Navigating Autonomy in Aerial Robotics: Creating a robust and expandable framework for UAVs

Christos Georgiades

***Index Terms*—microservices, multidrone, collaborative systems, relative positioning, detection and tracking, embedded systems, autonomous system, unmanned aerial system (UAS), computer vision, image streaming, controller, joystick**

CONTENTS

I. INTRODUCTION

In recent years, Unmanned Aerial Vehicles (UAVs) have emerged as transformative tools across a multitude of industries, from agriculture and surveillance to search and rescue operations. The rapid advancement of UAV technology has been made possible due to their affordable price tag which has allowed multiple industries and hobbyists alike to find various ways to utilize drones' power, as well as the maturity of the software and systems developed empowering unmanned aircraft to perform tasks with unparalleled precision and efficiency.

This paper aims to provide a basis and an aide for individuals who want to build a autonomous and collaborative UAV system. It aims to provide a comprehensive description of the integrated components that constitute the drone's controller, concurrently addressing potential challenges and proposing solutions. These ideas are evaluated across two different operational scenarios. First, an aerial-photography task, where a drone or a squadron of drones cover a predefined geographical area. Second, a mission centered on Collaborative Relative Positioning. The scenario tested, describes a drone performing an orthophotography task, when it loses GPS signal and requests the help of the drones in its squadron to ensure a proper return to the base

Some key focuses of the project are to be fault-tolerant, expandable, provide remote access and diagnostics, and have minimal setup time.[maybe move to intro] Due to the nature of the project, a display or human interface devices (HID) would not be available during run-time, thus any dependence on such devices would always be kept minimal and only for debugging/development purposes.

Complementary to the aforementioned contributions within the scope of this project, implementations of packages have been made for the ROS ecosystem. The ros-database-synchronization is available on github and includes useful tools

such as database synchronization, and message serialization. Furthermore a rewrite of the Raft consensus algorithm has been implemented for ROS.

Overall, the contributions of this work are summarized as follows:

- A description and an implementation of a modular and replicable UAV.
- A system that is able to be easily expanded and duplicated that is able to handle simple or complex tasks.
- Raft Consensus imported to ROS.
- The proposed system has been experimentally implemented and thoroughly evaluated and validated in an outdoor environment following a cooperative system configuration.

[summarize contrib]

II. Related Work

There exist various works in academic literature introducing frameworks for UAVs that can accomplish a wide arrange of applications. Currently lots of work is being made in automating tasks that would previously require human involvement such as infrastructure inspection and search and rescue. A Microservice based solution working in the ROS ecosystem is AEROSTACK [1], designed for search and rescue operations using small to medium sized drones. Matlekovic et al. [2] use their own Microservice-based solution, built upon Kubernetes, that is used to perform area inspection.

A study proposed by Barazzetti et al. [3], uses imagery captured from a drone to produce orthophotography recreate 3D models of buildings. Similarly, Ngadiman et al. [4] create a flat orthographical image of an area using a similar approach. The procedures of obtaining the data for such endeavors used by the two studies, have been made fully-autonomous in the context of this project, being implemented as two state-based services working simultaneously.

[Collaborative and autonomous task execution in a neat little package]

III. Hardware Model

This section aims to describe in brief the hardware aspects, and the software stack powering them, of the proposed implementation. The software system described in this paper was originally developed for an NVIDIA Jetson Xavier NX onboard computer. The Xavier NX, running Ubuntu 18.04 on the 4.9.253-tegra kernel, served as the computational backbone of our system, enabling advanced vision-based tasks. The onboard computer is attached to a DJI Matrice300 RTK system. The UAV was equipped with the DJI Zenmuse H20T, DJI Zenmuse L1, or the Altum Micasense, which enabled advanced functionality such as a rangefinder, lidar or multispectrum photography.

A. On-Board Computer Integration

The integration of the onboard computer into the UAV system involved establishing a connection via a DJI OSDK expansion pack. In our setup, a USB-to-USB and a serial UART interface were used simultaneously in order to connect the expansion pack to the third-party computer. This setup seemed to produce the most reliable results and seemed to be most resistant to faults such as Handshake fails.

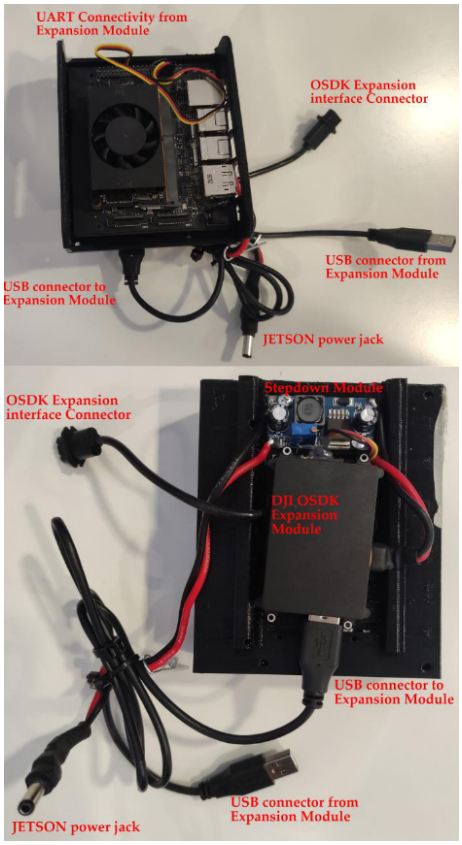

Fig. 1: Jetson Xavier NX, mounting case and connectors

As shown in fig. 1, the on-board computer is placed in a 3D-printed case. On the underside of the case the components needed for interfacing between the UAV and the computer are attached. The Matrice 300 RTK OSDK Expansion Module converts interfaces on the M300 RTK OSDK to standard interfaces. The USB and UART connectors are attached directly to the Nvidia Jetson. The XT30 power output allows 3.3V to be drawn from the drone's battery, which is passed through a stepdown module to a barrel jack, powering the Jetson Xavier and ensuring the on-board computer's power supply remains within safe limits throughout the duration of the flight. Furthermore, wiring the power jack in parallel with a XT30 connector, the UAV is able to provide power simultaneously the on-board computer and any third-party peripherals attached to the UAV system.

The described hardware configuration creates a complete and easily swappable solution for the project's computing needs. On the top side of the UAV, as can be seen in fig. 2, mounting rails are attached which hold the computer's case. Once in position, two pins can be used to lock the case and securely hold everything together. The mounting bracket serves not only as a housing solution for the onboard computer but

Fig. 2: Complete drone solution

also as a versatile platform for expansion modules, featuring a 7x7cm mounting grid where peripherals can be secured to via pins.

B. Networking and Connectivity

A connection scheme is needed that could facilitate both in-field development and debugging as well as the high level command and control of multiple UAV systems. UAVs are connected over cellular network, as shown in fig. 2, and communicate over VPN network, establishing a protected communication channel where the connected drones and any other ROS-enabled devices could interact via the ROS network. The integrated Wi-Fi module on the onboard computer acts as a debug entry point during the development phase acting as a Wi-Fi hotspot, facilitating real-time debugging and monitoring. A systemctl service is responsible for enabling hotspot connectivity at startup, if the osdk stack is enabled to launch or if there are no attached displays to the on-board computer. [network connectivity diagram]

IV. System Model

The collection of software developed for the purposes of this project consists of multiple scripts, acting as micro-services, working in conjunction and communicating via ROS network to achieve operation and automation of the drone's hardware. This approach is fault-tolerant by design, as an error or fault in one of modules of the system will not render the system as a whole inoperable.

The software and networking stack employed in this project is outlined as follows: ROS Melodic - 1.14.13

- DJI Onboard SDK-ROS - 4.1.0
- Ubuntu 18.04.6 LTS aarch64
- Jetpack 4.6
- Kernel: 4.9.253-tegra
- Python3 - 3.6.9
- Python2 - 2.7.17

GPU Libraries

- TensorRT - 8.2.1
- cuDNN - 8.2.1
- CUDA - 10.2
- OpenCV - 4.7.0-dev

A. System Initialization

The main entry point of the system is a shell script responsible for setting up the execution environment and launching the packaged microservices.

The script launches the project's modules in the following order:

1) roscore
2) ros master_discovery_fkie
3) ros master_sync_fkie
4) ros rosbridge_server
5) DJI Onboard-SDK-ROS
6) Connection to platform handler
7) Raft consensus client
8) Metric and Telemetry publishing
9) Logging and databasing
10) Drone Movement
11) Drone camera and gimbal control
12) Multispectral control
13) Peripheral component handler
14) CRPS Measurement Fusion and image detector

[better as diagram]

The project provides a systemctl service that launches the codebase during system initialization. An installer for the autostart service was also created, which modified the service file to work with different system with minimal human interference.

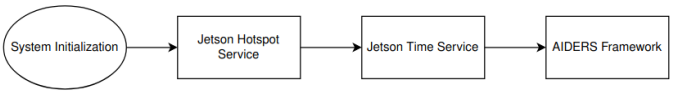

Fig. 3: Systemctl Service hierarchy

The service will setup a hotspot using the Jetson's Wifi module for debugging access, await for 3G and VPN access and then finally call AutostartMultimasterOSDK.sh which will launch the project. An important aspect of this methodology is that the onboard computer mounted on the drone remains accessible even if a fault occurs during the execution.

[Merge into one]

B. Topic onomatology

Another design-choice that allows for seamless scalability is a dynamic naming convention. When combined with ROS's topic-based communication, this strategy aligns seamlessly with the project's objectives. By applying this approach, Topics that belong to the same UAV system can be structured with an initial prefix, conveying the source or the ownership of a particular topic. Within the framework of project, the source of individuality used was the on-board computer's machine-id. The four initial characters of this id were concatenated to a

string denoting the model of UAV system to create a unique identifier used within ROS.

In Linux systems, this distinct machine ID is stored in the /etc/machine-id file and is configured during the initial installation. Another upside of using this identifier is that it can be easily regenerated which comes particularly useful when duplicating OS images.

This regeneration process can be initiated as follows:

```
rm -f /etc/machine-id
dbus-uuidgen --ensure=/etc/machine-id
```

Furthermore, topics shared among different drones, where instance-specific information varies, are identified by a common suffix, indicating that the format of the packets transmitted over this topic are related to a specific concept yet the content distinctly describes the state of an individual system. Building upon this concept, an infix, an element inserted within a word or string, can be used to group topics that are used to interact with an individual microservice.

To help explain the concepts further here are some specific examples using 2 drones with machine ids of aaaa and bbbb. These drones both publish GPS data in a topic called Telemetry and have a microservice that controls a weather sensor. Following the guidelines formerly stated would result in the following topic list:

```
/matrice300_aaaa/Telemetry
/matrice300_aaaa/WeatherSensor/Get_State
/matrice300_aaaa/WeatherSensor/Set_State
/matrice300_aaaa/WeatherSensor/Data

/matrice300_bbbb/Telemetry
/matrice300_bbbb/WeatherSensor/Get_State
/matrice300_bbbb/WeatherSensor/Set_State
/matrice300_bbbb/WeatherSensor/Data
```

C. Module Hierarchy - Late Initialization

Module hierarchy was an important problem that had arisen during development and the use of ROS as a communication medium applied certain restrictions to the required design. By convention, code that contains ROS callbacks must call spin, which is a provided implementation of an event processing sleep loop, furthermore only a single node is allowed per software package. Due to the problems formerly stated module hierarchy had to account for the sequence of initialization and the use of a common node held by the top-level module.

The created solution is based on the concept of late initialization, breaking down the process into distinct parts and allowing the imported scripts to inherit their parent's ROS node. Every instanced script at startup time will call setup_listener(). Modules will only setup some identifying variables during import, while top-level modules are responsible for calling their own init() function, which by convention calls init() calls to its imported ros-enabled modules. This causes an init() cascade which continues recursively until all modules are initialized. When that is done the top-level module is responsible for calling spin thus keeping itself and all its child modules alive.

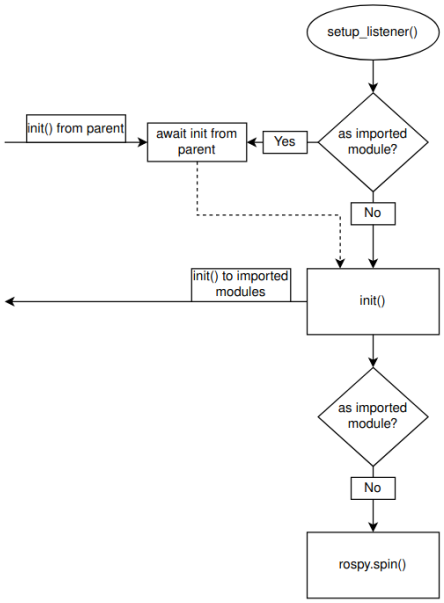

Fig. 4: Late Initialization flow diagram

Solving this problem allowed created modules to be imported and used by other scripts and permitted the creation of "common" classes, which were scripts containing commonly used callbacks and accessors pertaining to a certain subject. Eliminating the need for duplicate code and packaging important functions into an easy to use and complete script. In addition, modules could be used both as a standalone script and as a imported module, which aided the debugging process, allowed for independent functionality and simultaneous access by the parent module.

[Maybe a diagram of module hierarchy]

D. Microservice Design

Microservice based design offers a range of advantages over a monolithic design as it is easily scalable, modular, and flexible. It is easy to develop independently deployable applications and integrate code developed in isolation. Microservices can be modified or replaced without major disruptions to the entire application. This adaptability is valuable in the dynamic and evolving field of UAV technology. Microservices can isolate failures within individual services, preventing a failure in one component from affecting the entire application a crucial aspect for maintaining system stability and safety. Despite that, there is no perfect design and developers should consider both the upsides and the downsides of each design while having its final application in mind. For example, using Microservices it can be difficult to capture the state of a running system.

E. Multidrone combatibility

An integral focus of this project lies in the principles of scalability and inter-drone communication. each with its own authority and control, to manage and coordinate various

aspects of a complex system or network. In such an approach, these master nodes can distribute the workload, handle communication, and make decisions autonomously, enhancing system reliability, fault tolerance, and scalability. In the context of inter-drone communication and coordination, a multimaster approach might involve multiple drones having localized control over certain tasks, with the ability to communicate and synchronize their actions as a collective group. This approach is particularly relevant in swarm robotics, where multiple robots or drones work together on a common objective while maintaining individual decision-making capabilities.

The microservice based design facilitates this kind of development, as procedures can be developed in isolation, ensuring their correct operation and scalable nature, and then integrated into the codebase. This approach ensures that the system's architecture and protocols can effortlessly scale up to support a broader squadron of drones, maintaining efficiency and minimizing the need for extensive code modifications.

[kinda boring]

V. Backend

A. ROS core

Robot Operating System (ROS) is an open-source middleware framework designed for robotic system development. Introduced in 2007 by Quigley, Gerkey, and Ng, ROS provides a modular and flexible architecture for programming robots. The framework, describes a system of named buses called "topics" and "services" as well as offers a variety of other features.

Applications connected to the ROS network can setup subscribers and publishers which receive and publish messages respectively. The ROS wiki is a great tool for an introduction to ROS and its ecosystem. In the confines of this project it is used as a messaging system between discrete and connected microservices functioning as nodes. [Reduce]

The first four modules stated above[add pointer to list] form the basis of our ROS environment. The roscore module serves as the central hub for the Robot Operating System (ROS). It provides essential services, enabling communication and coordination among different ROS nodes and components. The master_discovery_fkie module is responsible for dynamically discovering ROS Masters in a network. The master_sync_fkie module provides synchronization functionality for ROS Masters. It ensures that multiple ROS Masters operating in a distributed system have consistent information about the state of the ROS network. The rosbridge server module acts as a middleware that facilitates communication between ROS and non-ROS systems. It translates messages between the native ROS format and other communication protocols, enabling interoperability with external applications. [maybe move to system model]

B. DJI Onboard-SDK-ROS

The DJI Onboard-SDK is a software development kit designed to integrate DJI's UAVs with ROS. The OSDK is responsible for handling the communication between the Onboard computer and the UAV, acting as an interface between DJI's flight controller and the ROS ecosystem. This codebase, written in C++, can be considered as the backend of the project's software stack. It provides access to a range of functionalities and sensor data provided by the UAV's flight control computer. By convention, low-level operations, such as navigation, actuator control and sensor data acquisition are written within this backend layer as it has direct access to the functionality the OSDK provides. These operations or data are then exposed to the frontend, where they can be accessed from either directly, indirectly via a wrapper script written in Python3.

DJI provides two versions of their Onboard Software Developement Kit (OSDK), a ROS enabled version and a standard one. The proposed approach uses OSDK-ROS which is compatible with ROS but lacks some features that are available in the non-ROS release. This was ameliorated by importing and rewriting code from the OSDK, into the OSDK-ROS.

[include layout of services and topics] [move to backend]

C. Drone camera and gimbal control

The OSDK-ROS provides interfaces and functions for the control of the camera and the gimbal out of the box, but depending on the model of drone you are using some changes might be needed to the original OSDK source code. Control and capture of the image stream of specific integrated cameras required augmentations to the provided C++ code that communicates between the drone's internal computer and the ros-network.

The cameras that are used by the Matrice300 drone can be separated into three conceptual categories. These categories are the "Main" cameras which use the SKYPORT interface in order to attach to the drone, the First Person View, or FPV, camera and the Stereo cameras which are integrated in the UAV.

1) Main Camera: The DJI-OSDK is already setup with a callback that handles the image stream from the camera without need for further changes. Nevertheless the application flow is analyzed in this section in order to provide a base example and highlight the differences between the main camera and the other cameras available from the drone.

The function startMainCameraStream included in the OSDK is used to setup a callback that handles the Image Stream from the "Main" camera.

```
response.result = startMainCameraStream(&publishMa
```

The function publishMainCameraImage included in dji_vehicle_node_publisher.cpp is used to package the image frame from the drone into a sensor_msgs/Image packet, which is then published using the an Image publisher which is already defined in the provided codebase.

```
main_camera_stream_publisher_.publish(img);
```

Finally, other applications can setup a subscriber using the above topic in order to utilize the camera stream as such:

```
rospy.Subscriber(dronename+'/main_camera_images',
```

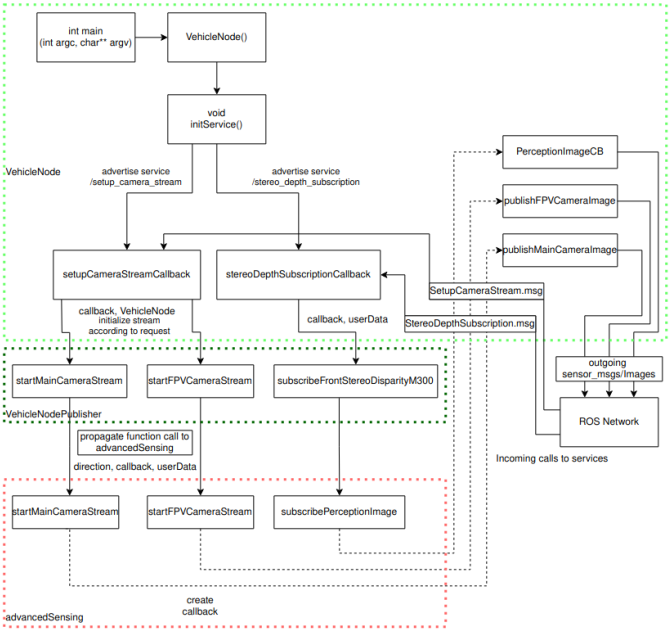

Fig. 5: Camera Services: Advertisement and setup

Where camera_stream_CB is a local definition or function

The first two categories can be enabled via ROS through a request to setupCameraStreamCallback defined in dji_vehicle_node.cpp For the stereo cameras some changes are needed to be made which will be outlined in the following subsections.

The service message SetupCameraStream.srv provided in the OSDK is as such:

```
#constant for vga image frequency
uint8 FPV_CAM  = 0
uint8 MAIN_CAM = 1

# use above constants to config freq.
uint8 cameraType

# 1 for start camera stream, 0 for stop
uint8 start

---
bool result
```

2) FPV Camera: The operation of the FPV camera is similar to the main camera attached to the UAV. A call needs to be made to the dronename+"/setup_camera_stream" service with the appropriate arguments. The service will then setup a callback function to publishFPVCameraImage defined in dji_vehicle_node.cpp as such:

```
response.result = startFPVCameraStream(&publishFPVCameraImage, this);
```

The function publishFPVCameraImage is then responsible for packaging and publishing the images to the ros network.

3) Stereo Camera: While the SDK includes methods for obtaining video from the integrated stereo cameras, some changes were needed in the underlying code to make it it work.

Initially, the service used to enable the Stereo cameras is different than the two other cameras. A call is needed to be made to the topic dronename+"/stereo_depth_subscription" which is calls stereoDepthSubscriptionCallback defined in dji_vehicle_node.cpp. Subsequently the following code needs to be added in stereoDepthSubscriptionCallback.

```
if (request.front_depth_240p == 1)
{
    this->stereo_subscription_success = false;

    if (isM300()) {
        // Create a userData struct
        VehicleStereoImagePacketType *userData = ne

        StereoImagePacketType *pack = new StereoIma
        pack->info = { 0 };
        pack->imageRawBuffer = NULL;
        pack->mutex = NULL;
        pack->gotData = false;

        OsdkOsal_MutexCreate(&pack->mutex);

        userData->stereoImagePacketType = pack;
        userData->vh_node = this;

        ptr_wrapper_->subscribeFrontStereoDisparity
    }
    else {
      ptr_wrapper_->subscribeFrontStereoDisparity(
    }
}
else
{
    ROS_WARN("no depth image is subscribed");
    return true;
}
```

With the code above we need to add a new function subscribeFrontStereoDisparityM300 in vehicle_wrapper.cpp.

```
void VehicleWrapper::subscribeFrontStereoDisparityM
{
    if (isM300()) {
        Perception::DirectionType direction = Perce

        vehicle->advancedSensing->subscribePercept
    }
}
```

Finally, we need to declare the PerceptionImageCB function responsible for publishing incoming stereo images.

breakatwhitespace

```
void VehicleNode::PerceptionImageCB(Perception::ImageInfoType info, uint8_t * imageRawBuffer,
                    int bufferLen, void * userData) {
    DSTATUS("image info: dataId(%d) seq(%d) timestamp(%llu) datatype(%d) index(%d) h(%d) w(%d) dir(%d) "
            "bpp(%d) bufferlen(%d)", info.dataId, info.sequence, info.timeStamp, info.dataType,
            info.rawInfo.index, info.rawInfo.height, info.rawInfo.width, info.rawInfo.direction,
            info.rawInfo.bpp, bufferLen);

    // Add camera Identifier

    StereoImagePacketType *pack;
    VehicleNode *node_ptr;

    if (userData) {
        VehicleStereoImagePacketType* vehImageType = (VehicleStereoImagePacketType*)userData;

        pack = vehImageType->stereoImagePacketType;
        node_ptr = vehImageType->vh_node;
        std::cout << "Got VehicleNode and pack\n";
    }

    if (imageRawBuffer && userData) {
        std::cout << "imageRawBuffer or userData exist\n";
        std::cout << "setting StereoImagePacketType *pack\n";
        std::cout << "calling OsdkOsal_MutexLock\n";

        OsdkOsal_MutexLock(pack->mutex);
        std::cout << "setting pack->info\n";
        pack->info = info;

        if (pack->imageRawBuffer) {
            std::cout << "calling OsdkOsal_Free\n";
            OsdkOsal_Free(pack->imageRawBuffer);
        }

        std::cout << "setting imageRawBuffer\n";
        pack->imageRawBuffer = (uint8_t *) OsdkOsal_Malloc(bufferLen);
        //memcpy(pack->imageRawBuffer, imageRawBuffer, bufferLen);

        std::cout << "creating image packet\n";
        sensor_msgs::Image img;
        img.height = info.rawInfo.height;
        img.width = info.rawInfo.width;
        img.data.resize(img.height*img.width);
        img.encoding = "mono8";
        img.step = img.width;
        uint8_t img_idx = 0;

        if (info.dataType == 1) {
            img.header.frame_id = "Left"; // recvFrame.recvData.stereoImgData->img_vec[img_idx].name;
        } else {
            img.header.frame_id = "Right";
        }

        img.header.seq = info.sequence;// recvFrame.recvData.stereoImgData->frame_index;
        img.header.stamp = ros::Time::now(); // @todo

        std::cout << "copying to packet\n";
        std::cout << "Img Size:" << std::to_string(info.rawInfo.width) << "x" << std::to_string(info.rawInfo.height) << "\n";
        std::cout << "bufferLen:" << std::to_string(bufferLen) << "\n";
        memcpy(&img.data[0], imageRawBuffer, bufferLen);
        //memcpy((char*)(&img.data[0]), imageRawBuffer, bufferLen);

        std::cout << "publishing....\n";
        node_ptr->stereo_depth_publisher_.publish(img);

        std::cout << "setting pack->gotData\n";
        pack->gotData = true;
        std::cout << "calling OsdkOsal_MutexUnlock\n";
        OsdkOsal_MutexUnlock(pack->mutex);
        std::cout << "calling OsdkOsal_MutexUnlock DONE\n";



    } else {
        std::cout << "imageRawBuffer or userData missing\n";
    }
}
```

how to enable Stereo Camera blah blah....insert stereo_camera_diagram

D. Rangefinder Handler

While the modular H20T camera is able to estimate and provide rangefinder data such as beam position and distance,

obtaining such values via OSDK-ROS is not as straight forward as it should be. Consequently, a workaround was devised, involving the utilization of an MSDK-enabled application operating on the DJI Smart Controller. This application accesses the required data, facilitated by the concurrent operation of a handler script on the UAV.

When necessitated, the handler establishes communication with the Android application, initiating the process of collecting and transmitting rangefinder data. The handler publishes afterwards the data from the rangefinder into a common topic making them available for the whole python3 layer.

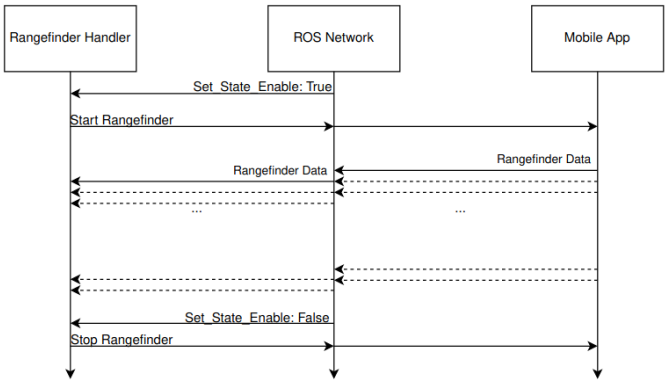

Fig. 6: Rangefinder data acquisition protocol

VI. Frontend

While the distinction is made between the C++ backend and the Python3 frontend, it does not mean the frontend is necessarily dumb. On the contrary most of the complex logic is contained within this layers due to its ease of use and wide access to libraries and plugins.

Since the onboard computers used for this paper are running Ubuntu 18.04 which supports up to ROS Melodic, some extra steps are needed in order to compile the codebase with proper Python3 support, taking advantage of the numerous enhancements and more actively supported plugin library. The alias below compiles the required packages for ROS-enabled application with support for Python3:

```
cat /sys/class/power_supply/BAT0/uevent blah blah...
```

[find alias]

Scripts in the front-end follow a common design philosophy. Immediately when a script is initialized it decides if it is run as an independent script or as an imported module. In python3 code this is done as such:

```
if __name__ == '__main__':
    as_module = False
else:
    as_module = True
```

The program then calls the listener method which setups the node's internal names, this process is discussed more in the Multimaster and Onomatology sections. If the script is running as an imported module it will await initialization from the parent, otherwise, if the script is running independently, it initializes a ROS node and calls the init() function. This function is responsible for setting up the subscribers and publishers the script uses as well as preparing any global variables, finally it calls the init() function of any imported scripts. This initialization hierarchy scheme allows ros programs to be importable, as otherwise rospy nodes in imported modules would cause conflict. Furthermore, taking advantage of this late initialization imported modules can take advantage of the parents ros-node and perform actions and exchange information on its behalf. Finally, any parent script should call rospy.spin() in order to infinitely wait and keep alive its set of listeners.

maybe a section about how the modules are designed and communicate

maybe a description of listener init import as module general coding principles

A. Metric collection

This module is responsible for collecting and publishing non-vital metrics. These metrics can be consumed by various applications or stored in a database for later processing.

Metric collection is either done via the on-board computer's sysfs or using pre-defined service calls to the UAV's internal computer. Sysfs is a virtual filesystem in Linux that provides a structured and intuitive way to interact with and retrieve information about devices, drivers, and kernel parameters.

Here are some examples that showcase the simplicity of obtaining or setting hardware attributes; You can use sysfs to check battery status on a laptop.

```
cat /sys/class/power_supply/BAT0/uevent
```

Here is another example that shows how to set an attribute;

```
echo performance > /sys/devices/system/cpu/cpu0/cp
```

This module is responsible for collecting and publishing non-vital metrics. These metrics can be consumed by various applications or stored in a database for later processing.

Metric collection is either done via the on-board computer's sysfs or using pre-defined service calls to the UAV's internal computer. Sysfs is a virtual filesystem in Linux that provides a structured and intuitive way to interact with and retrieve information about devices, drivers, and kernel parameters.

[include layout]

1) Database Implementation: Each on-board computer implements a microservice responsible for collecting, serializing and finally storing ROS packets into an SQL database. The developed algorithm utilizes worker threads, responsible for capturing published packets from a list of user-specified topics. When a packet is captured, it is serialized recursively into a list of primitive values, the array of values is stored in a buffer until the worker is ready to save the collected data into the database instance. This serialization algorithm from ROS packets to SQL seems to be a new contribution to the ROS codebase.

Scheduling access to the database for the running workers is done via a round-robin algorithm with mutual exclusion. The worker will queue if it has a non-empty buffer, until its their turn to obtain the mutex. When they can, they will save the buffer's content into the database and then release the mutex, letting the next worker inline.

2) Database Synchronization: Separate instances of the database micoservices can communicate and synchronize as a means of fault tolerance and disaster recovery. The protocol at use consists of two mechanisms, data propagation, which is done at packet arrival at the source drone, and data completion which is performed at the time of propagation by the receiving drones.

Data propagation is activated when a drone receives packets from a topic that is synchronized between database instances. The microservice will republish the incoming packet to topics sharing the same name but belonging to the different drones. Once that propagated packet is received by the synchronizing services it will trigger a data completion check.

Data completion is verified by the use of seq numbers. If a gap or multiple gaps are found in the sequence numbers, then a database packet request is created, specifying the sequency of packets which are missing, and published to the ROS network. Database instances can respond to that request by retrieving the entries from their database, deserializing them back into a ros packet and transmitted to the requester.

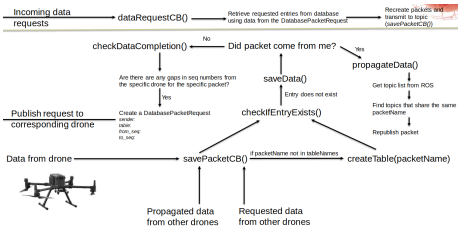

Fig. 7: Database Synchronization algorithm

B. Multi-channel Telemetry

Telemetry, refers to the data describing the current positional and rotational state of the UAV. The Telemetry system is designed with multiple or inconsistent sources of positional data in mind, such as the well-known GPS, RTK and CRPS.

In order to achieve at-will access the logic pertaining to the particular task is split into two core components. First, a Positioning System Prioritiser (PSP) is used to order and evaluate the confidence of each positioning system. This is done by connecting custom listeners, called TelemetrySource, to each positional topic. The listener can be instructed to inspect incoming Telemetry packets and evaluate them on programmed heuristics, such as the number of satellites and packet frequency. The parameters on which a source ranks the incoming data stream can be changed depending on the nature of the source. e.g. IMU based positioning systems do not take advantage of satellites. When a request is made for the latest Telemetry, the listeners are iterated through until a valid entry is encountered, which is retrieved and returned to the caller.

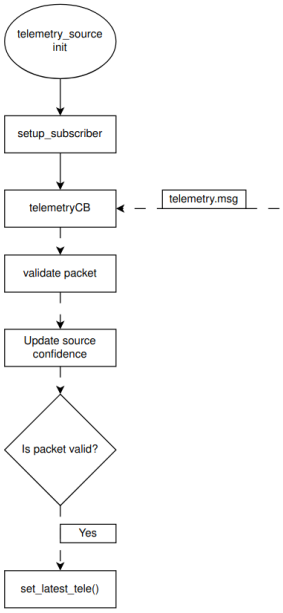

Fig. 8: Telemetry Source initialization and validation logic

The TelemetryTranslator serves as a single point of truth for the frontend layer for Telemetry data. At a specified frequency, namely 50hz, a worker is instructed to retrieve positional data from the PSP system and enhance them with sensor data from various components of the UAV. The resulting Telemetry message is published to the topic "../Telemetry" and serves as a quick snapshot of the drones state. The structure of a Telemetry message is as such;

```
uint32 rostime_secs
uint32 rostime_nsecs
uint32 flightTimeSecs
uint32 batteryThreshold
uint32 batteryPercentage
uint32 gpsSignal
uint32 satelliteNumber
float32 altitude
float32 heading
float64 velocity
float64 longitude
float64 latitude
float64 homeLatitude
float64 homeLongitude
string droneState
float32 gimbalAngle
string serialVersionUID
```

C. Drone Navigation

In the realm of UAVs, the ability to precisely control and manipulate the movement of these aerial platforms has emerged as a paramount challenge and opportunity. This section of the paper delves into the navigational usage of the DJI Matrice300.

1) Control Authority: This section about obtaining Control Authority is specific to the DJI ecosystem but products from other commercial manufacturers might contain similar systems, or it can inspire home-brew systems if such a protocol is desired. Movement for the OSDK involves a system of control authority that DJI has programmed into their Ecosystem. Control Authority in this context can be conceptualized as a flag or token that the internal computer of the UAV can pass to the devices which are in active communication. If a device controls this token then it can issue flight commands or missions to the drone's integrated MissionManager. Control devices have a prebuilt hierarchy system. For example, the drone's official controller, the DJI Smart Controller starts off with this token, then MSDK apps, and finally OSDK applications. The controllers high hierarchy comes useful when regaining control of the vehicle.

The provided navigation code by DJI was deemed too unreliable, requiring repeated attempts in order to accept and complete a mission, thus the decision to remake the navigation and mission/task systems.

The aircraft's flight controller arbitrates which source is in control at any one time. The different sources of control, and their control priority from highest to lowest is:

1) Remote Controller
2) Mobile Device (with a DJI Mobile SDK based application, connected to the Remote Controller)
3) Onboard Computer (with a DJI Onboard SDK based application, connected directly to the aircraft)

Furthermore there exist some prerequisites an application must meet before being granted control authority. For OSDK applications the light mode switch must be in correct position (P mode for M300), enabled Multiple Flight Modes within the DJI simulator, and having initialized the DJI Pilot app. The last requirement was not stated in official DJI documentation, causing the OSDK obtaining and immediately losing Control Authority resulting in a single abrupt start and stop motion.

Several prerequisites are required to enable or obtain control authority for the Onboard SDK:

- Flight mode switch must be in correct position (P mode for A3, N3, M600, M600 Pro and F mode for M100)
- Onboard computer application must have activated the Onboard SDK
- Onboard computer application must request and receive control authority

Having satisfied the previous requirements, a communication interface between the ROS layer and the drone's internal computer needs to be built. Taking advantage of the OSDK's built-in functions which include a ROS service for obtaining the Control Authority, a wrapper in the python3 layer can simplify the control authority process and merge all the procedures and protocols to a single point, and facilitate the exchange of information and control between the frontend and the backend.

In the OSDK layer, a ros-service is already setup to obtain and release control authority. A node in the python layer can collect requests from other nodes from a topic and pass it forward to the OSDK. Furthermore we can publish the state of Control Authority to different parts of the ROS network.

2) Autonomous flight controller: controlAuthorityHandler.py flightController.py flightStateController.py

The flight controller is the stack of applications responsible for the navigation and execution of tasks by the drone. It receives incoming tasks internally or from external applications and interprets them accordingly, calculating adjustments based on control and pathing algorithms, and issues commands to the vehicle's actuators.

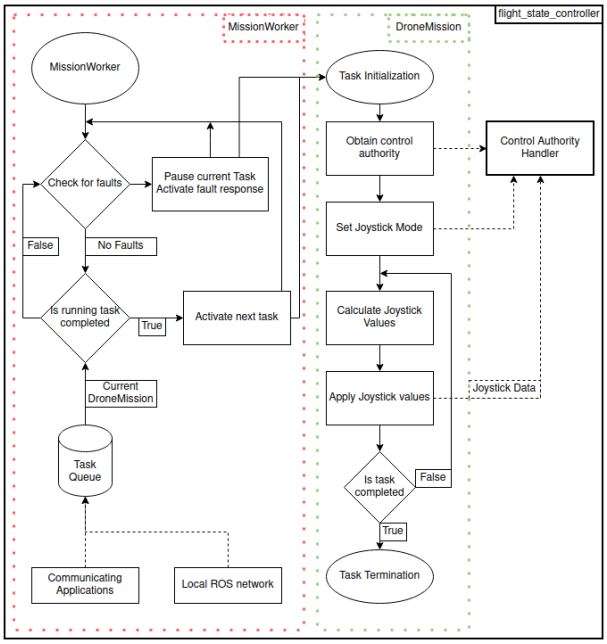

Fig. 9: Flight State Controller - Algorithms

At the forefront of this component is the MissionWorker thread. This worker continually checks on the internal state of the UAV in order to verify that it is mission capable by checking if any internal faults have occurred. If no faults are detected, the MissionWorker moves on to check the state of the current running task. If the running task is finished then it is removed from the Task Queue and the next task is activated. Otherwise, the worker repeats this process. In the case where the worker encounters an internal fault, the current task is paused and a fault response is activated. This fault response can be selected from a list of predefined actions during runtime.

Incoming tasks are parsed into a DroneMission object, which functions as a reference and an intermediary layer between the MissionWorker and the executing logic required for a task. Created DroneMissions are organized in a custom FIFO queue called a TaskQueue, allowing for the easy sequential execution of logic. The MissionWorker can monitor and control the progress of a task via the procedures the DroneMission provides. The DroneMission resembles a wrapper for a thread which executes navigation related code. The thread is passed a

reference of its parent DroneMission that it uses as it executes to update data regarding its execution and progress.

The navigation code executed by the aforementioned threads may encompass a spectrum of tasks, ranging from elementary operations like takeoff or landing to more intricate procedures such as a cartographical procedure.

Fig. 10: Flight State Controller - Algorithms

3) Manual controller schemes: Along with the autonomous control of the drone, a variety of control schemes with 3rd party controllers have been implemented as well. These control schemes serve as an alternate solution to controlling or recovering a drone when a communication channel is unavailable, furthermore, taking advantage of the ROS network and mobile internet the provided solution can be used as solutions for beyond line of sight control. This section will touch on the description of the implementation of these various controllers as well as the algorithms used in order to smooth discrete controller input into a continuous stream of input.

The packet datatype used for message interchange is sensor_msgs/Joy, a datatype provided by the ros-joy package. This datatype bundles axis of movement and button input from a linux-supported controller.

```
Header header
float32[] axes
int32[] buttons
```

[How is control authority process handled.]

a) Conventional Game Controller: The integration of a common consumer controller, a Logitech F710, is facilitated by the use of the joy node, which obtains and publishes the joystick data to the ROS network. A common laptop is used to remotely transmit input data via ROS network to the controlled UAV. A receiving microservice, running from the on-board computer, interprets the incoming input data and performs any programmed functionality, passing the incoming axis input as control directives to the UAV computer.

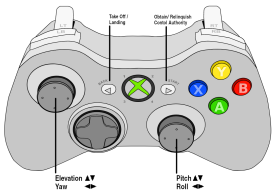

Fig. 11: Control scheme - Computer controller

The following diagram shows the control scheme on four axis for a UAV. The select button can trigger different functionality depending on the state of the motors and the drone's current altitude.

Algorithm 1 Decision Tree - Select Button

```
if altitude < 3.0m then
    if motors_active then
        land_uav()
        turn_off_motors()
    else if not motors_active then
        turn_on_motors()
```

b) Android Controller: The Android controller functions similarly to the conventional controller, but additional quality of life improvements were fitted in order to facilitate real-life use by pilots. Namely, a live video stream of the UAV's main camera and various indicators such as the current control authority and altitude and battery status. The controller layout is the same as the Logitech F710.

c) Keyboard Controller: The keyboard controller served as a recovery or debug tool, and was not tested thoroughly in the field, despite that its novel control scheme could form a basis for more advanced schemes later on.

Joystick input is simulated with the use of PIDs mapped to each movement axis, altitude, yaw, pitch, roll. The target velocities are fed into the UAV's navigation system at a specified rate. Every keyboard click modifies the target speed value by a set specified amount. That amount can be modified by setting a sensitivity value, which sets how many 'clicks' separate the minimum and possible maximum speed values.

Control Authority can be granted or relinquished via the numpad '5', which toggles the current control authority state.

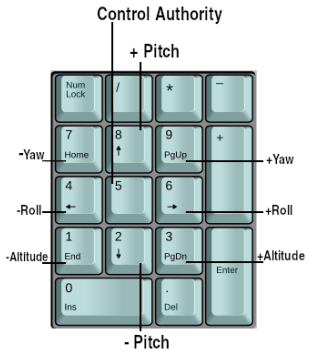

Fig. 12: Control scheme - Keyboard controller

4) Controller Input Quantization: [explain the need, explain how it smooths and improves pilot feedback, explain how sensitivity values were achieved] [diagrams and charts]

D. Inter-process communication

Connectivity with other systems is an important aspect in a UAV system whether as a monitoring tool or an alternate controller, or for seamless communication between multiple UAVs for collaborative missions. The system can interact via ROS with other ROS-enabled applications that are available in its local area network(LAN). Some of such devices would be

a dedicated platform running on a laptop or Android Devices running with DJI-Android-SDK or ROS...

The web-based platform serves as the central hub, enabling seamless coordination and control and orchestrating the UAV fleet. The bidirectional communication encompasses the transmission of mission parameters, and operational instructions from the web platform to the UAVs, while simultaneously receiving real-time telemetry data, sensor data, and status updates from the UAVs.

On the opposite site of the spectrum, the android enabled applications aim to function as the main or a replacement controller. Android apps establish a connection with only a single drone at a time and enable command and control of the UAV, either via autonomous missions that can be assigned to the onboard computer or via sending navigational input through an attached controller.

1) Raft consensus client: Consensus algorithms, exemplified by Raft, are pivotal in distributed computing, providing essential qualities like fault tolerance and coordination. In the realm of distributed systems, the Raft consensus algorithm has emerged as a fundamental and robust approach to solving the critical problem of achieving consensus among a group of nodes. Proposed by Diego Ongaro and John Ousterhout in 2013[citeeeee]. Raft was designed with a one node acts as the leader, responsible for coordinating the consensus process, while the other nodes follow the leader's instructions. The algorithm has been implemented, for the purposes of this project in ROS, using dynamically named topics and ROS-Messages. The implementation of Raft Consensus allows a squadron of drones to elect a leader, which is responsible for establishing a consensus and directing the squadron. This feature is useful for CRPS, as a team of drones has to work together and provide clear and time sensitive data to the faulty drone.

E. Minimal latency streaming

[describe livestream streaming] [FIll out]

1) Image over ROS:

2) RTMP:

3) HTML5 web video server:

VII. Application Scenarios and Experimental Results

Building upon the components formerly introduced, two scenarios were created to prove the capabilities of the created framework. The first scenario aims to gauge the framework's performance as a singular unit and its general readiness. Expanding on the first scenario, the second test case focuses on the scalability and flexibility of the developed framework and hopes to examine aspects such as task sequencing, collaboration and intra-squadron communication.

A. Orthocamera

The task of orthophotography demonstrates the capabilities of the built system not only as a unary system but as a scalable set of components. The application of orthophotography can be used in a variety of fields from search-and-rescue to surveillance and argiculture.

First of, an orthophotography scenario where a UAV is tasked with navigating a path of geographical nodes from a specified altitude while capturing images of the landscape below at a 90 degree angle, thus creating a tiled map akin to satellite imagery which can be displayed on an web-based platform. This task can be scaled by the web-based platform, dividing the nodes into paths covering non-overlapping areas and distributing them between a squadron of drones.

This scenario is activated when the FlightStateController receives a Mission from a ROS connected device. This triggers two threads to begin running in parallel, the FlightStateController will start a pathing algorithm which will iterate through the geonodes and notify the Orthocamera module to start its own thread, which is responsible for capturing imagery at calculated intervals. With these two components working together, the images, along with attached geographical data, are transmitted to a remote platform which is responsible for displaying the incoming images.

B. CRPS Measurement Fusion and image detector

Drawing from the capabilities of the modular mechanisms formerly described, an interconnected squadron of UAVs can be instructed to perform more complex and elaborate tasks. This section touches on the concept of a Collaborative Relative Positioning System, which approaches how reliable relative positioning can be achieved in GNSS-challenged application scenarios using a combination of signals of opportunity (SOPs), inertia and vision data. More generally the task undertaken will explore the design of a collaborative autonomous system that allows for the exchange of instructions and data between its nodes. This system can be useful for not only tracking and following UAVs but also for collaborating as a squadron in order to provide an alternative for a drone with faulty GPS data (Faulty Drone). Our implementation of CRPS is influenced by the research conducted by [Author's Name] as presented in [Paper Title], with our main focus on the protocols and the mechanisms needed to be implemented in order to automate this process, from initialization to returning to home with minimal human supervision. [blah blah...]

The basis of this system should be a reliable Relative Positioning System such as Monocular Depth Estimation[cite] which was implemented in the context of this project. To use MDE we must take an image of the object, in this case a UAV, at a specified and known distance and use that metric to work backwards in order to estimate the position of the object when new images are given to us.

- If the size of the object at distance D0 is S0
- What is D1 if the size now is S1
- D1 = D0 * S1 / S0

The algorithm above would give us the distance of the Faulty (or Rogue) drone from the RPS drone. Using this result and the RPS drones' valid GPS data we can estimate the position of the Faulty drone using the Haversine formula. Alas, the results of this process can be accurate only to a

certain degree, as projecting a 3D scene onto a 2D plane will result in depth information being inherently and permanently lost. We could counter this inherent inaccuracy by doing more calculations with multiple RPS drones and fusing these estimations into one single position the Faulty drone can use in order to navigate.

[diagram ppt]

This system is not quite yet autonomous as of now, some coordination protocols are required in order to initialize and prepare each drone for their task. In order to accomplish that another system is needed which is present in all the drones of the squadron, a collaboration handler, that is able not only to request for aid from the rest of the squadron but also be able to respond to these incoming requests from the other UAVs. What this system does is continuously monitor a UAV system for any GPS faults and in case a fault arises, pause its tasks and halt its movement and begin a CRPS request process by publishing a request to the rest of the squadron. The drones in the rest of the squadron will receive this request and respond whether they can provide assistance to the Faulty drone. When the faulty drone receives CRPS data from the squadron it activates the fault_response thread which is responsible for performing a fault_response task such as Return-to-home or Continue-Mission using the positioning data provided by the rest of the squadron

[collaborattion ppt]

For this protocol to function a custom ros message was made called CrpsRequest.msg which contains the following:

- string serialVersionUID - Identification of the drone system publishing this message
- bool request_active - Crps request active by the requester / Positive or Negative response to request by the requestee
- Telemetry telemetry - Position of the drone publishing this message

The above message is used by the Requester in order to request CRPS from the squadron. It provides the requesters serialVersionUID allowing requestees to identify which drone in the squadron needs help. The request_active allows not only the requester to activate or annul a CRPS request, but also for the requestees to accept or decline help. The telemetry field provides the last position of the requester before it encountered a fault with his GPS, allowing requestees to locate him and also the requester to recognize from how far help is coming.

The second scenario involves a loss of GPS data during an operation of orthophotography. Two other manned drones are responsible to act as helpers and provide relative positioning data to the faulty drone and guide it while it performs a return to home operation.

The order of events in this scenario starts off with a drone performing a orthophotography task as described above when a navigational error occurs. The fault-detector, running on the on-board computer of the faulty drone, detects a problem with satellite navigation, upon then he notifies the CRPS-Collaboration-Handler. This component will start an acknowledgment process with the rest of the drones of the squadron by transmitting and geotagged request for help. When CRPS capable drones in the squadron receive this message, they activate their internal monocular depth perception algorithms and begin transmitting position estimates between them. One drone in the squadron, decided by the raft consensus algorithm is responsible for fusing the outputs from the helping UAVs, transmitting it to the faulty member. The incoming positioning estimates help guide the drone safely back to its home base.

C. Livestream Protocol Comparison

D. Controller Input Comparison

E. Database Performance

VIII. Conclusions

The union of the microservice based design, focused primarily on modular and interconnected components, a reliable backend and an easy to use communication middleware forms an expandable and fault tolerant basis for developing and executing robotic and UAV tools. The proposed framework allows for easy replication between machines aiding the creation of larger squadrons and their smooth operation while simultaneously offering easy control via ROS connected applications. The capabilities of the DJI drone are made easily available allowing users to intuitively create UAV based applications. A number of in-field experiments were performed to validate the performance of the proposed approach which has further demonstrated the system's reliability and ability to retain gathered data.

Acknowledgment

Discuss

References

[1] J. L. Sanchez-Lopez et al., "AEROSTACK: An architecture and open-source software framework for aerial robotics," 2016 International Conference on Unmanned Aircraft Systems (ICUAS), Arlington, VA, USA, 2016, pp. 332-341, doi: 10.1109/ICUAS.2016.7502591.

[2] Matlekovic, L., Juric, F., Schneider-Kamp, P. (2022). Microservices for autonomous UAV inspection with UAV simulation as a service. Simulation Modelling Practice and Theory, 119, [102548]. https://doi.org/10.1016/j.simpat.2022.102548

[3] Barazzetti, Luigi Brumana, R. Oreni, Daniela Previtali, M. Roncoroni, F.. (2014). True-orthophoto generation from UAV images: Implementation of a combined photogrammetric and computer vision approach. ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences. II-5. 57-63. 10.5194/isprsannals-II-5-57-2014.

[4] Ngadiman, Norhayati Kaamin, Masiri Sahat, Suhaila Mokhtar, Mardiha Ahmad, Nor Farah Atiqah Kadir, Aslila Razali, Siti. (2018). Production of orthophoto map using UAV photogrammetry: A case study in UTHM Pagoh campus. AIP Conference Proceedings. 2016. 020112. 10.1063/1.5055514.