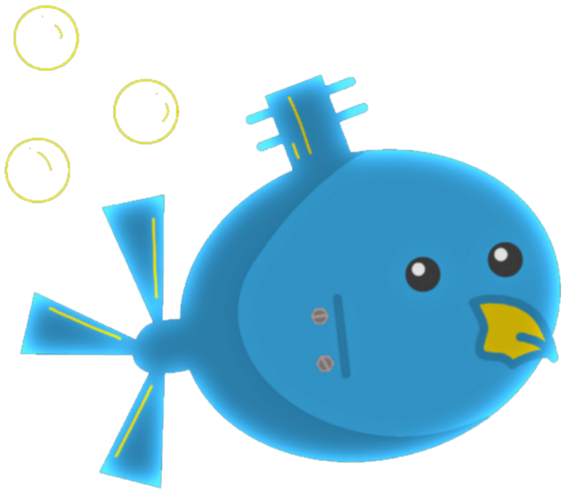
CALIFORNIA STATE UNIVERSITY, LOS ANGELES

Software Design Document



ROBOSUB

Members
Thomas BENSON, David CAMACHO, Bailey CANHAM, Brandon CAO,
Roberto HERNANDEZ, Andrew HEUSSER, Hector MORA-SILVA,
Bart RANDO, Victor SOLIS

Monday 6th March, 2023

Table of Contents

Revision History

Version	Description	Date
1.0	First release of Software Design Document.	9 December 2022

Table 1: Revision History

1 Introduction

1.1 Purpose

This document represents the software component of Lanturn, an Autonomous Underwater Vehicle made by the Robosub Senior Design Software and Hardware/Electrical teams. This will break down into further sub teams which put together make up the entire RoboSub Senior Design Software Team. These sub teams include Autonomy, Computer Vision, Controls, and Navigation. Some modules used for software include: the ROS module and the Arduino module.

1.2 Document Conventions

The standards/conventions that were used to write this document are very common formatting such as bold headings, highlighted words that are important, and smaller fonts for paragraphs.

1.3 Intended Audience and Reading Suggestions

The types of readers that this document is intended for are future developers of the RoboSub Senior Design years, developers who are interested in robotics, and hobbyists. This document will cover the four different sub teams that make up the RoboSub Senior Design Team working on Lanturn: Autonomy, Computer Vision, Controls, and Navigation.

Autonomy covers state machines, Computer Vision covers machine learning and image processing, Controls covers movement and IMU readings, Navigation covers sensor readings and mapping/localization.

1.4 System Overview

The Autonomy/Mission Planning sub team is to cover the SMACH model developed to control the AUV subsystems through each task at the RoboNation Competition.

The Computer Vision sub team is to recognize specific images, such as badges or dollar signs, in underwater environments and then process them through a machine learning model and send the relevant data to Autonomy/Mission Planning.

The Controls sub team is to control Lanturn, the sub, with its thrusters using PID controllers. This is done using three different axes: Pitch to tilt forward or backward, roll to move side to side, and Yaw to rotate left or right.

The Navigation sub team is to gather data from the sensors such as barometer and then also send relevant data to Autonomy/Mission Planning.

2 Design Considerations

This section describes many of the issues which need to be addressed or resolved before attempting to devise a complete design solution.

2.1 Assumptions and Dependencies

Lanturn will have a TX2 module, a small form-factor embedded computing device, for its main computer. The TX2 module comes flashed with Ubuntu 18.04 and comes packaged with Jetpack 4.6.1, a set of libraries designed for AI computations designed for the TX2 module.

Over the Operating System base and the Kernel overlay will be ROS2 Foxy, a piece of middleware that sets an environment for the different processes that will run on Lanturn.

The second computing device that will exist in the Lanturn system is a microcontroller, a teensy 4.1. The microcontroller will be used to interface with the onboard sensors, motors, and actuators.

2.2 General Constraints

Ubuntu is the only operating system that can run on the TX2 module. This requires any drivers or other implemented interfaces to be designed for, or to be compatible with Ubuntu versions.

The TX2 module and its accompanying software is designed for Ubuntu 18.04. Even though it is designed for Ubuntu version 18.04, it is possible to upgrade to a later Ubuntu version; however, the only Ubuntu version which is officially supported is Ubuntu 18.04. The Ubuntu version the TX2 will be running on is Ubuntu 20.04 Focal; all constraints that come with running on an unsupported version of Ubuntu on the TX2 module will exist in this project.

The middleware running on top of Ubuntu is a ROS2 version, ROS2 Foxy, which runs on Ubuntu 20.04. ROS2 Foxy comes with restrictions with the programming languages that can be used. The API that is used for development are rclcpp (C++ library) and rclpy (Python library).

The second computing device, Teensy 4.1, must be programmed in C++ using one of: teensyduino with Arduino IDE or platformio for Visual Studio Code.

2.3 Goals and Guidelines

This document will have the following goals and guidelines in mind:

- The Lanturn project must have a functioning autonomous system that can execute competition tasks by July 2023.
- All code written for this project should be documented.
- Programs written for this project should be refactored periodically to improve efficiency.

2.4 Development Methodology

The software team will be broken down into 5 sub-teams, each responsible for one module of the software system. Each week, the sub-teams show progress made and communicate any difficulties that have come across.

As sub-teams gain deeper understanding of their module, they will communicate to the other modules what data they can provide and what data they cannot.

This system will ensure that none of the modules depend on each other and can be switched out, if need be, in future versions of Lanturn.

3 Architectural Strategies

Each module will be programmed and packaged as a ROS2 package, except for the controls module which will exist as firmware on the microcontroller. Even though the microcontroller won't be a ROS package, it will still have all the characteristics of a ROS2 package; it will be entirely modular, using the same communication system provided by ROS2.

The controls module will interface with actuators, sensors and thrusters. The computer vision module will interface with cameras. All other modules will exist as software and will go through the controls and computer vision modules to interact with the environment.

Additionally, there will be a watchdog service that will monitor the liveliness and diagnostics of the system.

4 System Architecture

The software on Lanturn is organized into a number of modules, each of which is responsible for a specific task. These modules are build on the ROS framework and communicate with each other using ROS messages.

4.1 Overview of the System

Computer Vision shall output to: Mapping, Localization. Mapping shall output to: Autonomy, Localization. Localization shall output to: Autonomy Autonomy shall output to: Controls. Controls shall output to: Mapping, Localization.

Autonomy will manage time, manage state machines, read and interpret mapping/localization data, read and filter computer vision data, control claw, shoot torpedoes, release dropper, autonomously navigate map, and position/orient/center to desired orientations.

Comp Vision will publish raw images from front and bottom cams, detect and classify all task objects, provide distance from objects, calculate angle of incidence of objects.

Controls will read and publish data from Bar30 barometer, VN-100 IMU, Teledyne DVL, Sonar. Controls will also implement a PID library, generate PWM values that will move the sub to desired position, output PWM values to thrusters. Controls will also control a mechanical claw, shoot torpedoes, and release a ball from dropper all on command.

Mapping will subscribe to all computer vision data and Sonar data. Mapping will also implement a Kalman Filter, generate a map of the environment, position all task objects in map, and publish map.

Localization will subscribe to IMU data topic, Barometer data topic, both camera topics, DVL data topic, and map topic. Localization will position and orient the sub inside the map, and publish localization data.

4.2 Autonomy

The autonomy part of the software is responsible for completing the robosub goals. This is done using a state machine and using the ROS software to navigate the data to move from one state to another until each goal is completed, battery runs out, or time limit of 20 minutes is exceeded. The state machines are subscribed to all hardware components to be able to determine ubliches the next goal, and to controls to reach the next goal or complete the task. Each task/goal is broken down into their own states in their own class. Future implementation will move from state machine to behavior tree which allows for system to be goal oriented instead.

4.3 Computer Vision

The DFD Level 1 will start with the Camera sending information to the Image Processor, after processing the image it will send the information to Object Detection. Object Detection will process that information and once it's able to detect the object it will send the data to Object Classification, and it will define the location of the object. Once Object Classification processes the data of the object detected, it will send a Message Output. The Message Output will give out the data of the Object ID and the Bounding Box.

4.4 Controls

The controls code has the same form as any other microcontroller code. It starts with a setup() function then starts executing a loop() function until told to otherwise.

In the setup() function, sensors, actuators and the thruster motors are initialized and general setup like importing libraries and configuring the PID controller is done here. This function is executed once then control is turned over to the loop() function.

The loop() function contains the code for the PID controller, sensor reading and actuator control that will run independently. The flow of the data is this: grab sensor data, fix setpoints to account for circular rollover error, compute PWM values, combine the PWM values, output them to the ESCs and loop back to the beginning.

4.5 Localization

The localization node takes input data from the various environmental sensors (IMU, Sonar, Barometer, DVL, hydrophones). Localization will process this data and then publish the processed data to the entire system. Any node that subscribes will have access to the data.

4.6 Mapping

TBD.

5 Policies and Tactics

The main influences on Lanturn's Software design has already been established by previous years.

Some adjustments have been made by individuals to suit their ideas and programming style, but most of the system is inherited from previous years.

5.1 Specific Product Used

- System
  - Operating System
  - Ubuntu 20.04
  - Build System
  - Colcon
  - Ament_cmake
- Autonomy
  - Visual Studio Code
  - Groot
  - SMACH
  - BehaviorTree.CPP
- Controls
  - Visual Studio Code
  - Platform.I0
- Computer Vision
  - LabelIMG
  - Google CoLab

  - YOLOv4
  - Darknet
  - OpenCV
- Mapping and Localization
  - Gazebo
  - Visual Studio Code
  - Cubik Studio

5.2 Requirements Traceability

Each module of the system will have its own git repository. The sub-team assigned to the module will be responsible for keeping detailed documentation for the process of setting up, installing and using the software module.

5.3 Testing the Software

- Navigation
  - Create Gazebo Simulation for localization testing
  - Unit test for processing sensor data
  - Unit test for noise filter on sensor data
  - Unit test for publishing position data
  - Unit test for subscribing to position data
- Autonomy
  - Create unit test for states.
  - Create unit test for publishers
  - Create unit test for subscribers.
  - Simulate test environment for states
  - Simulate test environment for publishers
  - Simulate test environment for subscribers
  - Use testing AUV (Blastoise)

5.4 Engineering Trade-Offs

ROS is the leading opensource robotics software in use today. There are no engineering trade-offs with hardware and software support being mutigenerational.

5.5 Guidelines and Convetions

Using the selected IDE and choice of CPP instead of python ensures that the languages general convention is enforced.

5.6 Protocols

By using ROS the built in API of the standardized subscriber and publisher model is always enforced.

5.7 Maintianing the Software

No plans to maintain software, only fix bugs. The software is specially attached to the hardware, the only way to maintain software or update is to upgrade hardware. If we do that then we need to refactor or use a different ROS version.

5.8 Interfaces

There is no interface for users, fully autonomous, web GUI subscribes to publishers for user visualization.

5.9 System Deliverables

5.10 Abstractions

6 Detailed System Design

6.1 Autonomy Module

6.1.1 Responsibilities

The main responsibility of autonomy is to design and implement the logic software of the AUV to complete each task of the competition, currently using SMACH and in the future BehaviorTree. In addition, autonomy is responsible for receiving and publishing data. Autonomy receives data from computer vision, localization, and mapping components. Based on the current state of the AUV, to maneuver throughout the competition, the data collected is then published to the control's component.

6.1.2 Contraints

- Autonomy will be responsible for system checks prior to starting goal search, and aborting mission if critical component failure occurs.
- Autonomy will run until either all goals at met, battery capacity runs out or time limit exceeds 20 minutes.
- Autonomy will assume all data published is correct, data is formatted and validated prior to being published.
- Autonomy is responsible for completing tasks that lead to completing each goal successfully, based on its current state and retry if it fails the goal state and moving on to the next goal/state.
- Failure to complete tasks in a timely matter will result in SMACH failing.
- SMACH is not meant to be used as a state machine for low-level systems that require high efficiency, SMACH is a task-level architecture.
- SMACH is linear, performing one task at a time, unless it is designed to run in concurrence

6.1.3 Composition

- ROS2: Robot Operating System V2
- SMACH: State Machine

- SMACH Viewer: State Machine graphical Viewer
- Behavior Tree: Goal oriented instead of state-based logic
- BehaviorTree.CPP: API for implementing Behavior Tree
- Groot: Graphical Interface for building Behavior Trees.

6.1.4 User Interactions

- Define the current state of the AUV
- Subscribe to receive data from other components of the AUV
- Publish data to controls component of the AUV
- Define transition between sub-states of a state machine
- Define the transition between different state machines
- Pass user data between different state machines

6.1.5 Resources

- SMACH – A ROS-independent Python library to build hierarchical state machines `http://wiki.ros.org/smach/Documentation`
- SMACH Viewer – GUI that shows the state of hierarchical SMACH state machines. `http://wiki.ros.org/smach_viewer#Documentation`
- BehaviorTree.CPP - Implemented using C++, assembled using a scripting language based on XML. Behavior Trees are composable. You can build complex behaviors by reusing simpler ones. `https://www.behaviortree.dev/`
- Groot - "IDE for Behavior Trees". Allows users to visualize, create and edit Behavior Trees, using a simple drag and drop interface and lets Trees to be monitored in real-time. `https://www.behaviortree.dev/groot`

6.1.6 Interface/Exports

- Autonomy interfaces only through other systems that directly interact with the hardware and doesn't directly interface with any hardware.
- Autonomy shall interact with the user interface to provide the current state of the AUV
- Autonomy interacts with all other components to receive data to determine the current state of the AUV
- Autonomy interacts with controls to provide instructions to maneuver the AUV

6.2 Computer Vision Module

6.2.1 Responsibilities

The primary responsibility of the computer vision is to utilize the cameras attached to the RoboSub in order to identify various competition items. The computer vision program will then output a bounding box around the identified object and send that information to the navigation controls. The computer vision model should have a high enough accuracy to allow us to be confident in the result. Additionally, the program should output the distance and angle the robot is from the identified object and send this data to the navigation controls.

6.2.2 Contraints

The main constraint on the computer vision model is time and storage. We must train the computer vision model on pictures of each object we want it to be able to identify. However, in order to do this, we must have hundreds of pictures available and saved for when we train the model. Therefore, we must consider how much storage we have available on the computer. Additionally, training the computer vision model takes quite a bit of time. In our first training session, we only got through around 500 iterations of training, and it took over 5 hours. Due to this, we must consider how much time we have and carefully plan it out in order to ensure we have enough time for the model to train enough to be reliable.

6.2.3 User Interactions

6.2.4 Composition

The first subcomponent is Google CoLab, which is a collaborative python programing space that provides access to cloud computing. We use Google CoLab for its GPU to train and test our computer vision model. The second subcomponent is YOLOv4, which is an object detection algorithm. It works by dividing images into a grid system with each cell in the grid responsible for detecting objects within itself. The third subcomponent is Darknet, which is a neural network framework written in C and CUDA. We use this in conjunction with YOLOv4 to complete object detection. The fourth subcomponent is OpenCV, which contains an optimized computer vision library, tools, and hardware all aimed at real-time object recognition. We are testing this as a second option to compare the results with YOLOv4 and Darknet. The fifth and final subcomponent is LabelIMG, which is a graphical image annotation tool. We use this to label our images prior to using them to train our computer vision model.

6.2.5 Resources

The other component that will be receiving the data is Navigation, either by sending them the data of the output of the bounding box as well as the angle and distance that the robot is from the identified object, which will be whatever object they present us with during the competition. The only side-effects that we could experience in this component would be if we were to mislabel an input from the camera.

6.2.6 Interface/Exports

- Export bounding box to Navigation Controls
- Export distance from the robosub to the object to Navigation Controls
- Export angle between robosub and object to Navigation Controls

6.3 Controls Module

6.3.1 Responsibilities

Controls acts as the primary interface between the hardware and the software for communications to the main motherboard and ROS operating system. It is responsible for allowing the various software components to communicate back and forth with all the sensors, motors, and actuators attached to the robot as necessary.

6.3.2 Contraints

The main constraint on the controls module is the hardware. The hardware is limited to the software libraries that are provided by the manufacturer and the communication protocols that are supported by the hardware.

6.3.3 Composition

The controls module is composed of the following subcomponents:

- Vector Nav VN-100 IMU
- Bar30 Pressure Sensor
- Blue Robotics T200 Thruster
- Blue Robotics Ping1D Sonar
- Teledyne Pathfinder DVL

6.3.4 User Interactions

- Retrieving data from the IMU sensor
- Retrieving data from the DVL sensor
- Retrieving data from the Barometer
- Retrieving and processing the data from the Hydrophones
- Sending commands to motor controller
- Sending data through the Sonar
- Publishing data to ROS

6.3.5 Resources

- IMU `https://www.vectornav.com/products/vn-100`
- Barometer `https://www.bluerobotics.com/store/electronics/bar30-pressure-sensor-r1/`
- Thruster `https://www.bluerobotics.com/store/electronics/t200-thruster-r2/`
- Sonar `https://www.bluerobotics.com/store/electronics/ping1d-sonar-r2/`
- DVL `https://www.teledyne-rdi.com/products/instruments/pathfinder-dvl/`

6.3.6 Interface/Exports

- Publish IMU data to ROS
- Publish DVL data to ROS
- Publish Barometer data to ROS
- Publish Hydrophone data to ROS
- Publish Sonar data to ROS
- Publish Thruster data to ROS

6.4 Localization Module

6.4.1 Responsibilities

The primary responsibility of this component is to allow the RoboSub to know with a certain degree of certainty its location within a three-dimensional space. This component will take in sensor data from the inertial measurement unit (IMU), sonar, hydrophones, and barometer and use these data to estimate a position. It will then broadcast this position information through an ROS node to make it available to the rest of the software systems.

6.4.2 Contraints

The main limitation in this component is that localization is probabilistic due to the inherent noise in the sensor data. Sensor noise is unavoidable therefore solutions that seek to minimize the noise's impact in the system will help to reduce the propagation of errors in the estimates providing an overall better "guess" at the RoboSub location.

6.4.3 Composition

TBD.

6.4.4 User Interactions

The localization component will be used by the following:

- Autonomy: Pathfinding and decision-making.
- Computer vision: Landmark discovery.
- Mapping: S.L.A.M.

6.4.5 Resources

The localization component requires sensor data from the Controls module. Localization will process this data into position information. This component will work in conjunction with the Mapping component to produce position and environment information.

6.4.6 Interface/Exports

TBD.

6.5 Mapping Module

6.5.1 Responsibilities

The primary responsibility of this component is to create a three-dimensional map of the space surrounding the RoboSub. This component will also be able to place landmarks or key locations within this map. The goal is to allow the RoboSub to search its environment and keep track of everything of the boundaries and landmarks found within those boundaries.

6.5.2 Contraints

6.5.3 Composition

6.5.4 User Interactions

6.5.5 Resources

6.5.6 Interface/Exports

7 Detailed Lower Level Component Design

7.1 Name of Class or File

7.1.1 Classifications

7.1.2 Processing Narrative(PSPEC)

7.1.3 Interface Description

7.1.4 Processing Details

7.1.5 Design Class Heirarchy

7.1.6 Restrictions/Limitations

7.1.7 Performance Issues

7.1.8 Design Contraints

7.1.9 Processing Detail For Each Operation

8 Database Design

Not applicable.

9 User Interface

9.1 Overview of User Interface

At the end of this project, there will be a custom user interface designed specifically for Lanturn to aid in interacting with the running ROS system.

While custom interfaces are implemented, there are already implemented ROS integrated User Interfaces, RQT and Rviz.

RQT is the one that can be used by any ROS system and be the one that will be demonstrated here.

9.2 Screen Frameworks or Images

9.2.1 RQT

RQT is a user interface that comes packages with ROS and other visualization tools. Ir can be used to read, send and interpret data. It allows a user to interact with the ROS system the GUI is connected to. It uses the concepts of plugins to allow the user to view different forms of data and interpret them in different ways.

In this image, there are six plugins open.

1. Top Left. The Process Monitor plugin helps the user see all processes interacting with or through the ROS system.

2. Top Center. The MatPlot plugin can interpret data in a grid form and, in the context of Lanturn, it can be used to interpret stability, velocity and other odometry data.
3. Top Right. The Message Publisher allows the user to send different messages to nodes in the ROS system.
4. Bottom Right. The Console plugin displays log messages the nodes in the ROS system are saving.
5. Bottom Center. The Image View plugin displays images that are being streamed through a topic.
6. Bottom Right. The Topic Monitor shows the topics in the current ROS system and allows users to see what messages are going through the topic.

9.3 User Interface Flow Diagrams

The RQT user interface uses plugins to display information giving the user full control as to what is displayed and what is not. This means the user decides on the flow of the user interface, meaning, any flow diagram displayed here would miss the point of the modularity of this user interface design.

10 Requirements Validation and Verification

11 Glossary

12 References