DTEK2084 - Aerial Robotics and Multi-Robot Systems

Open Project - Final Report

Search and Rescue

Journey Through the Ups and Downs of Learning Multi-Robot Systems

---

GROUP 8

Tuisku Polvinen, Sandra Ekholm, Lipei Huo

The group members are Tuisku Polvinen, Sandra Ekholm and Lipei Huo.
Tuisku was a self-appointed team leader and architect as well as GitHub manager. Lipei was the master of physical robots and Sandra the navigation expert.

One of the main challenges would be to learn not only Multi-robot Systems, but even Multi-human Systems as our teamwork flow had not been the most effective in the previous project. We had aced working parallelly but a bit wrong: instead of developing different parts, we often worked on the same things at the same time unknownst to each other. This time we aimed to be more efficient.

Starting the Project

Initial Confusion

After our success with drone racing, we were ready for our next challenge. From our previous project we had learned that getting all of the different software working together was one of the biggest challenges. This time we wanted to minimize struggling with software, and thought at first that the approach with least problems would be using real robots – as Tuisku had had problems with getting Gazebo running, Sandra had gotten to the point where her ROS2 installation refused to cooperate and Lipei discovered that his graphics card didn't have what it took to simulate all the graphics.

Initial Project Plan: Application and Main Techniques

We are designing a search and rescue robot team, where a drone locates a person and calls a turtlebot equipped with medical aid to the site. When the turtlebot has arrived, the drone rises higher up to be visible for human rescuers who are looking for the site. A robotic search and rescue team is essential in real-world applications such as saving lives after a disaster like earthquake or wildfire. The robotic team can speed up the rescue operation and get to locations which can be inaccessible by humans, allowing the team to save human lives

which might be lost due to slow human rescue teams or due to the hazardous environment in which the human rescue team must operate in.

In our project, the drone is chosen due to its capability to survey large areas efficiently. This ensures fast location of the victims. But as the drone struggles with carrying heavy loads, a ground robot is needed to carry first aid medical stuff, such as water, medicine, sanitary towels etc., to the victim before the human rescuers arrive at the location. As a ground robot, we identify the usage of a larger robot (like Husky) would be beneficial providing robust movements on the unsteady ground. A Husky-robot would be able to carry even heavier loads, but for this project we choose to use a Turtlebot as the ground robot. The main reason behind choosing the turtlebot instead of husky, is that we have just started our robotics development journey which is why we believe utilization of simpler robots would allow us to learn more about a multi-robot system development. Furthermore, we have some experience of using turtlebots, thus configuring the robot might become easier allowing us to focus on the application development. Additionally, in a real world scenario, a turtlebot might be a more suitable choice over a husky due to its smaller size. The smaller size enables the turtlebot to make it to places where the bigger husky cannot reach such as a collapsed building in which only small entries allow the robot to enter the building.

Main techniques we identified as the most significant to carry out the project are multi-robot communication, autonomous moving, obstacle avoidance, computer vision, localization and environment mapping. Communication between the robots is essential to coordinate the task efficiently, and it is realised through the use of ROS2 topics which provide a flexible real-time message exchange mechanism. Autonomous movement allows for robots to navigate without human control, which enables fast and safe way to victim detection and system response. Intuitive obstacle avoidance of the robots ensures the robots remain operable, and they won't cause any damage to the environment which could complicate the rescue process. Computer vision is essential to detect the victim to be saved. Naturally, to navigate precisely and coordinate actions between the team members, localization is one of the key mechanisms in this project. In case, a precise map from the environment is not available, environmental mapping is necessary to build the map of the unknown environment, which helps in safer navigation improving path planning by taking into account the static obstacles.

However, as the project scope is wide, we decide to start the project with a step by step approach. The project scope is narrowed down first to only essential parts, after which we plan to incrementally decrease the scope. This ensures that we will have at least a base solution available when the project deadline approaches.

Teacher Comments on Initial Project Plan

After talking with the instructors we learned that our robot cooperation setup might require even more "advanced configuration" if we went the real robot route. One of the big problems they presented was that both of the robots required the control computer to connect to them via wifi – we would need to connect to two wifis at the same time, which sounded like a hassle. After a brief meeting, we decided that we could try to get the best of both worlds – Tuisku and Sandra would set up a simulator on their laptops and work on one robot each, while Lipei would make real-world development on his and work for dual-robot development.

The teacher deemed our scenario and approach good. They pointed out that given the severity of our considered scenario, no robots without the ability to localize itself would be used unless in a remote GNSS-denied location. Thus, we could use their own location information and share that between the robots quite easily. The teacher recommended we look into Pose messages.

Simulator Preparation

Instead of using Gazebo introduced during the course, we decided to learn and utilize Webots. During the course presentation sessions, one presentation was given on Webots, which convinced us on utilizing the simulator instead of Gazebo. Webots was deemed to be more user-friendly and easier to configure, making it a more deployable choice for a simulator as our primary interest during this project is to develop a multi-robot communication application instead of learning how to configure a simulator for a robotics project. Additionally, Webots allows for more realistic robotic behavior simulations with minimal configurations, due to its in-built models for a wide range of sensors and actuators.

In the beginning, we encountered interoperability problems with different ROS2 and Webots versions. The issues were solved by upgrading our Ubuntu operating systems to Ubuntu 24, and ROS2 versions to Jazzy Jalisco. By having a common and the newest operating system and ROS2 setup, interoperability of our codes could be maximized with up-to-date support from the ROS2 community.

We decided to use existing robots (drone and turtlebot) present in contemporary tutorials, to allow most of our focus to be on the actual development of the application instead of building and configuring the robots in simulators. One of the mainstream Webots packages for ROS2 was decided to be utilized throughout this project as the main tutorial (https://github.com/cyberbotics/webots_ros2), since it provides detailed functionality to work with ROS2 and Webots.

Tuisku's Experience of Struggling with Webots

I installed the Webots simulator on Ubuntu 20.04 where I had ros2 foxy fitzroy installed. The simulator seemed easy to use and intuitive, which made me quite happy! Some problems arose though when I tried to get it working with ros2. I installed the webots_ros2 package following the instructions in https://snapcraft.io/install/webots/ubuntu but could not get it to work. I decided to reinstall ros2 using a newer version (Jazzy Jalisco), and then install Webots again, this time according to the instructions from Jazzy Jalisco's documentation: https://docs.ros.org/en/jazzy/Tutorials/Advanced/Simulators/Webots/Installation-Ubuntu.html.

To ensure that a version compatible with my ros2 would be installed, I did not install Webots beforehand but waited until the webots_ros2 package installation notified that it was not installed and asked if I wanted to install it. To have the ~/ros2_ws/src folder mentioned nearing the end of the installation instructions, I chose the install webots_ros2 from sources option.

Success! Now the next challenge was to get something else than the demo to work. Luckily the Webots tutorials had covered this, and the next step would be to go through them.

Sandra's Experience of Struggling with Ubuntu and ROS2 Galactic

During the common project for this course, I managed to break my ROS2 distribution in such a way that building any ROS2 related projects led to a fatal failure. The fatal failure appeared even though the ROS workspace that was tried to be built was perfect and shouldn't have contained any issues (example workspace tried with *colcon build*: https://github.com/TIERS/drone_racing_ros2).

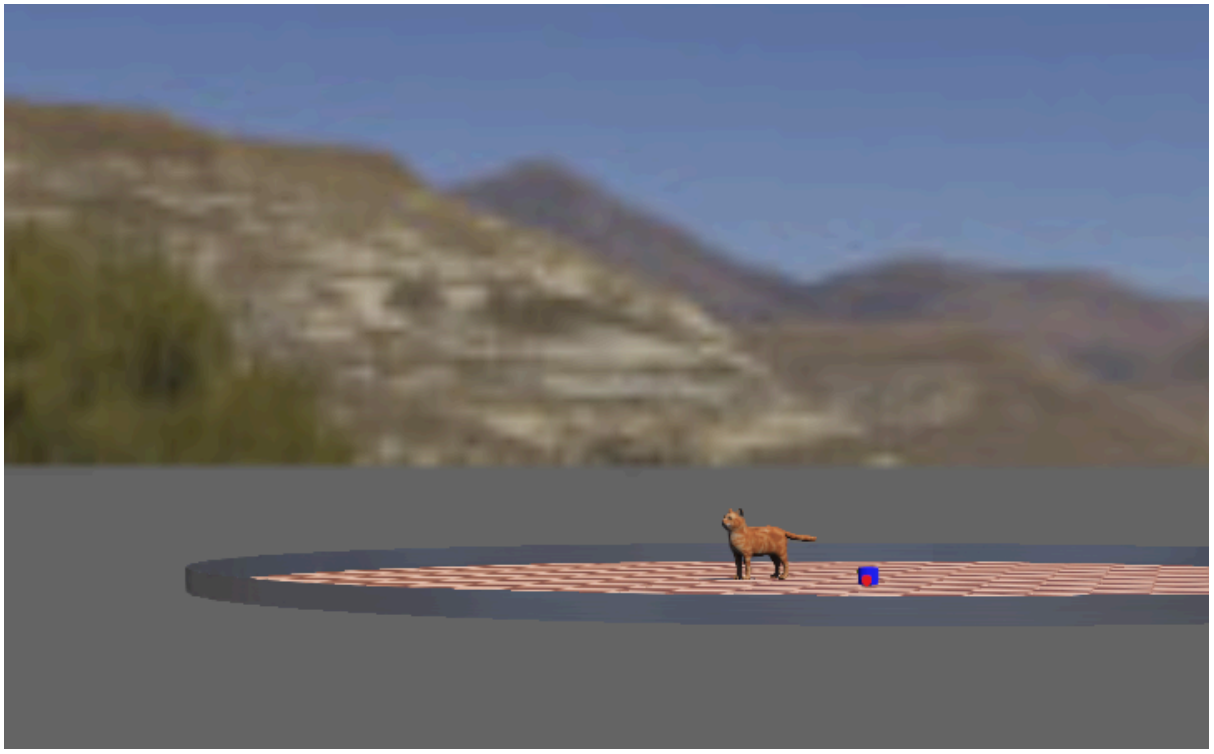
After trying to fix the issue by myself for almost five hours, I decided to ask for help from a lab instructor who looked at the terminal for five minutes and surrendered. ChatGPT was also consulted leading to no success.

For this project, we decided to try fixing the issue on my laptop by upgrading the Ubuntu version to 24 and by reinstalling ROS2. This was deemed to be a good idea since this way the ROS-version could be configured to be the same for me and Tuisku. Having the same ROS distribution to work with would lead to less interoperability issues caused by different ROS distributions! This would then lead to improved project team collaboration as both me and Tuisku would be working with the simulations. The ROS-nodes created wouldn't need to be modified for multiple ROS-versions. However, as Lipei would be working with real robots which operate with different ROS-versions, we expect that some modifications to the ROS-nodes must be done at some point to support all utilized distributions.

After successful installation of Ubuntu 24 and ROS2 Jazzy Jalisco, the same instructions for installing Webots simulator were followed as Tuisku had suggested. This led to a successful Webots simulator demonstration! Also the *colcon build* issues were left behind. Next step would be to cover the Webots tutorials in order to ensure the setup is actually working and to understand the simulator environment better.

Getting the Right Robots to the Simulation

By the help of tutorials, we managed to get the tutorial robot to function in the simulation. The next step was figuring out how to get the drone and the turtlebot there instead.



The new world with a tutorial robot and a cat

Finding the URDF for the Tello drone was easy. It had been provided for the first assignment so that everyone could use it in Gazebo. The question was, would the URDF model work just like that in Webots too? Upon closer inspection we found Gazebo-specific parts from the URDF file. To make things more straightforward, we decided to instead use the DJI Mavic drone that was included in the Webots tutorials and had all the sensors we needed.

For the turtlebot, we also decided to go with a model that was included with the Webots simulator. Using the ready-made robots from the tutorials would allow us to focus on the actual application and multi-robot communications. How these robots would work out with Lipei's findings with the actual turtlebot and drone we would find out later.

While Sandra and Tuisku were learning about the Webots simulator, we came upon very basic questions. How would we create a world with two robots? We started to solve this by creating a new package with its own world.

The new package was based on the Webots tutorial. This time, when creating a world, a bigger world from the drone tutorial was used. The world present in the drone tutorial was added to this new package as is. This world would now present a world with a drone and a suitable search and rescue environment. However, this world was now missing the turtlebot.

After some examination of the webots_ros2 package, we realized that the world needs the robot definitions as .urdf files. In the tutorial, .urdf files were stored in the resources folder. Both the mavic_webots.urdf and the turtlebot_webots.urdf were added to our new package to represent the two needed robots.

Adding the turtlebot to our drone world was straightforward – just modifying the world file to include the turtlebot (and its externproto) was enough.

To make them both work during launching required more adjustments. The universal_robot package in the tutorials had a two-robot setup, so looking at its launching file felt helpful. The drone's URDF file got added to the resource map and controller initiation to lobot_launch.py:

```
   package_dir = get_package_share_directory('my_package')
       robot_description_path  =  os.path.join(package_dir,  'resource',
'my_robot.urdf')
   print(robot_description_path)
   webots = WebotsLauncher(
       world=os.path.join(package_dir, 'worlds', 'my_world.wbt')
   )

   my_robot_driver = WebotsController(
       robot_name='my_robot',
       parameters=[
           {'robot_description': robot_description_path},
       ]
   )

       robot_description_path  =  os.path.join(package_dir,  'resource',
'mavic_webots.urdf')
   print(robot_description_path)
   mavic_driver = WebotsController(
       robot_name='Mavic_2_PRO',
       parameters=[
           {'robot_description': robot_description_path},
       ],
       respawn=True
   )
```

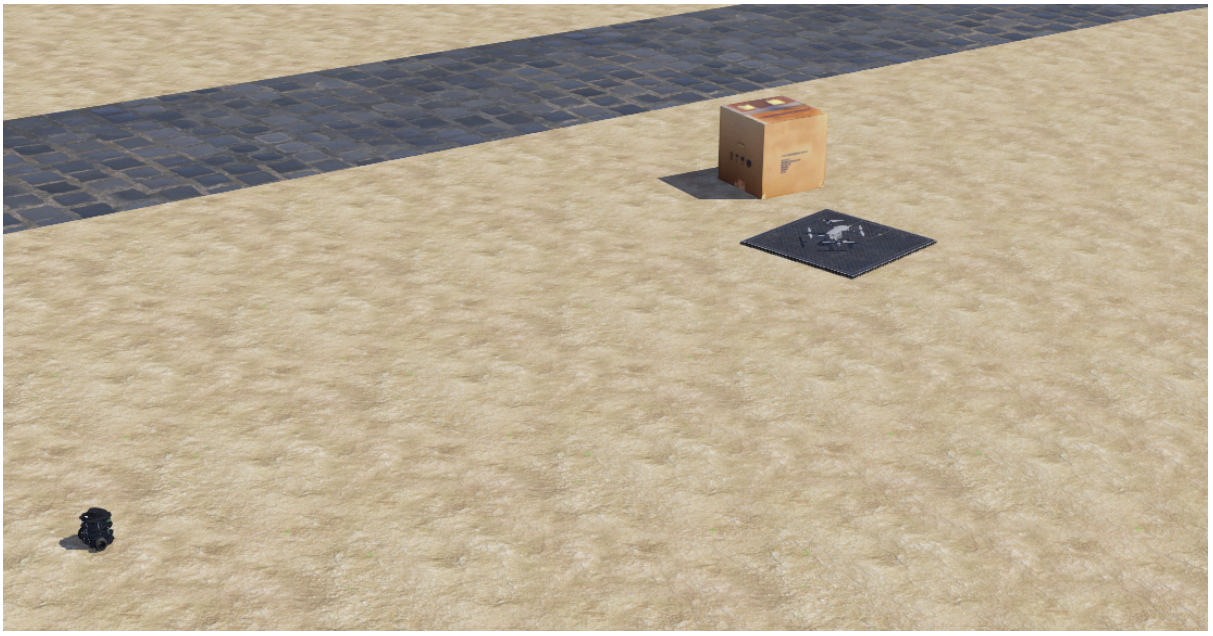
In similar way, the turtlebot's URDF file was created, and its controller was initiated:

```
    turtlebot_description_path  =  os.path.join(package_dir,  'resource',
'turtlebot_webots.urdf')
   turtlebot_driver = WebotsController(
       robot_name='TurtleBot3Burger',
       parameters=[
           {'robot_description': turtlebot_description_path},
       ],
       respawn=True
   )
```

The URDFs had to also be added to the setup.py. If not, the following errors would manifest:

INFO: 'Mavic_2_PRO' extern controller: connected.
INFO: 'Mavic_2_PRO' extern controller: disconnected, waiting for new connection.

Now, the drone and turtlebot launched! Tuiskus next task would be creating a node to read the drone's camera. Sandra's next task would be creating a node to move the turtlebot. Working with one robot each would allow us to efficiently work in parallel without implementing the same things. At this point, we also created a git repository with two branches, one for Tuisku's drone and another for Sandra's turtlebot. At a later point, we would migrate both to the main branch to deliver the complete project. But first we would learn how to control our respective robots.



The drone and a turtlebot in the same world

Development in Simulation

Basics

As this project is built with a step-by-step approach, we needed to define the base functionalities from the whole project scope. From the main techniques described in the first chapter, the following functionalities were highlighted: multi-robot communication, autonomous moving, computer vision and localization.

As this project's main purpose is to learn about multi-robot systems, multi-robot communication is definitely one of the key characteristics needed to implement. The communication between the robots in this application concerns exchanging location information, thus localization is considered another main functionality. The drone needs to inform the turtlebot, where the victim is, thus both of the robots must be aware of their location. In order to search the victim and to come rescue the victim, both robots must be

able to move. To recognize the victim, at least the drone must have some sort of computer vision algorithm.

Thus, the minimal functionalities we started to work with were the following:

- Drone:
    - Detect a given color (the color of the shirt of the missing person, to be later replaced with more sophisticated recognition)
    - Fly to the detected person
    - Send own location
- Turtlebot:
    - Listen to location
    - When location is received, drive to the location
    - When arrived, broadcast arrival

From here on we would then refine the functionalities. The drone should later actively fly around and search for the person, the turtlebot should avoid obstacles and try to recognize the missing person when it approaches the location received from the drone. But we decided to start from a simple approach to not get overwhelmed.

To test the application, obstacle avoidance and more fine tuned path planning were considered secondary, as to find the person and go to the location, the path could be chosen in a way that nothing is blocking the way to the location. In real world scenarios, obstacle avoidance together with sophisticated path planning are essential to allow fast and reliable operations. Thus these two would be our next task.

Environmental mapping would be our last target, as to simply navigate by reactively avoiding obstacles, a suitable approach can already be found even though a map would provide the application with more efficient navigation capabilities.

We divided the tasks between ourselves, Tuisku would start working on the drone and Sandra on the turtlebot, while Lipei would research the real-world robots.

Drone

- Reads its own position from /Mavic_2_PRO/gps and /Mavic_2_PRO/imu
- Creates a lawnmower pattern of coordinates to fly to to search around the area.
- Publishes control commands to /Mavic_2_PRO/cmd_vel
- If all coordinates have been visited, the drone expands the search area.
- Reads camera image from /Mavic_2_PRO/camera/image_color
- Detects a person from the camera image
- If a person is detected, it flies nearer to the target.
- When target is reached, publishes to /target/pose and /target/found
- Listens to the turtlebot and when it announces arrival, rises high

Reading Drone Camera

- In this part we had much help of what we had learned earlier

- Using a pre-trained model was simple
- We learned how to access the package's share folder

To work on the drone, Tuisku created a new node called mavic_node. The mavic_driver node was also copied from the mavic tutorial so that it would be contained in the search and rescue package, and that the package would contain everything needed for the task. The driver was initially defined in the drone's .urdf file as being in the webots_ros2_mavic package. We updated it to my_package.mavic_driver.MavicDriver. The entry points were also updated:

```
    entry_points={
        'console_scripts': [
            'my_robot_driver = my_package.my_robot_driver:main',
            'mavic_driver = my_package.mavic_driver:main',
            'mavic_node = my_package.mavic_node:main',
        ],
    },
)
```

After the drone racing assignment, reading the camera image was easy to implement. From inspecting the ros2 topic list, we found out that the topic to subscribe to was named after the drone, camera and image type. The subscription was created as follows:

```
        self.subscription = self.create_subscription(
            Image,
            '/Mavic_2_PRO/camera/image_color',
            self.listener_callback,
            10)
```

.
Tuisku also added a pedestrian to the world, in the line of sight from the drone's initial position. The next task was to examine the image in the drone node.

Looking at the pedestrian in the world through the camera revealed that going with the shirt color would be a bit cumbersome, as the shirt was green and not easy to recognize amongst the foliage. Tuisku decided to look into pre-trained Haar cascade models for human detection instead.

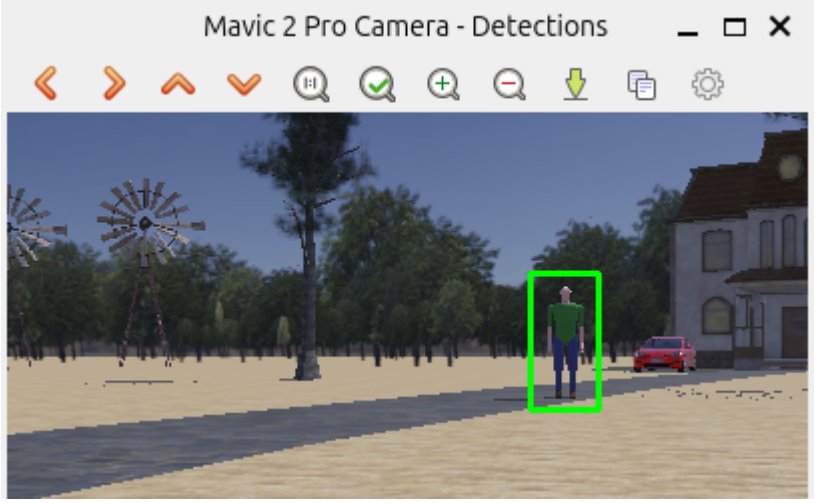
There were some problems with where to save the model as we hadn't used much external files previously. We wanted the model to be included in the repository and placed it in a folder inside the package that we called 'data'. We figured quite quickly that to include the file it needed to be added to data files in setup.py, but how to access it from the node required more research.

```
data_files.append(('share/' + package_name + '/data',
['data/haarcascade_fullbody.xml']))
```

After some research, we figured out that by using the ament_index_python, we could get the share directory of the package and find the files from there.

```
        # Use haarcascade_fullbody.xml from the share directory
        package_share_directory =
ament_index_python.get_package_share_directory('my_package')
        haarcascade_path = os.path.join(package_share_directory, 'data',
'haarcascade_fullbody.xml')
        human_cascade = cv2.CascadeClassifier(haarcascade_path)
```

After the model was successfully loaded, detection seemed to function really well. Tuisku had wondered if a model trained to detect real people from images from real life would have problems detecting the pedestrian model, but at least initially it seemed that we would not need to worry.



The drone has detected a human

Flying the Drone, Reading GPS and Implementing a Search Flight

- Flying to a detected target was easy
- Flying around without a visual target was not

The next step was to fly the drone. The first functionality to develop at this stage was that the drone should be able to fly nearer to the detected human. The next, more advanced but critical function would be that it should fly around until it detects a human. Last but not least, the drone would need obstacle avoidance.

Flying nearer to the detected human would be very similar to flying closer to a gate or stop sign from the previous assignment, but this time it would be important that the drone does not fly too close to the target and accidentally hit the human. The strategy would be to detect if the midpoint of the detected human was left or right of the midpoint of the drone camera

image and turn left or right accordingly, and then fly closer to the detected target if the bounding box was smaller than a value that we would find out by testing.

To find the human in the first place, the drone needed a functionality to fly around. In the drone racing assignment, the drone had always started its flight so that it had the first gate in its sight when it took off, but in our search and rescue project, the drone would need to fly around without its goal in sight.

At first, Tuisku came up with two strategies for the initial search flight. The possibilities were that the drone would either fly around aimlessly or with a predefined search pattern. The aimless flight would probably be easier to implement, but really bad for the task. The search pattern would require the drone to have an understanding of where it was and where it should go.

To understand where the drone was, we decided to use GPS. As the teacher had said, in our scenario the used drones would have GPS if possible, as human lives would depend on the accuracy of the localization. The drone published GPS-related data to the following topics:

/Mavic_2_PRO/gps
/Mavic_2_PRO/gps/speed
/Mavic_2_PRO/gps/speed_vector

After first reading about navigation and gps messages in ROS, Tuisku found info about NavSatFix messages and tried to use them in the GPS subscription. It didn't work, and inspecting the topics with ros2 topic list revealed that the drone instead published them as PointStamped from geometry_msgs. After this adjustment, getting the GPS data from the simulation caused no problems in itself.

The more problematic thing was actually using the GPS data in a smart way. Tuisku tried to implement a system where an array of coordinates was created in a so-called lawnmower pattern and the drone would then fly from one point in the array to the next, while simultaneously checking if it finds a human. The creation of the grid was successful, but flying the drone from point to point was not.

We realized soon that to be able to fly to a coordinate, the drone did not only need its own position but even direction. We looked into the compass messages the drone sent, but also found information online that the direction could also be obtained from its IMU. In the ros topic list there was an IMU topic, but it seemed ambiguous if it was for the drone or the turtlebot. Looking at the .urdf file revealed that the topic name was set there in the element topicName, and Tuisku changed it to Mavic_2_PRO/imu to match the other drone topics.

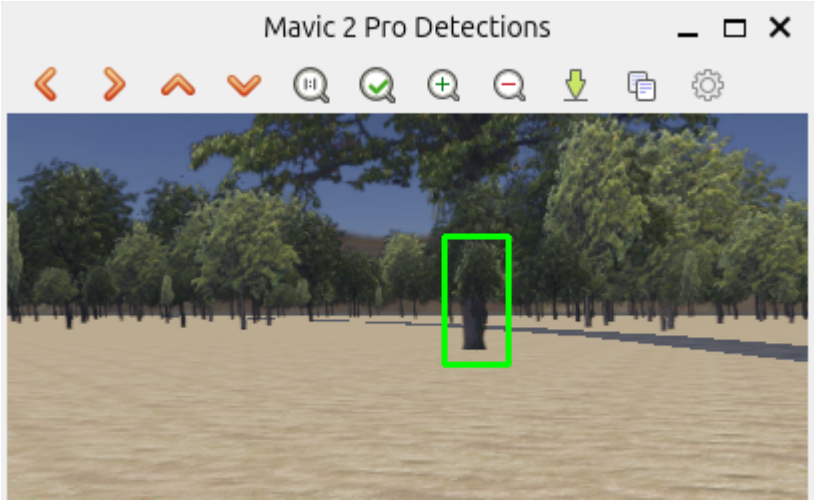
At this point, the drone node had three subscriptions: image, GPS and IMU. Introduction of the GPS callback had resulted in Tuisku moving the flying around function there, but the image callback still had the functions to move closer to a detected human. At this point we realized this would have been easy to split into two tasks that two group members could have been developing in parallel, one the initial search and the other the movement of approaching the target.

Publishing Data

For publishing the position data when the drone had arrived, we decided to try out the Pose messages suggested by the teacher. Researching the Pose messages, we found out that they include the location of the drone and its position in quaternions.

Problems

- The drone was slow, but when the velocity was adjusted up, it became unstable
- The camera misrecognized some trees as people



The drone has detected a …human?

Most of the problems encountered while developing the drone were fixed relatively easily. The main problems that remained were wrongly recognizing trees as humans and the slowness of the drone.

After some research, the slowness was found to be a result of the time step settings of the world. The setting basicTimeStep was 8 in the tutorial world we had chosen to use as the base of our simulation, and the simulation ran at approximately 20 % of normal speed. When increasing basicTimeStep the speed of the simulation increased, but the drone started to behave in a really unstable way. We realized that the drone should have been tested in a world that had a more normal speed to have it work properly.

Turtlebot

Getting the Turtlebot Move in the Simulation

Considering the turtlebot, at this point, the turtlebot robot definition is a straight copy from the tutorial webots_ros2_turtlebot (github: cyberbotics/webots_ros2/webots_ros2_turtlebot). In this project, the robot definition is located at **resource/turtlebot_webots.urdf**. The turtlebot was added to the drone's world by simply following the same tutorial; from the example world provided in the webots_ros2_turtlebot, the parts concerning the turtlebot were

transferred as is to the drone world. The world incorporating both the drone and the turtlebot is located at **worlds/turtle_world.wbt**.

However, simply adding the turtlebot's Webots Controller to the launch file (**launch/both_robot_launch.py**) was not enough to actually control the turtlebot. This is where the true challenges had begun…

The investigation process was not straightforward. The first fatal error was that the /robot_description topic in Webots, was not capable of receiving the turtlebot description file correctly. When Webots couldn't receive the turtlebot definition, it was not able to initialize the robot even though the simulator window showed the turtlebot in the world correctly. In the launch file, the turtlebot_driver was supposed to load the robot description .urdf file, and pass it automatically to the /robot_description topic. But as we saw the following warning in terminal, we knew the turtlebot description was not correctly given to the topic:

```
[webots_controller_TurtleBot3Burger-4]                    [WARN]
[1748780901.310409667] [controller_manager]: Waiting for data on
'robot_description' topic to finish initialization.
```

Almost two days were spent on trying to find the cause for this issue. We tried to analyze the tutorial's launch file, and get only the necessary code parts to our launch file as we didn't want to add anything unnecessary to avoid redundancies in our project. All the trials and reconfigurations led nowhere. We gave up with the tutorial's launch file and decided to add the turtlebot to our world manually. This also led to failures, since when adding the turtlebot manually to the world, a robot controller must be defined. Implementing a robot controller code was not feasible as in our case, the robot controller must be ros2-based to get all the system components working interoperably. The webots_ros2 package contains all the necessary robot controllers and drivers to operate between the webots and ros2. Thus, again, we were at step one. We had to analyze the turtlebot tutorials launch file in more detail.

After a comprehensive analysis of the tutorial's launch file, the issue was finally solved. An initial robot description was passed to the webots with a robot_state_publisher node:

```
    robot_state_publisher = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        output='screen',
        parameters=[{
                            'robot_description':  '<robot  name=""><link
name=""/></robot>'
        }],
    )
```

And the actual turtlebot description is passed to Webots by configuring the set_robot_state_publisher parameter to True.

```
    turtlebot_driver = WebotsController(
        robot_name='TurtleBot3Burger',
        parameters=[
            {'robot_description': turtlebot_description_path,
             'set_robot_state_publisher': True},
```

During initialization, webots expects something to be published immediately to the /robot_description topic. If nothing is published, the topic is somehow blocked and it does not receive the correct turtlebot description. This is why an initial robot state publisher node is configured to publish a dummy robot description to the topic. When the turtlebot has been correctly configured by the webots_controller, then the true turtlebot description is finally passed to the /robot_description topic!

At this point, it was expected that the turtlebot had some sort of /cmd_vel topic which would be used to control its movements. But no, this was not the case! The turtlebot didn't have any topics by which it could be controlled. However, it had some GPS topics, which meant that we would be close.

As we had already analyzed the tutorial's launch file carefully, we had realised that the turtlebot description defines a motor and wheels for the robot. But there is nothing to control them with. After further examination of the tutorial's codes, we found a ros2control.yml file which seemed to define drive controllers for the wheels. This tutorial ros2control.yml file was added to our project tree as is for further testing into the **resource/ros2control.yml**.

Then, we ended up back again to analyze the tutorial's launch file. Now Sandra noticed that two nodes that were also defined in the ros2control file, were initialized in the launch file! The diffdrive_controller and joint_state_broadcaster were added to our launch file as well. Also, the turtlebot_driver was configured to use the ros2control file parameters in order to connect to the two nodes controlling the turtlebot movements.

```
    # ROS control spawners for turtlebot hardware in simulation (wheels,
motor)
    controller_manager_timeout = ['--controller-manager-timeout', '50']
    diffdrive_controller_spawner = Node(
        package='controller_manager',
        executable='spawner',
        output='screen',
        arguments=['diffdrive_controller'] + controller_manager_timeout,
    )
    joint_state_broadcaster_spawner = Node(
        package='controller_manager',
        executable='spawner',
        output='screen',
                                arguments=['joint_state_broadcaster']    +
controller_manager_timeout,
```

```
    )
```

The challenges didn't end here. When launching the simulation now, the turtlebot didn't still have the /cmd_vel topic. But when investigating the terminal, some hints were given that the joint_state_broadcaster and diff_drive_controller were not correctly initialized.

```
[webots_controller_TurtleBot3Burger-5] [ERROR]
[1748882080.950037667] [controller_manager]: The 'type' param was
not defined for 'joint_state_broadcaster'.
[spawner-7] [FATAL] [1748882080.952416824]
[spawner_joint_state_broadcaster]: Failed loading controller
joint_state_broadcaster

[webots_controller_TurtleBot3Burger-5] [ERROR]
[1748882081.652762404] [controller_manager]: The 'type' param was
not defined for 'diff_drive_controller'.
[spawner-8] [FATAL] [1748882081.699611895]
[spawner_diff_drive_controller]: Failed loading controller
diff_drive_controller
```

First Sandra tried to tackle the issues by modifying the ros2control.yml file. The issue was debugged for multiple hours without any success. Then Sandra had to analyze the tutorial's launch file, AGAIN. Among the additional navigation nodes which were not needed in our project at this point, we noticed a small detail. The two nodes are launched after the turtlebot_driver has been initialized. This is a little similar issue than with the /robot_description topic. If the two nodes are initialized in webots before the turtlebot has been configured, they won't work as webots is not capable of connecting these control nodes to the turtlebot. This is why the turtlebot_driver must be first configured by the webotscontroller, and then these two nodes can be established. This way, the controller nodes are able to connect to the turtlebot correctly.

```
    waiting_nodes = WaitForControllerConnection(
        target_driver=turtlebot_driver,
        nodes_to_start= ros_control_spawners
    )
```

After remapping the turtlebot's /cmd_vel topic to turtlebot/cmd_vel, modifying the setup.py to hold the necessary data files (ros2control.yml) and adding necessary dependencies in the package.xml, the turtlebot was finally controllable. This was proven with a simple controller node that connects to turtlebot's /cmd_vel topic and moves the turtlebot in a circle.

Next step would be to subscribe to the drone's location to which the turtlebot should go to, get the turtlebot's own location, and based on this information move to the location autonomously. For now we decided to leave the obstacle avoidance for later project stages.

Getting the Drone's Location Information

To start with the main purpose of this project (multi-robot coordination), first we wanted to implement the communication topics between the two robots. Thus, a subscriber to the topic where the drone publishes its current location was implemented on turtlebot. As the topic handles Pose-type messages, the turtlebot's subscriber captures only the necessary information from the message, the X and Y coordinates in the world frame, to avoid redundant information storage.

Tuisku implemented the drone in a way that the target location messages are published frequently. Thus another topic from the drone had to be subscribed. This topic informs whether the target has been found.

These two topics led to the fact that the turtlebot updates its target location only once when the drone confirms in the other topic that the target has been found. So, when the turtlebot receives a message from the drone confirming the victim has been found, the target location is set and locked to prevent the target location drifting which might cause issues in the navigation part. However, this design choice leads to the fact that the turtlebot's target location can only be updated after the turtlebot has reached the destination. But as in the simplest form of this application, we assume only one person shall be rescued and the drone is capable of finding the victim precisely, thus this design fact is sufficient for now. The information from the drone about the victim has been found also triggers the turtlebot to move to the target location since otherwise there is no purpose for the turtlebot to move.

Getting the Turtlebot's Own Location

To move to a meaningful location, the turtlebot's own location was next implemented instead of starting to move to the location immediately. For the current position, as told by the course instructors, we can assume the GPS is accessible which is why the turtlebot's GPS topic was subscribed. Only the X and Y coordinates are captured from the topic to reduce redundant information storage. This design decision leads to the fact that this search and rescue approach is not suitable for GNSS-denied environments, which can be seen as a huge drawback as for example the application might be needed to search for victims in forests. In case GPS is not available, localization approaches like visual odometry using cameras or particle filters estimating the location based on noisy odometry data could be considered.

For getting the turtlebot's orientation, odometry is used. As GPS provides more accurate positioning, the odometry is used to only get the orientation. For this particular application and simulator, odometry seems a more suitable choice than IMU to track the orientation since IMU data are prone to drift over time causing more accumulated errors. Odometry is also prone to similar drifting and may introduce errors due to wheel slippering in uneven terrains. However, in this particular setup, turtlebot's IMU sensor needed further configurations to work, thus to actually implement the application, odometry was decided to be used for orientation.

Moving the Turtlebot to the Target Location

Finally, the point was reached when the turtlebot was ready to be configured to move autonomously to the victim's location. The core ideas to move the robot to the destination were to update the rotation towards the target, move forward and stop the robot when the target was close enough. The rotation and forward going must be iterated over to adjust the path. This is a similar idea utilized also during drone racing. However, the approach is not the most optimal and fastest to reach the target, but we consider the more fine tuned path planning to be one of the next steps in this project. For now, it is sufficient to get the core application working.

The forward movement was something we considered easy. But to actually move forward to the right direction was something that required more in-depth research. Several orientation updating approaches were searched online. However, the approaches derived the orientation updates using a predefined map which we didn't have. Finally, a simple approach was found which derived the amount of needed rotation for the robot to face the target. To calculate the yaw angle to face the target, a simple maths function atan2 is utilized. It computes the angle between the robot's current position and the target point by taking the arctangent of the ratio between the difference in y and x coordinates effectively giving the direction to the target in radians. The amount of rotation needed for the robot to face the target was calculated by subtracting the robot's current rotation from the target orientation. After carefully reading about the proposed approach, it was deemed to be efficient and simple to be utilized in our project. As we were not sure how the turtlebot's odometry and the atan-function would behave in radians over time (are the values between -pi and pi, or between zero and 2*pi, or multitudes of these), we added a function to determine the amount of rotation needed to be between -pi and pi. This calculates the shortest amount of rotation needed making it easier for us to determine whether the orientation to the target is sufficient enough.

We decided that the turtlebot's orientation must be sufficiently aligned before starting to move forwards. This way the turtlebot won't move unnecessarily to the wrong direction leading to the turtlebot arriving faster to the destination. But after the orientation to the target is ok, the turtlebot is controlled to move towards the destination iteratively checking the alignment with the target is still sufficient.

To control the turtlebot's movement with its /cmd_vel topic, the orientation is updated by taking into account the amount of yaw needed to face the target. This ensures faster orientation alignment as with a bigger rotation change needed, the turtlebot turns faster, while during smaller rotation needed, the turtlebot turns slower just to fine tune the orientation. Similarly, the distance to the target calculated with maths hypot-function affects the forwarding speed. When there are a lot of meters to travel, the turtlebot moves with the fastest stable speed. But when the distance to the target gets smaller, the turtlebot moves slower to not collide with the victim. When the distance to the target location is small enough, the turtlebot stops and remains still for the victim to get the water and other first-aid stuff.

What is worth mentioning is that the turtlebot starts to move only after the drone has provided the turtlebot with a confirmed location. This prevents the unnecessary moving of

the turtlebot, saving its resources. However, we could consider that the turtlebot starts to search for victims within a smaller area around the starting location to assist the drone and make the rescue application become faster.

The turtlebot movement controls are implemented in a timer callback to allow continuous movements during the navigation.

Publishing the Turtlebot Status

To demonstrate multi-robot communication in both directions, from drone to turtlebot and vice versa, the turtlebot sends status information to the drone. When the turtlebot is close enough to the target location, it stops and broadcasts the information about the arrival to the drone. This allows the drone to confirm the turtlebot arrival and move higher, becoming more visible to the human rescuers.

The turtlebot also updates the status topic frequently when it starts to move towards the target location and during its traveling phase to the destination. This information is useful to confirm the turtlebot has received the target location and is trying to get to the location.

Testing the Turtlebot Setup

To ensure the turtlebot controller works as expected, it was first evaluated without the drone. The drone's topics about the location and confirmation were published from the terminal. These tests ensured the turtlebot would not move until both of the topics were published. As the code was created carefully and the implementation was simple at this point, the turtlebot worked as expected!

However, a couple of things were realised at this point. We had assumed that only one person would be rescued. How about a situation where multiple humans would need to be assisted? The drone would remain in its current position until the turtlebot arrives, and the turtlebot does not yet have functionality to update the target location after reaching the initial target. As we started to run out of time for this project, the turtlebot specific research had to be ended, and we needed to move on to integrating the two robots.

Two Robots Working in the Same World

- Went mostly smoothly after initial confusion
- Both robots used cmd_vel messages by default
    - The drone driver was modified to listen to Mavic_2_PRO/cmd_vel
    - The turtlebot driver was modified to listen to turtlebot/cmd_vel
- When everything was up and running, the robots cooperated just as designed

Adding both robots to the same world had required some editing of both the world and the launch file. Editing the world to include both robots was straightforward, as it only required adding the objects to the world file.

Editing the launch description had required a bit more reading, as we had never before launched two robots simultaneously with the same launch file. Luckily, one of the tutorials had a two robot setup, and we could examine it to learn how to do it. We had to create WebotsController objects for both robots and give them as elements in the list given when creating and returning a LaunchDescription. More about this can be read in the Getting the right robots to the simulation chapter.

The biggest obstacle we encountered when having both robots in the same world was that they both used the same cmd_vel topic to listen to commands! We found two ways to work around this. For the Mavic drone, we edited the drone controller to subscribe to Mavic_2_PRO/cmd_vel instead. For the turtlebot, a mapping was created to use turtlebot/cmd_vel instead.

```
        # ROS interface
        rclpy.init(args=None)
        self.__node = rclpy.create_node('mavic_driver')
        self.__node.create_subscription(Twist, 'Mavic_2_PRO/cmd_vel',
self.__cmd_vel_callback, 1)
```

At this point, both robots had been developed independently. But when the finished robots were finally run in the same world, they performed really well together. Even if we had mostly developed each from a different location, we had defined clearly which topics and what kind of messages we would use for robot to robot communication, and communicated about it actively when developing the multi-robot parts. It was really nice to see the robots cooperating in the simulation effortlessly!

Development in Real World

The objective of this project is to develop a multi-robot system consisting of a Tello drone and a TurtleBot, where the drone is used to search an area for injured individuals, and the TurtleBot subsequently delivers medical supplies to the identified location, marked by person.

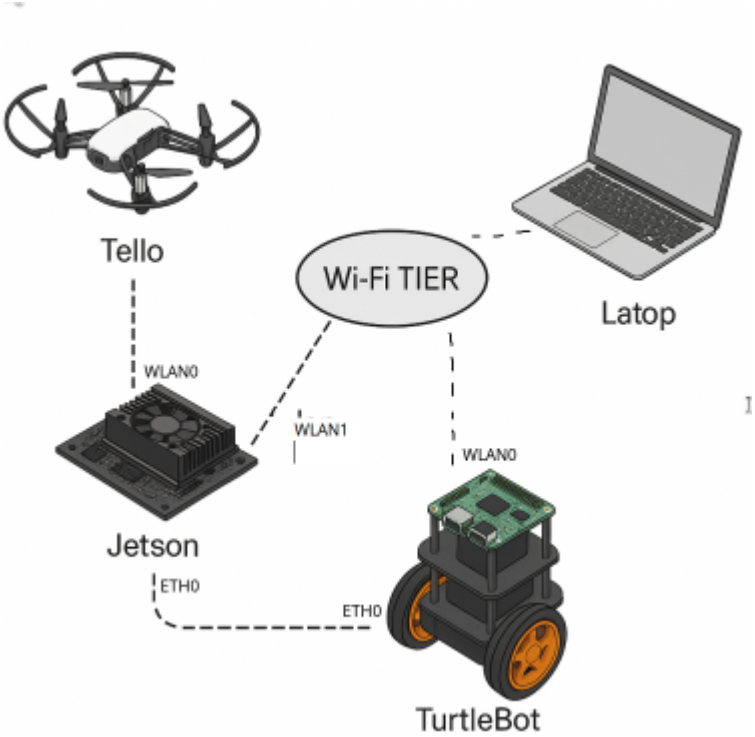
At first glance, the concept appears straightforward, with clearly defined roles for each robot. However, implementing the system on real hardware presented several challenges. Our initial setup involved using a Tello drone and a Raspberry Pi-powered TurtleBot. However, we quickly encountered a limitation: the Raspberry Pi lacks a GPU, making it unsuitable for running convolutional neural networks (CNNs) required for human detection. Without deep learning capabilities, our core objective would be unattainable.

Fortunately, we discovered that the TurtleBot is equipped with a Jetson board,which provides GPU acceleration. However, the Jetson and the Raspberry Pi were not initially connected. After consulting with the teacher, we decided to connect the Jetson to the Raspberry Pi via Ethernet using USB-to-Ethernet adapters. This allowed the Jetson to act as the computational hub for the system, running the CNN models for human detection.

This decision introduced another issue: in order for the Jetson to process images from the Tello drone, it needed a dedicated Wi-Fi connection to the drone. The Jetson already used its built-in Wi-Fi (wlan0) to connect to the Wi-Fi network (TIER) for SSH access. To resolve this, we added a second Wi-Fi adapter to the Jetson, allowing wlan1 to connect to the Tello drone while wlan0 remained connected to the TIER network. Also this multi connection needs to be configured through the IP route table, to make sure the data to Tello is going through WLAN1, and the data to Turtle Bot is going through ETH0.

Final Hardware Setup

1. Tello Drone connects via Wi-Fi to Jetson wlan0
2. Jetson uses wlan1 to connect to Wi-Fi TIER (for SSH access from a PC)
3. Jetson eth0 connects to Raspberry Pi eth0 (via USB-to-Ethernet adapters)
4. Raspberry Pi wlan0 connects to Wi-Fi TIER (for SSH access)
5. Laptop connects to Wi-Fi TIER to remotely access both Jetson and Raspberry Pi via SSH



System Setup and Problem Solving

After completing the hardware setup, we encountered several challenges during the integration and configuration of the system. Below is a summary of the key problems and how we resolved them.

1. Accessing the Jetson Module

The first issue was the lack of access credentials for the Jetson module. Initially, we were only able to interact with it through a running Jupyter web server hosted on the Jetson, which allowed limited command execution. Eventually, we managed to retrieve the correct username and password, enabling full SSH access and system control.

2. Wi-Fi Adapter Connectivity Issue

The second major problem involved the Jetson's secondary Wi-Fi interface (wlan1), which was unable to connect to the Tello drone. At first, we suspected that the Jetson hardware lacked the necessary kernel support for the USB Wi-Fi adapter. After reviewing kernel logs via dmesg, we found that the adapter was not being recognized at all, suggesting a hardware failure. Fortunately, we had a backup Wi-Fi adapter on hand, which worked immediately upon connection, resolving the issue.

3. ROS 2 Version Incompatibility

Another challenge arose due to version mismatches between the ROS 2 installations on the TurtleBot and the Tello drone. To address this, we implemented a communication bridge between the Jetson and the TurtleBot. The Jetson was set up with the same ROS 2 version as the Tello, while an intermediate layer was created to translate and forward messages to the TurtleBot, effectively bypassing the version compatibility issue. (But eventually we didn't use ROS2 with Tello, because there was a problem with Tello driver)

4. Tello Driver Compatibility on Jetson

Our next goal was to run the TIER Tello ROS 2 driver on the Jetson. However, the driver assumes an x86_64 architecture with a graphical user interface (GUI), while the Jetson runs headless Ubuntu 20.04 on ARM architecture. Since OpenCV GUI APIs were used to display live video feeds, we modified the driver by commenting out the GUI-related code. Despite these changes, we encountered further issues with video decoding and missing dependencies.

Because the Jetson's system and other critical applications also rely on OpenCV, we could not safely recompile or install a new version without risking system stability. To avoid this, we opted to use the official Tello Python library, DJITelloPy, from GitHub. This library allowed us to establish communication with the Tello drone and successfully retrieve image streams and send control commands directly from the Jetson.

5. Controlling the TurtleBot via Jetson

Once the Tello control pipeline was functional, we focused on controlling the TurtleBot from the Jetson. We established a UDP connection between the Jetson and the Raspberry Pi on the TurtleBot to send motion commands. On the TurtleBot side, we created a ROS 2 publisher node that publishes to the topic /tb4_08/cmd_vel. This node receives velocity commands (x for forward motion and z for angular velocity) via UDP from the Jetson.

To ensure safety and precise control, we implemented a 0.5-second timer that stops the TurtleBot if no new command is received within that interval. This prevents unintended movement in case of communication dropouts.

6. Wi-Fi connection issue

During testing, one recurring inefficiency was the need to manually connect the Jetson's secondary Wi-Fi interface to the Tello drone's Wi-Fi network before each session. This repetitive task significantly slowed down the development workflow.

To streamline the process, we automated the Wi-Fi connection by encapsulating the necessary commands into a shell script. This script ensures that the Jetson automatically connects to the Tello's Wi-Fi network each time we initiate a test session. As a result, setup time is reduced and testing becomes more efficient and reliable.

Multi-Robot Navigation and Coordination

Following the successful setup of communication and control with both the Tello drone and the TurtleBot, our next objective was to implement coordinated navigation. The goal was to have the Tello drone search a designated area for a human target, and upon successful detection, guide the TurtleBot to that location to deliver assistance.

Location Estimation and Navigation Strategy

Initially, our approach involved having the Tello drone autonomously fly and search the area. If a person was detected, the TurtleBot would navigate to the estimated position of the drone for the rescue task. However, this proved difficult because the Tello drone lacks GPS capabilities. To overcome this, we experimented with step-based dead reckoning—estimating the drone's position by calculating distance based on movement commands. Unfortunately, the estimation errors increased significantly over longer distances, making this method unreliable. To simplify the navigation problem within the limitations of our hardware, we adopted a revised approach: Predefined target locations were specified before each mission. Both the drone and the TurtleBot were directed to move to these coordinates. Once in the search area, the Tello drone conducted human detection, and if a person was found, it would track the person's movement continuously, enabling the TurtleBot to follow and assist accordingly. In a real-world deployment, GPS would provide a far more accurate and scalable solution. However, in our setup, predefined coordinates served as a practical workaround.

Parallel Processing for Smoother Operation

All image processing was performed on the Jetson module. Initially, we implemented a sequential pipeline where Tello and TurtleBot images were processed alternately. After each detection cycle, control commands were sent to the respective robot. This approach resulted in poor performance, as it forced one robot to wait for the other: The TurtleBot, being slower,

caused the drone to pause frequently. The Tello's blocking API calls further delayed the control loop.To improve concurrency and responsiveness, we implemented a multi-threaded control architecture: A dedicated thread was assigned to each robot (Tello and TurtleBot). A thread-safe lock mechanism was introduced to coordinate access to the Jetson's neural network for image detection. This allowed both robots to move and respond to detections simultaneously, greatly improving system fluidity and real-time responsiveness.

To enhance the robustness and smoothness of the system, we performed a final round of parameter tuning and behavior optimization. This involved adjustments across several key components. Movement Step Sizes: We adjusted the motion step size for both the Tello drone and the TurtleBot to achieve smoother navigation. Smaller, more controlled movements improved precision during tracking and overshooting near the target.

Human-Following Mode

Once the Tello drone and TurtleBot reached the target area and successfully detected a human, both robots entered a human-following mode. To ensure safety and maintain an appropriate distance from the person. We analyzed the horizontal ratio of the human's bounding box within the image frame to estimate proximity. If the human appeared too large in the frame (indicating close distance), both robots slowed down or stopped to avoid getting too close or causing accidental contact.

This visual feedback loop enabled dynamic, human-aware navigation without relying on depth sensors or GPS. It provided a practical safety mechanism suitable for our hardware constraints environment.

Concluding the Project

Results

Simulated Approach

After successfully installing Webots, creating a suitable world environment for search and rescue missions with the desired robots in it and implementing appropriate controllers for them, we started the journey of learning to develop multi-robot applications.

The drone determines its own location in the world, creates a series of coordinates to fly through and analyzes its camera feed while flying through the waypoints. When a human is detected, it starts publishing in two topics to communicate the found status and its own coordinates to the turtlebot. It also listens to the turtlebot to determine when help has arrived.

The turtlebot was successfully configured to listen to the drone for confirmed target location, while simultaneously localizing itself via GPS and odometry. When the turtlebot receives the target from the drone, it starts moving to the destination robustly by first aligning the

orientation towards the target and only after that moving steadily to the location. During arrival, the turtlebot slows the speed down to not collide with the victim and eventually stops near the human.

During the project, we developed various new skills. The simulator preparation and configuration requires a lot of time, in this case even more time than the actual robotic application development! The robots and their sensors must be carefully defined, the robots actuators' controllers must be implemented, the world characteristics must be clearly described and even the simulator parameters must be considered. For example, the simulation running speed affects the stability of the robots, thus it shall be carefully selected.

The actual robotic application development felt a little easier, as computer vision and robot movement control through the /cmd_vel topic were already familiar to us from the previous courses and the drone racing project. However, we learned to localize a robot using GPS coordinates! In the simulation, the characteristics of the GPS coordinates are a bit different than in the real world scenario. Still, we believe the learned skills should translate to real life quite easily, giving us a good first touch in localization.

Finally, we realised that multi-robot communication is not actually that difficult since the same ROS2 messages and topics can be used similarly as with subscribing and publishing to robot sensor and actuator topics. What we found more important was the timing of the messages exchanged. One must ensure the order of the messages between different topics is correct, and it must be considered that the messages arrive at the appropriate time in critical applications.

Real-World Approach

Despite significant hardware constraints, we successfully implemented a functional multi-robot system using a Tello drone and a TurtlebBot, coordinated through a jetson module. Our system demonstrates real-time, autonomous cooperation between aerial and ground robots for human search and rescue.

By leveraging the Jetson board's GPU capabilities, we were able to run deep learning models for human detection efficiently. We established robust bidirectional communication: receiving real-time image data from Tello drone and sending control commands back, while simultaneously sending navigation commands to the TurtleBot over a dedicated UDP link. Both the Tello drone and TurtleBot were able to navigate autonomously and in parallel, track human targets and maintain safe, adaptive behavior based on vision.

For the real world implementation improvements, the main thing is to enhance and optimize the capabilities of hardware. Currently the tello does not provide IMU data, so movement is estimated, and the error accumulates. Also the movement of the turtlebot is not smooth, which is something that could be optimized. Finally, the detection model is targeted for jetson cam, it's not optimized for images from tello, thus the number of false detections increases.

Future Extensions

During the project we realised that our implementation is capable of rescuing a single person. In many real-world scenarios, a disaster leads to more than one human's life to be under risk. Thus, an extension would be appropriate. If only one pair of drone and turtlebot is available, the drone could start researching for more victims after the first one has been found. The turtlebot could be programmed to stay with the first victim for a specific amount of time after which it would travel to the next confirmed victim. But, as the amount of supplies one turtlebot is capable of carrying is restricted, when the drone finds the second victim, another turtlebot could be called for aiding. Also multiple drones can be utilized for more efficient search of the victims. The drones could publish the target locations to one topic, and the turtlebots would announce the locations they are heading to one topic to prevent more than one turtlebot from going to a specific location.

When it comes to the actual algorithms used, we didn't have enough time to implement all the functionalities we wanted. One of the main reasons for this was that the simulator and real-world robots' configurations required more time than anticipated. Thus the functionalities we didn't manage to do must be left for the future. This includes collision avoidance, sophisticated path planning and environment map creation. If an accurate map exists, optimized path planning algorithms such as A* and RRT* could be considered. A-star would be useful with the turtlebot, since it provides an optimal path with the help of heuristics, allowing the turtlebot to get to the target as fast as possible. With the drone, Rapidly-exploring Random Tree star would be beneficial due to its optimized functionality for high-dimensional environments providing an optimal path to the target over time. These advanced path planning algorithms would provide our application with smoothened movements and faster operations. The map would allow us to create paths with in-built static obstacle avoidance. For dynamic obstacle avoidance, a LiDAR sensor could be useful. LiDAR provides real-time distances to surrounding objects, allowing a robot to detect moving objects and respond to the changes in environment by updating the path efficiently.

A predefined map would also allow us to localize a robot in GNSS-denied environments by utilizing e.g. Monte Carlo Localization in which a particle filter is used to estimate the location based on the sensory readings and the map. However, the issue is that a map is not always available, especially when rescuing people in non-urban areas or when a disaster has modified the environment in such a way the maps are outdated. Thus the robots must create a map on the fly with e.g. cameras and LiDARs. A simultaneous localization and mapping (SLAM) could be implemented to tackle issues of localization and map creation in one go. But as we have encountered multiple articles saying SLAM is difficult, we believe one entire project should be about implementing a robust SLAM-algorithm.

We had paid less attention to integrating the simulator approach to the real world approach since we were so into implementing a working search and rescue application. To integrate the two implemented approaches, we believe huge code modifications are required as the ROS versions used in the simulator and real world differ. Additionally, one significant aspect we identify is that in the real world the sensors (odometry, IMU) introduce way more noise than in the simulations. Thus, some more advanced orientation definition approaches (sensor fusion, extended kalman filter) must be considered to provide more robust navigation. A kalman filter could also be used to tackle one of the main problems in the real

world robotics scenario. When developing the application with real robots, the tello drone was shaking during flying even when just hovering. To make it more robust and improve the navigation accuracy, the kalman filter could be applied to filter out sensor noise and provide more stable estimates of position and orientation.