MY TECHNICAL REPORT

AUBRY Célien
(supervisor: BEAUDOIN Laurent, AVANTHEY Loïca)

Technical Report n^ooutput, January 2023
revision

In this technical paper, I present synchronisation system, its embedded architecture and its potability. I also talk about embedded frugality, which is an innovation approach that aims to create simple, affordable and effective solutions by maximizing the use of available resources. It is based on simplicity, adaptability, collaboration and openness, and is applied in many fields, such as technology, agriculture, energy, health and many others.

Dans ce document technique, je présente un système de synchronisation simple permettant la detection d'une flotte de catamaran. J'expose aussi sur la frugalité embarquée qui est une approche d'innovation qui vise à créer des solutions simples, abordables et efficaces en maximisant l'utilisation des ressources disponibles. Elle repose sur la simplicité, l'adaptabilité, la collaboration et l'ouverture, et trouve son application dans de nombreux domaines, tels que la technologie, l'agriculture, l'énergie, la santé et bien d'autres.

Keywords
Robot Operating System, Synchronisation, on-board frugality

Laboratoire de Recherche de l'EPITA
14-16, rue Voltaire – FR-94276 Le Kremlin-Bicêtre CEDEX – France
Tél. +33 1 53 14 59 22 – Fax. +33 1 53 14 59 13
caubry@lre.epita.fr – http://www.lre.epita.fr/

Chapter 1

Introduction

The SEAL is an exploratory laboratory. It is made up of two teacher-researchers: Laurent BEAUDOIN and Loïca AVANTHEY. One of their aims is to map areas by recovering data in image form. They survey places, whether on land, at sea or in the air, to recover their relief and appearance. They process the images recovered from their exploration sessions into 3D models. The images are recovered using exploration robots that patrol the terrain in an optimized manner. So there are several bodies of work in the SEAL laboratory. There's the optimization of robot control: patrolling quickly, without pitfalls and with optimized routes, with as little backtracking as possible. But there's also image processing, once the image data has been collected, to create maps, linked to the use of artificial intelligence to refine the results.

To carry out these missions successfully, robots need to be equipped with a way of perceiving the environment in which they are evolving. In this way, they will be able to adjust their trajectory to new instructions, bypass obstacles or compensate for the effect of current without human intervention, for example.

The information gathering I'll be using in my research projects will be essentially visual, so as to be applicable to all robots regardless of whether they're on land, in the air or at sea. I'll be using cameras adapted to on-board constraints, and using image processing, I'll be able to retransmit appropriate instructions to the robot.

So that my developments can benefit all robots, I'll make sure that the software and hardware are compatible with the ROS 2 inter-logical architecture currently used by SEAL robots.

Initially, I'll be focusing on setting up an environment that supports ROS2-Humble and is practical and easy to duplicate. This will enable me to go back in time, as the software takes a long time to install. Next, I'll tackle the implementation of the ROS nodes that will enable data from the GPS sensor to be retrieved, as well as data collected by the diving wing immersed in water. And then send them to a Raspberry "Beacon" so that the data can be processed by it.

Chapter 2

State of the Art

2.1 Wireless Communication

Communicating with a robot while it's running is very important. It allows you to interact with it, even if it operates "independently". Of course, the robots I'm going to use are autonomous, as we want them to evolve on their own in their environments.

There are several important aspects to communicating with an autonomous robot during operation:

Monitoring and control: Communication with an autonomous robot during operation enables human operators to monitor its activities and take control if necessary. This is crucial to ensure that the robot functions correctly, avoiding potential errors, responding to unforeseen events and ensuring safe operations.

Diagnosis and troubleshooting: In the event of a problem or malfunction, real-time communication with the robot enables technicians or human operators to carry out remote diagnosis. They can identify problems, provide instructions for necessary repairs, and even make adjustments to resolve issues without completely interrupting operations.

Interaction with the environment and humans: Some autonomous robots interact with the environment and humans. Communication during operation can be crucial to managing these interactions appropriately. For example, in the case of service robots or autonomous vehicles, communication can enable robots to understand and react to signals, movements and instructions from people around them.

Learning and continuous improvement: Communication with an autonomous robot during operation can be used to collect real-time data. This data can be used to improve the robot's algorithms and capabilities. Interaction with its environment and feedback from exchanges with humans can contribute to the robot's continuous learning and adaptation to new or complex situations.

In short, communication with an autonomous robot during operation is crucial to monitor its activities, guarantee its safety and reliability, resolve any problems, improve its performance and facilitate its interaction with the environment and human beings. It helps to ensure that the robot operates smoothly, maximizing its benefits while minimizing the risks associated with its use.

It also helps the robot to make the right choices. Certain messages will have certain levels of priority in its control. A bit like a human. For example, if you give it an order such as "Don't move here", the robot will do it, but we can give it a certain amount of freedom, like telling it that one of your comrades has already passed this way. The aim is to let it choose whether or

not to move here, depending on whether or not it feels it's important to do so, based on the information available to it.

2.2 Control / Command

Control engineering is a field of engineering concerned with the design, implementation and management of systems that regulate the behavior and operation of machines, devices or dynamic systems. Its main aim is to maintain or modify the behavior of a system to achieve specific objectives.

More specifically, control involves the use of sensors to measure relevant variables in a system, such as speed, position, temperature and so on. These data are then processed by control algorithms to generate command signals which are transmitted to actuators to influence system behavior.

Control principles are used in many fields, including electrical, mechanical, chemical, aerospace, automotive and robotic engineering, and even in more specific applications such as temperature control systems in buildings, flight controls for aircraft, automatic navigation systems and so on.

Types of control vary according to the specific needs of the system to be controlled. They include continuous control, discrete control, linear control, non-linear control, predictive control and adaptive control, among others.

Control is therefore a fundamental field of engineering concerned with regulating and influencing the behavior of dynamic systems, using sensors, data processing algorithms and actuators to achieve specific objectives efficiently and precisely.

2.3 Contextualizing data synchronization in embedded systems

Embedded systems refer to computer systems integrated into larger non-computer devices. They are specially designed to perform specific tasks within a larger system. These systems are autonomous, often limited in size and resources, and optimized for specific performance.

In a variety of contexts, embedded systems play a crucial role:

Robotics: In the field of robotics, embedded systems are at the heart of robot control. They are responsible for managing sensors, actuators, data processing and real-time decision-making to enable robots to interact with their environment and accomplish specific tasks.

Navigation: In the field of navigation, embedded systems are used in GPS (Global Positioning Systems) to determine precise position, speed and time, providing essential location information for accurate navigation and mapping.

GPS is a satellite-based positioning technology widely used in embedded systems. It uses a network of orbiting satellites to transmit signals to receivers on Earth. These signals are picked up by GPS receivers to calculate their exact position in terms of latitude, longitude, altitude and time-stamp.

The essential role of GPS in precise positioning and navigation is fundamental to many applications, such as

Car navigation: to provide drivers with precise routes and directions. **Maritime and air navigation**: for ship and aircraft navigation.

Mobile applications: For location, geolocation, tracking and mapping services in smartphones and other portable devices.

Geolocation systems for logistics: For tracking and managing shipments, vehicle fleets and more.

2.3.1 Data synchronization issues

GPS data synchronization in a multi-robot environment presents a number of major challenges and issues, not least spatial information consistency. Indeed, robots operating in the same environment must have a **coherent perception of the space around** them. This implies precise synchronization of location and mapping data obtained from GPS signals. Differences in spatial perception can lead to discrepancies in trajectory planning and potential collisions between robots. But there is also a question of **"latency" in data transmission**. The transmission of GPS data between robots can lead to variable delays. These delays introduce temporal differences in the reception of location information, which can impair coordination between robots, especially in scenarios requiring synchronized movements or concerted actions.
There is **GPS signal failure or loss**. In complex environments (indoors, dense urban areas, tunnels, etc.), GPS signal reception can be compromised or even lost. This can lead to interruptions in localization, affecting robots' ability to position themselves accurately and navigate reliably.

Then there's the **complexity of multi-robot management**. Coordinating the actions of multiple robots based on synchronized location data requires sophisticated management algorithms. This includes the design of robust communication protocols, conflict resolution mechanisms and cooperative strategies for safe and efficient navigation. [3] [2]

One of these is **time synchronization**. Precise time synchronization is essential if robots are to share and use consistent location data. Time differences between GPS data acquisitions can lead to inconsistencies in information fusion and influence the decisions made by robots, affecting their ability to cooperate effectively. To achieve synchronization between these distributed systems, we need to define a very precise reference. Despite the selection of a good base time, hardware clocks count time at different rates.
Moreover, the quality of the oscillator is also a factor to be taken into account in order to avoid so-called clock drift. To avoid or mitigate this, we can choose from various options, such as using GPS (Global Positioning System), which provides precision of several tens of nanoseconds, NTP (Network Time Protocol), widely used today to set our PC clocks, and PTP (Precision Time Protocol), which offers high precision thanks to local networks. [5]

2.3.2 Recent advances and gaps in GPS data synchronization between robots

The current state of research in the field of positioning and navigation systems has been marked by significant advances. Recent advances include the integration of artificial intelligence (AI) to predict and correct errors in GPS data. Machine learning algorithms are increasingly used to improve location accuracy by taking into account environmental conditions and interference.

The fusion of data from multiple sensors has become essential for improving the accuracy of location and navigation systems. Research has been carried out into advanced sensor fusion techniques, such as the use of extended Kalman filters and neural networks for more accurate localization, particularly in complex environments. [4]

Multi-robot systems, such as fleets of autonomous vehicles, have also received attention to ensure their resilience to interference and error. More robust algorithms and communication protocols have been developed to guarantee continuous operations even in the event of sensor failure or loss of communication between robots.

New approaches have also emerged, such as vision-based localization, simultaneous mapping and localization (SLAM), and the use of non-GPS signals for navigation. Despite these advances, challenges remain, particularly in terms of indoor navigation where GPS signals are limited, and in terms of security to prevent spoofing or jamming attacks. [1]

These recent advances have considerably improved the accuracy and reliability of positioning

and navigation systems. However, further research is needed to address persistent challenges and emerging needs, such as indoor localization and securing systems against potential attacks.

2.3.3 Review

In summary, GPS data synchronization in a multi-robot environment is con- fronted with challenges related to spatial coherence, transmission latency, interference, signal loss, time synchronization and robot management complexity. Solving these challenges is crucial to ensure accurate and reliable coordination between robots, essential in applications such as collaborative robotics, autonomous logistics, or team surveillance and rescue.

2.4 ROS2, an essential tool for robotics

Embedded systems play an essential role in a wide range of applications, from medical devices and autonomous vehicles to drones and industrial robots. These systems often require complex management of sensors, actuators and real-time control algorithms to operate reliably and efficiently in dynamic and varied environments.

With this in mind, Robot Operating System 2 (ROS2) is emerging as a promising software platform for the design and deployment of complex embedded systems. ROS2, the successor to ROS, offers a modular, flexible architecture for managing communication between different software components within a robotic system. Its scalability, native multi-robot support, enhanced security and ability to operate in distributed environments make it an attractive choice for demanding embedded applications.

Embedded systems based on ROS2 benefit from several advantages. Firstly, ROS2's modular approach enables a more flexible design, where software modules can be developed, tested and deployed independently. In addition, its compatibility with popular programming languages such as C++ and Python facilitates the development of embedded applications for a diverse range of hardware.

Notably, ROS2 also offers functionalities adapted to the specific constraints of embedded systems, such as management of limited resources in terms of memory and computing power. This optimization makes it possible to implement complex robotic solutions even on hardware platforms with restricted capacities, while maintaining acceptable performance.

In this review, we explore the contributions and applications of ROS2 in embedded systems. We examine case studies, experiments and recent advances illustrating the successful use of ROS2 in specific embedded contexts. This review will highlight the challenges faced and solutions provided by ROS2 to meet the critical requirements of embedded systems, as well as future prospects for its integration to improve the performance and reliability of these systems.

2.5 Micro-ROS, an ROS2 extension for micro-controllers

Embedded systems play a fundamental role in a wide range of applications, from automobiles and the Internet of Things (IoT) to medical devices and drones. Efficient management of sensors, actuators and real-time control algorithms is crucial to ensure the smooth operation of these systems in varied and often demanding environments.

With this in mind, Micro-ROS is emerging as an innovative and promising technology for the deployment of distributed embedded systems, offering optimal integration with Robot Operating System 2 (ROS2). Micro-ROS is an extension of ROS2 designed specifically for micro-

controllers and resource-limited environments, offering the advanced functionality of ROS2 adapted to the constraints of small-scale embedded systems.

Embedded systems using Micro-ROS benefit from its significant advantages. Firstly, Micro-ROS offers a lightweight, optimized architecture, enabling efficient execution on micro-controllers with restricted hardware capabilities in terms of computing power and memory. This specific adaptation to the hardware constraints of embedded systems guarantees efficient use of resources while maintaining acceptable performance.

In addition, Micro-ROS facilitates the modular development of embedded applications by integrating ROS2 functionalities adapted to micro-controllers. This modular approach enables a flexible design where software modules can be developed, tested and deployed independently, offering greater adaptability to the specific needs of each application.

In this review, we will explore the contributions, applications and advances of Micro-ROS in embedded systems. We will analyze case studies, experiences and recent developments that highlight the effectiveness of Micro-ROS in specific embedded applications. This review will also examine the challenges faced and solutions provided by Micro-ROS to meet the requirements of small-scale embedded systems, while considering future prospects for its integration to improve the robustness and efficiency of these systems in various application areas.

Chapter 3

Achievements

3.1 Projet

This project focuses on data synchronization in an onboard environment. The laboratory has a Camataran robot which allows exploration on the water surface. The principle is to submerge a device underneath it to attach a camera for probing the seabed, while retaining the convenience of a non-submerged part of the system.

The submerged part of the robot is a shaft directly inlaid on the Catamaran's plate, on which the camera device is implemented. An underwater wing is added to counter underwater disturbances. The wing is controlled by a servo-motor, which rotates on an axis to stabilize the submerged platform. A Raspberry 3B+ board, to which the camera is connected, and an Arduino board, to which an IMU (Inertial Measurement Unit) is connected, embedding an accelerometer and a gyroscope to control the stability of the submerged system, are positioned on this plate.

The camera's video data are retrieved using a raspberry, which also retrieves data from the IMU via the arduino's serial port. This data is sent back to the surface of the catamaran on a raspberry card, which itself collects data sent to it by the GPS. These three data are then retrieved and synchronized with other nearby catamarans via Wi-Fi.
The following diagram describes the various elements of the Catamaran project:

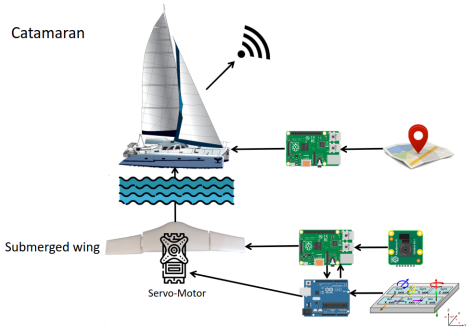

Figure 3.1: Architecture of the Catamaran entities

To synchronize data, we use an external Wi-Fi access point where all Catamaran entities are connected and send their data. The GPS and IMU data is then analyzed and sent back to the

other Catamarans, so that each Catamaran has access to the GPS data of the other entities. This will optimize the exploration of the Catamarans, perfecting their trajectories to cover a larger area in less time.

The Wi-Fi access point will be on a floating platform at the center of exploration. It will interact with all catamarans connected to the network.

This arrangement means that Wi-Fi can be used as a means of communication. In fact, in water, waves have difficulty propagating and it would not have been possible to use them in these conditions.

3.2 Installation

3.2.1 References

- Model : Raspberry 3B+
- Raspberry's Operating System : Ubuntu 22.04.03 LTS Server (No desktop)
- Imager : Raspberry OS Imager 1.8.1
- Configuration :
  - Username : pi
  - Password : raspberry
  - Hostname : raspberrypi.local
- Carte SD which contain OS : SanDisk 32Go et 64Go
- OS of my computer to setup environments : Ubuntu 22.04

3.2.2 Setup of operating systems

We're going to run ROS2 on a Raspberry, so we need to prepare an Operating System to run it. We're going to configure an Ubuntu 22.04 Server OS on the Raspberry 3B+ board, with no screen or peripherals. The aim will be to find a way to install the applications we need and create an object to make instances of them. Why Ubuntu 22.04 Server? The ROS2 application we're going to use is resource-intensive, so it's important not to use a version with a graphical interface that would consume resources unnecessarily, bearing in mind that not all our applications should need them. As far as Ubuntu 22.04 is concerned, it is compatible with the Humble version of ROS2, which is not the case with the Operating System made for Raspberry Boards named Raspberry Pi OS, and the version is known as LTS (Long-Term Support), which is important for maintaining the application over several years.

To install the OS, I use the Raspberry OS imager application and configure my settings as described above. Then I install and connect via ssh to the Wi-Fi network configured in the imager.

Once the SD card has been initialized, I simply put it in the Raspberry. It connects to the desired network and I access it via ssh.

For this step, there were some problems with the Raspberry Pi OS I had started with. In fact, the operating system is practical because it's not very resource-hungry, which makes it a very good choice when we're constrained by embedded resources. But it's not compatible with the ROS2-Humble software.

3.2.3 Setup of ROS2-Humble on the Raspberry

To install ROS-humble, I follow the information on this site:
https://docs.ros.org/en/humble/Installation/Alternatives/Ubuntu-Install-Binary.html
I go through the binary version because going through the source is far too long for the Raspberry, which already has almost all the work already done in the binary.

To get the binary, I get it via the following command :

wget https://github.com/ros2/ros2/releases/download/release-humble-20231122/ros2-humble-20231122-linux-jammy-arm64.tar.bz2

I choose the ARM package which is adapted to the raspberry architecture. For the unpack, I download the lbzip2 library, without which decompression can't be made :

sudo apt install lbzip2

In the rest of the installation, on the 'rosdep install ...' command, I added '–rosdistro humble' to handle package conflicts. Downloading the dependencies takes about 2 hours. There will be about thirty messages due to the scan of the linux image that it does at each installation, to which you'll have to press ENTER to get past the warnings so that the installation can continue.

I had to go through this step several times. Initially, installation via the ROS2 software sources available at
https://docs.ros.org/en/humble/Installation/Alternatives/Ubuntu-Development-Setup.html
didn't work. In fact, the sources compiled for hours without ever producing a viable result, as they were too resource-intensive, and the compilation always stopped before completion.

3.2.4 Setting up a Static IP

I'd like to set a Static IP so that I can create a network between all the Raspberries. Indeed, the network is the communication key of the project. When the catamarans are on the water, their underwater section will be able to use wifi to communicate between the various network entities via a Raspberry "Beacon", which will retrieve the information to be synchronized and broadcast it to the other catamarans. For this reason, it's best to set the IP address of the "Beacon" using a Static IP which will be known by the catamarans, so you don't have to change the catamaran codes if the Beacon's IP is modified.
To set the Static IP address, I modify the file at this location: /etc/dhcpcd.conf by adding :

interface eth0
static ip_address=192.168.0.100/24
static routers=192.168.0.1
static domain_name_servers=192.168.0.1

3.3 Implementing communication with ROS

In the previous section, we saw how to set up the work environment on which we'll be working. Now we can set about creating the ROS2 nodes within the various entities that make up the project. We're going to talk about the two different instances we need. First, we'll have the catamarans, which can be any number of catamarans, and where they'll communicate with the

master board for sensor information. The second part will be the implementation of the code on the master board, which will have the same basic environment, but will be able to acquire data from each catamaran and then broadcast it "Multicast" to all the entities in the catamaran fleet. The code for the ROS2 nodes I'm going to build will be written in C++.

3.3.1 Working on the catamaran side

The aim here is to start by creating a ROS node that will enable the publication of a data structure. To do this, I start by creating a workspace that will serve me throughout the creation of the nodes. Then I create my first package in the src folder of this workspace, which will enable the GPS data to be published. I do this using the ROS2 command lines:
ros2 pkg create –build-type ament_cmake catamaran_nodes
Once the package is created, I'll create my node codes directly in the src folder of the new package. So I want to create my GPS node, which I'll call GPS_node_publisher.cpp, and write the classic ROS2 publisher code inside. However, the data to be transmitted is the data structure returned by a GPS sensor. It's in the form of altitude, longitude, latitude. In the software documentation, I found a structure called NavSatFix in the sensor_msgs library. I import the structure and fill it with the data to be sent:
#include "sensor_msgs/msg/nav_sat_fix.hpp"

I create a GPSPublisher class that inherits from the Node class. In its constructor, I initialize a publisher object using the "create_publisher" API function, which I template with my NavSatFix type structure. I tell it I want to publish on the "gps_topic". I'll have to make sure that the node retrieving this data is reading from the same topic! Then I initialize a timer using the "create_wall_timer" function. The timer calls a CallBack function when its time has elapsed. So you need to give it a time as an argument, which in this case is 2000 milliseconds. I create the Callback function, call it publishGPS and insert the following logic: Create the data structure of the message to be sent and publish it. The aim is to publish it on the topic listed above. The node code looks like this:

```cpp
#include "rclcpp/rclcpp.hpp"
#include "sensor_msgs/msg/nav_sat_fix.hpp" // Contains GPS variables return by the sensor

class GPSPublisher : public rclcpp::Node
{
 public:

  GPSPublisher() : Node("gps_publisher_node")
  {
    publisher_ = this->create_publisher<sensor_msgs::msg::NavSatFix>("gps_topic", 10);
    timer_ = this->create_wall_timer(
                                     std::chrono::milliseconds(2000),
                                     std::bind(&GPSPublisher::publishGPS, this));
  }

private:

  void publishGPS()
  {
    auto gps_msg = std::make_unique<sensor_msgs::msg::NavSatFix>();
    gps_msg->latitude = 48.8566;  // Latitude Paris, France
    gps_msg->longitude = 2.3522;  // Longitude Paris, France
    gps_msg->altitude = 35;  // Altitude Paris, France
    RCLCPP_INFO(get_logger(), "Publishing GPS - Latitude: %f, Longitude: %f", gps_msg->latitude, gps_msg->longitude);
    publisher_->publish(std::move(gps_msg));
  }

  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<sensor_msgs::msg::NavSatFix>::SharedPtr publisher_;
};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<GPSPublisher>());
  rclcpp::shutdown();
  return 0;
}
```

Figure 3.2: Node GPS code

I compile with **colcon build** at the root of the workspace, then initialize the commands with **source install/setup.bash** Then I launch the node with the following command:
ros2 run catamaran_nodes imu_node_publisher

For the IMU data node, it's pretty much the same, except that the data structure will change. In fact, there's also a data structure for acceleration and angular velocity, which are the data collected by the IMU in the submerged section and which we want to send back to the other entities in the fleet.
It's also in sensor_msgs and I import it with :
#include "sensor_msgs/msg/imu.hpp"

I publish the messages on another topic in order to differentiate the types of message sent. So I send the messages to an "imu_topic" topic. I put this code in a file I call IMU_node_publisher.cpp. Here's a preview of the IMU file:

```cpp
#include "rclcpp/rclcpp.hpp"
#include "sensor_msgs/msg/imu.hpp" // Contains GPS variables return by the sensor

class IMUPublisher : public rclcpp::Node
{
 public:

  IMUPublisher() : Node("imu_publisher_node")
  {
    publisher_ = this->create_publisher<sensor_msgs::msg::Imu>("imu_topic", 10);
    timer_ = this->create_wall_timer(
                                      std::chrono::milliseconds(2000),
                                      std::bind(&IMUPublisher::publishIMU, this));
  }

private:

  void publishIMU()
  {
    auto imu_msg = std::make_unique<sensor_msgs::msg::Imu>();
    imu_msg->angular_velocity.x = 2;
    imu_msg->angular_velocity.y = 8;
    imu_msg->angular_velocity.z = 10;

    RCLCPP_INFO(get_logger(), "Publishing IMU - Angular x: %f, Angular y : %f", imu_msg->angular_velocity.x, imu_msg->angular_velocity.y);
    publisher_->publish(std::move(imu_msg));
  }

  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<sensor_msgs::msg::Imu>::SharedPtr publisher_;
};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<IMUPublisher>());
  rclcpp::shutdown();
  return 0;
}
```

Figure 3.3: Node IMU code

I compile with **colcon build** at the root of the workspace, then initialize the commands with **source install/setup.bash** Then I launch the node with the following command:
ros2 run catamaran_nodes imu_node_publisher

3.3.2 Working on the beacon side

The beacon, on the other hand, is a multicast acquisition and broadcast device, simulated by another device on the same Wi-Fi network, for which all data is redirected to it. It therefore needs to listen to both the gps_topic and imu_topic topics. To do this, it needs to register on both topics, and when data is received on one of the topics, it is processed and another piece of data is waited for. To do this, I need to create a node with two subscribers, one for each of the topics, and they each have a callback that processes their own message. I use the **create_subscription** function, which lets me create an object that listens for a certain type of message on a certain topic. So I create two, one that I template with the NavSatFix data structure and link to the "gps_topic" topic, and one that I template with the "imu" data structure and link

to the "imu_topic" topic. I link the two to a Callback function that processes the data according to whether it's IMU or GPS data.

Here's how the resulting code looks :

```cpp
#include "rclcpp/rclcpp.hpp"
#include "sensor_msgs/msg/nav_sat_fix.hpp" // Contains GPS variables return by the sensor
#include "sensor_msgs/msg/imu.hpp"

class DataSubscriber : public rclcpp::Node
{
    public:

        DataSubscriber() : Node("data_subscriber_node")
        {
            gps_subscriber_ = this->create_subscription<sensor_msgs::msg::NavSatFix>(
                "gps_topic", 10, std::bind(&DataSubscriber::GPSCallback, this, std::placeholders::_1));
            imu_subscriber_ = this->create_subscription<sensor_msgs::msg::Imu>(
                "imu_topic", 10, std::bind(&DataSubscriber::IMUCallback, this, std::placeholders::_1));
        }

    private:

        void GPSCallback(const sensor_msgs::msg::NavSatFix::SharedPtr gps_msg)
        {
            RCLCPP_INFO(get_logger(), "Reveive GPS - Latitude: %f, Longitude: %f", gps_msg->latitude, gps_msg->longitude);
        }
        void IMUCallback(const sensor_msgs::msg::Imu::SharedPtr imu_msg)
        {
            RCLCPP_INFO(get_logger(), "Receive IMU - Angular x: %f, Angular y: %f", imu_msg->angular_velocity.x, imu_msg->angular_velocity.y);
        }
        rclcpp::Subscription<sensor_msgs::msg::NavSatFix>::SharedPtr gps_subscriber_;
        rclcpp::Subscription<sensor_msgs::msg::Imu>::SharedPtr imu_subscriber_;
};

int main(int argc, char * argv[])
{
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<DataSubscriber>());
    rclcpp::shutdown();
    return 0;
}
```

Figure 3.4: Receiver GPS & IMU code

I compile with **colcon build** at the root of the workspace, then initialize the commands with **source install/setup.bash**. Then I launch the node with the following command:
ros2 run master_nodes receiver_data

3.3.3 Network communication

If we try to launch all the nodes, we find that communication is not possible.The nodes are on two different devices.However, they're on the same network, so we can specify that we're using communication on a particular port. Communication to the receiving node takes place over the Wi-Fi network. To do this, we need to specify a few arguments for the connection to take place. In fact, there's an environment variable that lets you define a connection ID and link devices over the network rather than locally. By defining the ROS_DOMAIN_ID environment variable as follows:
export ROS_DOMAIN_ID=42 This will enable devices to communicate according to the following characteristics:

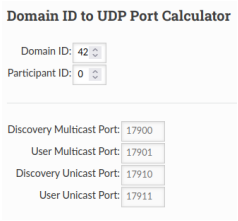

Figure 3.5: UDP port calculator depends ID Domain (here 42)

Of course, ID 42 can be different and has been arbitrarily chosen here. Above all, it must be different from 0.

3.3.4 Multicast data return on Tag side

3.3.5 Identification by message

Now that the beacon has retrieved the data from the Catamarans. We now need to separate messages from several different sources. To do this, we need to add a unique identifier, such as a UUID, to each message sent. This will overwrite the last data received by the same sender. The aim is to send the last data closest to the actual position. The idea is to create your own message that will wrap the "sensor_msgs" structures I'm currently sending.

To achieve this, all we need to do is create a new package and prepare it with several parameters: - In the package.xml file, I add:
<build_depend>rosidl_default_generators</build_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
<depend>sensor_msgs</depend>
<depend>unique_identifier_msgs</depend>
- In the CMakeLists.txt, I add:
find_package(rosidl_default_generators REQUIRED)
find_package(sensor_msgs REQUIRED)
find_package(unique_identifier_msgs REQUIRED)
rosidl_generate_interfaces($PROJECT_NAME "sensor_msgs/NavSatFix.msg"
DEPENDENCIES sensor_msgs unique_identifier_msgs)

Next, you need to add the message to be sent to a my_gps.msg file in a msg folder at the root of the package:
sensor_msgs/NavSatFix
unique_identifier_msgs/msg/UUID

Once a unique UUID has been generated in the publication nodes, I can now separate messages according to source.

3.3.6 History

To obtain a history of messages, we can store incoming messages in a file so that we can retrace the routes taken by each catamaran. In addition to this, we have the precise time, i.e. their evolution over time. With IMU data, we even have their orientation. With the OfStream library, we can easily do just that:
ofstream monFlux("Filename.log");
myFlow « "GPS and IMU data over time" « endl;

3.3.7 Publish grouped data

Now we need to republish all this information to the other catamarans. To do this, we need to do almost exactly the same thing as above. We need to create a "publisher" to publish the latest data, no more than 10 seconds old, on another "mapData" topic (to avoid sending data from a

robot that has disconnected). It would be preferable to create another interface containing IMU and GPS data at the same time, so that all information can be sent directly!

3.4 Grouped data reception

Finally, you just need to create a node on the catamaran side to subscribe to the "mapData" topic you created earlier, in order to retrieve the data and adapt their trajectories according to what they receive. Of course, the catamaran's value must be removed from this data, by matching the UUIDs with its own UUID to remove its own value.

Chapter 4

Conclusion

In conclusion, embedded frugality is positioning itself as an essential area of research and development in the field of embedded systems. It aims to optimize the use of limited hardware resources while guaranteeing acceptable performance, thus offering solutions suitable for a diversity of applications, from IoT to autonomous devices. In this study, we explored the main concepts, strategies and approaches adopted to design frugal embedded systems. We have highlighted techniques for managing energy consumption, optimizing algorithms and reducing hardware complexity, all of which contribute to system efficiency while retaining functionality. However, it should be noted that despite significant progress in this field, challenges remain. The design of frugal embedded systems requires a holistic approach that takes into account trade-offs between performance, energy consumption and hardware constraints. What's more, adapting these principles to specific applications remains a complex challenge requiring ongoing, in-depth research.

Due to time constraints, this study could not cover all aspects and recent advances in embedded frugality. Further investigations are needed to further explore the opportunities and limitations of this approach in specific fields such as robotics, connected automobiles and other mission-critical applications.

In sum, embedded frugality remains an exciting and promising area of research, offering innovative solutions to the growing challenges of resource-constrained embedded systems. We hope that this study will stimulate further research and development aimed at fully exploiting the potential of embedded frugality to create more efficient, sustainable and high-performance systems. Despite the time limitations encountered, this study lays the foundations for future in-depth investigations in this crucial area of embedded technologies.

Chapter 5

Bibliography

[1] Andréa Macario Barros, Maugan Michel, Y. M. G. C. F. C. (2020). A comprehensive survey of visual slam algorithms. *LCAE (CEA, LIST) - Laboratoire Capteurs et Architectures Electroniques (CEA, LIST)*. (page 5)

[2] Luis Costa, Alisson V. Brito, T. P. N. T. B. (2017). *Integration of Robot Operating System and Ptolemy for Design of Real-Time Multi-robots Environments*. PhD thesis, UFPB - Universidade Federal da Paraiba. (page 5)

[3] Maxa, J.-A. (2020). *Architecture de communication sécurisée d'une flotte de drones*. PhD thesis, ENAC - Ecole Nationale de l'Aviation Civile. (page 5)

[4] Merlet, J.-P. (2019). *Influence of uncertainties on the positioning of cable-driven parallel robots*. PhD thesis, HEPHAISTOS - HExapode, PHysiologie, AssISTance et Objets de Service. (page 5)

[5] MKACHER, F. (2017). *Optimization of Time Synchronization Techniques on Computer Networks*. PhD thesis, LIG - Laboratoire d'Informatique de Grenoble. (page 5)

Copying this document

Contents