed by the application to build and run the source code. In this way, a package with the development environment is created through one or a series of configuration files, named *"Dockerfile"* and *"docker-compose.yml"*, which tell the underlying deamon to perform a series of actions to build the application. To share this *container* all is needed is to share these configuration files and the source code; the Docker deamon will handle the creation of a uniform container.

We built our custom container from *Ubuntu Noble 250404* as the base image, as discussed section 1.1.1. Then we installed some additional software for development, such as: ROS2, a web browser, the preferred code editor and *noVNC*. The latter is an open-source Virtual Network Computing (VNC) browser client, built with Hyper Text Markup Language (HTML) 5, WebSockets and Canvas, which can connect to a remote desktop, in this case our Docker container, from any modern web browser. This graphical desktop sharing system, provided by VNC, was needed because ROS 2 has a lot of graphical interface tools (from *Gazebo* simulations to image and graph rendering) and using them via a command line interface was unthinkable. Additionally, we implemented a series of scripts to automatize the set up processes of the system, in this way with a command we can start a new container and build inside of it our ROS 2 code. Finally, we made sure that Docker copies a local directory with the source code to the newly created application.

In this way, we created a uniform development platform for ROS 2 Jazzy that depends only on Docker and a modern web browser, making

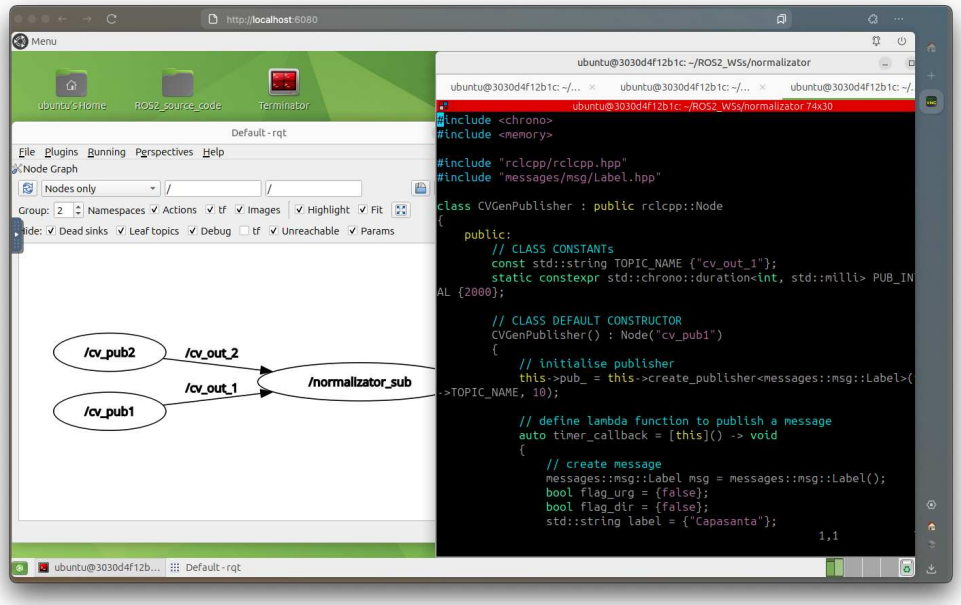

Figure 1.1: Development environment on Docker with noVNC on web browser

it easier for us developers to share and test our code, without the need for additional hardware.

1.3 Physical setup

In our first setup, we tried to use a *Nvidia Jetson Nano* as our main computational board, because it is a small, powerful and energy-efficient computer developed by *Nvidia*, built especially for Artificial Intelligence (AI) computing and image processing. However, we soon find out that our cameras, *Raspberry Camera module 3*[4], were not compatible with the Jetson, because they were "to new" for our old board.

To overcome this incompatibility, we had to add a second board to

[4]https://www.raspberrypi.com/products/camera-module-3/

the setup: the Raspberry Pi 5 discussed in section 1.1.1, since they were compatible with the cameras. The idea is that the Raspberry Pi 5 is responsible for the image acquisition part through the cameras, while the Jetson is responsible for the image processing part, since it is more efficient. TODO To make the image available from the Raspberry to the Jetson, we had to set up an internal wired network (via ethernet cable) between the two boards. After adding some Network Address Translation (NAT) rules and specifying how to resolve specific hostnames, the two boards could communicate with each other. We also took advantage of the fact that when some data is published on a ROS 2 topic, every device connected to that same network can access it, making it easier to debug and test. In addition, each of the two boards had a heatsink with a 5 [V] fan attached to it, to properly dissipate heat; and a power bank to supply enough power for some hours.

However, similar to the problem of section 1.1.1, the Jetson was too old for the newest ROS 2 version and we could not install directly *Ubuntu Noble 24.04* on it. But in this case we were prepared: we already had a Docker image, described in section 1.2, which made able to create a container with ROS 2 Jazzy inside any operating system. We just had to add some more options, such as accessing the hardware from within the container. In the end, our Docker image became from a universal development option to an essential part of our system because we could execute the same ROS 2 code from basically every board we wanted.

The last component this setup needed to monitor the underwater wildlife was a waterproof enclosure to host all the other components in a

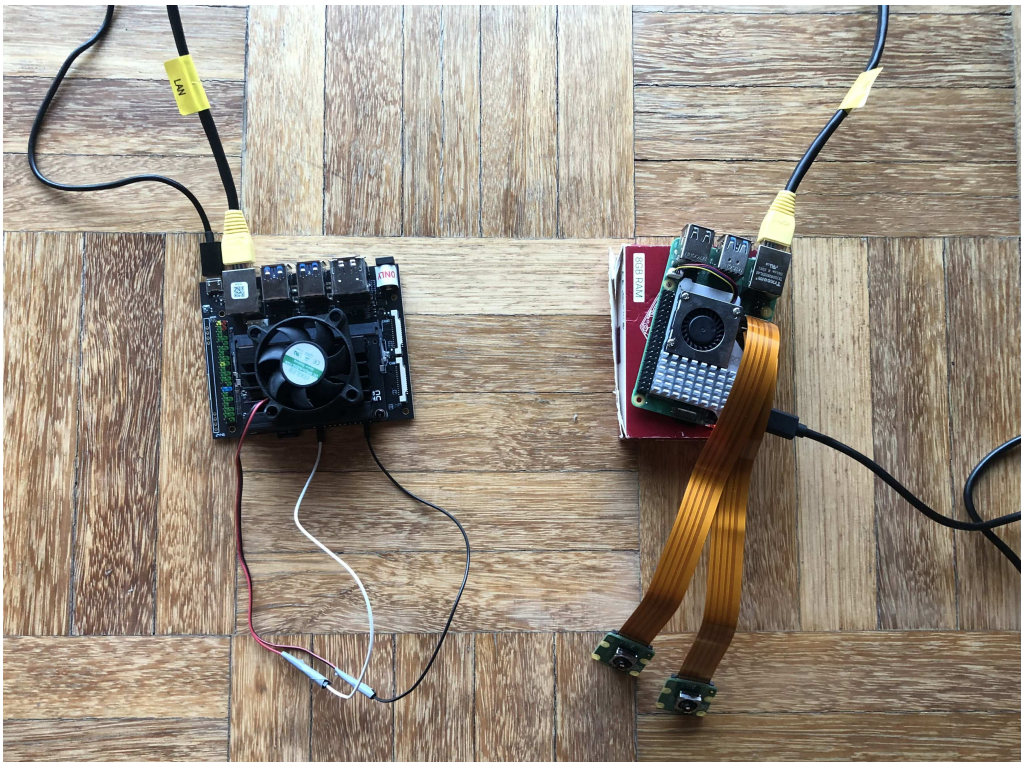

Figure 1.2: Physical setup wired together: on the left the Jetson Nano, on the right the Raspberry Pi 5 connected with the Module 3 cameras

single box. Fortunately, one member of the Nautilus team (TODO: should I name him?) developed and built this last piece of hardware, thanks to a 3D printer. However, this introduced two limitations. The first one being that he could not print whatever we liked, because the 3D printer had some size limits. For this reason, in the biggest box he built we could not fit the two boards, but only one (which we choose to be the Raspberry Pi 5, since it can access directly the cameras). He added some supports for the cameras, with a baseline of 12 [cm], and also provided 2 holes on the box's cover to pass through 2 cables: one to power the board and one ethernet to give wired connection to the board. The second was that, unfortunately, the box and its cover after the waterproof treatments were not completely waterproof, and some water could leak from the joints. For this reason, we were unable to completely submerge the system as originally intended without damaging the electronic components. In a future iteration, a better waterproof box can be designed and used, giving more space to the Jetson Nano, which can handle the computing part more efficiently, and other sensors that can monitor different parameters of the ecosystem.

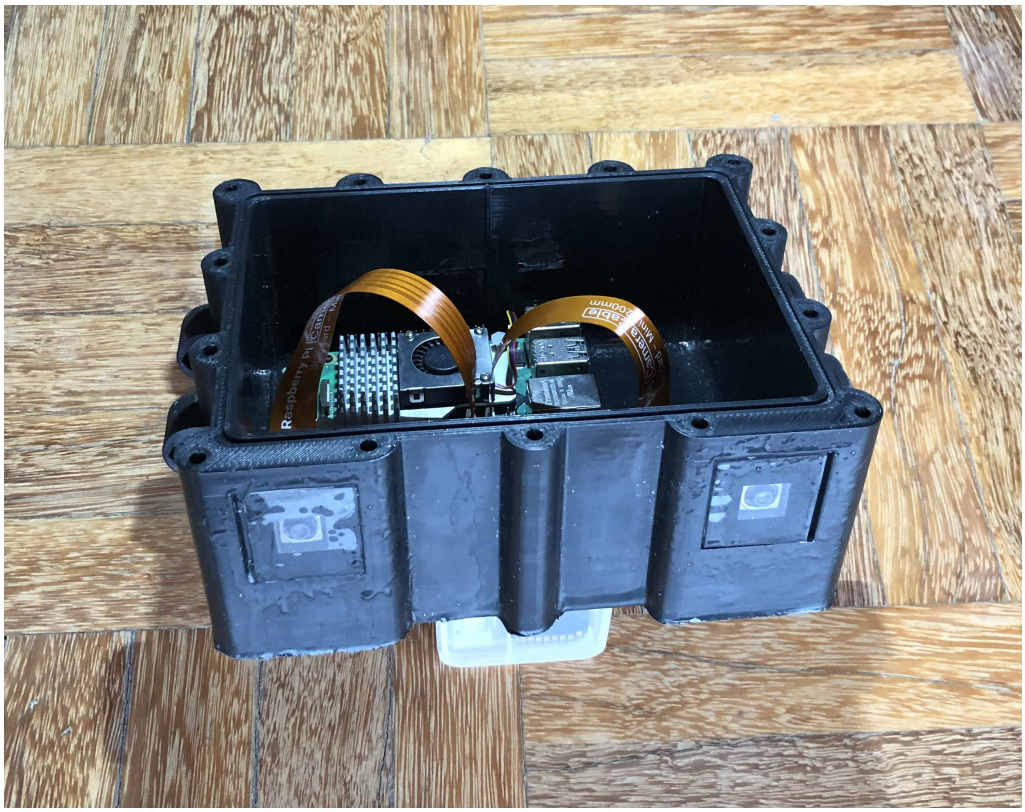

Figure 1.3: Final configuration used for the sysetm testing

2 System design

Before writing the code for the nodes, we had to define how these nodes should communicate with each other. In other words, we had to design the whole system before writing random code. We decided to hold a brainstorming session with all the subgroups to get an idea of where to start. The subgroups that we involved were:

- **Computer Vision (CV) algorithms**, responsible for analyzing the images captured by the cameras and producing useful outputs, such as image recognition and visual odometry.
- **Communications**, responsible for sending the results through an acoustic modem to the exterior.

Each subgroup had different needs, and our goal was to provide them with a consistent operating system that could capture images from the cameras, pass them to the CV nodes, take the different outputs and communicate them to the exterior through the modem. Thanks to this first description, we could already abstract three main components for our system, which we could design, build and test separately:

1. Image acquisition node.
2. CV nodes.
3. Modem communication node.

When every piece was ready, we joined them together, creating a first version of our system. Below are reported the design phase for each one of these 3 abstraction units.

2.1 CV nodes

These nodes are responsible for taking in input the images from the cameras and analyzing them. They can be considered the "brain" of our system because they require the most computational power to produce results on the underwater environment. The algorithms we had to integrate into our system were three, and they were independently developed by three different students from the CV subgroup as their bachelor's thesis. Here is a brief description of what they do:

- **Mono vision image recognition**, attempts to recognize an underwater animal through one camera, thanks to some AI model trained to recognize a specific underwater life form (the first model was trained to detect crabs). As output, it creates a box around the detected animal with a label and a confidence number on how accurate the recognition was. This node requires only one stream of images and has no limitations on the input Frames Per Second (FPS). However, the higher, the best are the results.
- **Stereo vision image recognition**, as the first algorithm, but works in stereo vision (the first model was trained to detect scallops). However, as output it does not add the detection box, but simply labels the frames with the name of the detected animal. This requires two streams of images and has no limitations over the FPS in input.

- **Visual odometry**, attempts to track the trajectory of the moving system due to translations and rotations detected by the cameras. Since our system is supposed to remain still and monitor underwater wildlife, this algorithm is useless. However, only for this iteration, we could move the box with the hardware inside only to test the algorithm. (TODO: should I mention it was moved to auto-docking mid phase?) This one cannot work in mono vision, as it requires two frames of the same image (left and right) to calculate a depth map. As output, it produces a 4x3 matrix of doubles, describing the rotation and translation changes from the previous frames. This requires two streams of images, and to produce an accurate estimation, it requires at least 10FPS. If this limit is not met, the output may look inconsistent and inaccurate.

The initial idea was to assign every CV algorithm to a node and use ROS 2 subscriber model to handle I/O. A subscriber to the camera topic would retrieve the frames, while a publisher would communicate the results to the modem. However, since ROS 2 topics are strictly typified and can communicate messages of only one type, we cannot use a common topic for the output, because every algorithm has its own output format. Having a common output would mean a less complex system, but to avoid this problem we had to define an output topic for every node (three in total) with the following naming convention: "*/cv_outs/algorithm_name*". This initial solution may not be the most efficient one, but it is a solid and modular approach to lay the foundations of this project. A fourth CV algorithm can be easily added to the system, and there is room for optimization in a second moment.

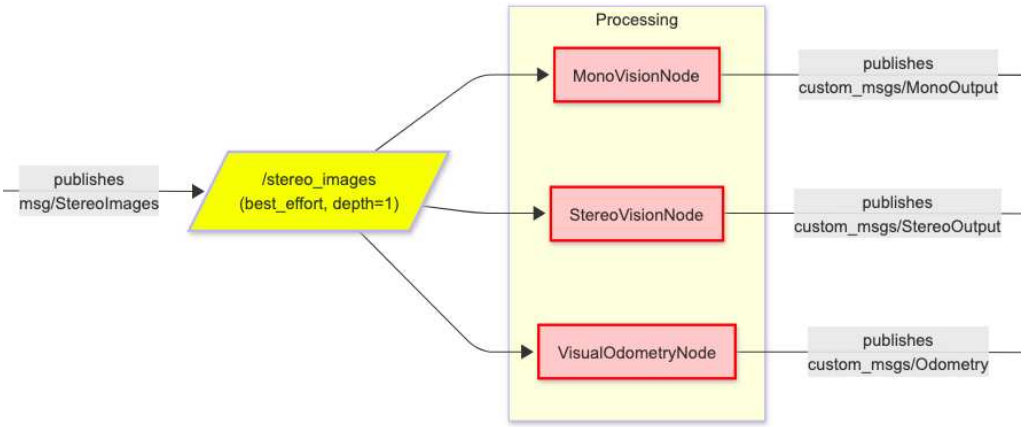

Figure 2.1: Diagram showing how the CV nodes communicate

2.2 Image acquisition

The images are acquired by the two cameras directly connected to the Raspberry Pi 5, and the raw images are published to the topics */camera1/image_raw* and */camera1/image_raw* using ROS 2 default message type for images: */sensor_msgs/Image*. However, before sending these to the CV algorithms, we performed some pre-processing operations:

- **Image compression**, introducing a compression algorithm (Joint Photographic Experts Group (JPEG)) we significantly reduced the size of the acquired frames, without modifying the output of the algorithms.
- **Image synchronization**, since the stereo algorithms require one left image and one right image (also called a "*stereo image*"), we make sure that these two are captured on the same small time interval, avoiding desynchronization problems for the algorithms, which is not optimal.

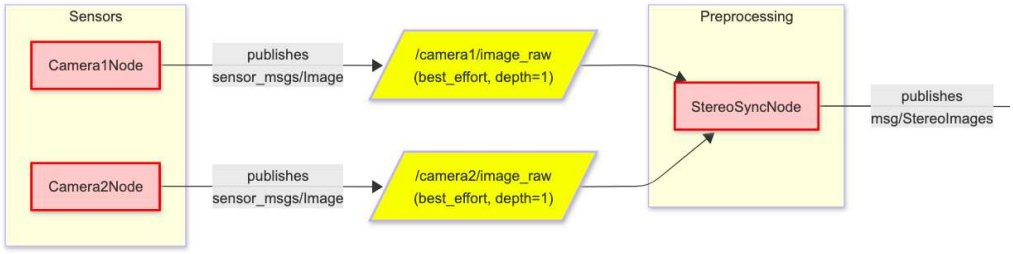

Figure 2.2: Diagram showing the image acquisition and pre-processing phase

The result of this "*PreprocessingNode*" is a custom ROS 2 message type with the stereo image and the relative timestamp, named "*/msg/StereoImage*", which facilitates the analysis performed by the CV algorithms. The output topic for after this phase is called */stereo_images*, and it will be accessed by the algorithms.

2.3 Modem communication

Complying with requests from the modem subgroup has been more difficult than the other. Since the system is designed to be underwater and the only communication with the exterior is provided by an acoustic modem, we cannot send too much information because it has high latency and high error rate. For example, sending frames analyzed by the CV algorithms was too demanding and considered inefficient by the subgroup. Another requirement was that the different messages received by the modem should be in the same format, in this way the modem has to worry only on receiving the inputs and retransmit them to land. The required format was JSON.

To comply with the first request, we decide to send only the results of

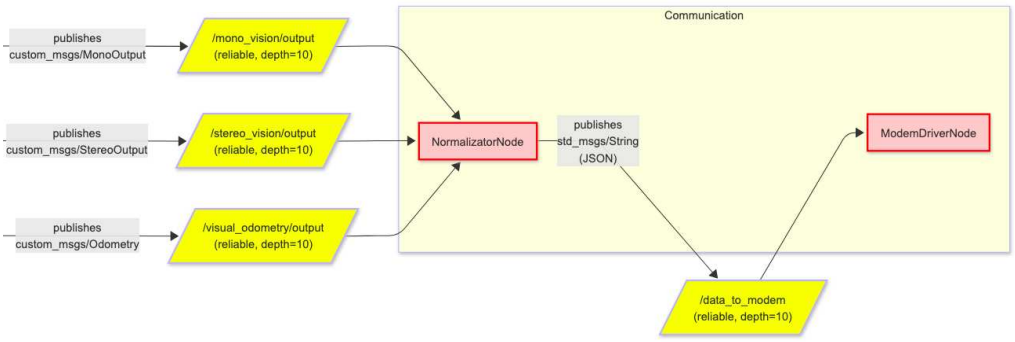

Figure 2.3: Diagram showing the communication phase with the modem

the algorithms, which are a much smaller data structure, instead of the analyzed frames. These can be easily represented as three custom ROS 2 messages, with different fields and values. For the second, we decided to introduce a "normalization" node between the CV ones and the modem. This node should subscribe to the output topics of the CV nodes, parse the result in a JSON format (which will be buffered in a string), and publish the normalized result on a dedicated topic for the modem. In this way, we did not only create a lightweight output format, but introduced a node that potentially could parse every output message in an acceptable form for the modem, making the system more scalable.

Since the modem does not require additional information for sending the data, such as networking header fields (checksum, total length, flags, etc.) the messages contain only the significant results of the CV algorithms. The only 'overhead' introduced in every message is a timestamp string, which gives a temporal reference to the data.

2.4 Custom messages

After this brainstorming, we defined a starting point for our system where, later on, we could change some aspects to introduce a more robust and optimized structure. Before building the nodes, with these informations we can create the required message interfaces using ROS 2 custom messages[1], which will be used to communicate between the nodes. Keeping in mind the requirements of both subgroups, we defined the following 4 custom messages.

MonoIR.msg:

Data type	Name	Description
string<=19	timestamp	Capture time
string<10	label	Detected animal name
float32	confidence	Prediction accuracy
float32[4]	box	Box's coordinates $\begin{bmatrix} x1 & x2 & y1 & y2 \end{bmatrix}$

Table 2.1: Structure of MonoIR.msg custom message

StereoVO.msg:

Data type	Name	Description
string<=19	timestamp	Capture time
float64[3]	translation	Translation vector, $\begin{bmatrix} x & y & z \end{bmatrix}$*
float64[9]	rotation	Rotation vector, $\begin{bmatrix} x1 & x2 & x3 & y1 & y2 & y3 & z1 & z2 & z3 \end{bmatrix}$*.

Table 2.2: Structure of StereoVO.msg custom message

Originally, the output is a 3x4 matrix that describes the differences in

[1]https://docs.ros.org/en/jazzy/Tutorials/Beginner-Client-Libraries/Custom-ROS2-Interfaces.html

the rotation and translation of the object, $\begin{bmatrix} x1 & x2 & x3 & x \\ y1 & y2 & y3 & y \\ z1 & z2 & z3 & z \end{bmatrix}$. However, since in ROS 2 matrices cannot be used as messages, we decided to break down the output into two vectors, one for translation and the other for rotation.

StereoIR.msg:

Data type	Name	Description
string<=19	timestamp	Capture time
string<10	label	Detected animal name

Table 2.3: Structure of StereoIR.msg custom message

2.5 Topics needed

In summary, the identified topics in the previous sections needed to build our system are as follows:

Topic name	Message type	QoS depth	QoS policy
/camera1/image_raw	sensor_msgs/Image	1	Best effort
/camera2/image_raw	sensor_msgs/Image	1	Best effort
/stereo_image	msg/StereoImages	1	Best effort
/outputs/cv_mono_ir	msgs/MonoIR	10	Best effort
/outputs/cv_stereo_ir	msgs/StereoIR	10	Best effort
/outputs/cv_stereo_vo	msgs/StereoVO	10	Best effort

Table 2.4: Structure of StereoIR.msg custom message

With ROS 2 latest version, we can manually configure some QoS parameters, such as the policy used, in our case "*best effort*" since we do not

have any particular requirement, and how many messages are queued before discarding the old ones (also called "queue size" or "queue depth"). For the last parameter, we decided to set the queue size of 1 for the images, since they occupy more disk space and are acquired with a high frequency, and 10 for the output of the CV algorithms, because it can be useful to memorize the last results.

3 Building the system

3.1 ROS 2 used functionalities

Before writing the actual code for the nodes, we divided which ones should be written in C++ or Python, since ROS 2 gives the opportunity to do so. In general, the first is faster and more efficient, in exchange for a more difficult syntax, while the second is more human readable but less efficient. In addition, we had to keep in mind the "development support" given to both languages. For example, if we ever wanted to add new features in the future, such as "*ComposableNode*"[1] to optimize our code, we need to have a C++ node, because there is no library to support such features in Python at the moment. To avoid rewriting a whole working node just to add new features, we elected C++ as the main language used. However, we had to use Python for the CV nodes, because the students who wrote the algorithms used Python, as it has more support in the libraries for image analysis. Since there is no easy way to embed Python code inside a C++ node, for simplicity, we decided to use Python for these nodes. Since we have enough computational power, this should not introduce system slowdowns.

[1] https://docs.ros.org/en/jazzy/Tutorials/Intermediate/Composition.html

3.1.1 Lifecycle nodes

In ROS 2, lifecycle nodes allows nodes to transition through different states during their execution time. This is particularly useful for managing the startup, configuration, activation, deactivation, and shutdown of nodes in a more controlled and predictable way. They will allow "*roslaunch*" to ensure that all components have been instantiated correctly before it allows any component to begin executing its behavior, and they also allow nodes to be restarted or replaced in case of error. Basically, they introduce a known interface that executes according to a known life cycle state machine; in this way, we had just to overwrite a series of methods that are executed after a certain transition is requested.

There are four primary states for a lifecycle node:

1. **Unconfigured**, it has just been instantiated, it is not yet ready to perform its activity.
2. **Inactive**, it has been instantiated, but it is currently not performing any processing.
3. **Active**, its currently processing.
4. **Finalized**, it is ready to be deallocated, it is a terminal state.

Transitions between states are triggered when certain callback functions are called. These are the ones that we had to overwrite in order to use this functionality:

1. **onConfigure**, allows the node to load its configurations, typically involving tasks that must be performed once during the node's life, such as obtaining permanent memory buffers and setting up subject publications / subscriptions that do not change.
2. **onCleanup**, clears all states and returns the node to a functionally equivalent state as when first created.

3. **onActivate**, prepares the node to execution, this may include acquiring resources that are only held while the node is actually active, such as access to hardware. Ideally, no preparation that requires significant time (such as lengthy hardware initialization) should be performed in this callback.
4. **onDeactivate**, reverses the *onActivate* changes.
5. **onShutdown**, performs any cleanup necessary before destruction.

When an error occurs, the node is sent to the inactive / finalized state (depending on the error type). All these transitions are managed through a "*lifecycle_manager*" node; it has the task of handling the transitions from the event and states it receives. Using this structure for the nodes allows one to define and separate the functions according to their objective (such as initializing the nodes).

3.1.2 Parameters & Configuration files

ROS 2 gives the opportunity to define parameters for the nodes, used to configure nodes at startup (and during runtime), without changing the code. The lifetime of a parameter is tied to the lifetime of the node. By default, the parameters are strictly tipified and ROS 2 pretends that they are all declared. For example, in our system we used parameters to define some values for the camera acquisition node (FPS, camera width and height...), and depending on their value the node undertakes different actions as response; or for the CV algorithms they specified the debugging level of the node or if it should track the computational times to process the frames.

ROS 2 provides an external command named "*ros2param*" to change these parameters at runtime, or they can be changed during node exe-

```
# Define parameters used to start the node
/cv_algorithms/stereo_ir_node:
    ros__parameters:
        debug:  true  # Prints additional debugging informations
        save:   false # Saves results as RosBags
        time:   true  # Calculates an average computational time per frame
```

Figure 3.1: Example of configuration file in YAML used to start *stereo_ir_node*

cution. However, to start the system with the desired parameters ROS 2 gives the opportunity to define some *configuration files* where we can specify all the parameters we need for the next run. In this way, it is easy to set up the system without worrying about using external commands to set the parameters. These files can be written in 3 different formats (Yet Another Markup Language (YAML), eXtensible Markup Language (XML) and in Python-like code). We decided to use YAML because it is the most modern and versatile of the options.

3.1.3 Launch files

As we implemented the nodes, the number of commands, arguments and even configuration files needed to start the system increased, making it complex to start again. Fortunately, there exists a "*launch system*" in ROS 2 meant to automate the running of many nodes with a single command. It helps the developer describe the configuration of their system and then execute it as described (such as which nodes to run and with which configurations). All of this information is specified in a "*launch file*", which can be written in YAML, XML or Python. In this case, we decided to use Python over a more portable markup language because it integrates better with the system and helps to speed up the process.

```python
def generate_launch_description():
    # Retrieve C.V. algorithms parameters from their configuration files
    param_mono_ir = PathJoinSubstitution([FindPackageShare(name_package), "config", "mono_ir_config.yaml"])
    param_stereo_ir = PathJoinSubstitution([FindPackageShare(name_package), "config", "stereo_ir_config.yaml"])
    param_stereo_vo = PathJoinSubstitution([FindPackageShare(name_package), "config", "stereo_vo_config.yaml"])

    # Create and configure every node parameters
    launch_mono_ir = LifecycleNode(
        package = name_package,
        executable = name_node_mono_ir,
        name = name_node_mono_ir,
        namespace = name_package,
        parameters = [param_mono_ir],
        output = "log",
        autostart = False)

    launch_stereo_ir = LifecycleNode(
        package = name_package,
        executable = name_node_stereo_ir,
        name = name_node_stereo_ir,
        namespace = name_package,
        parameters = [param_stereo_ir],
        output = "log",
        autostart = True)

    # Return the nodes
    return LaunchDescription([launch_mono_ir, launch_stereo_ir])
```

Figure 3.2: Example of launch file written in Python to automate the start up phase of CV nodes

For example, to execute the launch file given in figure 3.2 and start the CV nodes, we use the following from the command line interface:

```
$ ros2 launch cv_algorithms_launch.py
```

3.2 Modules implementation

3.2.1 Image acquisition

The "**sensing_capture**" node is the only one that actually communicates directly with the hardware, so its correct execution is crucial for the whole system. In this scenario, the lifecycle nodes are very useful: if, for some reasons, the frames cannot be acquired properly (for example: the cameras disconnect, the pre-processing node cannot synchronize the frames, etc.), the node propagates the error to the "lifecycle_manager", which in response deactivates the node (not the whole system), isolating

the error. When resolved, the node can be restarted, restoring the correct functioning of the system. This node also initializes the cameras through a "*GSTREAMER pipeline*" and sets up two publishers for the raw images.

Instead, the "**sensing_preprocessing**" node adds the compression algorithm and synchronizes the raw frames, making them ready for the CV processing. He sets up two subscribers, one for each raw image topic, and initializes one publisher for the synchronized stereo image.

Since we cannot directly see the input frames from the cameras (the Raspberry will not be connected to a monitor, and we cannot access in real time its saved frames), we decided to implement a simple Python web server using "*Flask*"[2]. This is a temporarily, but very useful solution when trying to debug the output of the algorithms with the input of the cameras.

3.2.2 CV publishers

As described before, we had three CV algorithms to integrate into our system, and they are named: "**lc_node_mono_ir**", "**lc_node_stereo_ir**" and "**lc_node_stereo_vo**", where "*lc*" stands for "*life cycle*", "*ir*" for "*image recognition*" and "*vo*" for "*visual odometry*". It is important to note that the visual odometry one was moved last minute to another subgroup of Nautilus ("*Auto-Docking*") because it needs a moving support to estimate its trajectory. But since our system will be static, during the testing phase we decided to make the switch. However, except for their outputs and different source code, they all shared the same components:

[2]https://flask.palletsprojects.com/en/stable/

```
[normalizator_node-1] {
[normalizator_node-1]   "mono_ir": {
[normalizator_node-1]           "timestamp": "2025-07-07 12:52:43",
[normalizator_node-1]           "label": "Granchio",
[normalizator_node-1]           "confidence": "0.71",
[normalizator_node-1]           "box": {"5.4", "7.8", "15.6", "18.1"}
[normalizator_node-1]   }
[normalizator_node-1] }
```

Figure 3.3: Formatted output for "lc_node_mono_ir"

1. One subscriber to the synchronized stereo images.
2. The main algorithm to execute when the node is active; if the node is deactivated it will still receive the frames but will not compute them.
3. One publisher to share the results of the main algorithm, every algorithm has its own topic since the results are in different formats.

Similarly to the image acquisition nodes, these also have a flask web server to share the annotated frames with the observers.

3.2.3 Normalize outputs

The "**normalizator**" node is responsible for formatting the different outputs of the CV algorithms into a JSON string that is easier to parse and transmit for the modem. Its made up of three subscribers, one for each CV output, three functions to convert the custom messages to a string, and one publisher to communicate the results with the modem.

For this iteration, since the modem subgroup was busy testing the hardware, we did not implement a driver node that handles communication between the system and the modem. At the moment the results are stored locally on the board main memory, but on the next iteration we plan to implement this missing feature a fully functional wireless camera trap.

```
[normalizator_node-1] [INFO] [1752166238.070824552] [normalizator.normalizator_node]:
[normalizator_node-1] {
[normalizator_node-1]   "stereo_ir": {
[normalizator_node-1]           "timestamp": "2025-07-10 16:50:38",
[normalizator_node-1]           "label": "Capasanta"
[normalizator_node-1]   }
[normalizator_node-1] }
[normalizator_node-1]
```

Figure 3.4: Formatted output for "lc_node_stereo_ir"

4 Testing the system

4.1 Computational tests

Before testing the whole system in the field, we decided to perform some computational tests for the CV algorithms. We ran the tests on different devices to get an idea of which frames could be analyzed at the same testing time.

The setup was easy: we replayed the same RosBag on the */stereo_images* topic, acquired the 29-05 at Piovego which contains approximately 30 seconds of video, acquired by the acquisition nodes. At the same time, we start the CV nodes in different orders (the ones to test) and the *normalizator* one to see the results. First we ran one node at time (mono vision image recognition ad stereo vision image recognition), then we ran them simultaneously and measured the time they took to compute the frames.

The devices we tested are: our Raspberry Pi 5 [table 4.3], our Jetson Nano [table 4.2] and my MacBook Air 2015 (with processor: 1.6GHz dual-core Intel Core i5) [table 4.1] used for development. To uniformize the results, we used the same Docker image for all devices, without varying the parameters of the system. Regarding the test on the Jetson Nano, when we tested this device, we could not use the Compute Unified

Device Architecture (CUDA) cores provided by *Nvidia* boards, which is a proprietary parallel computing platform that allows software to use certain types of graphics processing units for accelerated general purpose processing. For this reason the results can be inaccurate, we expect to speed up the algorithms if using these capabilities for image processing.

Algorithm	Analyzed frames	Mean execution time [s]
MonoIR only	35	0.5182
StereoIR only	14	13.1029
MonoIR in parallel	50	0.3847
StereoIR in parallel	12	13.3328

Table 4.1: Computational test on the development device

Algorithm	Analyzed frames	Mean execution time [s]
MonoIR only	184	0.1470
StereoIR only	12	49.5941
MonoIR in parallel	176	0.1513
StereoIR in parallel	12	49.8969

Table 4.2: Computational test on the Jetson Nano

Algorithm	Analyzed frames	Mean execution time [s]
MonoIR only	447	0.0601
StereoIR only	18	10.7203
MonoIR in parallel	430	0.0614
StereoIR in parallel	16	10.9782

Table 4.3: Computational test on the Raspberry Pi 5

From the results, we can see the differences in terms of computational power between the devices when the testing environment and the inputs are the same. Even if we expected the Jetson Nano to be more efficient, the fastest was the Raspberry Pi 5, most likely because with the first one we did not have access to the CUDA cores (which were "blocked" by

Docker). For a future iteration someone could try to include these cores and see how they impact the system and if they actually provide a speed up. For the field test, we decided to use only the Raspberry Pi 5 as the main board for the system, because the Jetson Nano seems to slow down the computational part, so we decided to put it aside for the moment.

4.2 Field tests

To test the whole system, we went to Chioggia for two days (6 and 7 July) with the aim of collecting data for future tests and to test the various components. Unfortunately, the enclosure was not waterproof, so we could not completely submerge the system. However, we were able to acquire videos underwater without damaging the board and the cameras.

The whole system was functional, and we were able to see in real time the cameras and also the results of the algorithms. Unfortunately, only the monovision image recognition algorithm correctly processed the frames and could identify the crabs; while the stereovision image recognition was affected by "*overfitting*", meaning that it recognized a scallop in every frame processed. The figures 4.1, 4.2 and 4.3 show some positive results of the system.

Figure 4.1: Real crab detected by the photo trap

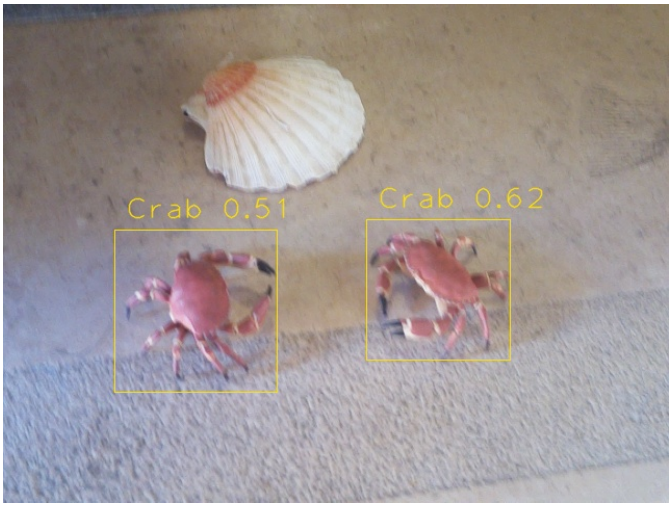

Figure 4.2: Two toy crab detected by the photo trap

Figure 4.3: A "swimming" crab detected by the photo trap

Conclusions and future

To sum up, the designing, implementing and testing of a ROS 2 operating system for an underwater monitoring system has been a rewarding project and, without a doubt, a great experience. We managed to create a fully operational operating system in ROS 2, which could acquire real-time images and perform, always in real-time, some CV algorithms to analyze the given frames and communicate in output some useful information, mainly the presence of crabs or scallops. Also, both the input frames and the elaborated one (the outputs of the algorithms) are displayed in a simple Python web server, in order to view and debug, from a local web browser connected to the same network as the main board, the inputs and the outputs of the system. This project has also been challenging because we had to interact with a group of students and complying with their requirements was not always that easy.

Although the system works, there is room for improvements in certain areas, for example:

1. The physical enclosure was not entirely waterproof, making it difficult to acquire underwater datasets for future experiments. An idea could be to re-design the enclosure, virtually adding space for more components, and making it waterproof.

2. Wireless communication was not introduced in this iteration due to time limitations. Adding this feature would mean receiving results

from the photo trap even if it is completely underwater and offshore, without physically accessing the device.

3. Enabling the computational power of the CUDA cores could speed up the processing part of the algorithms, making the system more efficient.

If these missing features could be implemented in future iterations, maybe from other students, it would mean creating a complete photo trap, which can be used in a real-case scenario to monitor underwater wildlife. Applications for such device are countless in the marine field. Having a student group supported by the University of Padova, with the aim of creating such a system, is very helpful, because it encourages the development, but also gives the opportunity to other students / researchers to benefit from this work, making it useful for someone. In my opinion, this is the most important thing when trying to develop something new.

For these reasons, I would like to continue the development of this project, by adding some missing features and making it more suitable for real-life scenarios.

List of Figures

List of Tables

Acronyms

AI Artificial Intelligence. 7, 14

API Application Programming Interface. 1

CUDA Compute Unified Device Architecture. 31, 32, 38

CV Computer Vision. vii, 13–18, 21, 23, 25, 27–29, 31, 37, 39, 47

DDS Data Distribution Service. 1

FPS Frames Per Second. 14, 15, 25

HTML Hyper Text Markup Language. 6

JPEG Joint Photographic Experts Group. 16

JSON JavaScript Object Notation. 4, 17, 18, 29

NAT Network Address Translation. 8

OSRF Open Source Robotics Foundation. 1

QoS Quality of Service. 3, 20

ROS Robotic Operating System. vii, 1–6, 8, 15–20, 23–26, 37, 47

VNC Virtual Network Computing. 6

XML eXtensible Markup Language. 26

YAML Yet Another Markup Language. 26, 39

Bibliography

[1] Steven Macenski et al. "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

Acknowledgments

I would like to thank Professor Damiano Varagnolo (supervisor of this thesis) for all the support he provided us during the course of this project, for the availability he offered us when we made the field test in Chioggia, and also for giving me the opportunity to join the Sensing Rigs of Nautilus group. Without him, this project would have been impossible.

I would like to thank all of the students who made this project possible. In particular, I thank Riccardo Fusari (our tutor in this project), who organized the whole group and took care of the project's bureaucracy with great commitment.

I thank Jacopo Toniolo, my work partner who helped me develop the ROS 2 operating system, and also for all the patience we shared when trying to debug the strangest of the errors we encountered.

I thank the CV sub-group, composed of Gianmarco Fossato, Giovanni Giaretta and Jacopo Falasco, who helped me configure and integrate their algorithms into a working ROS 2 node.

I thank Nicolas Ferraresso, for the time he spent working on the physical setup and also for the support he gave during the test days in Chioggia.

I would also like to thank all the people who supported me in these years, the results in this thesis are my way to say "thank you".