**Group Member NAMES: Ryan Sauer, Demetrius Johnson,**

**Jonathan Schall & Olivia Pellegrini**

**6-19-23**

***Software Design Specifications***

# **Table of Contents**

[**Table of Contents 1**](#_dkop2mmhn1fn)

[**1.0 Introduction 3**](#_30md879pwozd)

[1.1 Goals and objectives 3](#_fyfj3byser22)

[1.2 Statement of scope 3](#_3dhtn6vf96b9)

[1.3 Software context 3](#_e05yl2xxzhou)

[1.4 Major constraints 3](#_f86i0qmzyew8)

[**2.0 Data Design 4**](#_9mxro1azjkuv)

[2.1 Internal software data structure 4](#_sfshs0r2o3k5)

[2.2 Global data structure 4](#_kpw5k2t0bf7f)

[2.3 Temporary data structure 5](#_bta77rttz5ka)

[**3.0 Architecture and component-level design 5**](#_t7ftykkmekgp)

[3.1 Program structure 5](#_it5pafjtfwb1)

[ROS 6](#_l5uhw6m4yz9v)

[Installation 6](#_9upib38u4prk)

[Concepts 7](#_k61d4q1wb3sh)

[Nodes 7](#_mjrer1s9lk53)

[Topics 8](#_j455mbp7nliz)

[Services 10](#_mwus3umflg1u)

[Names 12](#_wonqv0f7rabz)

[Private name 12](#_dk0ch1k0hc9)

[Relative name 13](#_y45ul2t5par9)

[Working on several PCs 13](#_v35f2smqn1xq)

[MAVROS 13](#_1mvjbttehma1)

[Main services 14](#_h3atkblf490)

[Main published topics 14](#_r2805cygv1sl)

[Main topics for publication 15](#_rlj3udsh3gf)

[Topics for sending raw packets 16](#_yjdost3m9tme)

[Key Architecture Questions 17](#_esqifxqzz1h)

[Appending Our Drone-car Architecture: 21](#_v10lyzefa7lf)

[Description of GUI API: Kivy 2.2 21](#_59gwfjgacx25)

[Description of Drone-Car interfaces 23](#_ob7jbmdnpkvv)

[Description of Communication Protocols: 24](#_2fi792d1looy)

[Description of Computer Vision model and object segmentation: 26](#_d0z7ubwjci6s)

[Description of Car control - Robot HAT 26](#_lr03pyq5rscz)

[3.1.1 Architecture diagram 27](#_tqdq5wx2rog0)

[3.1.2 Alternatives 28](#_2omijfakfbeu)

[3.2 Description for component n 28](#_oictxa1pwrz3)

[3.2.1 Processing narrative (PSPEC) for component nm 29](#_3nit6sp66ha6)

[3.2.2 Component n interface description 29](#_nh95gkyanxrq)

[3.2.3 Sub-Component n.m processing detail 30](#_hhxu4i6zz7c6)

[3.2.3.1 Interface description 31](#_88awmkotokan)

[3.2.3.2 Algorithm model (e.g., PDL) 31](#_fyjg49z3cc6a)

[3.2.3.3 Restrictions/limitations 37](#_1tdj3q306ht5)

[3.2.3.4 Local data structures 38](#_qxwsll9h4nla)

[3.2.3.5 Performance issues 38](#_rsgjvnqtqcrb)

[3.2.3.6 Design constraints 39](#_th5i66ziidgf)

[3.3 Software interface Description 39](#_ksg20rj8uagg)

[3.3.1 External machine interfaces 39](#_tanrtwyw582a)

[3.3.2 External system interfaces 40](#_wthg733p6lk5)

[3.3.3 Human interface 40](#_dm60uttpdnde)

[**4.0 User Interface Design 40**](#_p9e67hckgi81)

[4.1 Description of the user interface 40](#_p8cx9e3a1pow)

[4.1.1 Screen images 40](#_66ktz4xr2akf)

[4.1.2 Objects and actions 44](#_orynomsfa4kc)

[4.2 Interface design rules 46](#_lutxab3f4hwv)

[4.3 Components available 46](#_w4j51ccznt2k)

[4.4 UIDS description 47](#_ehp6o0gmcvqd)

[**5.0 Restrictions, Limitations, and Constraints 47**](#_lrs31fy259mb)

[**6.0 Testing Issues 48**](#_v6fu8r62d9hs)

[6.1 Classes of tests 48](#_6r1a4ifvq)

[6.2 Expected software response 48](#_5fcjhpjlrpzu)

[6.3 Performance bounds 49](#_jo1v2yxa2gu9)

[6.4 Identification of critical components 49](#_1ozvqbw2ufhn)

[**7.0 Appendices 50**](#_aagouihbm7sr)

[7.1 Requirements traceability matrix 50](#_hhal1xv6ix37)

[7.2 Packaging and installing issues 50](#_an1blxbnycbp)

[7.3 Design metrics to be used 51](#_8e9ib1pclsva)

[7.4 Supplementary information (as required) 51](#_7763aws099ze)

[7.5 Hardware and Software Materials and Requirements 52](#_5ec32hlwqqvm)

[7.6 Research Background Sources 56](#_sfvxzmsri80c)

# 

# **1.0 Introduction**

## **1.1 Goals and objectives**

The main objective of our project is to develop a drone-car collaborative model that can be used to obtain data on drone-car collaboration. The model will use a drone camera to record the surroundings of the car and notify the car of potential hazards in real time. Some basic goals of our model are:

* Accurately detect a potential hazard in multiple environments using sensors from the car and drone, particularly the vision systems (camera).
* Maximize drone battery life and bandwidth, while minimizing latency (response time of the car to commands and information sent from the drone).
* Develop a working product that is open source and reproducible through our documentation to serve as a future reference/baseline for further research.

## **1.2 Statement of scope**

General Requirements of the model

* An image recognition algorithm running on the drone for detecting predefined objects
* A manual/automatic control mechanism for the car.
* A manual/automatic control mechanism for the drone.
* A communication channel between the car and drone (for sending commands).
* A side communication channel for the car and drone (for data collection).
* An image processing offloading (from the drone to the car) algorithm for the drone.
* A log system to record data points about the drones battery life, bandwidth, and latency.
* A GUI-based interface that can be run on a remote device to control the drone and car system (run experiment) as well as record logs.

## **1.3 Software context**

We will be applying software to a drone and car for research purposes. It is not intended to be widely available and only used by researchers for data creation and analysis. It is also meant to be used in a controlled environment with simple scenarios for gathering data.

## **1.4 Major constraints**

The project needs to be completed by August 15th, 2023. This gives us approximately six months to complete the model. However, after the model is finished, we would like to use it to collect research data of our own. In order to do this, the model would need to be finished before August.

Currently, we are working with the PiCAR-x and the Clover 4.2 Drone.

We were unable to obtain extra Raspberry-Pi’s in order to give everyone on the team the ability to work with one. There is a very limited budget and it is unlikely the remaining budget can afford extra drones, cars, or other such (relatively) expensive technology.

The Raspberry Pi for the PiCarX came in later than anticipated, preventing us from being able to work on it.

Additionally, drone flight regulations from University and state policies make finding and developing a more comprehensive test environment more challenging and limited.

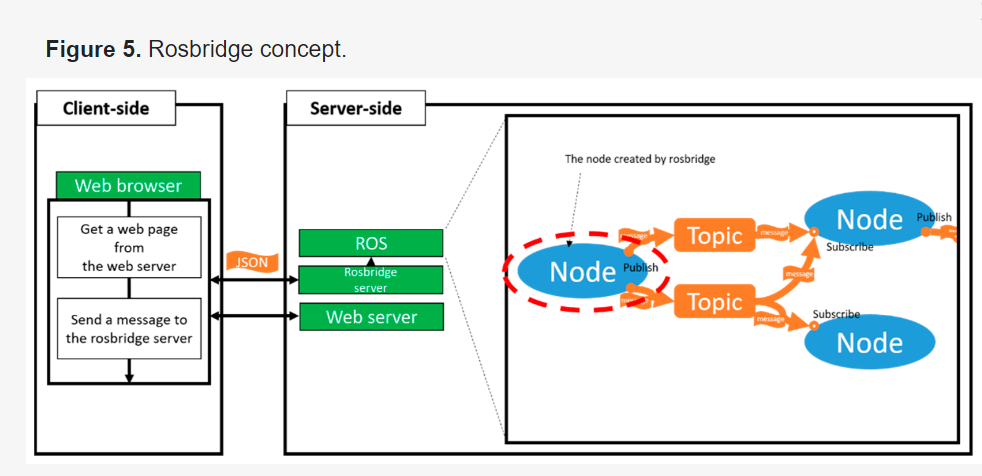
# 2**.0 Data** Design

## 2**.1** Internal software data structure

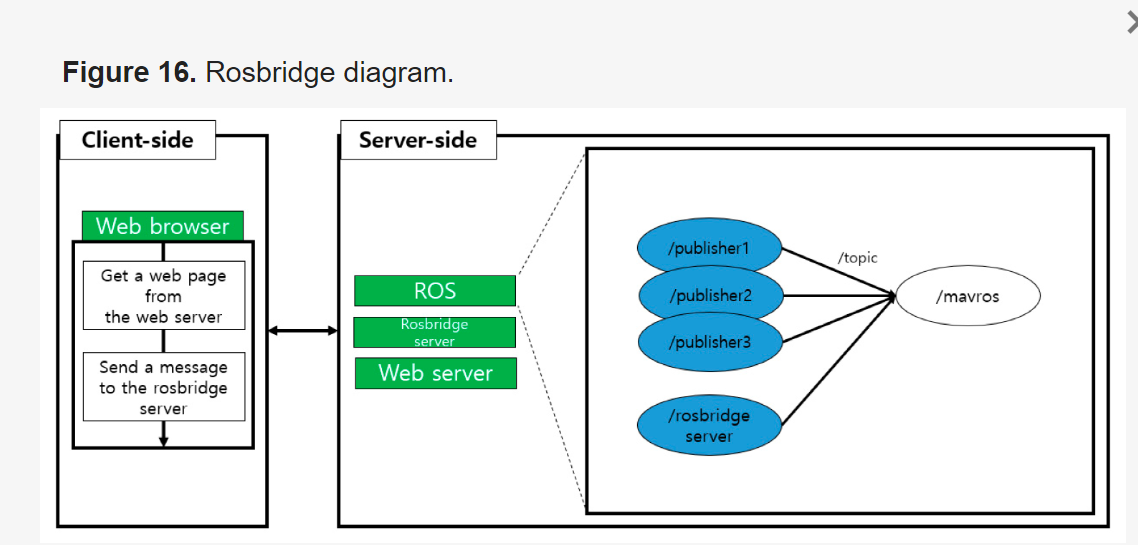
Our program is designed to allow data to flow through. We are utilizing the JSON module to listen to a data stream from the drone, and actively append to a file for the user. Our object detection program uses arrays to convert Javascript objects into OpenCV images, making them into the correct format for detection.

### **2.2 Global data structure**

Receiving JSON data from input streams that are written to the GUI. For the input streams, we are using the web interface messages provided by ROS topics that are published to the web server running on the drone. The Rosbridge package (from ROS) provides the integration of ROS topic messages and the web server:



For our design, instead of using the browser directly, we simply get the address of the server and the path to where the data is being published and feed that into our own GUI interface for display. We are using the Kivy 2.2 API to implement our GUI. Also, the Rosbridge interface allows us to send commands to the drone through ROS topics, which must be specifically translated to MAVLink messages (provided by the MAVROS library), which are the type of messages the flight control unit of the drone expects:



### **2.3 Temporary data structure**

Files created for interim use are described.

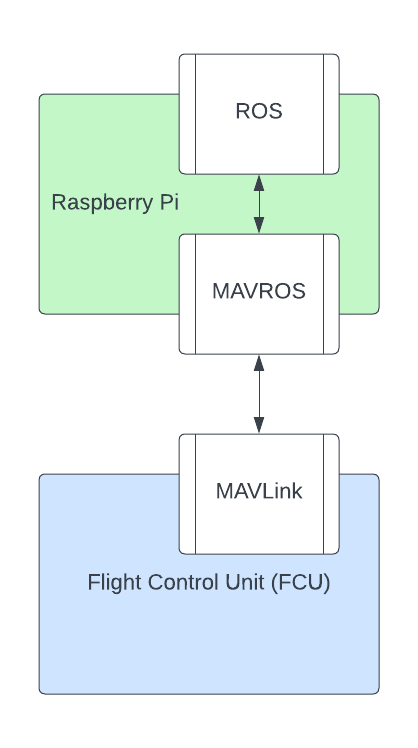
Many of our temporary files are user defined. Some examples of these files are the output file for data collection and the output file where the recording is saved. Where the recording is saved, is decided by the user. Other generated temporary files are stored in the /tmp/ folder.

# 3**.0** Architecture and component-level design

## 3.1 Program structure

A detailed description of the program structure chosen for the application is presented.

In order to implement our program, we needed to become familiar with ROS (Robot Operating System), and a special ROS library package called *MAVROS*. The reason why you need to use the MAVROS package is because of the kinds of messages the flight control unit of the drone (Pixracer FCU) expects. So, MAVROS provides an interface between the ROS software framework and the MAVLink communication framework:



During our design, some questions needed to be answered in order to understand exactly how the ROS/MAVROS/MAVLink frameworks interact with each other in order to understand our drone-car collaboration architecture that we designed.

Next, let us introduce a few excerpt sections from the documentation of the Clover 4.2 drone that we are using, which gives some background on ROS, MAVROS and MAVLink, and how they use the frameworks for their open-source drone that we are using in our program architecture:

### ROS

Main documentation: [https://wiki.ros.org](https://wiki.ros.org/).

**ROS** is a widely used framework for developing complex and distributed robotic systems. The [Clover autonomous flights platform](https://clover.coex.tech/en/programming.html) is based on ROS.

#### Installation

ROS is already installed on [the RPi image](https://clover.coex.tech/en/image.html) of the Clover Drone.

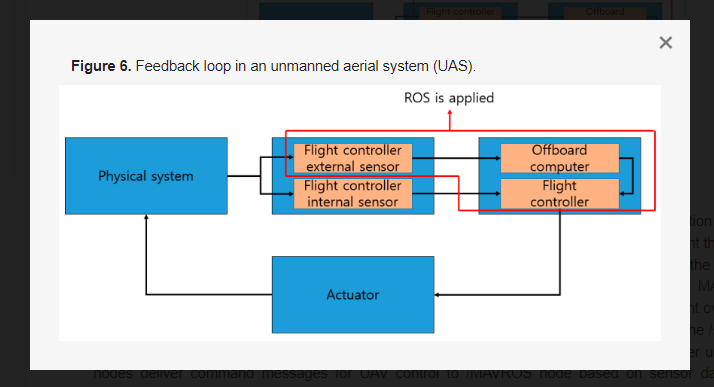
To install ROS on your PC you may address the [official installation documentation](https://wiki.ros.org/noetic/Installation/Ubuntu). For a quick start it's recommended to use [the virtual machine image with ROS and Clover simulator](https://clover.coex.tech/en/simulation_vm.html).

#### **Concepts**

##### **Nodes**

Main article: <https://wiki.ros.org/Nodes>.

ROS node is a special program (usually written in Python or C++) that communicates with other nodes via ROS topics and ROS services. Dividing complex robotic systems into isolated nodes provides certain advantages: reduced coupling of the code, increased reusability and reliability.



Many robotic libraries and drivers are made as ROS nodes.

In order to turn an ordinary program into a ROS node, include the rospy (Python) or roscpp (C++) library, and insert the initialization code.

An example of a ROS node in Python:

import rospy

rospy.init\_node('my\_ros\_node') # the name of the ROS node

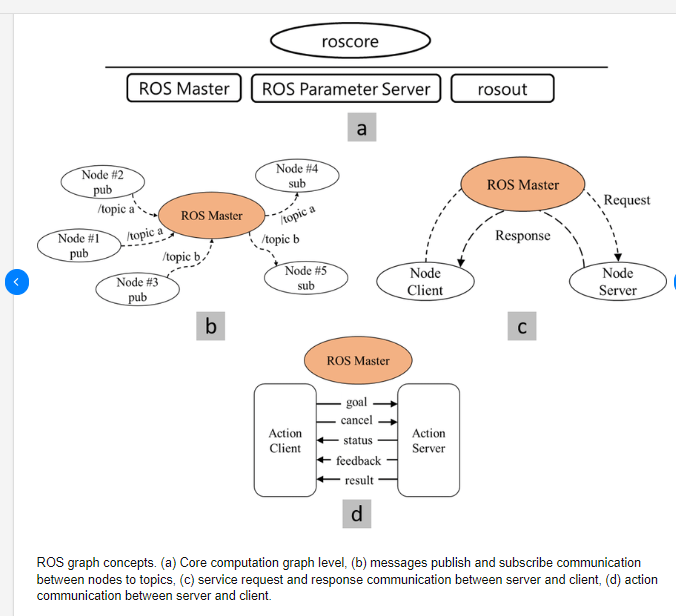
rospy.spin() # entering an infinite loop...

Any [autonomous flight script](https://clover.coex.tech/en/programming.html) for Clover is a ROS node.

##### **Topics**

Main article: <https://wiki.ros.org/Topics>

A topic is a named data bus used by the nodes for exchanging messages. Any node can *publish* a message to any topic, and *subscribe* to any topic.



Each topic has the a of messages it passes. ROS include a lot of standard message types, covering different aspects of robotics. Creating custom message types is also possible. Example of standard message types:

| **Message type** | **Description** |
| --- | --- |
| [std\_msgs/Int64](https://docs.ros.org/api/std_msgs/html/msg/Int64.html) | Integer number. |
| [std\_msgs/Float64](https://docs.ros.org/api/std_msgs/html/msg/Float64.html) | Double-precision floating-point number. |
| [std\_msgs/String](https://docs.ros.org/api/std_msgs/html/msg/String.html) | String. |
| [geometry\_msgs/PoseStamped](https://docs.ros.org/api/geometry_msgs/html/msg/PoseStamped.html) | Position and orientation of an object in a given [coordinate system](https://clover.coex.tech/en/frames.html) and a time stamp (widely used for passing the robot pose or some robot's part pose). |
| [geometry\_msgs/TwistStamped](https://docs.ros.org/api/geometry_msgs/html/msg/TwistStamped.html) | Linear and angular velocity of an object in a given coordinate system and a time stamp. |
| [sensor\_msgs/Image](https://docs.ros.org/api/sensor_msgs/html/msg/Image.html) | Image (see the [article on working with the camera](https://clover.coex.tech/en/camera.html)). |

See the rest of standard message types in packages: [common\_msgs](http://wiki.ros.org/common_msgs), [std\_msgs](https://wiki.ros.org/std_msgs), [geometry\_msgs](https://wiki.ros.org/geometry_msgs), [sensor\_msgs](https://wiki.ros.org/sensor_msgs), and others.

Example of publishing a message of type [String](https://clover.coex.tech/en/(https:/docs.ros.org/api/std_msgs/html/msg/String.html)) in a topic /foo in Python:

from std\_msgs.msg import String

rospy.init\_node('my\_ros\_node')

foo\_pub = rospy.Publisher('/foo', String, queue\_size=1) # creating a Publisher

foo\_pub.publish(data='Hello, world!') # publishing the message

Example of subscription to a topic /foo:

import rospy

from std\_msgs.msg import String

rospy.init\_node('my\_ros\_node')

def foo\_callback(msg):

print(msg.data)

# Subscribing. When a message is received in topic /foo, function foo\_callback will be invoked.

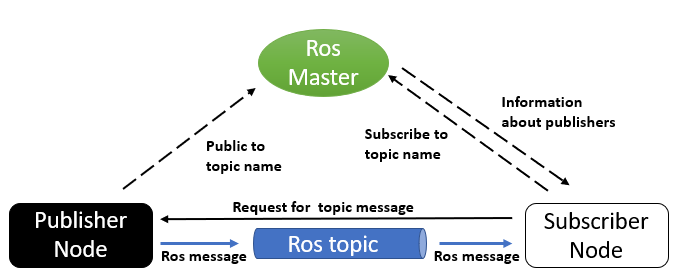
rospy.Subscriber('/foo', String, foo\_callback)

You can read a topic message once, using wait\_for\_message function:

msg = rospy.wait\_for\_message('/foo', String, timeout=3) # wait for a message in /foo topic with timeout of 3 seconds

You can also work with topics using the rostopic utility. For example, using the following command, you can view messages published in topic /mavros/state:

rostopic echo /mavros/state



The rostopic info command shows the type of messages in the topic, and rostopic hz shows frequency of published messages.

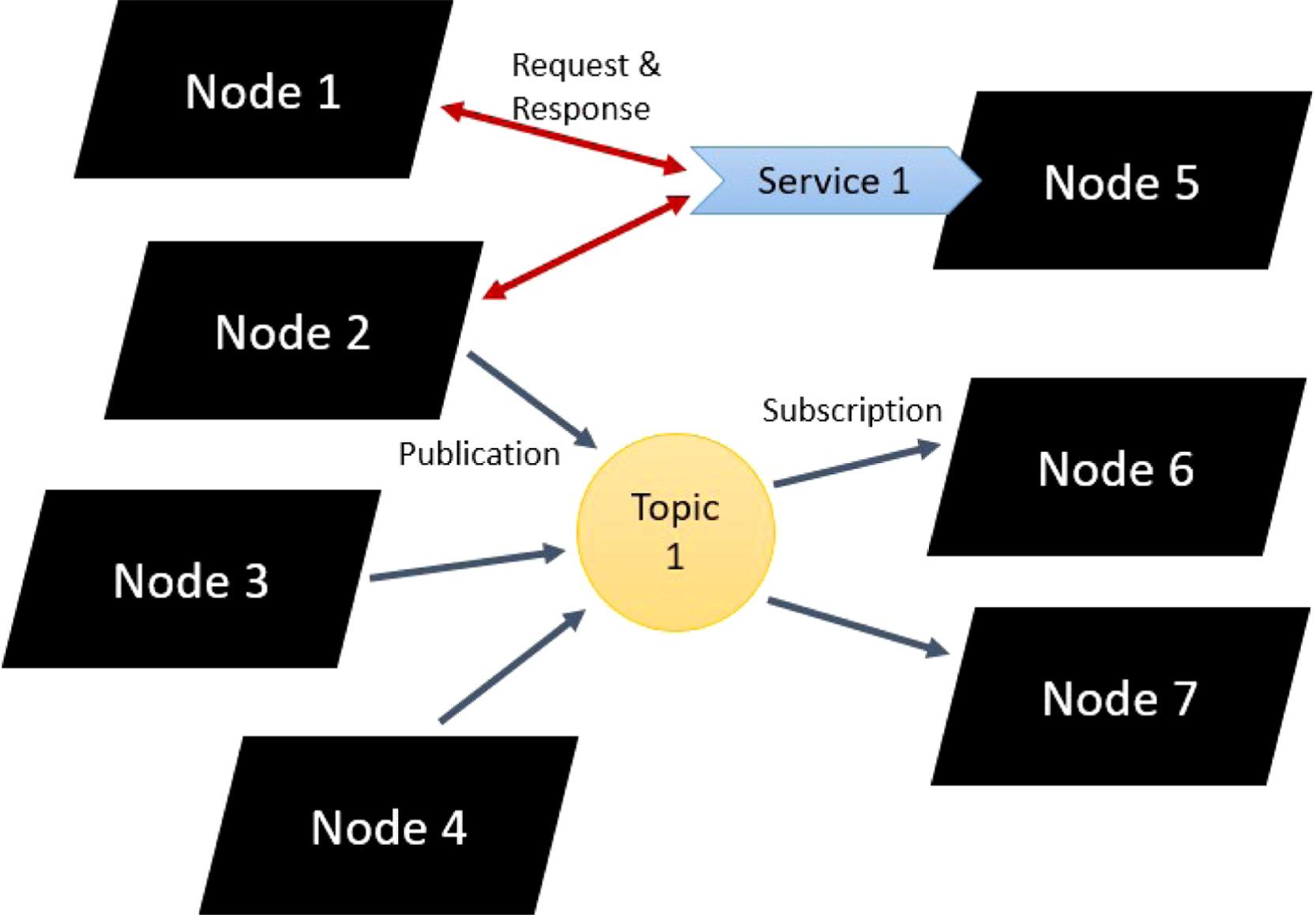
Also you can monitor and visualize topics using [graphical tools of ROS](https://clover.coex.tech/en/rviz.html).

##### **Services**

Main article: <https://wiki.ros.org/Services>

A service can be assimilated to a function that can be called from one node, and processed in another one. This allows for interprocess (inter-node) invocations, where one node (process) can communicate with another node (passing parameter data) specifically for the purpose of calling some function that a node can provide, and the called node can do some processing and send the result back to the calling node. The service has a name that is similar to the name of the topic, and 2 message types: request type and response type.

Thus, ROS services implement [remote procedure call (RPC)](https://en.wikipedia.org/wiki/Remote_procedure_call) pattern.



Example of invoking a ROS service in Python:

import rospy

from clover.srv import GetTelemetry

rospy.init\_node('my\_ros\_node')

# Creating a wrapper for the get\_telemetry service of the clover package with the GetTelemetry type:

get\_telemetry = rospy.ServiceProxy('get\_telemetry', srv.GetTelemetry)

# Invoking the service, and getting the quadcopter telemetry:

telemetry = get\_telemetry()

You can also work with the services using the rosservice utility. For instance, you can call service /get\_telemetry from the command line:

rosservice call /get\_telemetry "{frame\_id: ''}"

More examples of using the services for Clover quadcopter autonomous flights are available in the [documentation for node simple\_offboard](https://clover.coex.tech/en/simple_offboard.html).

##### **Names**

Main article: <https://wiki.ros.org/Names>.

Any topic, service or a parameter is identified with a unique name. A ROS name is hierarchical structure with a / symbol as a separator (which is close to a file name in a file system).

Examples of ROS names:

* / (global namespace)
* /foo
* /stanford/robot/name
* /wg/node1

This names are global (close to global names in a file system). In practice, it's recommended to use *private* or *relative* names.

###### **Private name**

Each node can use its own private namespace (corresponding its name) for its resources. For example, aruco\_detect node may publish such topics:

* /aruco\_detect/markers
* /aruco\_detect/visualization
* /aruco\_detect/debug

When a node is referring its private resource, instead of /aruco\_detect/ namespace it may use ~ symbol:

* ~markers
* ~visualization
* ~debug

Thus, creating a foo topic and the private namespace would look like this:

private\_foo\_pub = rospy.Publisher('~foo', String, queue\_size=1)

###### **Relative name**

Several nodes may group into a common namespace (for example, when there are several robots in the network). For referring topics and services in the current namespace, the opening / symbol is omitted.

Example of create a foo topic in the current namespace:

relative\_foo\_pub = rospy.Publisher('foo', String, queue\_size=1)

Generally, it's recommended to use private or relative names instead of global ones.

##### **Working on several PCs**

Main article: <http://wiki.ros.org/ROS/Tutorials/MultipleMachines>.

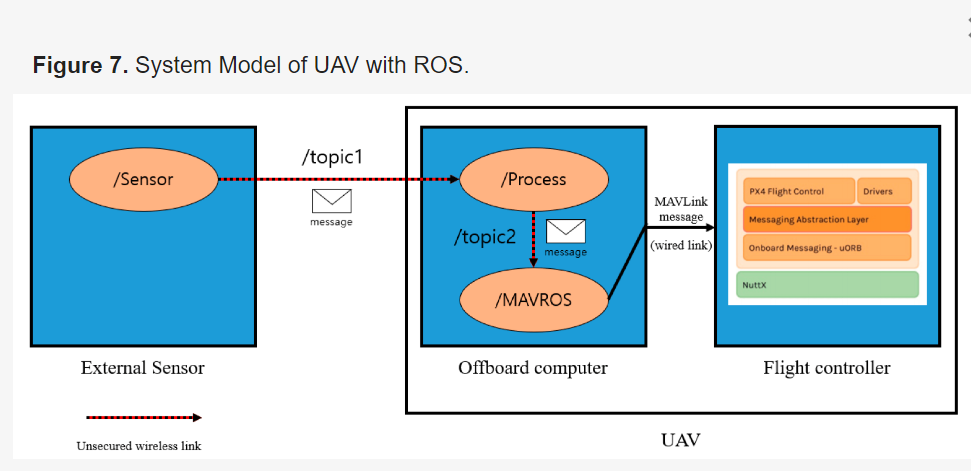
The advantage of using ROS is the possibility of distributing the nodes across several PCs in the network. For example, a node that recognizes an image may be run on a more powerful PC; the node that controls the copter may be run directly on a Raspberry Pi connected to the flight controller, etc.

### MAVROS

Main article is available in the official documentations: <http://wiki.ros.org/mavros>

MAVROS (MAVLink + ROS) is a ROS package that allows controlling drones via the [MAVLink](https://clover.coex.tech/en/mavlink.html) protocol. MAVROS supports PX4 and APM flight stacks. Communication may be established via UART, USB, TCP or UDP.

MAVROS subscribes to certain ROS topics that can be used to send commands, publishes telemetry to other topics, and provides services.



The MAVROS node is automatically started in the Clover launch-file. In order to [set the type of connection](https://clover.coex.tech/en/connection.html), change the fcu\_conn argument.

Simplified interaction with the drone is possible with the use of [simple\_offboard] package (simple\_offboard.md).

Some MAVROS plugins are disabled by default in the clover package in order to save resources. For more information, see the plugin\_blacklist parameter in /home/pi/catkin\_ws/src/clover/clover/launch/mavros.launch.

#### Main services

/mavros/set\_mode — set [flight mode](https://clover.coex.tech/en/modes.html) of the controller. Most often used to set the OFFBOARD mode to accept commands from Raspberry Pi.

/mavros/cmd/arming — arm or disarm drone motors (change arming status).

#### Main published topics

/mavros/state — status of connection to the flight controller and flight controller mode.

/mavros/local\_position/pose — local position and orientation of the copter in the ENU coordinate system.

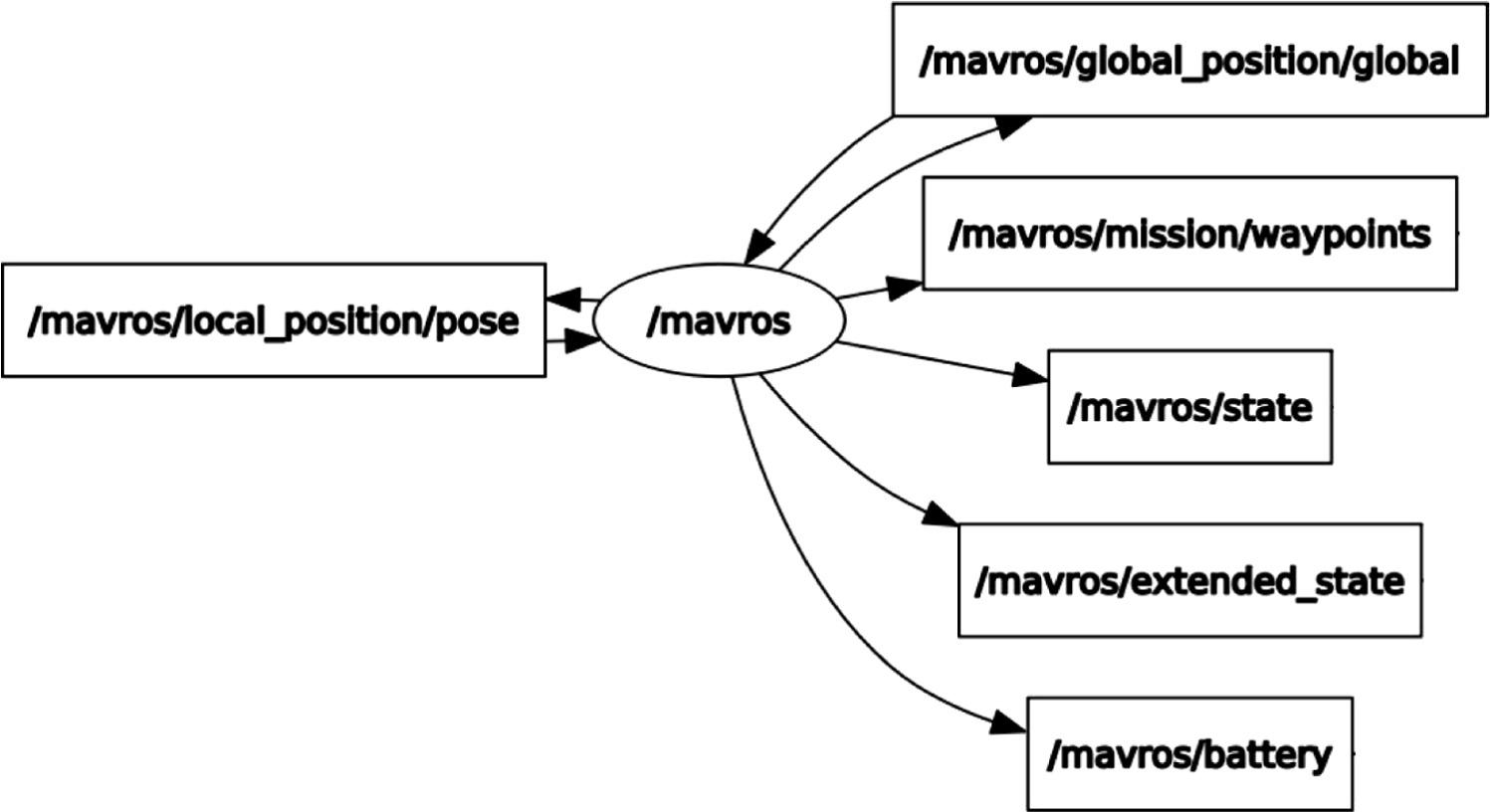
/mavros/local\_position/velocity — current speed in local coordinates and angular velocities.

/mavros/global\_position/global — current global position (latitude, longitude, altitude).

/mavros/global\_position/local — the global position in the [UTM](https://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system) coordinate system.

/mavros/global\_position/rel\_alt — relative altitude (relative to the arming altitude).

Messages published in the topics may be viewed with the rostopic utility, e.g., rostopic echo /mavros/state. See more in [working with ROS](https://clover.coex.tech/en/ros.html).



#### Main topics for publication

/mavros/setpoint\_position/local — set target position and yaw of the drone (in the ENU coordinate system).

/mavros/setpoint\_position/global – set target position in global coordinates (latitude, longitude, altitude) and yaw of the drone.

/mavros/setpoint\_position/cmd\_vel — set target linear velocity of the drone.

/mavros/setpoint\_attitude/attitude and /mavros/setpoint\_attitude/att\_throttle — set target attitude and throttle level.

/mavros/setpoint\_attitude/cmd\_vel and /mavros/setpoint\_attitude/att\_throttle — set target angular velocity and throttle level.

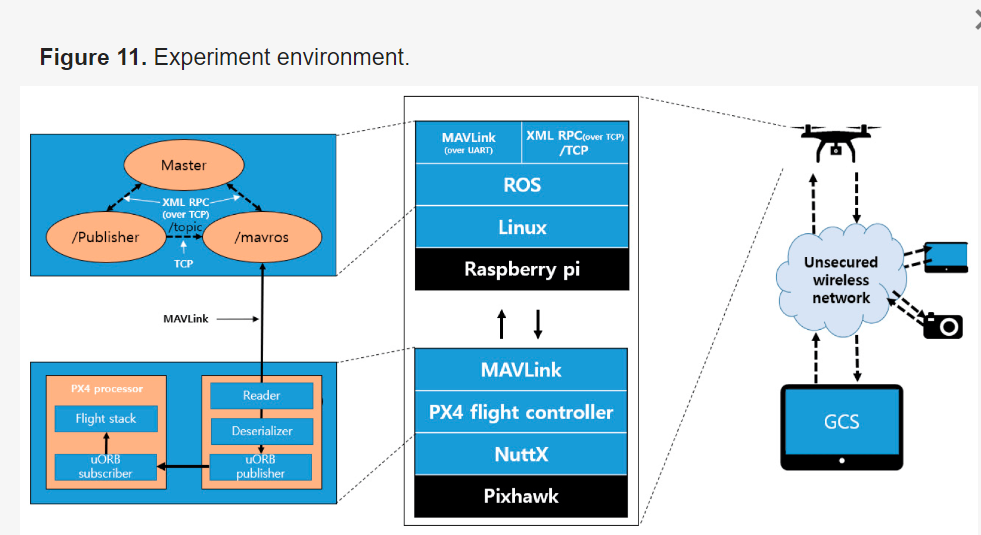
#### Topics for sending raw packets

/mavros/setpoint\_raw/local — sends [SET\_POSITION\_TARGET\_LOCAL\_NED](https://mavlink.io/en/messages/common.html#SET_POSITION_TARGET_LOCAL_NED) message. Allows setting target position/target speed and target yaw/angular yaw velocity. The values to be set are selected using the type\_mask field.

/mavros/setpoint\_raw/attitude — sends [SET\_ATTITUDE\_TARGET](https://mavlink.io/en/messages/common.html#SET_ATTITUDE_TARGET) message. Allows setting the target attitude /angular velocity and throttle level. The values to be set are selected using the type\_mask field

/mavros/setpoint\_raw/global — sends [SET\_POSITION\_TARGET\_GLOBAL\_INT](https://mavlink.io/en/messages/common.html#SET_POSITION_TARGET_GLOBAL_INT). Allows setting the target attitude in global coordinates (latitude, longitude, altitude) and flight speed.

In summary, here is a nice diagram that demonstrates how the drone framework we are using works, except we are using a secure communication link (unlike what the diagram shows) through a TCP connection:



### Key Architecture Questions

In the rest of this section, we will ask and answer 5 key architectural questions, and provide diagrams that demonstrate the architecture:

**1) What is the difference between ROS and MAVROS**

ROS (Robot Operating System) and MAVROS (MAVLink ROS Interface) are two related but distinct frameworks commonly used in the field of robotics and unmanned aerial vehicles (UAVs). Here's a breakdown of their differences:

*Purpose*:

* ROS: ROS is a flexible framework for developing robot software. It provides a collection of libraries, tools, and conventions to simplify the development of robot applications across a wide range of platforms.
* MAVROS: MAVROS is a ROS package specifically designed to provide communication between ROS and MAVLink-based autopilots, primarily used in UAVs. It acts as a bridge between the two systems, enabling the exchange of messages and commands.

*Functionality:*

* ROS: ROS offers a comprehensive set of capabilities for robotic application development, including message passing, hardware abstraction, sensor integration, motion planning, visualization, and more. It provides a modular and extensible architecture that facilitates the development of complex robot systems.
* MAVROS: MAVROS focuses specifically on integrating MAVLink-based autopilots, such as those used in drones, with the ROS ecosystem. It handles the translation of MAVLink messages into ROS messages and provides convenient ROS interfaces to control UAVs, access sensor data, and monitor the vehicle's status.

*Communication Protocol:*

* ROS: ROS uses its own communication protocol, known as ROS topics, to enable communication between different nodes (software components) within a ROS system. Nodes can publish and subscribe to topics, allowing for the exchange of messages.
* MAVROS: MAVROS utilizes the MAVLink protocol, a lightweight communication protocol widely used in the unmanned systems community. MAVLink is designed for efficient communication between onboard systems and ground control stations, and MAVROS extends this communication to the ROS environment.

*Application Scope:*

* ROS: ROS is a general-purpose framework applicable to a wide range of robotic systems beyond UAVs. It is used in industrial robots, autonomous vehicles, manipulators, mobile robots, and more.
* MAVROS: MAVROS is primarily used in the context of UAVs and is particularly valuable for developers working with drones or other MAVLink-based autopilots within the ROS ecosystem.

In summary, ROS is a comprehensive framework for robot software development, while MAVROS is a ROS package specifically tailored for integrating MAVLink-based autopilots into the ROS environment, primarily for UAV applications.

**2) Why are MAVLink messages different from ROS messages?**

MAVLink messages and ROS messages serve different purposes and are designed with different considerations in mind. Here are a few reasons why MAVLink messages and ROS messages differ:

*Communication Efficiency:*

* MAVLink: MAVLink is a lightweight communication protocol designed for efficient transmission of data between onboard systems and ground control stations in resource-constrained environments. It prioritizes minimal bandwidth and low latency to ensure real-time communication between the vehicle and the ground station.
* ROS: ROS messages prioritize ease of use and flexibility rather than communication efficiency. While efficiency is important, ROS is often used in more powerful computing systems where bandwidth and latency are not as critical. ROS messages provide additional metadata, such as time stamps and frame IDs, to support various functionalities within the ROS ecosystem.

*System Independence:*

* MAVLink: MAVLink messages are designed to be platform-independent and support interoperability across different autopilot systems. They provide a standardized format for communication between autopilots and ground control stations, enabling seamless integration of various UAV hardware and software components.
* ROS: ROS messages are primarily intended for communication within the ROS ecosystem. They are specific to the ROS framework and its data types. ROS messages are tailored to support the modularity and flexibility provided by ROS, allowing developers to easily exchange information between different software components within a robotic system.

*Message Content:*

* MAVLink: MAVLink messages are often focused on providing essential information related to UAV control and status. They include data such as GPS position, attitude, sensor readings, control commands, and system health status. The content of MAVLink messages is optimized for the needs of UAV operations.
* ROS: ROS messages are more generic and versatile, accommodating a wide range of robot-related data. They can represent sensor readings, control commands, image data, point clouds, robot state information, and more. ROS messages are designed to be flexible and extensible to support diverse robotic applications beyond UAVs.

While MAVLink messages and ROS messages have their specific design considerations, it's worth noting that the MAVROS package provides a translation layer between the two systems. It converts MAVLink messages into ROS messages, allowing developers to leverage the ROS ecosystem while working with MAVLink-based autopilots.

**3) Are MAVLink messages used to send controls to the flight control unit?**

Yes, MAVLink messages are commonly used to send control commands to the flight control unit (FCU) or autopilot in a UAV system. The FCU is responsible for managing the flight behavior of the UAV and executing the control commands received from the ground control station or other external systems.

MAVLink provides specific message types for sending control commands, such as:

* MAV\_CMD\_NAV\_TAKEOFF: Used to command the UAV to take off and reach a specified altitude.
* MAV\_CMD\_NAV\_LAND: Used to command the UAV to land at its current position.
* MAV\_CMD\_NAV\_WAYPOINT: Used to send waypoints or navigation targets to the UAV, specifying locations it should fly to in a mission.
* MAV\_CMD\_DO\_SET\_MODE: Used to change the flight mode of the UAV, such as switching between autonomous mode and manual mode.
* MAV\_CMD\_DO\_SET\_SERVO: Used to control servo motors on the UAV for various purposes, such as controlling camera gimbals or other onboard mechanisms.
* MAV\_CMD\_DO\_CONTROL\_VIDEO: Used to control video-related parameters, such as starting or stopping video recording.

These MAVLink control commands can be constructed and sent from a ground control station or any system capable of generating MAVLink messages. The FCU interprets these messages and executes the corresponding actions, enabling remote control and autonomous operation of the UAV.

**4) Are MAVLink messages sent over TCP?**

MAVLink messages can be sent over various transport protocols, including TCP (Transmission Control Protocol), but they are not limited to TCP alone. MAVLink is designed to be transport-agnostic, meaning it can be used with different communication protocols depending on the specific requirements and constraints of the system.

Here are a few common transport protocols used with MAVLink:

* *Serial:* MAVLink messages are often transmitted over serial connections, such as UART (Universal Asynchronous Receiver-Transmitter) or USB, between the autopilot or flight control unit and the ground control station or other onboard systems. This is a common method for direct communication between the UAV hardware and the ground station.
* *UDP (User Datagram Protocol):* UDP is a lightweight and low-latency transport protocol suitable for real-time communication. MAVLink messages can be sent over UDP for communication between the UAV and the ground station or other systems over a local network or the Internet. UDP is often used when low latency is crucial, but it does not provide reliability guarantees.
* *TCP (Transmission Control Protocol):* MAVLink messages can be encapsulated within TCP packets for reliable and ordered communication between systems. TCP provides mechanisms for error detection, retransmission of lost packets, and ordered delivery. It is commonly used for communication over the Internet or in situations where reliable and ordered message delivery is critical.

It's important to note that the choice of transport protocol depends on factors such as communication requirements, network infrastructure, latency constraints, and reliability needs. The specific implementation and configuration of MAVLink communication can vary based on the platform, software stack, and requirements of the UAV system.

**5) What is a callback function in ROS?**

In ROS (Robot Operating System), a callback refers to a function or method that is registered to be executed in response to a specific event or message. The callback function is called asynchronously when the event or message occurs, allowing the program to respond to that event in a timely manner.

Callbacks are an essential mechanism in ROS for handling incoming data, responding to events, and enabling communication between different nodes (software components). Here's how callbacks work in ROS:

*Subscriber Callbacks:*

In ROS, a subscriber is a component that listens to a specific ROS topic, waiting for messages published on that topic.

* When a message is published on the subscribed topic, the ROS middleware dispatches the message to the associated subscriber.
* The subscriber's callback function, registered during the setup of the subscriber, is then invoked with the received message as an argument.
* The subscriber's callback can process the data, update internal state, or trigger further actions based on the received message.

*Timer Callbacks:*

A timer callback is a function that is executed periodically at a specified frequency.

* In ROS, timers are commonly used for tasks that require regular execution, such as updating sensor readings, performing control calculations, or publishing data at fixed intervals.
* A timer callback is registered with a specific duration or frequency, and the ROS runtime system calls the callback function automatically at the specified intervals.

*Service Callbacks:*

In ROS, services provide a request-response mechanism for communication between nodes.

* A service callback is a function that gets invoked when a service request is received.
* When a node receives a service request, it dispatches the request to the appropriate service callback function registered for that service.
* The service callback processes the request, performs the necessary computations or actions, and sends back a response to the client.

Callbacks are fundamental to the event-driven and message-passing nature of ROS, enabling nodes to react to data or events as they occur. By registering callbacks, developers can define custom logic to handle specific events or message types, allowing for modular and flexible ROS system design.

### Appending Our Drone-car Architecture:

Now, we can discuss how we integrate our own program structure into the ROS/MAVROS/MAVLink framework.

#### **Description of GUI API: Kivy 2.2**

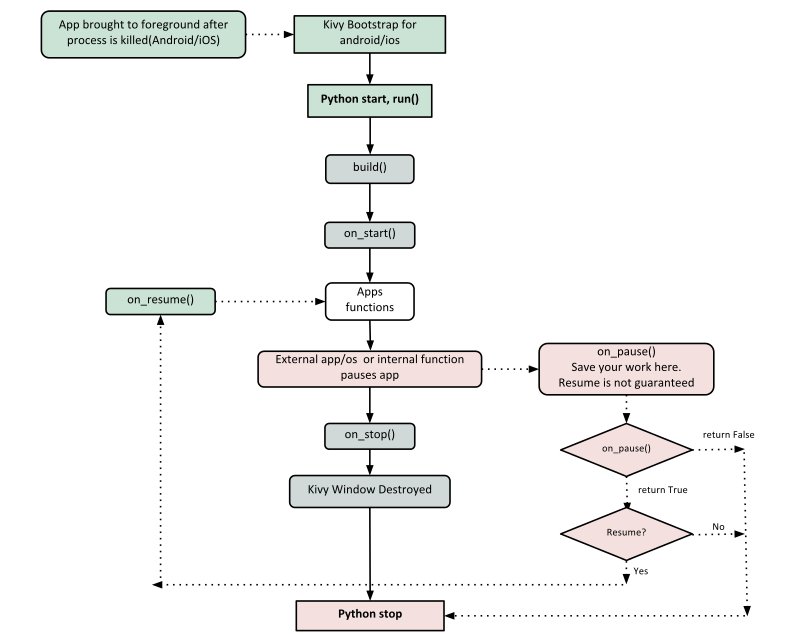
For our GUI, we are using Kivy 2.2 API.

The Kivy API provides functionality for creating and managing user interfaces, handling user input (such as touch, mouse, and keyboard events), managing graphics and animations, working with audio and video, accessing sensors (e.g., accelerometer), and much more.

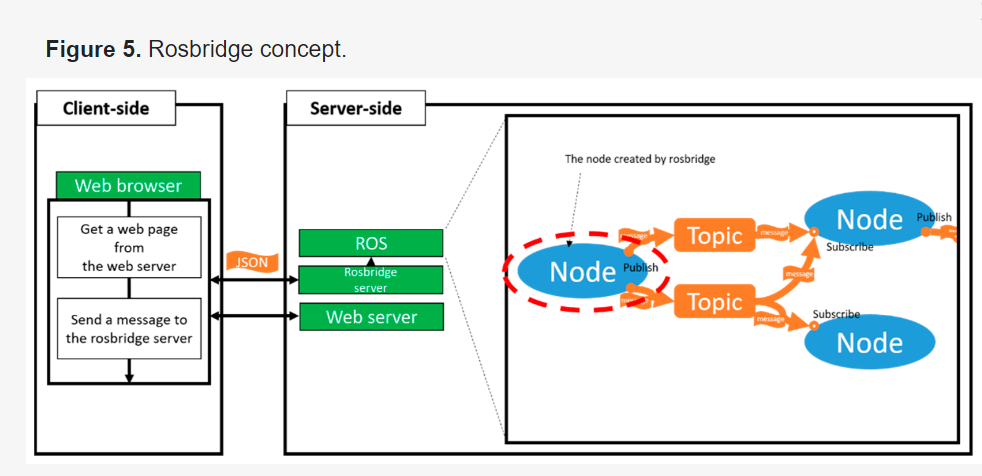
Some of the key components and classes in the Kivy API include:

1. App: The base class for creating Kivy applications.
2. Widget: The base class for all UI elements, representing visual components that can be displayed on the screen.
3. Layouts: Classes like BoxLayout, GridLayout, and FloatLayout that help with arranging and organizing widgets.
4. Event handling: Classes and methods for managing user input events such as touches and keyboard inputs.
5. Properties: A system for defining and managing properties of widgets and other Kivy objects.
6. Graphics: Modules and classes for drawing and manipulating graphics, including shapes, colors, textures, and shaders.
7. Animation: Classes for creating and controlling animations.
8. File system and resources: Modules and classes for working with files, directories, and resources (such as images and sounds) used in the application.
9. Widgets: A wide range of pre-built UI elements like buttons, labels, text inputs, sliders, progress bars, etc.
10. User interface language: Kivy provides a language called KV language that allows developers to define user interfaces in a declarative manner.

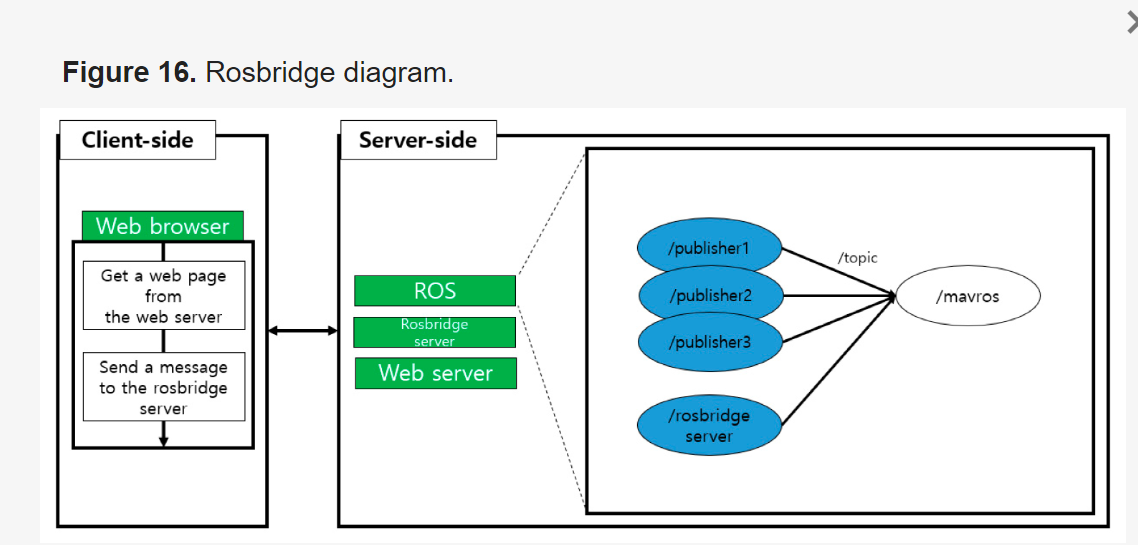
Here is an example of the Kivy app life cycle:



For the input streams, we are using the web interface messages provided by ROS topics that are published to the web server running on the drone. The Rosbridge package (from ROS) provides the integration of ROS topic messages and the web server:



For our design, instead of using the browser directly, we simply get the address of the server and the path to where the data is being published and feed that into our own GUI interface for display. We are using the Kivy 2.2 API to implement our GUI. Also, the Rosbridge interface allows us to send commands to the drone through ROS topics, which must be specifically translated to MAVLink messages (provided by the MAVROS library), which are the type of messages the flight control unit of the drone expects:



#### **Description of Drone-Car interfaces**

**1. RedDetect.py (run on drone):**

- This script is responsible for detecting red objects in a video feed using computer vision techniques.

- It establishes a connection with a car server using the drone\_client.py script.

- It initializes a ROS (Robot Operating System) node for red object detection.

- It subscribes to the video frames from the main camera and processes them.

- It detects red objects in the frames, draws bounding boxes around them, and publishes the modified frames to a ROS topic.

- If a red object is detected, it sends a message to the car server to stop the car temporarily and then continue driving.

- It uses the OpenCV library for image processing and the ROS framework for communication.

**2. Drone\_client.py (run on drone, through being called by RedDetect.py):**

- This script is a client-side module responsible for establishing a socket connection with the car server.

- It provides functions to send messages to the car server indicating different commands.

- It includes a wait\_timer() function to introduce delays in the program.

- It also contains a close\_socket() function to close the socket connection.

- It imports the Picarx library, which suggests that it may be used to control the car's movements.

**3. Car\_server.py (run on car/edge server):**

- This script acts as the server-side program running on the car.

- It listens for incoming connections from the RedDetect program.

- It establishes a socket connection with the RedDetect program.

- It receives commands from the RedDetect program indicating actions to be taken by the car.

- Based on the received commands, it controls the car's movements using the Picarx library.

- If instructed to stop, it stops the car. If instructed to continue driving, it resumes normal driving.

Overall, the three scripts work together to detect red objects in the video feed and control the car's movements accordingly.

RedDetect.py performs the image processing and communication with the car server,

drone\_client.py handles the socket connection and message passing,

and car\_server.py receives the commands and controls the car's actions.

Because the drone and car use Raspberry Pi model 4B computers which have limited processing power, we will use another machine for additional computational resources. This extra machine will serve as a standin for the car, since in a real-life framework, the car would have much more battery power and computational resources than a drone. We needed to do this specifically because the computer vision models that we will run to detect color and then shapes of objects will be too much to process, especially for the drone which must process messages sent between the flight control unit and the Raspberry Pi, as well as manage a socket connection between the car and any other communication nodes.

#### **Description of Communication Protocols:**

In order to create a communication channel between the drone and the car (and the additional edge server machine to assist with the cars computational power), we use the DNSMasq and WPA Supplicant APIs. This allows all nodes to communicate on the same IP network. This includes any computers on the network that can remote into the car or drone (via SSH), or that may run the GUI interface that connects to the car and drone. This is all done through a TCP/IP connection, and a UDP side channel which will be used specifically to send and receive video messages to and between the drone and car, and reserve a TCP socket used for command communications.

Here is some more background on the two communication components. Note that DNSMasq is only ran on the drone in order for the drone to act as a router and DHCP server, and the WPA Supplicant tool is ran on both the drone and the car to connect to (or broadcast) a network:

DNSMasq and WPA Supplicant are two important components that can be used together to enable a Raspberry Pi to function as a router. Let's understand their roles and how they contribute to the routing functionality:

1. DNSMasq: DNSMasq is a lightweight DNS (Domain Name System) forwarder and DHCP (Dynamic Host Configuration Protocol) server. It provides DNS caching and DHCP services, making it a useful tool for small networks or situations where a full-fledged DNS server is not required.

When configured as a router, DNSMasq can handle DNS queries and provide DHCP services to devices connected to the Raspberry Pi. It can assign IP addresses dynamically, act as a gateway for connected devices, and forward DNS requests to appropriate DNS servers on the internet. By acting as a DNS and DHCP server, DNSMasq enables the Raspberry Pi to manage network traffic and provide network services to connected devices.

2. WPA Supplicant: WPA Supplicant is a cross-platform tool that manages Wi-Fi connections on Linux systems. It supports various authentication methods and encryption protocols used in Wi-Fi networks, such as WPA, WPA2, WPA3, and EAP (Extensible Authentication Protocol).

By using WPA Supplicant, the Raspberry Pi can connect to an external Wi-Fi network and function as an access point or a bridge to extend the network. It allows the Raspberry Pi to create a Wi-Fi hotspot that other devices can connect to, essentially turning the Raspberry Pi into a wireless router.

The combination of DNSMasq and WPA Supplicant allows the Raspberry Pi to provide DNS and DHCP services while simultaneously acting as a Wi-Fi access point. This enables the Raspberry Pi to route network traffic between connected devices and the internet, allowing devices to access the internet through the Raspberry Pi's network connection.

Additionally, we will be running Iperf3 scripts to measure bandwidth between the drone and the car. Both the drone and car must have *Iperf3* installed:

***Iperf3 Server Script:***

#!/bin/bash

echo "Running iper3 server script..."

iperf3 -s -p 5201 -V --logfile iperf3\_TCP\_Test\_server.txt

#note that iperf3 appends to the log file rather than overwrite it.

***Iperf3 Client Script:***

#!/bin/bash

echo "Running iperf3 client script for 30 seconds..."

iperf3 -V -c 192.168.11.1 -t 45 --logfile iperf3\_TCP\_Test\_client.txt &

iperf\_pid=$!

for i in {1..45}; do

echo "$i seconds passed."

sleep 1

done

# Kill the iperf3 process

kill $iperf\_pid

wait $iperf\_pid 2>/dev/null

echo "iperf3 client script completed."

#### **Description of Computer Vision model and object segmentation:**

Besides our first layer of detection (color, specifically red), the next layer of image processing involves detecting a class of shapes:

1. Star
2. Triangle
3. Square
4. Circle
5. Oval
6. Hexagon
7. Pentagon

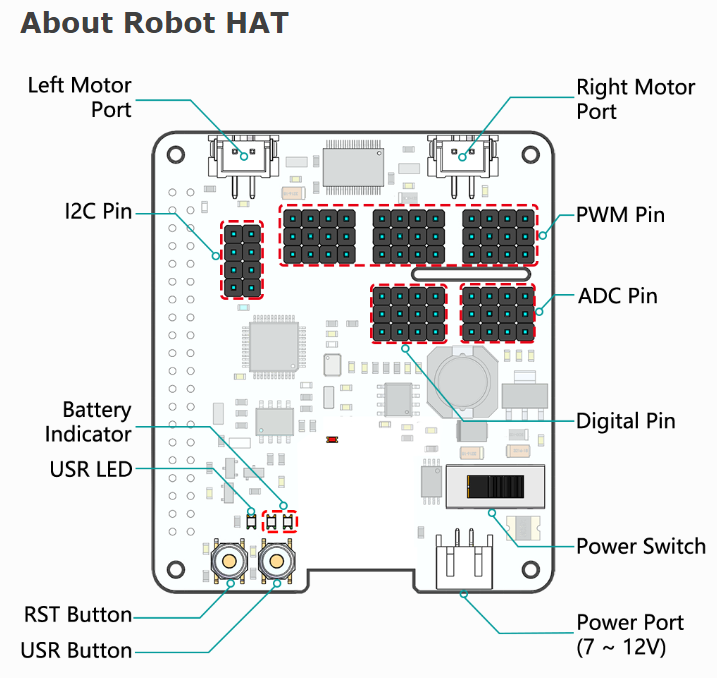
We will do this through using ROS topics in the same way we implement the red detection software component. If red is detected in the frame, the drone will run the YOLOv4 model that we trained to detect the shapes as mentioned above to process the area where red was detected (representative of a critical event). The rest of the frame will be sent to the car’s edge server (the edge server machine that will serve as a standin for the car) for processing shapes. If the frame does not contain red, the entire frame will be transmitted to the edge server for processing shape.

#### **Description of Car control - Robot HAT**

The "Robot HAT" is specifically designed for the "Pi Car X," it is a custom hardware add-on board or expansion module intended for use with the Pi Car X robot car kit.

It is common for robot car kits to come with specialized HATs or motor driver boards. These boards typically provide motor control, sensor interfaces, and other necessary components to control the robot's movement and interact with its environment.

The Robot HAT for Pi Car X is designed to interface with the Raspberry Pi's GPIO pins and provide motor control functionality, as well as support for sensors, servos, or other components required for the robot car's operation. It includes features such as motor drivers, voltage regulation, input/output ports, and connectors specifically tailored for the Pi Car X kit.



In order to control the car, a python program is run on the raspberry pi of the car that calls the Pi Car X library, which is a library written to interface with the ROBOT HAT that controls the motors of the car. The program we created makes a socket connection between the drone and car, so that the drone can send commands to the car.

### 3.1.1 Architecture diagram

#### 

### 3.1.2 Alternatives

A discussion of other architectural styles considered is presented. Reasons for the selection of the style presented in Section 3.1.1 are provided.

We considered not introducing an additional machine to serve as an additional computational power representative for the car. However, our object detection models require a lot of processing power, and there are no alternatives to the raspberry pi computer that has a practical price or implementation timeline for what we are trying to achieve.

However, adding in the machine solves our problems the most effectively without really affecting too much change or the purpose of our drone-car collaboration model. Our framework still stands to help researchers test and implement a unique drone-car collaboration model.

## 3.2Description for component n

A detailed description of each software component contained within the architecture is presented. Section 3.2 is repeated for each of n components

### 3**.2.1** Processing narrative (PSPEC) for component nm

A processing narrative for component n is presented.

3.2.1.a) Object detection: this is used to detect our desired object, (shapes for example), from video frames collect from the drone’s camera. We use Darknet, an open source software, as a way to train our YOLOv4-tiny objects for detection. The YOLOv4-tiny model can take our input as a jpeg images and an XML files with bounding box coordinates to train our models. We use these images and XML files in Darknet to create weight files to be used in later programs to provide object detection accuracy. Another handy software, we used for annotating our dataset was OpenLabeling which automatically generated the corresponding XML file for our jpeg images.

3.2.1.b) Data collection: The data we need to collect can be taken from the drone and car. This includes things such as battery status, network status, etc. For this, we can just implement file write in Python to get these variables and display them in a txt file for users to easily read and then close the file after we are done running the experiment.

3.2.1.c) GUI component: the GUI provides an easy way for the user to run, then experiment and view the video stream. The GUI includes a start button, connect button, standby button, and restart button. There are also testing and settings tabs for the user to click between.

3.2.1.d) Video Recording: the video recording component is used to capture live video streams from a camera on the drone. We plan to use this program to get a bird’s eye view of the ground around the PiCar.

3.2.1.e) Flying the drone: This program is used to fly the drone, allows it to land on an Aruco marker, and hover in a certain area for some time. This program will allow the drone to follow the PiCar to capture a video stream of its surroundings.

3.2.1.f) Car/Drone Communication Program: This allows communication between the car and the drone. It enables the drone to capture and process video frames, and send commands to the PiCar. The program creates a server/socket connection in order for the communication to be possible.

### 3**.2.2** Component n interface description

A detailed description of the input and output interfaces for the component is presented.

3.2.2.a) Object detection: The input is jpeg and XML images of the object and bounding boxes. We also need a weights file which contains the best weights from training the data set. During training, we also needed a configuration file as input. The output includes bounding boxes with a confidence level percentage for detection that are overlaid on our video frames.

3.2.2.b) Data collection: The data collected takes input from the drone and the output is a txt file that contains status information.

3.2.2.c) GUI component: The input relies on the user to press buttons that will send parameters to the drone to conduct the experiment or connect. The output would be the results of what action the user chooses from the GUI, which could be actually running/connecting to the drone.

3.2.2.d) Video Recording:The program takes input from the user of the path they want to save the video, and the output is the video frames being displayed/saved to the path (ros publisher).

3.2.2.e) Flying the drone: The clover SDK runs on the pi which in return controls FCU. For the Aruco marker landing program, we can input the countdown timer we want to use, change the speed, and the amount of time the drone will hover over the marker for. The hover program requires the user to input how long the hover time should be, so the output of this result is the drone actually hovering in the area for the desired time.

3.2.2.f) Car/Drone Communication Program: We need the server IP and port to allow connection from the drone to the car. Input to the server comes from the client in the form of a warning. The server (car) can execute these commands to avoid obstacles.

### 3.2.3Sub-Component n.m processing detail

A detailed algorithmic description for each sub-component within the component n is presented. Section 3.2.3 is repeated for each of the m sub-components of component n.

3.2.3.a) Object Detection: This algorithm uses darknet files to run detection on the image, converting it from a Javascript object from the camera to an image readable by OpenCV. It creates a bounding box byte stream to create an overlay for the object in the video stream. The then uses the coordinates to actually create the bounding box on the overlay. Then the video is output for the user to see the object with the bounding box and object name on it.

3.2.3.b) Data collection: This component has functions that all do the same thing, which is to get data and write to a file. The subcomponents include getting the status of the battery, getting the flight status, and activity status. They output to the same log folder.

3.2.3.c) GUI component: The video stream that holds the content fed into the system is shown on the Simulation Tab for the user to see. All test results are also stored at the location specified by the GUI’s settings.

3.2.3.d) Video Recording: This component is quite simple and is only used to start and store the video stream on the drone, and does not contain multiple components.

3.2.3.e) Flying the drone: This component is initiated when the python file is run in the Linux terminal. It includes an Aruco program, which allows the drone to hover over an Aruco maker and land on the marker. The program uses the Aruco’s frame id and drone frame id, which is used to find the center of the drone. This program gradually decreases the drone’s height as it hovers over the Aruco marker. Another component of flying the drone includes hovering, the input for this program is how long the user wants the drone to hover, which should be greater than 5 seconds. It includes a countdown timer, and directions to fly forward until landing is initiated.

3.2.3.f) Car/Drone Communication component: This component consists of two programs, a server program and a socket program. The socket program’s main function is to connect to the server and send commands to the server, sending warnings if there is an obstacle in the way of the car (the server). The server program should receive these commands sent from the socket program and maneuver the car around any obstacles or stop/slow down the car.

#### **3.2.3.1 Interface description**

A description of sub-component m inputs and outputs is presented.

3.2.3.1.a) Object detection: the input for detection are individual video frames which can be converted from pixel to base64 to create bounding boxes on the video stream. The output will include the bounding box and confidence of the detection as a percentage.

3.2.3.1.b) Data collection: The input is data taken from the drone, including battery and network status. The output is the information being written to a file.

3.2.3.1.c) GUI: At the top of the GUI the connection status of the drone and the car are displayed, this output value is taken from the drone and car after our programs have successfully allowed connection between the two. The start button’s input is the user pressing the button to initiate the experiment, and then the output is sending the command to execute the experiment/actually run the experiment. The live video stream takes input from the drone’s camera, and the output is the video frames to the GUI, which will include bounding boxes from the object detection if the object is present in the frames. The settings tab has multiple options for users to change certain settings on the drone, the user can just input information, and the output is the change to the settings (such as drone username and password). In the testing tab, the user can press the “run drone test” or “run car test” and input the port number to connect and test each separately.

3.2.3.d) Video Recording:The program begins by having the user input the path to save the recording. Then, the recording can begin and once a keyboard interrupt occurs the video can be saved to the path.

3.2.3.e) Flying the drone: The drone flight component has no interface itself but can be initiated by pressing the start button on our GUI. This will send a command to the drone to begin flight. The Aruco program is strictly for landing our drone safely, and uses the Aruco marker as a sign to be descending from flight. The hover flight involves the user more by having them input how long they want the drone to hover for.

3.2.3.f) Car/Drone Communication component: The input the client program takes is the server IP address and port number, and with that information, it is able to create a connection with the server. The server program gets a command sent from the client program, and this command allows the car (the client) to avoid obstacles.

#### **3.2.3.2 Algorithm model (e.g., PDL)**

The pseudocode listing for sub-component m is presented. We have written our pseudo code in PDL format, which uses English rather than a specific programming language to show the design of the program.

3.2.3.2.a) Object detection:

| Load in network architecture()  Run darknet on image(img,width, height)  Adjust image size  Convert to openCV color  Adjust ratio to get sizing of bounding box  Adjust height  Adjust width  Convert image to from JS to OpenCV img()  Convert bytes to array  While object detected is true  Convert to OpenCv image  Create overlay for box  For Above Confidence threshold  Left, top, right, bottom of bounding box  Left, top, right, bottom of bounding box  Label box with class //name of object detected  Show confidence of detection (a percentage) |
| --- |

3.2.3.2.b) Data collection:

| file.open()  While Drone Program is running  Get CurrentBatteryPercentage()  If (battery percentage has changed)  file.write(“filename.txt”, data)  Else  Continue  While Drone Program is running  Get CurrentFlightStatus()  If (flight status has changed)  file.write(“filename.txt”, data)  Else  Continue  While Drone Program is running  Get Activity Status()  If (drone activity status has changed)  file.write(“filename.txt”, data)  Else  Continue  file.close() |
| --- |

3.2.3.2.c) GUI:

| If Connect\_Pressed(hardware\_dest):  ssh.connect(hardware\_dest)  If Standby\_Pressed(hardware\_dest, args):  ssh.run(hardware\_dest, program, args)  If Restart\_Pressed(hardware\_dest):  ssh.run(hardware\_dest, restart)  If Start\_Pressed():  If run is true:  ssh.run(stop\_program, car)  ssh.run(stop\_program, drone)  log.stop()  If run is false:  ssh.run(start\_program, car)  ssh.run(start\_program, drone)  log.start |
| --- |
| If ip.stream == true:  Show ip.video  Else:  Show black |
| For item in new\_messages.pop:  Show item  old\_messages.append(item) |
| server\_cmd = ‘iperf3 -s ’  Client\_cmd = ‘iperf3 -c ’  If is\_server:  s\_cmd(server\_cmd)  Else:  c\_cmd(client\_cmd)  s\_cmd(cmd):  If port != null:  Cmd += ‘-p {port}’  If is\_logfile:  Cmd += ‘-l {logfile location}’  If is\_verbose:  Cmd += ‘-v’  ssh.run(cmd)    c\_cmd(cmd):  Cmd += ‘-t 30’  If port != null:  Cmd += ‘-p {port}’  If is\_logfile:  Cmd += ‘-l {logfile location}’  If is\_verbose:  Cmd += ‘-v’  ssh.run(cmd) |
| If ssh\_send(var):  ssh.run(var) |
| If new\_input(var):  If validate(var):  self.set(var)  else:  Show error\_msg |
| If drone\_status == connected and car\_status == connected:  Color = Green  Elif drone\_status == not connected or car\_status == not connected:  Color = Red  Else:  Color = orange |

3.2.3.2.d) Video Recording:

| ros\_command(command):  try:  Run command  exception:  Print “Error message”  main():  Get path input name from user  Make timestamp  Save file to path  try:  if \_\_name\_\_ ==main()  if there is a KeyboardInterrupt:  Print “Recording stopped by user” |
| --- |

3.2.3.2.e) Flying the drone:

| **Aruco marker oscillating program**:  Set timer (seconds):  Print "Waiting for {seconds} seconds..."  For each second:  rospy.sleep(1)  Print "{seconds-i} seconds remaining..."  Print "FLIGHT INITIATED!"  Initiate flight  Set Velocity()  Set attitude()  Get telemetry data = rospy.ServiceProxy('get\_telemetry', srv.GetTelemetry)  Navigate = rospy.ServiceProxy('navigate', srv.Navigate)  Navigate Global = rospy.ServiceProxy('navigate\_global', srv.NavigateGlobal)  Set Position = rospy.ServiceProxy('set\_position', srv.SetPosition)  Set velocity = rospy.ServiceProxy('set\_velocity', srv.SetVelocity)  Set attitude = rospy.ServiceProxy('set\_attitude', srv.SetAttitude)  Set rates = rospy.ServiceProxy('set\_rates', srv.SetRates)  land = rospy.ServiceProxy('land', Trigger)  # Get number of oscillations from user, and also hover time and offset  Input oscillation  Input hover time  Input aruco offset in type float  if oscillations <= 0 or hover\_time < 5 or x\_aruco\_offset < 0.5:  Print “number of oscillations <= 0, exiting program…”  Exiting…  Start timer for 10 seconds  Print 'Take off and hover 1 m above body frame'  Beginning flight  Sleep for 5 seconds  for i in range(oscillations):  Print 'Performing oscillation {i + 1}...'    print('Hover 1 m above aruco marker 113')  Hover at speed speed=0.2 and at 1m over aruco marker  Print ‘Hovering for {hover\_time} seconds...'  Sleep for 10 seconds  Print 'Hover 1 m above, {x\_aruco\_offset} m right (x offset) of aruco marker 113'  navigate(x=x\_aruco\_offset, y=0, z=1.5, speed=0.2, frame\_id='aruco\_113')  Print 'Hovering for {hover\_time} seconds...'  Sleep for 10 seconds  Print 'Performing Landing...'  navigate(x=0, y=0, z=0.2, speed=0.1, frame\_id='body')  Sleep for 10 seconds  Land drone() |
| --- |
| **Drone hovering program**:  Timer(Second inputted by user):  Print "Waiting for {seconds} seconds..."  for i in range(seconds):  Sleep for a second  Print “\_ seconds remaining...")  Print "FLIGHT INITIATED!"  Initiate flight()  Set Velocity()  Set attitude()  Get telemetry data = rospy.ServiceProxy('get\_telemetry', srv.GetTelemetry)  Navigate = rospy.ServiceProxy('navigate', srv.Navigate)  Navigate Global = rospy.ServiceProxy('navigate\_global', srv.NavigateGlobal)  Set Position = rospy.ServiceProxy('set\_position', srv.SetPosition)  Set velocity = rospy.ServiceProxy('set\_velocity', srv.SetVelocity)  Set attitude = rospy.ServiceProxy('set\_attitude', srv.SetAttitude)  Set rates = rospy.ServiceProxy('set\_rates', srv.SetRates)  land = rospy.ServiceProxy('land', Trigger)  User inputs hover time…  if hover time is less than 5:  Print “invalid hover time...exiting program…”  Exiting program  Count down time(35 sec)  #BEGIN FLIGHT:  Print “Take off and hover 1 m above the ground for {hover\_time} seconds…”  navigate(x=0, y=0, z=1, speed=0.2, frame\_id='body', auto\_arm=True)  Sleep for x seconds(hover time)  Print “drone is landing…”  Landing drone() |

3.2.3.2.f) Car/Drone Communication Program:

| **Socket program:**  Assign server IP = XXX.XXX.XXX.XXX  Assign server port = XXXX  Set Default speed = 0  If getWarning()  If car speed is not 0 and there is an obstruction  Slow down car by half speed  If obstruction to right  Car turn left  If obstruction to right  Car turn left  If unavoidable obstruction  Car stops  startCarDrivingAgain  Allow car to begin to drive after stop  main()  Connect to socket  Create TCP Connection    While connected  Receive command  If obstruction  Stop car  If car sends command = 1  Continue to drive  Else if receive warning  Adjust the speed  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  **Client program:**  Assign server IP = XXX.XXX.XXX.XXX  Assign server port = XXXX  Create the Client Socket Creation  close\_socket():  Close the socket  def message\_car(var):  Send socket message  Connect client and socket() |
| --- |

#### **3.2.3.3 Restrictions/limitations**

The external environment and/or infrastructure that must exist for sub-component m to

operate correctly is provided.

3.2.3.3.a) Object Detection: for our program to run properly we rely on the OpenCV library, darknet and YOLOv4 to be able to train our data set and run a detection on a live video. We also rely on Google Colab for a GPU for training.

3.2.3.3.b) Data collection: the drone needs to be properly running for us to log correct information into these files. So we must first be sure we are receiving correct information from the drone.

3.2.3.3.c) GUI: The program must be run on a Windows system that has the embedded python used to run all dependencies. There will be default properties, and thus all buttons assume correct properties have been input. If not, errors are produced and displayed.

3.2.1.d) Video Recording: For this component to be useful to our project we need a camera on the drone that can aid in object detection. Our project requires the drone to be flying in order to take bird-eye view pictures.

3.2.1.e) Flying the drone:A limitation for this program is that for it to work properly, we need the drone to be configured correctly.

3.2.1.f) Car/Drone Communication Program: The car and drone must be configured correctly. The car must be able to function as a server to communicate with the drone.

#### **3.2.3.4 Local data structures**

The data structures used within sub-component m are presented.

3.2.3.4.a) Object Detection: It takes an image from Javascript and converts it to an OpenCV image by converting bytes to an array.

3.2.3.4.b) Data collection: There is no real data structure present, its only purpose is to write to a file.

3.2.3.4.c) GUI: The GUI utilizes the txt format for log and output files. Temporary stdout data is stored in lists and all input will be stored in framework proprietary properties. All saved defaults will be stored in a .ini file.

3.2.3.4.d)Video Recording: No data structures necessary for this component.

3.2.3.4.e) Flying the drone: No data structures necessary for this component.

3.2.3.4.f) Car/Drone communication: No data structures necessary for this component.

#### **3.2.3.5 Performance issues**

Information on topics that may affect the run-time performance, security, or computational

accuracy of this sub-component are presented.

3.2.3.5.a) Object Detection: Accuracy of the object detected can be affected by closeness of the drone camera, camera quality, lighting and the environment around the object. The size and quality of our trained dataset also affects our object detection.

3.2.3.5.b) Data collection: The only issues that would cause this component to fail is if the program failed to open the file that we want to write to, or it is writing incorrect logs.

3.2.3.5.c) GUI: The run-time performance can be severely affected by the video stream lagging the GUI, this is mitigated by use of plugins to help lower load. Performance is also bounded by the strength of an SSH connection to the hardware from the GUI.

3.2.3.5.d)Video Recording: Lagging video streams can affect our object detection algorithm, which as a result affects our whole experiment.

3.2.3.5.e) Flying the drone: The hovering program requires the user to choose a hovering time over 5 seconds for the program to execute correctly. If bad data is inputted the program will exit and flight will not begin.

3.2.3.5.f) Car/Drone communication: If this connection fails at runtime, the rest of our experiment is affected and will not work properly. Connection must be guaranteed so our drone can follow the car to perform object detection.

#### **3.2.3.6 Design constraints**

Attributes of the overall software design (including data structures, OS features, I/O, and

interoperable systems) that constrain the design of this sub-component are presented

3.2.3.6.a) Object Detection: For training, our program only uses jpeg images. Our detection needs a clear video frame of the object we want to be detected for it to be successful.

3.2.3.6.b) Data collection: There is no design constraint since this program just writes to a file.

3.2.3.6.c) GUI: The GUI requires Windows. It is known to be compatible with Windows 10 and 11. It also requires Kivy 2.2 with a distributed python version.

3.2.3.6.d)Video Recording: There are no design constraints since this program is fairly simple and only used recording on the drone.

3.2.3.6.e) Flying the drone: The only constraints in this program is what the user inputs, if bad data is inputted the program will exit and flight will not begin.

3.2.3.6.f) Car/Drone communication: We are using python socket programming for the connections which limits the robustness of the connection

## 3**.3** Software interface Description

The software's interface(s) to the outside world are described.

The interface, or GUI, the user sees consists of the live video stream from the drone’s camera. A section for testing, settings, and simulation to run the experiment. The simulation tabs include connect, standby and restart buttons for both the drone and the car. The user can also alter the CV offloading. Battery life of the drone and the car are displayed in this tab. The settings can be used to change things like IP addresses, file/directory names, and other information as seen in section 4.1. The testing file allows the user to change the server from car to drone, or vice versa. It also has a section to connect to the drone, by entering the IP address, username and password.

### 3.3.1 External machine interfaces

Interfaces to other machines (computers or devices) are described.

In our experiment, we use a car and a drone, both having a RPi to run our programs on. There is no interface on the car or drone themselves, but rather they are all controlled by the single GUI we have made and described previously.

### 3.3.2 External system interfaces

interfaces to other systems, products, or networks are described.

Both the drone and car had to be configured before being able to run any of our programs on them successfully. We set up the car to act as a server for the drone to connect to and to allow communication between the two. To make changes to the server or other settings, the user can use the GUI we created.

### 3.3.3 Human interface

An overview of any human interfaces to be designed for the software is presented. See Section 4.0 for additional detail.

Our GUI was created using Kivy. It is a way for the user to see video footage from the drone, and see things like battery status of the drone, connection status of the drone and car, and alway the user of our testbed to initiate the experiment.

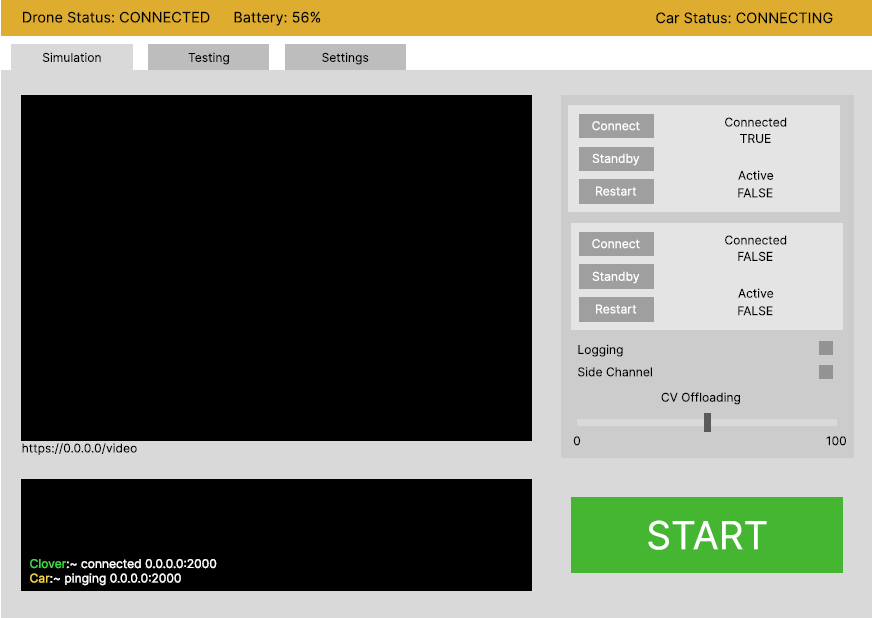
# 4.0 User Interface Design

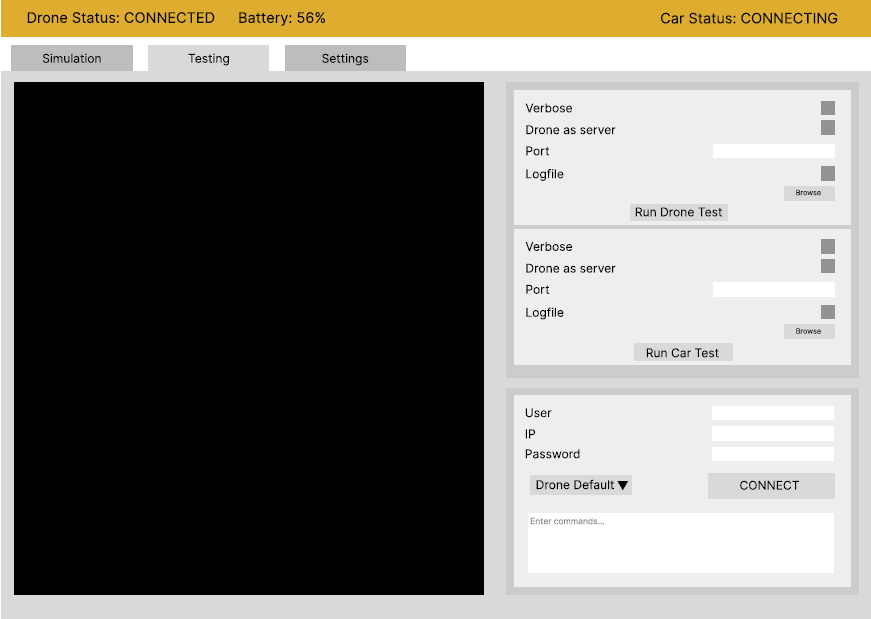
## **4.**1Description of the user interface

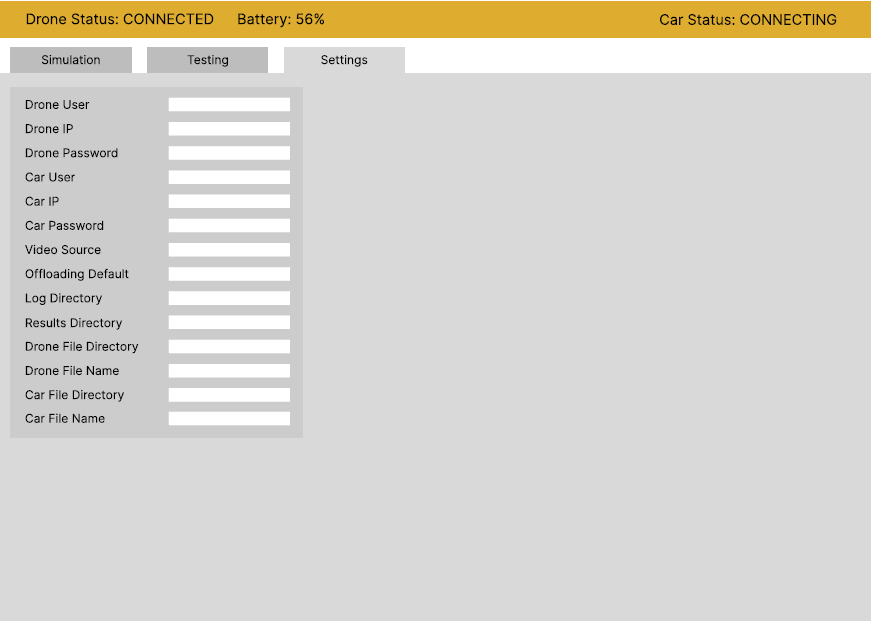
There are 3 main interfaces for this program. There is one for the general simulation testing use case, one for interfacing with the hardware, and one for assigning default settings for the interface and tests. The simulation testing interface is aimed at allowing the user to have easy control of the system with simple clicks of buttons, console output, and visible feedback to inform the user of the current functionality. The hardware interface has several forms to allow the user to directly ssh to the hardware and configure the systems as needed along with console output. The final settings interface provides a large form for user input to configure the connections, parameters, and overall user interface.

### 4.1.1 Screen images

The Simulation Testing Interface:

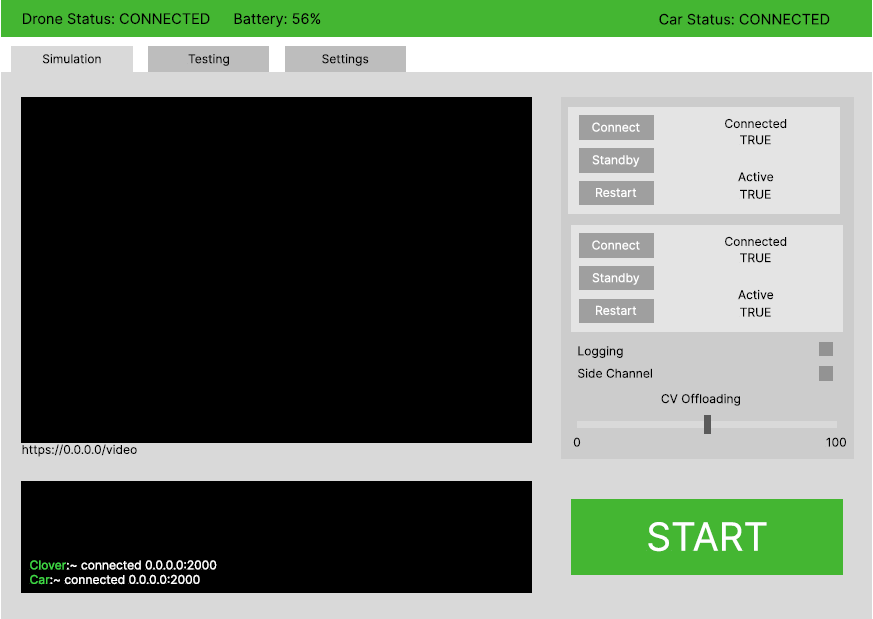


The Hardware Communication Interface:

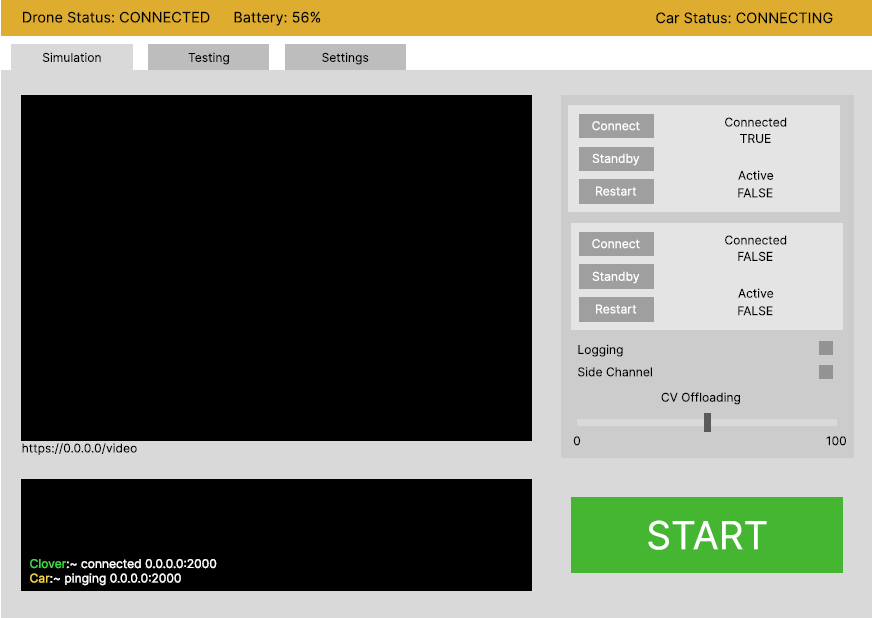
The Settings Interface:

Status Bar:

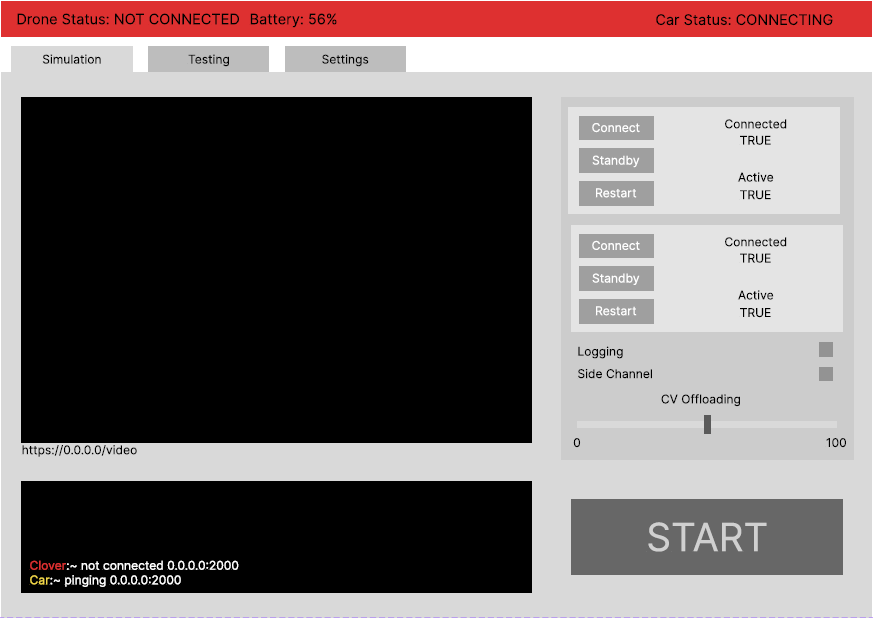
Connected:



Connecting:



Not Connected:



### 4.1.2 Objects and actions

Status Bar:

* Drone Status: Displays ‘CONNECTED’, ‘CONNECTING’, or ‘NOT CONNECTED’ depending on the status of an ssh connection.
* Battery: Displays a % or ‘NOT CONNECTED’ based on receiving data from the drone’s battery topic
* Car Status: Displays ‘CONNECTED’, ‘CONNECTING’, or ‘NOT CONNECTED’ depending on the status of an ssh connection.
* Color: The color of the status bar changes between red, yellow, and green depending on if everything is connected, connecting, or disconnected, always favoring the least connected option.

Simulation Tab:

* Video Viewer: The video screen displays whatever video is found at the link provided by the user setting for the drone’s video source.
* Video Link: Displays the source address used for the video feed
* Terminal Window: Display only window for any output from the GUI, drone connections, and car connections to allow the user to have feedback from the system
* Drone Connection
  + Connect Button: Establishes an SSH connection to the drone using parameters set in settings
  + Standby Button: Begins the drone program located on the drone at the given location from settings.
  + Restart Button: Sends a restart command ot the drone
* Car Connection
  + Connect Button: Establishes an SSH connection to the car using parameters set in settings
  + Standby Button: Begins the car program located on the car at the given location from settings.
  + Restart Button: Sends a restart command ot the car
* Logging Checkbox: Toggles the logging of the parameter data and testing results
* Side Channel: Toggles the use of an optional side channel flag for the program
* CV Offloading Slider: Determines the degree of computer vision offloading from the drone for the simulation
* Start Button: begins the test, turning gray while running. Clickable to cancel or end the test.

Hardware Configuration Tab:

* Terminal Window: Displays the output from the GUI, drone, and car terminals
* Drone IPerf3 Test:
  + Verbose Checkbox: determines the -v option
  + Drone as Server: Decides if the drone or GUI is the server for Iperf testing
  + Port Form: Determines the port used for the test, if left blank, a default is used
  + Logfile: Determines whether output should be just on the terminal or also to a logfile
  + Run Drone Test Button: Runs an IPerf3 bandwidth test on the drone using the given options
* Car IPerf3 Test
  + Verbose Checkbox: determines the -v option
  + Car as Server: Decides if the Car or GUI is the server for Iperf testing
  + Port Form: Determines the port used for the test, if left blank, a default is used
  + Logfile: Determines whether output should be just on the terminal or also to a logfile
  + Run Car Test Button: Runs an IPerf3 bandwidth test on the care using the given options
* SSH Session:
  + User Form: Username destination for SSH
  + IP Form: IP destination for SSH
  + Password Form: Password for destination host
  + Choice Dropdown: Allows user to decide between “Drone Default”, “Car Default”, and “Manual” login credentials
  + Connect Button: Establishes an SSH connection to the given IP
  + Input Form: Allows the user to run commands through the connection

Settings Tab:

* Drone User: Default username for a drone SSH connection
* Drone IP: Default IP for a drone SSH connection
* Drone Password: Default password for a drone SSH connection
* Car User: Default username for a car SSH connection
* Car IP: Default IP for a car SSH connection
* Car Password: Default password for a car SSH connection
* Video Source: IP/link used for video preview display
* Offloading Default: Default amount of CV offloading for a test
* Log Directory: Set the default location for the log files to be stored
* Results Directory: Set the default location for test results to be stored
* Drone File Directory: Default location for the drone’s test program to be stored
* Drone File Name: Default name of the drone’s test program
* Car File Directory: Default location for the car’s test program to be stored
* Car File Name: Default Name of the car’s test program

## 4.2 Interface design rules

Interface design is focused on making our product easier to use and to allow users to control the whole system from a single point of view. Our product is research based and thus a user interface is not required or high priority, but having a simple, flexible, and attractive interface will make our product much more appealing to researchers in the field. We utilized the following design rules when creating our user interface:

1. Easy to understand
2. Flexible
3. Give verbose feedback
4. Unfettered access to the hardware and test environment

## 4.3 Components available

The UI package used is Kivy 2.2. We utilize a variety of components from their default packages including, but not limited to:

* Button
* Boxlayout
* Dropdown
* Label
* Modalview
* Slider
* Stacklayout
* Tabbedpanel
* Textinput
* Videoplayer
* Widget
* Window

We are also utilizing the gstreamer dependency for our video, this is an extra addon that allows our program to function with real time video as expected.

For more information on these components and everything Kivy, see the Kivy Docs linked in section 7.

## 4.4 UIDS description

The UIDS used for Sky Socket is Kivy 2.2 for Python. It allows for fairly reactive programs and most standard components for modern UI. Some menus aren’t to our UI design goals, such as the file system menu. Because of this, we are using win32ui for file dialogs. Our UI is intended only for Windows and modern systems, so the limitations this provides do not impact our intended user base.

# 5**.0 Restrictions, Limitations, and Constraints**

**Time:**

We only have two semesters to get the software, hardware, and tests done from scratch with little to no experience with the subject matter beforehand. We have a lot of possible additions to the scope, but we may be limited by time. One of these additions is the addition of autonomous flight by the drone.

**Expertise:**

We are working with a lot of hardware and new interfaces/libraries we aren’t familiar with. We will have to use up time to learn how to use and work with these interfaces.

**Access:**

We have limited hardware (computing and vehicles) as well as limited access to them. We can work hands on with the drone and car when on campus and when we are given access to the testing facility, so we must be intentional about when we can be on campus and work with the hardware together.

# 

# 6**.0** Testing Issues

The tests we are running are focused on black box testing. They are aimed to test the functionality of the system with less emphasis on the performance. Performance is the goal of the project/research, so the testing does not require a certain level of performance above being functional.

## 6.1 Classes of tests

Unit Testing:

We will test each component of our system independently. We have tested each program independently without the rest of the system to ensure each part functions. This is a combination of white and black box testing for the given component. Naturally our ML is mostly black box, and our GUI + software is mostly white box testing.

Integration Testing:

We have a lot of pieces that have to work together and talk. After we’ve ensured the quality of our components, we slowly test combinations. Combinations of the program will be systematically cleared using majority white box testing.

Functional Testing:

Once all parts are together, we will be running many black box tests to ensure the product is acceptable and functioning in all required aspects.

## 6.2 Expected software response

GUI Expected Response:

Each component will be tested. All input and buttons are expected to do simple validation for viable input and responsive feedback. The tabs may have some rendering lag, but they are expected to be just as responsive as the buttons. The status bar is expected to update regularly and independently of the rest of the GUI. It should react to status updates and adjust its appearance accordingly without user intervention. The Video window should react to user input and update to show video output when detected. There may be some delay here, there is less we can do to improve this performance. The Command line feedback should print out all stdout and stderr output from the entire system. Identifiers for which system components should be given in tag form before the message to help with user comprehension and standard terminal formatting should be viewable. System feedback should be relatively responsive, any gaps or delays in communication should be mitigated via terminal feedback. Logging and test results should appear in the assigned folder as soon as the process related to the given log/file is finished. Any change to the save location should be observed by files created after the change but not by files made prior to the change. Errors in the GUI should print out in the terminal and not cause the program to crash. Should a critical error occur that prevents further execution, an error window should appear alerting the user.

System Expected Response:

The drone and car should react to programs being run quickly and as programmed. Any errors or issues should be reported to stderr/logs for the user to observe and debug. The car should drive according to given directions and stop when commanded. The program terminating should stop the car from driving. The drone should do any CV detection requested and offload the given amount requested within the bounds. It should also establish a connection with the car’s server. When the drone program stops, it should terminate its program and the connection.

Object Detection Expected Response:

We expect our object detection to be accurate and precise. Our goal is to get around 90% mean average precision (mAP). This should then be deployed in two versions, one for a server and one for the embedded system. They should perform at the performance bounds and output predictions in the expected manner.

## 6.3 Performance bounds

Special performance requirements are specified.

There are many aspects of the GUI that depend on the system performance and communication delays. However, GUI bound feedback should be done within 2 seconds of the action. This includes system print statements, button feedback, and launch. The system should respond with feedback within 2 seconds that a test is about to begin and the test simulation should begin within 10 seconds of that command being sent.

As mentioned above, our main goal was accuracy when it comes to object detection. The threshold for detection is 50% now. This means if confidence is 50%, our program will create a bounding box around the object we want to detect. Our detection works by analyzing each individual video frame, resulting in fast detection. Through our testing, we have found that performance can also improve depending on things like camera quality and implementing a larger data set with high quality jpeg images.

## 6.4 Identification of critical components

Those components that are critical and demand particular attention during testing are identified

Car:

The car runs software for receiving offloaded video as well as information from the drone’s vision system.

Drone:

The drone runs as a wifi router for the system so all communication goes through the drone. In addition, the drone will run a CV detection algorithm that triggers communication to the car.

Object Detection Algorithm:

We are training a custom object detection algorithm that will require extensive testing and training to ensure it is up to standard. It will also require preparation and testing for both a server and embedded implementation. Both will then require more testing.

User Interface:

The user interface must communicate with the drone and car as well as display video, car output, and drone output. All functionality must be tested for use and efficiency. The UI program also holds the logic for logging which requires separate and thorough testing.

# 7.0 Appendices

## 7.1 Requirements traceability matrix

| **ID** | **Use Case** | **Requirements** | **Priority** | **Depends on (ID)** |
| --- | --- | --- | --- | --- |
| **1** | **Run an Experiment** | * **Car driving** * **Drone video** * **Drone following car** | **HIGH** |  |
| **2** | **Retrieve and Analyze Data** | * **Develop API to collect data** * **Route the data to another API for analysis** | **HIGH** | **1,4** |
| **3** | **Change Car and Drone Parameters** | * **Develop API to change variable values** | **MEDIUM** | **1** |
| **4** | **Collect/track battery voltage of car and drone** | * **System voltage measurement interface** | **MEDIUM** | **1** |
| **5** | **Detect an Object using the system** | * **Develop an object detection system** | **HIGH** | **1** |

## 7.2 Packaging and installing issues

Special considerations for software packaging and installation are presented.

An issue we have run into is getting the object detection program able to run on the Clover drone image. The main reason is that the Clover drone image used in Ubuntu uses an older version of CMake (3.16), and our initial object detection program requires a new version (3.18 or above) to run darknet for object detection. For this reason we have decided to try and use TensorFlow and YOLOv4 so it is possible to run a program on the Pi, while having compatibility with all the other programs we plan to run. We are still working towards successfully finding a way to allow compatibility and implementing adequate computing power to run our detection program.

Due to our GUI requiring a video feed, the packages required for such infrastructure inhibits most packaging software from creating a functional EXE. To avoid this issue, we have created a custom embedded python solution that will allow our program to be distributable and runnable via a simple EXE call.

## 7.3 Design metrics to be used

A description of all design metrics to be used during the design activity is noted here.

## 7.4 Supplementary information (as required)

Kivy API Documentation: <https://kivy.org/doc/stable/api-index.html>

# 

## 7.5 **Hardware and Software Materials and Requirements**

* **GUI API: Kivy 2.2**
  + <https://kivy.readthedocs.io/en/master/guide/basic.html>
* **Raspberry Pi model 4B**
  + <https://www.raspberrypi.com/products/raspberry-pi-4-model-b/>
  + Used as the onboard computing device for both the car and the drone (each has their own Raspberry Pi computing device) for autonomous driving/flying by taking input from the sensors and processing the information, and also for communications (networking) with between devices (vehicles), such as wireless communications, i.e. WIFI.
* **Raspberry Pi Ai Car Kit (PiCar-X) for Intermediate**
  + <https://www.sunfounder.com/products/picar-x>
  + <https://docs.sunfounder.com/projects/picar-x/en/latest/introduction.html>
  + Used as the physical device for the car including motors, frame, driving mechanism, and sensors.
  + The Raspberry Pi OS imager should be used to image the sd card that will serve as the nonvolatile memory unit the Raspberry Pi computer of the car:
    - <https://www.raspberrypi.org/software/>
  + Here is the Repository that is cloned onto the Raspbian OS image of the Raspberry Pi; it will contain all of the Python installation files needed to program and control the car (Robot Hat Python Library); **Robot HAT** is the name of the software stack (a python library is essentially the interface) and the name of the circuit board attached to the raspberry pi to help manage sensors and the control of the car’s motors:
    - <https://github.com/sunfounder/robot-hat>
    - <https://docs.sunfounder.com/projects/picar-x/en/latest/about_robot_hat.html>
  + Here is the Repository containing user manuals for the PiCar X:
    - <https://github.com/sunfounder/picar-x>
* **Clover Drone 4.2** 
  + <https://clover.coex.tech/en/>
  + Used as the physical device for the drone including motors, frame, propellers, sensors, Electronic Speed Controllers (ESC), GPS, etc.
  + Includes Pixracer R15 Mini Pixracer Autopilot Xracer FMU V4 V1.0 PX4 Flight Controller
    - <https://docs.px4.io/main/en/flight_controller/pixracer.html>
  + PDF version of the documentation:
    - <https://clover.coex.tech/clover_en.pdf>
* **Python Programming Language**
  + <https://www.python.org/>
  + We will use the Python programming language for both the car and the drone.
* **OpenCV-Python Library**
  + <https://pypi.org/project/opencv-python/>
  + This is a Python vision analysis library that has been adapted from a library originally written for C++
  + We will use it to analyze vision data collected from cameras on both the car and the drone.
* **Raspbian OS builds with Linux Kernel**
  + <https://www.kernel.org/>
  + Both the car and the drone have their own onboard computing device (Raspberry Pi model 4B) with a custom modified version of the Raspbian Operating System image that uses the Linux Kernel.
  + Additionally, the Clover 4.2 Drone uses the ROS robotic framework used for advanced robotic distributed systems.
    - <https://wiki.ros.org/>
  + Here is the image used for the Clover 4.2 Drone Raspberry Pi computer:
    - <https://github.com/CopterExpress/clover/releases/tag/v0.23>
    - Image features:
      * Raspbian Buster
      * [ROS Noetic](http://wiki.ros.org/noetic)
      * Configured networking
      * OpenCV
      * [Mavros](http://wiki.ros.org/mavros)
      * Periphery drivers for ROS ([GPIO](https://clover.coex.tech/en/gpio.html), [LED strip](https://clover.coex.tech/en/leds.html), etc)
      * Aruco\_pose package for marker-assisted navigation
      * Clover package for autonomous drone control
* **Q Ground Control**
  + <https://docs.qgroundcontrol.com/master/en/>
  + This is an open source software used to communicate with and calibrate and configure a drone’s flight controller firmware. We will use this to calibrate the drone and manage the flight controller’s parameters and how the flight system of the drone uses and responds to sensor data.
  + Here is the firmware image used for our flight controller:
    - <https://github.com/CopterExpress/Firmware/releases/tag/v1.8.2-clover.13>
* **Clover Drone Simulation virtual machine (VM) image**
  + <https://github.com/CopterExpress/clover_vm>
  + <https://github.com/CopterExpress/clover_vm/releases/tag/v1.3>
  + <https://github.com/CopterExpress/clover_vm/releases>
    - Link to actual vm .ova vm image file
  + This is the virtual machine image (using Virtual Box) used to run the simulation software used to simulate programmed autonomous flights for the Clover 4.2 Drone.
  + Image contains:
    - Ubuntu 20.04 Focal.
    - ROS Noetic.
    - PX4 autopilot, QGroundControl.
    - Preinstalled [Clover](https://github.com/CopterExpress/clover) and Clover simulation packages.
    - Shortcuts for running Clover simulator.
    - VSCode.
    - Useful robotics-related software.
* **Drone Simulation Environment (Using Gazebo software)**
  + The simulation environment is based on the following components: [Gazebo](http://gazebosim.org/), a state-of-the-art robotics simulator;
    - <http://gazebosim.org/>
  + [PX4](https://px4.io/), specifically its SITL (software-in-the-loop) components;
    - <https://px4.io/>
  + [sitl\_gazebo](https://github.com/PX4/sitl_gazebo)  package containing Gazebo plugins for PX4;
    - <https://github.com/PX4/sitl_gazebo>
  + ROS packages and Gazebo plugins
  + **Note:** all of the above components are installed on the Clover Drone Simulation VM in order to do simulation programming without the Raspberry Pi on board computing device of the drone. This allows for programming and simulation without needing the physical drone present.
* **Etcher - Flashing Software**
  + <https://www.balena.io/etcher>
  + This is the software used to flash the micro-SD card with the respective OS (drone or car) used as the nonvolatile memory unit for the Raspberry Pi computers.
* **What is ROS (Robot Operating System)?**
  + <http://wiki.ros.org/>
  + *ROS (Robot Operating System) provides libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more. ROS is licensed under an open source, BSD license.*
  + [*http://wiki.ros.org/ROS/Technical%20Overview*](http://wiki.ros.org/ROS/Technical%20Overview)
    - *Link to technical overview of ROS Topics*
* **MAVROS (package)**
  + <http://wiki.ros.org/mavros>
  + MAVROS -- MAVLink extendable communication node for ROS with proxy for Ground Control Station.
  + This package provides a communication driver for various autopilots with the MAVLink communication protocol. Additionally it provides a UDP MAVLink bridge for ground control stations (e.g. QGroundControl).
* **MAVLINK (package)**
  + <http://wiki.ros.org/mavlink?distro=noetic>
  + MAVLink message marshaling library. This package provides C-headers and C++11 library for both 1.0 and 2.0 versions of protocol. For pymavlink use separate install via rosdep (python-pymavlink).
  + This package provides a communication library for various autopilot systems. This package contains both C-headers and pymavlink.
  + **mavros\_msgs/Mavlink Message**
    - <http://docs.ros.org/en/api/mavros_msgs/html/msg/Mavlink.html>
      * The above link is the header file to the mavrov\_msgs class which is used as a message transport class to convert between ROS and MAVLink message types.
      * Used to transport mavlink\_message\_t via ROS topic
    - **Here is the documentation for the mavros\_msgs class:** <http://docs.ros.org/en/api/mavros_msgs/html/index-msg.html>
    - mavros\_msgs defines messages for [MAVROS](http://wiki.ros.org/mavros)
    - And here is another important class **sensor\_msgs**: <http://wiki.ros.org/sensor_msgs?distro=noetic>
  + At least this package uses its own bundled (or installed by pip) mavlink headers or pymavlink: mavlink\_ros, rospilot, roscopter, autopilot\_bridge, px4-ros-pkg.
* **ros\_core**
  + Here is the documentation for **r**[**os\_core**](http://wiki.ros.org/action/fullsearch/ros_core?action=fullsearch&context=180&value=linkto%3A%22ros_core%22)class**:** <http://wiki.ros.org/ros_core?distro=noetic>
  + A metapackage to aggregate the packages required to use publish / subscribe, services, launch files, and other core ROS concepts.
* **libmavcon**
  + <http://wiki.ros.org/libmavconn?distro=noetic>
  + MAVLink communication library. This library provide unified connection handling classes and URL to connection object mapper. This library can be used in standalone programs.
* **catkin library**
  + <http://wiki.ros.org/catkin?distro=noetic>
  + <http://wiki.ros.org/catkin/conceptual_overview>
  + Low-level build system macros and infrastructure for ROS.
* **Working with ROS and MAVROS topics:**
  + <https://www.youtube.com/watch?v=ojboPhCwEtc&list=PLYLlj2SKN41WS49f-EJcvR0bWC2qIfcWw&index=11>
    - Go to around 52min mark in the video.
* WPA Supplicant Issue
  + Need to make sure WPA supplicant.config file contains the following for the *proto* variable:
    - proto=RSN WPA
  + Here is an example of how the file should look:



* Adding throttled image topic to subscribe to for obtaining and capturing live video feed to reduce processor resources needed to run our open cv python color detection program:
  + Add this to the m\_camera.launch file, found along path **/home/clover/catkin\_ws/src/clover/clover/launch:**



* ...

## 7.6 **Research Background Sources**

1. S. A. Hadiwardoyo, E. Hernández-Orallo, C. T. Calafate, J. -C. Cano and P. Manzoni, **"Evaluating UAV-to-Car Communications Performance: Testbed Experiments,"** 2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA), Krakow, Poland, 2018, pp. 86-92, doi: 10.1109/AINA.2018.00025.
   * **Abstract:** Vehicular networks are gradually emerging due to the expected benefits in terms of enhanced safety and infotainment services. However, outside main metropolitan areas, little infrastructure currently deployed, which may hinder these services. To mitigate this problem, Unmanned Aerial Vehicles (UAVs) are envisioned as mobile infrastructure elements, supporting communications when fixed infrastructure is missing. This way, in emergency situations, UAVs can offer services to vehicles including broadcasting alerts or acting as message relays between ground vehicles. Our work attempts to be a first step in this direction by presenting experimental measurement results regarding communications quality between cars and UAVs. In particular, we varied the altitude of the drone and its antenna orientation, and the car's antenna location to assess their impact on performance. Based on the experimental results achieved, we find that UAVs communicating in the 5 GHz band using IEEE 802.11 technology are able to deliver data to moving cars within a range of more than three kilometers, achieving more than 0.5 of packet delivery ratio up to 2.5 kilometers under the optimal configuration settings.
   * **URL:** <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8432227&isnumber=8432202>
2. J. Yoon, I. Kim, W. Chung and D. Kim, **"Fast and accurate car detection in drone-view,"** 2016 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia), Seoul, Korea (South), 2016, pp. 1-3, doi: 10.1109/ICCE-Asia.2016.7804775.
   * **Abstract:** With the development of drones, aerial images are used in a variety of applications. We propose a way to detect cars in drone-view fast and accurately. For this purpose we propose a feature called G-ORF for effective feature description. Also we designed a pose classifier and bin-specific weighted Linear Discriminant Analysis (wLDA) classifier for pose classification and binary classification of each pose respectively. Our method showed real-time performance in HD (1280×720) video in a PC environment with high detection accuracy.
   * **URL:** <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7804775&isnumber=7804716>
3. Yildiz, Melih, Burcu Bilgiç, Utku Kale, and Dániel Rohács. 2021. **"Experimental Investigation of Communication Performance of Drones Used for Autonomous Car Track Tests"** Sustainability 13, no. 10: 5602.
   * **Abstract:** Autonomous Vehicles (AVs) represent an emerging and disruptive technology that provides a great opportunity for future transport not only to have a positive social and environmental impact but also traffic safety. AV use in daily life has been extensively studied in the literature in various dimensions, however; it is time for AVs to go further which is another technological aspect of communication. Vehicle-to-Vehicle (V2V) technology is an emerging issue that is expected to be a mutual part of AVs and transportation safety in the near future. V2V is widely discussed by its deployment possibilities not only by means of communication, even to be used as an energy transfer medium. ZalaZONE Proving Ground is a 265-hectare high-tech test track for conventional, electric as well as connected, assisted, and automated vehicles. This paper investigates the use of drones for tracking the cars on the test track. The drones are planned to work as an uplink for the data collected by the onboard sensors of the car. The car is expected to communicate with the drone which is flying in coordination. For the communication 868 MHz is selected to be used between the car and the drone. The test is performed to simulate different flight altitudes of drones. The signal strength of the communication is analyzed, and a model is developed which can be used for the future planning of the test track applications.
   * **URL:**

<https://doi.org/10.3390/su13105602>

1. Barbeau, Michel, Joaquin Garcia-Alfaro, and Evangelos Kranakis. 2022. **"Research Trends in Collaborative Drones"** Sensors 22, no. 9: 3321.
   * **Abstract:** The last decade has seen an explosion of interest in drones—introducing new networking technologies, such as 5G wireless connectivity and cloud computing. The resulting advancements in communication capabilities are already expanding the ubiquitous role of drones as primary solution enablers, from search and rescue missions to information gathering and parcel delivery. Their numerous applications encompass all aspects of everyday life. Our focus is on networked and collaborative drones. The available research literature on this topic is vast. No single survey article could do justice to all critical issues. Our goal in this article is not to cover everything and include everybody but rather to offer a personal perspective on a few selected research topics that might lead to fruitful future investigations that could play an essential role in developing drone technologies. The topics we address include distributed computing with drones for the management of anonymity, countering threats posed by drones, target recognition, navigation under uncertainty, risk avoidance, and cellular technologies. Our approach is selective. Every topic includes an explanation of the problem, a discussion of a potential research methodology, and ideas for future research.
   * **URL:** <https://doi.org/10.3390/s22093321>
2. Textbook: **Software Engineering A PRACTITIONER ’ S APPROACH EIGHTH EDITION, by Roger S. Pressman, Ph.D. and Bruce R. Maxim, Ph.D.**
3. **Limitations of Camera and Radar Based ADAS**
   * <https://www.cepton.com/twitter/limitations-of-camera-and-radar-based-adas>
   * This is a link to a project that contains a threaded discussion about the failures of camera, lidar, and other sensors on Tesla vehicles as Tesla races towards making autonomous vehicles a reality. We can use this to help us keep track of some of the difficulties leading industries like Tesla faces in computer vision systems for autonomous vehicles.
4. This website gives a powerpoint and article that introduces ROS (Robot Operating System)
   * <https://sir.upc.edu/projects/rostutorials/1-ROS_basic_concepts/index.html#corecomponents-label>
   * <https://sir.upc.edu/projects/rostutorials/pdf-files/ROS_core_components_slides.pdf>
5. **RoBMEX: ROS-based modelling framework for end-users and experts**
   * <https://www.sciencedirect.com/science/article/pii/S1383762121000746>
   * This source was used to help understand the ROS and MAVROS frameworks
6. Lee, H.; Yoon, J.; Jang, M.-S.; Park, K.-J. **A Robot Operating System Framework for Secure UAV Communications**. Sensors 2021, 21, 1369.
   * <https://doi.org/10.3390/s21041369>