T.R.A.L.A.L.E.R.O

(Tatical Recon and Aerial Logistics for Autonomous Localization Engagement and Routing Operations)

Universidade de Aveiro(*UA*)

Departamento de Electrónica, Telecomunicações e Informática(*DETI*)

Mestrado em Engenharia de Computadores e Telemática (*MECT*)

Tatical Recon and Aerial Logistics for Autonomous Localization Engagement and Routing Operations

Authors

Anderson Lourenço - 108579
aaokilourenco@ua.pt

António Almeida - 108250
ant.almeida@ua.pt

May 31, 2025

Abstract

T.R.A.L.A.L.E.R.O (Tactical Recon and Aerial Logistics for Autonomous Localization Engagement and Routing Operations) is a project aimed at improving autonomous vehicle navigation in urban environments by addressing the challenge of finding available parking. The system employs a drone as an aerial scout on demand, capable of detecting open parking spaces using computer vision and transmitting these data to an autonomous vehicle via DENM messages over MQTT. The architecture integrates ROS2 and Autoware to process inter system communication and enable the vehicle to autonomously react and navigate to the designated spot. The project implementation includes a custom drone algorithm for scanning parking lots, converting GPS data to Cartesian coordinates, and sending actionable information to the vehicle. Simulation results validate the feasibility of this coordinated approach, setting the groundwork for real world deployment in smart city ecosystems.

Index

Chapter 1

Introduction

In recent years, the capabilities and affordability of unmanned aerial vehicles (UAVs) have advanced significantly, positioning them as practical assets in a wide range of applications, from logistics to environmental monitoring and intelligent urban infrastructure. This project explores the integration of UAVs into autonomous vehicle (AV) ecosystems to enhance their environmental awareness and operational effectiveness, particularly in urban parking scenarios.

Autonomous vehicles often struggle to locate available parking in dynamic city environments due to limited onboard sensor visibility. Static maps and short range sensors are insufficient for reliably identifying parking availability beyond the vehicle's immediate field of view. To address this limitation, the T.R.A.L.A.L.E.R.O project (Tactical Recon and Aerial Logistics for Autonomous Localization Engagement and Routing Operations) introduces an aerial reconnaissance system using a drone to identify open parking spots and relay that data to the AV in real time.

The system combines advanced technologies such as computer vision (YOLOv8), inter system communication using MQTT and DENM, and robotics frameworks including ROS2 and Autoware. The drone autonomously navigates above a designated area, detects and classifies parking space occupancy, moves onto the closest parking spot and transmits their GPS coordinates to the vehicle, which then converts to Cartesian coordinates and updates its route to park accordingly.

This paper details the development and integration of these technologies within a simulation environment. Ultimately, the project aims to transition this proof-of-concept from simulation to real world testing, highlighting the potential of UAV assisted navigation as a scalable solution for enhancing autonomous mobility in smart cities.

Chapter 2

State of the Art

In this section, we will showcase all the previous work that was used in our project. We will begin with the existing research on drone control and then move on to the application used to making available a user's geolocation.

2.1 A mission planning framework for fleets of connected drones

On their master's thesis [1] introduced a drone control framework and mission planning solution made specifically for fleets of aerial drones. Most public, non-commercial solutions offer limited functionality for mission planning, typically consisting of a linear mission flow for a predeter- mined aerial drone, where actions taken at each step are chosen from a non- extensible set. This framework overcomes those limitations and offers flexible mission planning, with non-linear missions that can change the course of a drone at any time during its sequence of actions while being able to request data from its own sensors and reacting to that data. All of this was achieved using a groundstation that is responsible for the con- trol and management of every drone. For the mission planning it was developed a Domain-Specific Language (DSL) built on top of the Apache Groovy language, allowing for logical conditional plans and providing user-friendly methods to de- clare common actions such as taking off, moving, turning, retrieving sensor data, landing, and returning to the home position. Every command was then translated and sent to the Drone trough ROS2.

2.2 Object Detection for Parking Space Analysis

2.2.1 YOLO Models

YOLO (You Only Look Once) is a widely adopted real time object detection algorithm known for its balance between speed and accuracy, making it highly suitable for deployment in resource constrained environments such as UAVs.

In this project, YOLOv8 is employed to identify and classify parking spaces as either occupied or vacant. This task is critical for extending the perception capabilities of autonomous vehicles, allowing them to make parking decisions beyond the limitations of their onboard sensors.

For training and evaluation, the project utilizes a combination of the public `PKLot.v2-640.yolov8-obb` dataset, specifically designed for parking lot detection , using oriented bounding boxes, and a private dataset collected from real world, open air parking scenario. This hybrid dataset ensures that the detection model generalizes well across varying

perspectives and parking configurations.

YOLOv8's support for OBB is particularly beneficial in aerial views, where perspective distortion can render traditional axis-aligned bounding boxes ineffective.

2.3 MQTT and DENM

Reliable and low latency communication between distributed agents is essential for coordinated autonomous systems. In the context of T.R.A.L.A.L.E.R.O, communication between the UAV and the autonomous vehicle is facilitated using MQTT and DENM.

2.3.1 MQTT

MQTT (Message Queuing Telemetry Transport) is a lightweight, publish-subscribe messaging protocol designed for low bandwidth and high latency networks. It has become a standard in IoT and robotics due to its minimal overhead, simplicity, and reliability. MQTT enables asynchronous communication between loosely coupled systems, making it well suited for dynamic environments such as drone vehicle coordination.

In this project, MQTT is used to transmit parking availability data from the drone to the autonomous vehicle. The drone publishes processed data (parking spot coordinates), while the vehicle subscribes to the relevant topics to receive updates in real time.

2.3.2 DENM

DENM (Decentralized Environmental Notification Message) is part of the ETSI ITS-G5 protocol suite. It is used for event based communication between vehicles and infrastructure, such as hazard warnings or situational alerts.

In T.R.A.L.A.L.E.R.O, DENM is adapted to encapsulate parking coordinates as an event message, allowing the autonomous vehicle to interpret the data and move to the corresponding position.

Together, MQTT and DENM form a hybrid communication layer MQTT providing the transport mechanism and DENM offering structured semantics for interpreting spatial events. This layered approach ensures efficient communication between the drone and vehicle, enabling real time, cooperative, decision making in urban navigation scenarios.

2.4 Autonomous Vehicle Parking Systems

2.4.1 Autoware Universe

Autoware Universe serves as a foundational pillar within the Autoware ecosystem, playing a critical role in enhancing the core functionalities of autonomous driving technologies. This repository is a pivotal element of the Autoware Core/Universe concept, managing a wide array of packages that significantly extend the capabilities of autonomous vehicles.

2.4.2 Parking Component Overview

Inside of the Autoware Universe there is a built in component that allows the vehicle to park on a pre-defined parking spot. This is the component on which we will develop.

2.4.3 Purpose and Capabilities

The Parking module in Autoware Universe is designed to automate parking operations including:

- Parallel and perpendicular parking
- Pull-over maneuvers
- Vacant space detection
- Path planning and execution within constrained areas

The Parkign functionality integrates multiple subsystems, which the following are crucial for this project:

- **Behavior Planning**: Determines whether parking is possible in the identified region and selects the most appropriate parking strategy.
- **Motion Planning**: Utilizes optimization-based planners like Reeds-Shepp or hybrid A* to generate feasible trajectories in tight spaces.
- **Control**: Converts planned trajectories into steering and speed commands, ensuring smooth, accurate execution with low-speed dynamics in mind.

These systems work together through ROS 2 nodes, exchanging information in real time. The parking-related nodes often subscribe to topics like */localization/pose*, and publish to */control/command*.

In a typical deployment, the parking scenario is initiated via user interface commands or scenario runners. Vehicles can autonomously detect parking lots or receive designated space locations through V2X communication. This aligns well with T.R.A.L.A.L.E.R.O's goals of combining infrastructure support (e.g., drone-base free space detection) with onboard intelligence.

The parking module also supports both fully autonomous parking (with onboard perception and planning and cooperative parking (with external guidance such as cloud-based maps or infrastructure sensors). This modularity makes it particularly suitable for research and development projects in smart cities or connected environments.

Chapter 3

Conceptual Modeling

3.1 Functional Requirements

- **Drone Navigation and Area Scanning :** The drone must autonomously navigate to a predefined parking area and maintain stable flight for visual scanning.
- **Parking Spot Detection :** The system must detect and classify parking spaces as either free or occupied using YOLOv8.
- **Data Communication :** The drone must send parking coordinates to the autonomous vehicle using MQTT.
- **DENM Message Encoding :** Parking coordinates must be formatted and transmitted as DENM messages to ensure compatibility with ETSI standards.
- **Message Reception and Parsing by Vehicle :** The autonomous vehicle must subscribe to the correct MQTT topic, receive DENM messages, and extract relevant parking information.
- **Route Planning and Goal Setting :** The vehicle must be able to autonomously plan a route to the identified free parking spot using Autoware.
- **Full Mission Execution :** The system must be able to complete an end-to-end mission cycle: drone launch, detection, communication, vehicle response, and parking.

3.2 Non-Functional Requirements

- **Real-Time Performance :** The system must operate with low latency to ensure timely delivery of parking data and vehicle response.
- **Reliability :** The system must ensure a high level of uptime for both the drone and communication components.
- **Scalability :** The communication architecture must support potential expansion to multiple drones or vehicles in a larger smart city deployment.
- **Modularity :** Each subsystem (vision, communication, navigation) must be independently testable and replaceable without major redesign.
- **Interoperability :**The system should conform to ITS communication standards to ensure compatibility with future smart infrastructure.

- **Maintainability :** The software and configuration should be well documented and organized for easy debugging, updating, or upgrading.
- **Resource Efficiency :** The drone must perform inference and communication using limited onboard computational resources without significant performance degradation.

3.3 Actors

- UAV (Drone)
- Autonomous Vehicle
- Human Operator (for initial mission setup and monitoring)

3.4 Use Cases

In a smart city scenario, an autonomous vehicle requires parking near a specific destination but lacks sufficient sensor visibility to detect available spaces beyond its immediate surroundings. To address this, a UAV is dispatched to perform aerial reconnaissance of the parking area.

Using YOLOv8 for real-time object detection, the drone identifies and classifies parking spots as occupied or free, moving to the free parking spot. Once in position, it transmits its GPS coordinates to the AV via a DENM message over MQTT. Upon receiving the data, the AV processes it through its Autoware based stack, and autonomously navigates to it.

Chapter 4

Architecture

In this chapter, we will talk about the architecture of our project. Our entire architecture is shown on figure 4.1.

As our project was simulated, all the components of the architecture are running on two computers.The base architecture had the Ground Station and the Drone Core. We build upon it adding the Communication and Data Vision, connecting it through Wi-Fi (as both the test computers were not ITS-G5 capable) to an MQTT Broker in the AV simulated environment (Autoware).

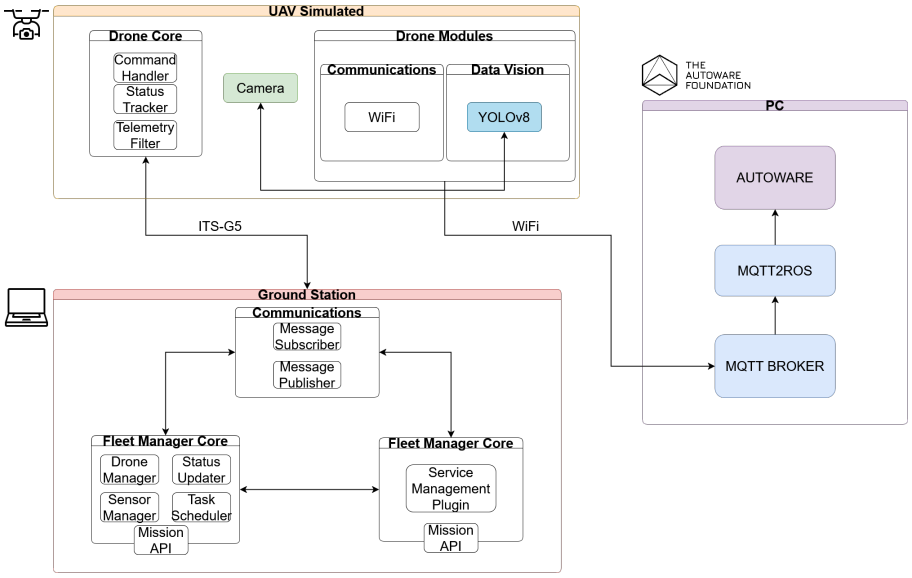


Figure 4.1: Architecture

4.1 UAV Subsystem

The UAV is responsible for aerial reconnaissance of a predefined parking area. Its subsystem includes the following functional components:

- **Autonomous Navigation:** The UAV is pre-programmed to fly to a known parking lot and maintain stable flight during the mission. The navigation is simulated, but replicates real UAV behavior.
- **Parking Spot Detection:** The UAV processes aerial imagery using the YOLOv8 object detection model to identify parking spots. Inference is conducted onboard, and the first available spot is selected.

- **Positioning Above Target Spot:** After identifying a free parking spot, the UAV moves directly above it and holds its position. This allows its GPS coordinates to represent the center of the available spot.
- **Communication via MQTT and DENM:** The UAV encodes its current GPS position into a DENM message and publishes it via MQTT to a topic subscribed to by the AV. This message acts as an event notification identifying the location of the available parking spot.

4.2 AV Subsystem

The Autonomous Vehicle (AV) receives location data from the UAV and navigates to the corresponding parking spot. Its subsystem includes:

- **MQTT Subscription and Message Handling:** The AV subscribes to the MQTT topic used by the UAV. When a DENM message is received, the AV extracts the GPS coordinates of the indicated parking spot.
- **Coordinate Conversion and Goal Setting:** The GPS coordinates are converted into Cartesian coordinates to match the vehicle's Lanelet2 map. The Autoware stack is used to set the new goal pose.
- **Path Planning and Execution:** Once the goal is set, Autoware plans an optimal route and the AV executes the navigation task autonomously. This includes path following, obstacle avoidance, and final parking.
- **Simulation and Visualization:** The AV operates in a fully simulated environment with all behavior visualized using RViz. The results are used to validate end-to-end functionality of the T.R.A.L.A.L.E.R.O system.

4.2.1 Communication and Integration

The communication between UAV and AV is facilitated by an MQTT broker operating in the simulation environment. The UAV acts as the publisher and the AV as the subscriber. DENM messages are used for structured event communication, conforming to Intelligent Transport System (ITS) standards. The integration of ROS2, YOLOv8, and Autoware enables real-time perception, communication, and planning in a modular and scalable architecture.

4.2.2 Simulation Environment

All components are tested in a simulated smart city environment using ROS2 and Autoware. A custom Lanelet2 map models the road and parking area. UAV and AV behavior is emulated to validate their interaction in a controlled, reproducible setup. The simulation framework allows for full visualization, debugging, and iterative development without the risks or limitations of physical hardware.

Chapter 5

Implementation

5.1 UAV Mission Implementation

In this section, we will explain what we added in order to make the drone able to fly, analyze the parking spot availability, and communicate with the AV.

5.1.1 Autonomous Flight Path

At the start of the mission, the drone is required to navigate to a predefined position near the parking lot to initiate the scanning and detection phase.

Leveraging the existing message passing infrastructure, the drone can be directed to a specific coordinate using the "GOTO" ROS2 message. Once at the target location, it must adjust its orientation and reposition itself to the nearest available parking spot.

This phase of the operation takes place after the drone has completed the analysis of parking spot availability. The core navigation algorithm responsible for positioning the drone above the selected parking spot is structured as follows:

1. Rotate to the closest available spot
2. Move forward until :
   (a) The UAV rotational value is greater than 10 OR
      i. If so, the drone will rotate and start the forward movement again.
   (b) The UAV position is in a radius of 50px to the center of the bounding box
3. If the position is reached, send the coordinates to the AV

The following pseudocode outlines the core logic that serves as the foundation for the drone's movement algorithm:

```
    def main():
        GOTO(x_coordinate,y_coordinate)
        WAIT(finish)
        ...
        while(true)
            TURN(degrees)
            WAIT(finish)
            MOVE(foward)
            while(angle < 10 and distance > 50)
                continue
            if(distance < 50)
                break
        send message
```

5.1.2 Parking Space Detection Module

While the drone is in flight, a parallel process continuously runs YOLO based bonding box object detection on the video feed captured by the onboard camera.

Training

To address the limited diversity in the initial dataset, we applied data augmentation techniques,most notably rotation, to simulate different drone angles and improve the model's robustness to varied perspectives. This augmentation process helped increase the dataset's variability and made the YOLOv8 model more resilient to orientation changes in the input frames. The final dataset was used to train the model over 30 epochs, using a batch size of 16 frames per set.

Detection

Once the model was trained, we developed a parallel processing module designed to continuously capture the video feed, analyze each frame using the pre-trained YOLO model, and extract key data. Specifically, this module identifies the nearest available parking spot and outputs its relative distance and orientation (rotation) with respect to the drone's current position. To avoid introducing latency in the drone's navigation logic, the YOLO detection module is executed as a separate parallel process. It communicates with the movement script via a local MQTT broker, allowing real-time data exchange without blocking or delaying the drone's flight operations.

Figure 5.1: Yolo Model after Training

Resume of the algorithm until now

This two scripts run in conjunction to achieve it's final goal of reaching the middle position of the closest available spot.

```
1     def on_message(data):
2         yawRateToTurn = data["yawRate"]
3         distance = data["distance"]
4
5     def main:
6         ...
7          while(true)
```

```
            TURN(degrees)
            WAIT(finish)
            MOVE(foward)
            while(angle < 10 and distance > 50)
                continue
            if(distance < 50)
                break
        send message
```

```
    def yolo:
        frame = readVideo()
        results = YOLO(frame)
        closest = results[0]
        for(result in resutls)
            if(result["angle"] + result["distance"] * Weigh < closest)
                closest = result

        send closest
```

5.2 Communication Implementation

5.2.1 DENM Message Generation

A base DENM message structure was defined within the code, allowing for efficient reuse by updating only the necessary fields such as coordinates and the `eventType`. Before transmission, the message is dynamically updated with the drone's current GPS coordinates,specifically, the latitude and longitude values,ensuring that the information accurately reflects the location of the identified parking spot.

The UAV communicates with the AV over a Wi-Fi connection, publishing the DENM message to an MQTT broker hosted on the AV's system. The AV, which is subscribed to the appropriate MQTT topic, receives the incoming message, extracts the GPS coordinates, and uses this data to set its navigation goal.

5.2.2 Autonomous Vehicle Reception

How does the autonomous vehicle receive the DENM message?

The autonomous vehicle receives DENM messages through an MQTT-based communication system implemented in the `GoalRemapper` ROS2 node. The reception process involves:

1. **MQTT Subscription**: The vehicle subscribes to the MQTT topic `autoware/goal_remap` (configurable via launch parameters) where DENM messages are published by the drone system.

2. **Message Reception**: The `on_message` callback function is triggered when a new MQTT message arrives:

```
def on_message(self, client, userdata, msg):
    try:
        payload = msg.payload.decode('utf-8')
        self.get_logger().info(f"Received MQTT with a DENM message: {
    payload}")
        msg_dict = json.loads(payload)

        # Detect DENM by structure (no 'command', but has 'management' and
    'eventPosition')
        if (
```

```
            'command' not in msg_dict and
            'management' in msg_dict and
            'eventPosition' in msg_dict['management']
        ):
            self.process_denm_message(msg_dict)
        else:
            self.process_mqtt_message(msg_dict)
    except json.JSONDecodeError:
        self.get_logger().error(f"Failed to parse MQTT message as JSON")
```

Software/Hardware on the Vehicle Side

Software Components:

- **ROS2 Framework**: The core communication middleware running on the autonomous vehicle
- **Goal Remapper Node**: A custom ROS2 node (`goal_remapper_node.py`) that handles DENM message reception and processing
- **Autoware**: The autonomous driving software stack that receives remapped navigation goals
- **MQTT Client**: Paho MQTT client for communication with the drone's MQTT broker
- **Mosquitto Broker**: MQTT message broker for inter-system communication (configurable via `mosquitto.conf`)

Hardware Requirements:

- Network interface capable of Wi-Fi/Ethernet communication with the drone system
- Computing platform running Ubuntu with ROS2 (as evidenced by the launch configuration)

Message Parsing and Interpretation

The vehicle's control system parses and interprets DENM messages through the following process:

1. **Message Structure Detection**: The system identifies DENM messages by their specific structure (containing `management` and `eventPosition` fields without a `command` field).
2. **Geographic Coordinate Extraction**: The `process_denm_message` function extracts critical information:

```
def process_denm_message(self, msg_dict):
    try:
        event_pos = msg_dict["management"]["eventPosition"]
        latitude = float(event_pos["latitude"])
        longitude = float(event_pos["longitude"])
        altitude = float(event_pos["altitude"]["altitudeValue"])

        # Default heading
        heading = 0.0
        if (
            "location" in msg_dict and
            "eventPositionHeading" in msg_dict["location"] and
            "value" in msg_dict["location"]["eventPositionHeading"]
        ):
            heading = float(msg_dict["location"]["eventPositionHeading"]["
    value"])
```

3. **Coordinate Transformation**: The system converts GPS coordinates (lat/lon) to local map coordinates using the `latlon_to_xy` function with predefined map origins:
   - Latitude: 40.634646
   - Longitude: -8.659986
   - Elevation: 0.0
4. **Goal Message Creation**: A ROS2 `PoseStamped` message is created for Autoware:

```
# Create PoseStamped message
goal_msg = PoseStamped()
goal_msg.header.stamp = self.get_clock().now().to_msg()
goal_msg.header.frame_id = 'map'
goal_msg.pose.position.x = x
goal_msg.pose.position.y = y
goal_msg.pose.position.z = altitude
goal_msg.pose.orientation = self.heading_to_quaternion(heading)
```

5. **System Integration**: The processed goal is published to Autoware's navigation system, and after a 3-second delay, the system automatically engages Autoware using the `engage_autoware` function:

```
def engage_autoware(self):
    try:
        self.get_logger().info("Publishing engage command to Autoware...")
        command = '''ros2 topic pub /autoware/engage
    autoware_auto_vehicle_msgs/msg/Engage '{engage: True}' -1'''
        subprocess.run(command, shell=True, stderr=subprocess.DEVNULL)
    except Exception as e:
        self.get_logger().error(f"Failed to publish engage command: {str(e)
    }")
```

The entire system is launched using the `goal_remapper_launch.py` file, which configures the MQTT broker settings and map origin parameters for proper message reception and processing.

5.3 Autonomous Vehicle Control Integration

Specific Commands and API Calls for Vehicle Control

The autonomous vehicle control integration utilizes Autoware's ROS2-based API system through the `GoalRemapper` node. The specific commands and API calls include:

1. Initial Pose Setting

The system first establishes the vehicle's initial position using ROS2's topic publication:

```
def publish_initial_pose(self):
    """Publish initial pose command"""
    try:
        self.get_logger().info("Publishing initial pose to /initialpose3d (once
    )...")
        subprocess.run([
            "ros2", "topic", "pub", "--once", "/initialpose3d", "geometry_msgs/
    PoseWithCovarianceStamped",
            "{header: {stamp: {sec: 1748258635, nanosec: 780595328}, frame_id:
    'map'},"
            "pose: {pose: {position: {x: 36.7392578125, y: -46.01408004760742,
    z: 2.450119733810425},"
```

```
            "orientation: {x: 0.0, y: 0.0, z: -0.3658047030231997, w:
    0.9306916348856418}},"
            "covariance: [1.0, 0.0, 0.0, 0.0, 0.0, 0.0,"
            "0.0, 1.0, 0.0, 0.0, 0.0, 0.0,"
            "0.0, 0.0, 0.01, 0.0, 0.0, 0.0,"
            "0.0, 0.0, 0.0, 0.01, 0.0, 0.0,"
            "0.0, 0.0, 0.0, 0.0, 0.01, 0.0,"
            "0.0, 0.0, 0.0, 0.0, 0.0, 0.2]}}"
        ], check=True, stderr=subprocess.DEVNULL)
    except Exception as e:
        self.get_logger().error(f"Failed to publish initial pose: {str(e)}")
```

2. Goal Position Publishing

When a DENM message is received, the system publishes navigation goals to Autoware using the `/planning/mission_planning/goal` topic:

```
def publish_goal(self):
    """Publish the remapped goal once"""
    if self.remapped_goal is not None and not self.goal_published:
        self.goal_pub.publish(self.remapped_goal)
        self.goal_published = True
        self.get_logger().info(f'Published remapped goal: x={self.remapped_goal
    .pose.position.x:.2f}, '
                               f'y={self.remapped_goal.pose.position.y:.2f}')
    else:
        self.get_logger().warn('No goal to publish or goal already published')
```

3. Vehicle Engagement Command

After setting the goal, the system automatically engages Autoware's autonomous driving mode:

```
def engage_autoware(self):
    """Publish engage command to Autoware after delay."""
    try:
        self.get_logger().info("Publishing engage command to Autoware...")
        command = '''ros2 topic pub /autoware/engage autoware_auto_vehicle_msgs
    /msg/Engage '{engage: True}' -1'''
        subprocess.run(command, shell=True, stderr=subprocess.DEVNULL)
    except Exception as e:
        self.get_logger().error(f"Failed to publish engage command: {str(e)}")
```

Navigation and Parking Logic

1. Coordinate Transformation Logic

The system processes GPS coordinates from DENM messages and transforms them into the vehicle's local coordinate system:

```
def latlon_to_xy(self, lat, lon):
    """Convert lat/lon to x/y coordinates using map origin"""
    R = 6371000  # Earth radius in meters

    # Convert to radians
    lat_rad = math.radians(lat)
    lon_rad = math.radians(lon)
    origin_lat_rad = math.radians(self.map_origin_lat)
    origin_lon_rad = math.radians(self.map_origin_lon)

    # Calculate differences
```

```
    dlat = lat_rad - origin_lat_rad
    dlon = lon_rad - origin_lon_rad

    # Convert to meters
    x = R * dlon * math.cos(origin_lat_rad)
    y = R * dlat

    return x, y
```

The map origin is configured in the launch file with specific coordinates:

- Latitude: 40.634646
- Longitude: -8.659986
- Elevation: 0.0

2. DENM Message Processing Logic

The vehicle processes incoming DENM messages with the following logic:

```
def process_denm_message(self, msg_dict):
    """Process incoming DENM MQTT messages and remap goal."""
    try:
        event_pos = msg_dict["management"]["eventPosition"]
        latitude = float(event_pos["latitude"])
        longitude = float(event_pos["longitude"])
        altitude = float(event_pos["altitude"]["altitudeValue"])

        # Default heading
        heading = 0.0
        if (
            "location" in msg_dict and
            "eventPositionHeading" in msg_dict["location"] and
            "value" in msg_dict["location"]["eventPositionHeading"]
        ):
            heading = float(msg_dict["location"]["eventPositionHeading"]["value
    "])

        # Convert lat/lon to x/y using map origin
        x, y = self.latlon_to_xy(latitude, longitude)

        # Create PoseStamped message
        goal_msg = PoseStamped()
        goal_msg.header.stamp = self.get_clock().now().to_msg()
        goal_msg.header.frame_id = 'map'
        goal_msg.pose.position.x = x
        goal_msg.pose.position.y = y
        goal_msg.pose.position.z = altitude
        goal_msg.pose.orientation = self.heading_to_quaternion(heading)
```

3. Automated Parking Sequence

The parking maneuver follows a timed sequence:

1. **Immediate Goal Setting**: Upon receiving a DENM message, the system immediately sets the parking goal
2. **3-Second Delay**: A timer is started for automatic engagement
3. **Automatic Engagement**: After 3 seconds, Autoware is automatically engaged to begin navigation

```
# Start a timer to engage after 3 seconds (one-shot)
self._engage_timer = self.create_timer(3.0, self._engage_autoware_oneshot)

def _engage_autoware_oneshot(self):
    self.engage_autoware()
    self._engage_timer.cancel()
    self.get_logger().info("Goal remapping cycle completed - waiting for next
    MQTT message")
```

4. Integration with Autoware Planning System

The system is designed to work with Autoware's planning simulator, as evidenced by the `runAutoware.sh` script:

```
cd /home/pixkit/pix/pixkit/Autoware
source install/setup.bash
ros2 launch autoware_launch planning_simulator.launch.xml map_path:=$HOME/RSA-
    Project/mapa_rsa vehicle_model:=sample_vehicle sensor_model:=
    sample_sensor_kit
```

The system uses the map configuration from the `mapa_rsa` directory, which includes:

- `lanelet2_map.osm`: OpenStreetMap format road network
- `map_config.yaml`: Map configuration parameters
- `map_projector_info.yaml`: Coordinate projection information

This integration allows the autonomous vehicle to receive parking coordinates from the drone system via DENM messages and automatically navigate to and park at the designated location using Autoware's sophisticated path planning and vehicle control capabilities.

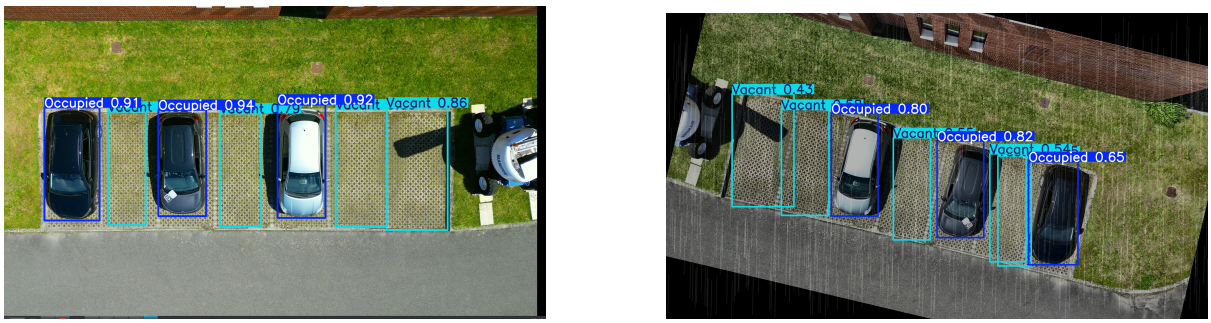
Chapter 6

Results and Evaluation

The T.R.A.L.A.L.E.R.O system was tested in a simulated, using a previously taken video of a parking spot, to evaluate detection accuracy, communication reliability, and overall system coordination.

6.1 Parking Space Detection

Using YOLOv8 with oriented bounding boxes, the drone accurately identified free parking spots from aerial views. A combination of public and real-world datasets, enhanced with data augmentation, allowed the model to generalize well across different angles. The detection module ran in parallel with drone navigation to ensure real time responsiveness.6.1



(a) Normal Frame

(b) Augmented Frame

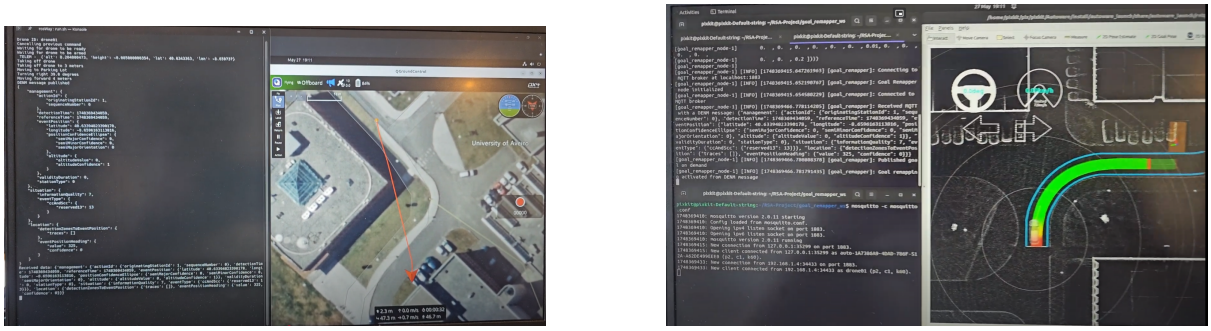
Figure 6.1: Parking Spot Detection in Different environments

6.2 Communication Performance

The system used MQTT for transport and DENM for structured messages. This hybrid approach enabled low-latency, reliable communication between the UAV and the autonomous vehicle. All DENM messages were successfully received and interpreted, triggering the correct navigation responses.6.2

6.3 End-to-End System Performance

The full cycle,from drone deployment, parking space detection, message transmission, to vehicle parking, was executed seamlessly. The AV converted GPS data into local coordinates, planned a path using Autoware, and completed the parking maneuver autonomously. Visualization in RViz confirmed the system's effectiveness. 6.3



(a) Sending DENM Message From UAV

(b) Receiving DENM Message In AV

Figure 6.2: DENM Message Through MQTT

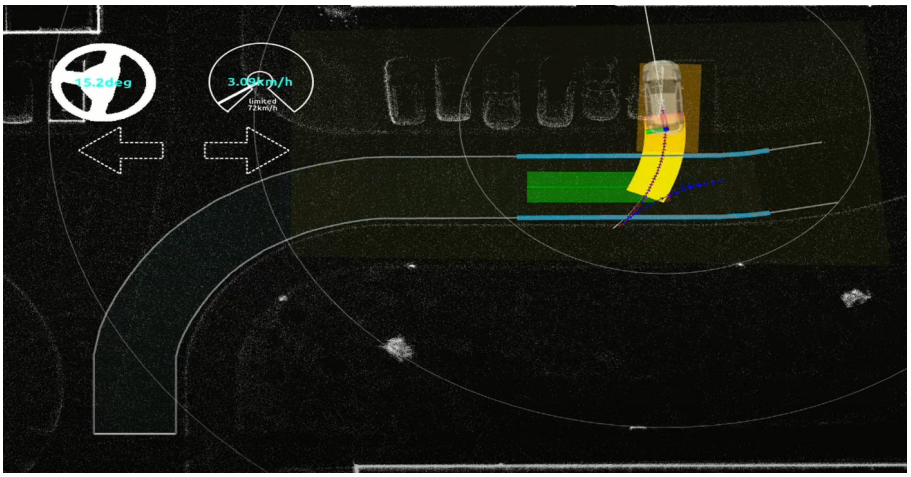


Figure 6.3: Finish Postion in RViz

6.4 Challenges and Lessons Learned

Throughout the project, several challenges emerged. Initially, the YOLO script was embedded within the main script, but due to performance issues and slow execution, it was later separated to improve efficiency.

The model underwent several training iterations to best fit our use case. Initially trained only on a public dataset, it failed to meet expectations, specifically, it struggled to analyze the parking spots effectively. Incorporating a private dataset improved accuracy, but the model could only accurately detect spots that were parallel to the camera view. It was only after applying data augmentation that the model became capable of reliably identifying parking spots from various angles.

Chapter 7

Conclusion and Future Work

The T.R.A.L.A.L.E.R.O system successfully demonstrated the integration of aerial reconnaissance with autonomous vehicle navigation for real time parking space detection. By combining UAV based computer vision, MQTT and DENM communication, and the Autoware platform, the system achieved end-to-end coordination, from detecting a free parking spot to guiding the vehicle to park autonomously.

Despite challenges such as managing performance limitations and ensuring accurate spatial mapping, the project achieved its core objectives. The modular architecture, efficient communication design, and improved object detection model laid a solid foundation for real world applications in smart city environments.

This work not only highlights the potential of cooperative systems in urban mobility but also opens the door for future enhancements, more complex parking scenarios, and real world deployment.

Bibliography

[1] Ana Margarida Oliveira da Costa e Silva. (2021). *A mission planning framework for fleets of connected drones*. University of Aveiro.

[2] Autoware Universe Online documentation. HERE

Appendix

Links

- Demo Video: T.R.A.L.A.L.E.R.O
- Github: RSA-Project