Drone Swarm

4/25/2023

Team Members: Alex Hamade, Kassidy McDowell,

Christopher K. Moore, Calvin Wu

Sponsor: Dr. Adrian Lauf

Table of Contents

Introduction/Executive Summary

The purpose of UofL’s Drone Swarm project is to develop an ad hoc network of drones that communicate with each other in a democratic fashion in order to organize itself into a best-fit formation based on a number of factors such as battery life or onboard sensors (e.g., cameras or lidar sensors).

The primary use case for this project is for its use in search and rescue, which is notoriously time-consuming and dangerous for human rescuers. Search and rescue can encompass many different environments which require different resources to accommodate.

Currently, search-and-rescue operations vary from setting to setting. In urban settings, thermal imaging and grid-searches are favored methods of looking for survivors after a catastrophe. While effective in a dense city location, the primary bottleneck is the availability of manpower and equipment to conduct such searches. In mountainous and dense forest settings, helicopter searches and tracking dogs may be used to try and locate the missing persons, but these methods are unreliable and time consuming. In aquatic settings, helicopters and search boats are popular methods of search, but given the nature of bodies of water, missing persons can be led astray by hundreds of miles in a short period of time by the tides and currents.

The drones, however, would be able to thrive in most, if not all, environments: from cities, oceans, forests, and mountains. The drones could be designed to fly in specific patterns and formations based upon the terrain and number of drones in the fleet to map the search area in the most efficient way possible. Importing tide and current data could help search crews cast a wider net in vast aquatic settings. By increasing the number of units conducting the search and reducing the decision-making time by making them fully autonomous, the drones would vastly improve the time it takes to fully conduct search operations in all environments. This project looks to make the process more efficient, less costly, and more potent in terms of its search capabilities.

This also has applications in locating and examining contamination zones involving hazardous materials. Currently, hazmat crews will use sensors to determine the type of spill and the extent to which it reaches. The wind and water currents can affect the direction that a contaminant spreads, which is taken into account during the investigation. Locating the source of a spill is more complicated work, and can be done primarily through sensor monitoring, where sensors are used to gauge the concentration of a contaminant, and tracing, where crews will attempt to follow a spill to its origin. Remote sensors, such as satellite imagery or infrared cameras can be used to further gauge the extent of a spill.

Rather than putting humans in danger and equipping them with expensive hazmat gear, the drones can be equipped with these sensors instead to determine the source of contamination leak. This allows cleanup crews to gather valuable information while determining the best course of action autonomously and safely for containing the spill. Satellite imagery could be used to automatically map out a search area for the drones to examine, while the onboard sensors would allow the drones to engage in concentration monitoring and tracing to locate the source. This approach would ideally save organizations time, money, and most importantly, improve upon hazmat-related illness and injury within those crews.

The Drone Swarm project has been in development for a number of years, but development was recently reset, due to design, maintenance, and scope issues. It was decided that the best step to continue the project would be to discard previous capstone teams' work and essentially start the design process over. Because of this, only the work from the two most recent semesters was included in this project's background research.

The first capstone team worked on developing an algorithm in Python for the drones to use while in flight. This algorithm would determine where a drone would need to be to optimize the overall fitness of the swarm's formation. Note that this algorithm did not calculate or simulate specific flight patterns, just optimal positions in space – flight paths will be left up to the onboard flight-controller when physical drones are set to start flying. A simulated environment was created to test the algorithm's effectiveness during drone flight, which also included a simulated ad hoc network. The team decided to use a greedy algorithm to determine fitness and decision making amongst the drones.

The most recent capstone team worked on porting that Python-based algorithm into C++ to make it increase efficiency and make it more robust with other onboard systems. This was recommended by the previous team since Python was used for it's simpleness rather than its speed or compatibility. This team also began the implementation of hardware onto the physical drones to get the drones out of the simulation and into the world. They also began setting up the ad hoc network that the drones would use to communicate with through B.A.T.M.A.N (Better Approach to Mobile Ad-Hoc Networking) protocol.

As for this current capstone team, the original semester goals included getting multiple drones flying in the air, using the algorithm to alter their formation in real-time. After beginning work on the project and doing preliminary research, the goals had to be modified to fit our constraints. The team decided to instead focus on enhancing the testing environment and to improve upon documentation so future teams would need less time getting set up and working. The focus was to create an environment where real drones (not simulated) could easily be kept track of. Indoor testing is ideal because it creates a more consistent testing environment and is generally safer since LARRI has a new rig designed for indoor drone flight.

System Description

Needs Assessment/System Requirements

The Drone Swarm setup is comprised of a set of drones, each with an onboard computer, a flight controller, a GPS, and any other onboard peripherals relevant to its specified purpose, with the primary example being optical cameras and/or lidar scanners. The onboard flight controller can be used to take in both coordinates from the GPS, and instructions from the onboard computer to interpret for and instruct the drone to fly based on its position, and what the algorithm determines through its state such as what peripherals are onboard on the drone or the general health of the drone, namely battery life.

The onboard computer will be the device that processes the more complex calculations based upon the scenarios that present themselves. It will be the main driver that determines the position of the drone and process important information such as image data gathered from the camera or sensor data built into the flight controller or placed on the drone. The computer will house the main algorithm as well as serve as the platform for the ad-hoc network imperative for the swarm algorithm to function as intended by flexibly allowing drones to enter and leave the network and make use of swarm intelligence as much as possible.

The GPS is used to pinpoint the location of the drone and allow for much of the maneuvers required for the drones to function. For testing, this may be replaced with any system that can provide external locational and orientational data such as motion capture or vision inertial odometry in order to provide local positional data to the drone to isolate sources of error such as the environment or the non-inconsequential error brought in by a GPS, especially when attempting to fly a drone indoors.

The drones themselves should be able to function no matter the shape or size within reason. As mentioned before, the drones should also house appropriate sensors by scenario. The drones should also be fitted with extensive battery life for long, uninterrupted segments of flight. Furthermore, methods of manual control of flight will be needed in order to retrieve drones in critical condition, either through remote control or through connection from a ground station.

The onboard computer should also be fitted with robust security and encryption in order to prevent cyber-attacks that may interfere with the drone swarm. Due to the ad-hoc nature of the network, it is especially imperative to implement some form of security, as without it, it could be both simple to do perform a targeted attack and be extremely disruptive to the network.

Initial Systems Specification

The Drone Swarm must first go through a preliminary phase of intensive testing. This will require some form of setup that deviates from the final iteration in order to better accommodate for this testing phase. The University of Louisville had set up and provided a new Optitrack system, which can be used to provide external locational data. As stated before, the use of GPS inside a building would be vastly unfeasible due to the immense amount of error compared to the relatively low amount of space, as well as the fact that GPS signals are much more inconsistent indoors. Optitrack, on the other hand, is even more responsive, with a ~20 ms delay and consistent packet streaming of up to 120 frames per second, giving very accurate positional and orthogonal data suitable for testing.

The drone utilized was provided by Professor Lauf. It is a custom-built quadrotor drone, designated “FatMax”, with a LiPo rechargeable battery, a Pixhawk4 (PX4) flight controller, a GPS, of which will not be used, and a Raspberry Pi installed with a lightweight Ubuntu 20.04 server OS. The operating system includes a number of installed packages, including Robot Operating System 2 Foxy Fitzroy (ROS 2), a package that allows for a machine to receive packet data from a publisher, of which may send messages to subscribers that subscribe to any specified topic. This can allow for quick, organized communications between nodes. ROS 2 will play an imperative role in allowing Optitrack to function as an external positional data source for FatMax.

Optitrack is a motion capture system that utilizes multiple infrared cameras to pinpoint the location of rigidbodies set in a designated cage with extremely high accuracy. The system also supports the recording of multiple rigidbodies, which may allow for the pinpointing of multiple nodes, or in this case, drones. This makes indoor testing possible where GPS would not be suitable. As stated before, the cameras may read up to 120 Hz and latency boasts low latencies down to 20ms. The Optitrack cameras will send the positional data to the Motive API, where it is calculated and converted to usable data. It can then utilize NatNet to take and broadcast and publish the data to the local network for the drones’ onboard computers to subscribe to based on which drone they actually are.

ROS 2 by itself will not be sufficient enough to support external positional data. Some officially supported packages will be required in order for Optitrack to communicate with the onboard computer on FatMax. “mocap_optitrack” is a package originally for Robot Operating

System (ROS), a legacy version of ROS 2. However, implementation of functionality is still possible, despite the version differences, and ROS 2 can run the mocap_optitrack package without issue. Another package utilized is MAVROS, a combination of MAVLink and ROS used to communicate with the onboard Pixhawk4 flight controller and push the positional data to appropriate uOrb topics that will then allow the PX4 to interpret the local positional data and determine a flight path as if it were using GPS.

Batman-adv is also installed as the drone's primary form of ad-hoc network communication. This will be the main avenue for different drone entities to communicate with one another in order to better inform the built in algorithm. This is necessary as it will be the most realistic implementation of the communication between the drones.

For future edits to the setup, the Ubuntu image on the Raspberry Pi was flashed into an img file and archived for use by future Capstone team projects. Users may flash the image on as many bootable disks as needed in order to have the same/similar setups on multiple drones, streamlining the process of setup and allowing for additional changes to be built on top of the existing image to maintain all progress from previous semesters.

Final Specifications

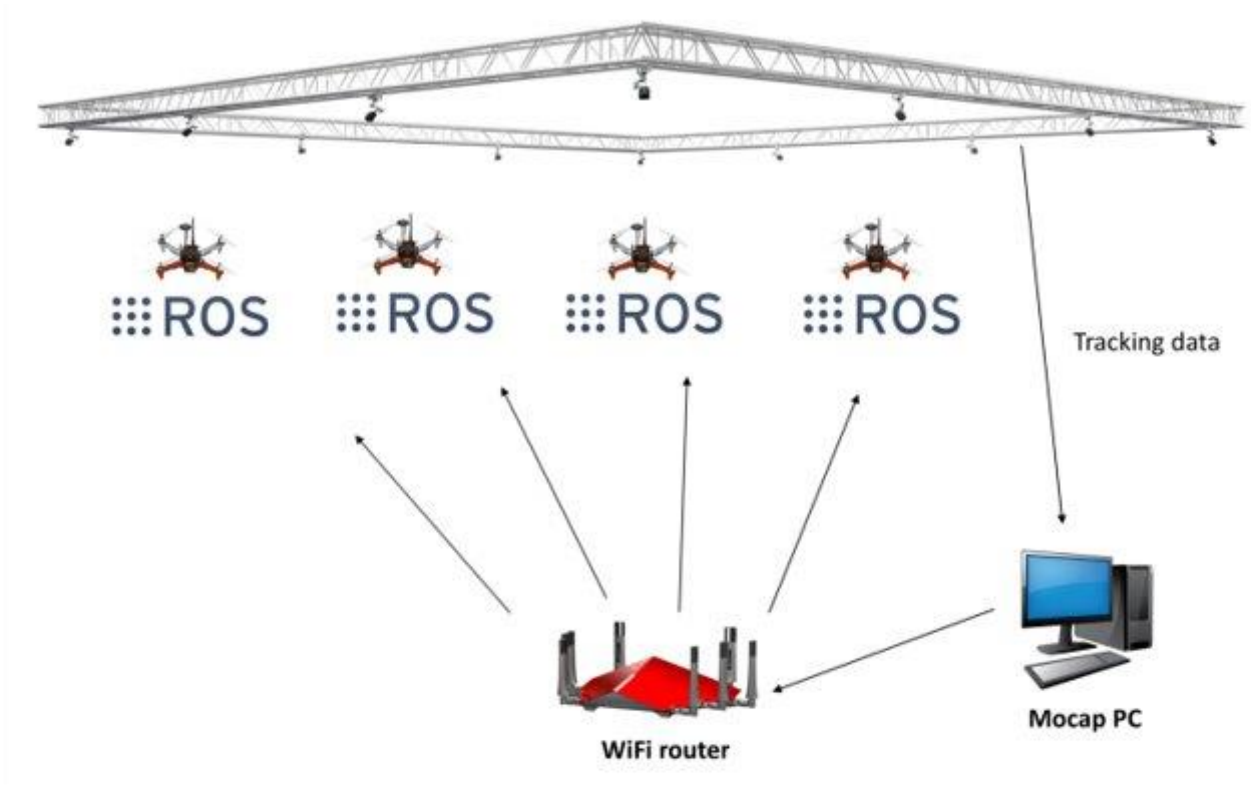
Currently, the Raspberry Pi 3+ is set up with ROS 2 and mocap_optitrack to communicate with the Optitrack camera system, batman-adv as the ad-hoc network for multiple drones to communicate over, and MAVROS for the Raspberry Pi to communicate with the Pixhawk4 flight controller. There are currently issues with MAVROS being unable to communicate with the PX4 flight controller, as MAVROS, though supported through unofficial avenues, is specifically made for the original ROS, and has no official support as of now. Another way to communicate with the PX4 controller through the Raspberry Pi is the XRCE-DDS middleware that is supported for the latest version of ROS 2. However, this does not have official documentation on how to send Optitrack topics from the Raspberry Pi to the PX4 controller as pose data. This needs to be researched. Another option is to downgrade to ROS Kinetic Kame, which would require the installation of Ubuntu 16.04. Most functionality is officially supported in this environment. This would allow for indoor testing of drone swarm flight patterns.

The current implementation of batman-adv ad-hoc network can receive messages from other onboard computers, but the messages themselves are currently unsuited for interpretation. Currently, the command “ros2 topics echo /mavros/mocap/pose” currently sends a continuous stream of pose data instead of sending one frame at a time. This prevents batman-adv to update its published message in a consistent, frame by frame stream, as it is given pose data in chunks, rather than frame by frame. There is currently functionality that is able to resolve this through ROS 2 Humble Hawksbill, the newest iteration of ROS 2. It adds the tag “--once” which will only return one packet at a time. This would allow the B.A.T.M.A.N. network to communicate a drone’s position one frame at a time.

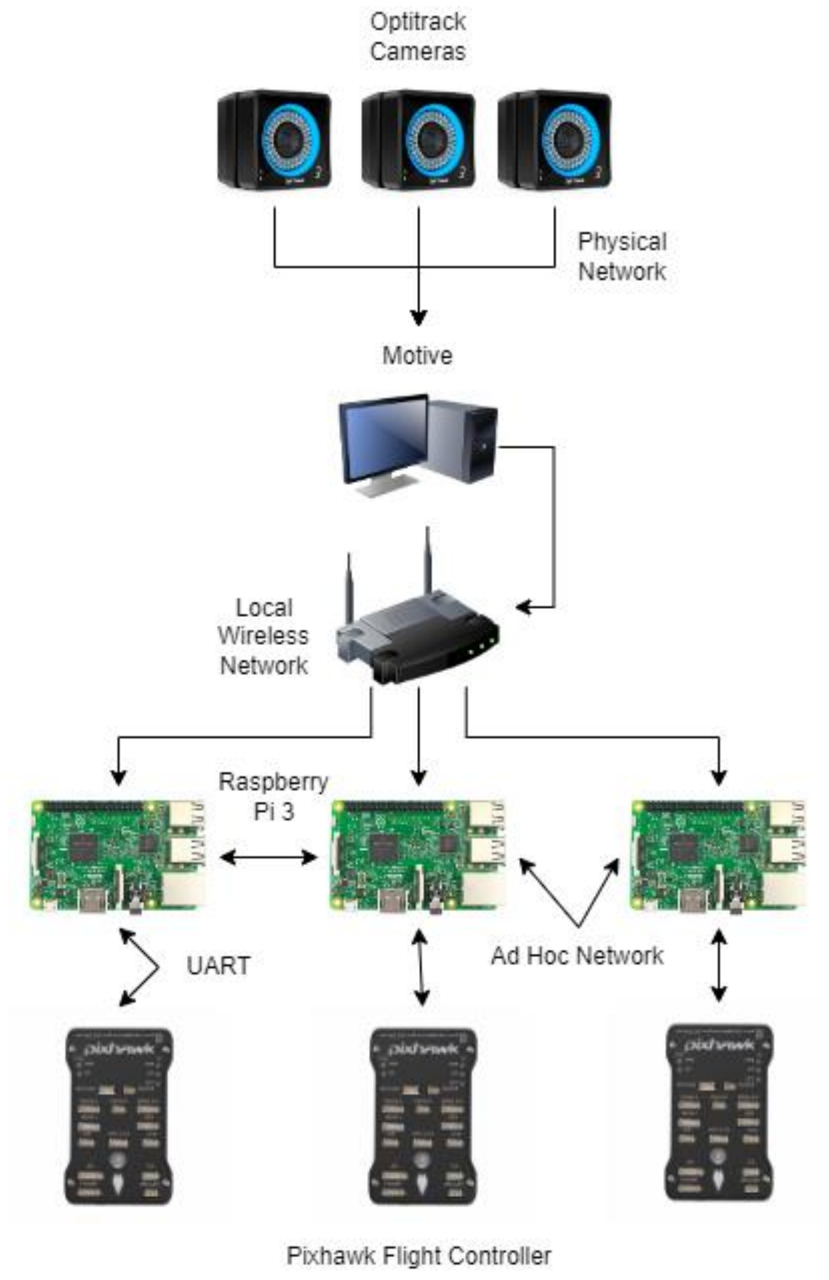
As development of the algorithm progresses, there would need to be a pivot to the GPS system in order to test the algorithm in more realistic situations. The FatMax includes the Holybro Pixhawk4 GPS, which is officially supported by the Pixhawk. This can provide the onboard computer with real positional data to calculate flight patterns as well as give more room for drones to fly, as there are no space constraints that are present as opposed to testing indoors with a motion capture system would have.

The Pixhawk 4 flight controller would still be the main driver for the drone, with connections with the UART of the Raspberry Pi through its TELEM2 port. The Pixhawk 4 will be able to take the Pi’s flight path instructions and reinterpret them as real flight paths for the drone to take. It will also take the GPS positional data from the Holybro GPS and send it to the Pi for it to process in its calculations. Along with the pivot from motion capture to GPS, it may be valuable to investigate the XRCE-DDS middleware to implement, as it is the currently supported middleware for ROS 2. There would be no need to maintain usage of previous, legacy features.

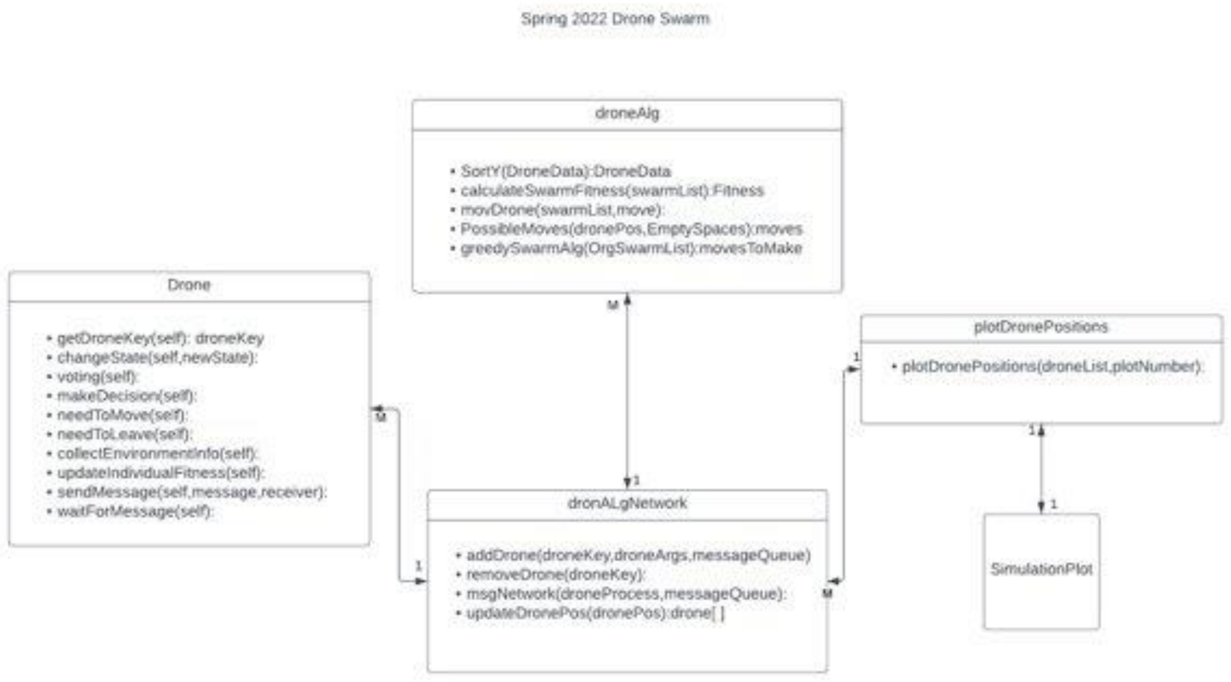
System Diagrams



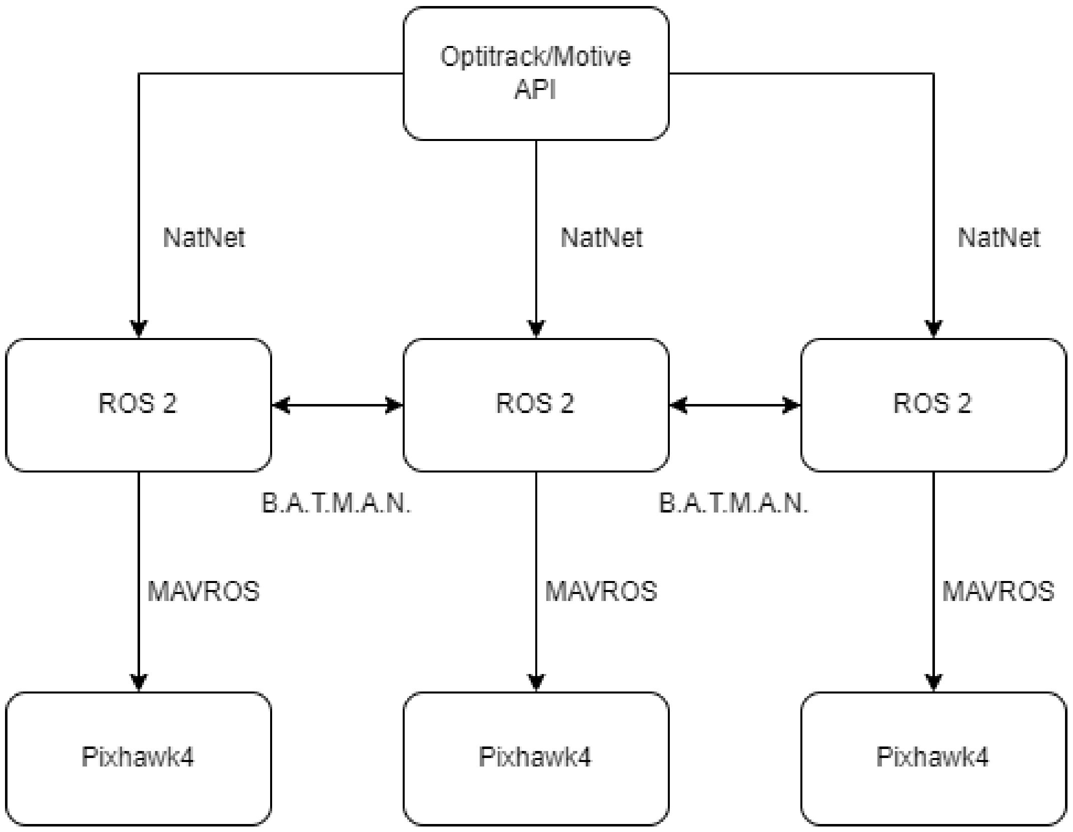
Hardware Overview Diagram



Software Overview Diagram


Spring 2022 Drone Swarm
droneAlg
• SortY(DroneData):DroneData
• calculateSwarmFitness(swarmList):Fitness
• movDrone(swarmList,move):
• PossibleMoves(dronePos,EmptySpaces):moves
• greedySwarmAlg(OrgSwarmList):movesToMake
Drone
• getDroneKey(self): droneKey
• changeState(self,newState):
• voting(self):
• makeDecision(self):
• needToMove(self):
• needToLeave(self):
• collectEnvironmentInfo(self):
• updateIndividualFitness(self):
• sendMessage(self,message,receiver):
• waitForMessage(self):
plotDronePositions
• plotDronePositions(droneList,plotNumber):
dronALgNetwork
• addDrone(droneKey,droneArgs,messageQueue)
• removeDrone(droneKey):
• msgNetwork(droneProcess,messageQueue):
• updateDronePos(dronePos):drone[ ]
SimulationPlot
M
1
M
1
1
M
1
1

This first diagram demonstrates Spring 2022 Capstone team’s class structure of its drone algorithm. It’s a simulation that projects the movements and desired arrangements of the swarm utilizing a greedy algorithm. It takes into account the ad-hoc nature of the network, allowing for nodes to be added to the network and taking account of the shape of the formation with these freely moving drones. Each node is assigned a specific amount of fitness to simulate based on a multitude of hypothetical factors such as onboard peripherals, battery life, or drone health.



This diagram showcases much of the work done this semester and displays much of the functions brought from it. The focus of this semester was to utilize Optitrack pose data to simulate local position for the Pixhawk 4 to interpret and use for flight. Optitrack would send pose data to each node individually based off of different topics for each drone (drone1, drone2, etc.). ROS 2 will receive the data and place them into these topics. Using MAVROS, the data is then sent to the Pixhawk 4 flight controllers to fly with.

Economical, Technical, and Time Constraints

There were a number of constraints that the team experienced during the Capstone project. Initially, this was a project that was fresh off of restarting in almost its entirety, so there was little foothold on the current state of the project. Much of the documentation was either scattered, rudimentary, or incomplete, and failed to truly explain the scope of the project as a whole. Initially, the team worked towards implementing a working setup of batman-adv on virtual environments. However, there were existing issues that occurred potentially due to the virtualized environments that we were utilizing such as Virtual Machines, Windows Subsystem for Linux, Docker, etc. There was a while where there was no access to physical hardware, namely Raspberry Pis. Combined with the fact that the existing documentation had specific nonfunctioning aspects, the endeavor took longer than it should have been.

In general, there were multiple points during the project where resource issues took a while to address. Bootable SD cards were provided by the team members themselves in order to actually boot the Raspberry Pis with Ubuntu 20.04. A wire to connect the TELEM2 port on the Pixhawk 4 flight controller to the UART of the Raspberry Pi was only acquired only a few days before the presentation. This was a major contributing factor as to why a flyable demo of the drone utilizing mocap pose data was not possible, as there was little time to set up the Pixhawk 4 to receive the pose data as well as troubleshoot issues with both the drone and the setup of the implementation.

There was also a bit of misdirection in the setup. Initially, the team attempted to install ROS 2 through multiple environments. Initially Debian was used with an older version of ROS. The source installation process was extremely intensive and prone to multiple points of failure, thus making it difficult to pinpoint what exactly was causing issues in the installation process. There was then a pivot to ROS 2 Foxy with Ubuntu 20.04. Initially, ROS 2 was in the process of being installed through source, but it was later found that ROS 2 had existing binary installation procedures which was a much more streamlined process. Much time would have been spaced if binary installation was learned about earlier. That being said, a ROS 2 source installation could potentially be helpful and even necessary to develop custom packages that suits the needs of the Drone Swarm project.

The team was mostly able to contact the sponsor, Professor Lauf, with little issue. There were points where he was unable to respond due to his schedule or personal issues, which would slow down progress where inquiry was needed. More specific direction for the project was a bit late to come, but by the second half of the semester, there was a relatively clear goal in mind. According to the team, work towards setting up batman-adv was of very little importance and was only a small part of the semester's work.

Part of the reason that work on the Optitrack system was late was also the fact that the system was still under construction during the first half of the semester. There was little way for the team to work on it, even if there was a direct effort on implementing the external positional data flow to the drones. Only until after the Optitrack system was fully setup was when work towards ROS 2 and mocap_optitrack was possible.

Detailed Implementation

Hardware Detailed Implementation

The drone swarm system currently uses a Raspberry Pi 3+ set up with ROS 2 and mocap_optitrack to communicate with the Optitrack system. For communication between drones, an ad-hoc network is implemented using B.A.T.M.A.N. and A.L.F.R.E.D. Finally, MAVROS is installed, which will eventually facilitate communication between the Raspberry Pi and the PX4 flight controller. Each of these components are described in detail below.

Raspberry Pi 3B

The Raspberry Pi 3B was the microcomputer chosen for its availability as Professor Laug already had one for the project. The microcomputer on board the drone is the heart of the system. It is a mediator for all of the components of the system. It is responsible for communicating with the flight controller, communing with nearby drones, and receiving motion capture data. The microcomputer does not have to be a raspberry pi as long as it is able to cover the same functions the microcomputer needs to accomplish in the system.

The Microcomputer needs a wireless internet interface for communicating with the optitrack server as well as using B.A.T.M.A.N for communicating in an ad hoc network with other drones nearby. The Microcomputer will also need to have a UART interface for communicating with the flight controller over that physical connection on the drone. Most microcomputer available on the market are going to have these peripherals and will most likely to be running on Linux anyways. Raspberry Pi's have been in short supply, so an exploration into alternatives.

Pixhawk 4 Controller

The Pixhawk 4 controller can be configured to use one of two methods for facilitating communication between the pi and the flight controller: MAVLink and XRCE-DDS. Both were researched and the necessary software is installed onto the pi for either form of communication, but neither were fully integrated nor tested. The team's recommendation is to use MAVLink because it offers MAVROS which has built-in features useful for communication with Optitrack. However, since XRCE-DDS is officially recommended by Pixhawk to be used with ROS 2 and the team could not fully test either method, both are described. It will be a future team's responsibility to test and choose a form of communication between the Raspberry Pi and the Pixhawk 4 flight controller.

Optitrack

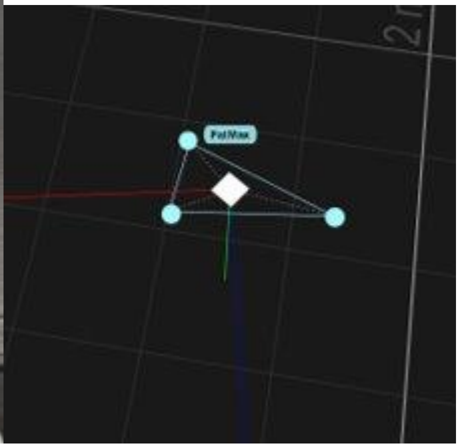
The Optitrack motion capture system was acquired through a grant to aid in testing the Drone Swarm algorithm. It is used to collect locational data that is used in place of GPS data for indoor testing. The locational data is gathered by the Optitrack cameras which capture the position of identifier markers. These markers can be placed on an object to create a rigid body. There must be at least three markers on any object in order for Optitrack to create a plane of the object which will be used to calculate the orientation and position of the object. There are five data types that a collection of markers can be formed into:

1. Marker Set Data - A named collection of identified markers and their positions (X,Y,Z).

2. Rigid Body Data - A named segment with a unique ID, position, and orientation data, and the collection of identified markers used to define it.
3. Skeleton Data - A named, hierarchical collection of rigid bodies.
4. Force Plate Data - Rebroadcast force and moment data.
5. Device Data - Rebroadcast data from analog devices such as DAQ devices.

The rigid body data type is what we use to classify our drone "Wide Max" in Optitrack. This rigid body data is collected as the "Wide Max" moves within the field of vision of the Optitrack cameras. This data can then be received through NatNet SDK. NatNet SDK is a client/server-based architecture that allows client applications to stream data received by the Optitrack cameras. This can be used directly on Windows operating systems, or it can be depacketized for any other operating system. Our first attempt at getting data with NatNet SDK was to use the depacketized version directly.

This was successful in sending data about every rigid body to the Raspberry Pi, however, there was no easy separation of rigid bodies and messages nor was there a great way to translate these messages into a useful form to be used with the Pixhawk flight controller. It would be inefficient to have each drone receive positional data for all drones and then have to parse the message to get their own positional data. Latency can be a particular issue for the drone swarm because slow reaction times could cause drones to collide with each other. For this reason, another tool was needed to subscribe to the positional data sent from the Optitrack system. ROS 2 was the solution to this issue which uses the optitrack_mocap package to format the messages.



The images here show the FatMax, and its rigid body shown side by side. The markers on the drone are what allows the Optitrack cameras to track the position of the drone and then send it over to the computer running Motive API for it to process.

Software Detailed Implementation

MAVROS

MAVROS is an extension package off of ROS 1 that utilizes MAVLink to communicate with drone related peripherals, ground stations, and autopilots. As the Pixhawk 4 flight controller utilizes an autopilot, it is possible to take advantage of it for direct communication with the Pixhawk 4 to then take instructions from the onboard computer to fly in whatever way is needed. It is one of the main packages needed to make mocap_optitrack function properly and is the one that converts the mocap_optitrack pose topics into uORB topics for the Pixhawk 4 flight controller to take up and reinterpret.

There is still a lot of work to be done regarding MAVROS. Even though the physical connection between the microcomputer and flight control is there, the team encountered issues in getting the connection to actually be established. MAVROS should easily allow the use of ROS topics to control the drone. MAVROS has a topic that it subscribes to called "/mavros/mocap/pose" that is used for the drone's local position using motion capture. The mocap_optitrack package should publish the drone's position data to this topic. The "/mavros/set_position/local" topic can be published to with a local position in the mocap space. Once published to, the Pixhawk flight controller will receive commands from the MAVROS package that sends the drone to that specified location.

XRCE-DDS

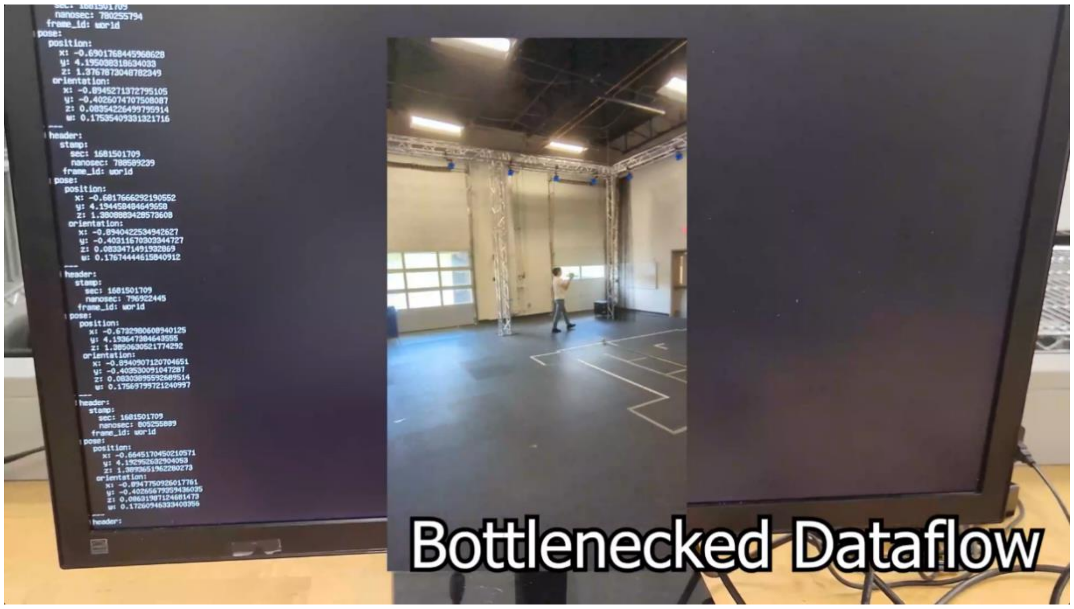
XRCE-DDS acts as the middleware between the flight controller and the raspberry pi in conjunction with ROS 2 and is only supported by PX4 versions 1.14 and newer. The PX4 uses an XRCE-DDS client which should be configured to automatically build at runtime. The XRCE-

DDS agent is installed on the pi along with the message definitions for the controller in the primary ROS 2 Workspace. Both the client and the agent must be running to send communication between the Raspberry Pi and the flight controller. The data can be sent over serial, UDP, TCP or a custom link.

Messages are sent in the form of uORB Topics which are defined by XML files. This communication is bidirectional, allowing the pi to subscribe or publish messages to the flight controller. These messages, when published, can act as commands for the drone. Code could be developed that takes positional data from Optitrack collected using ROS 2 and uses it to determine what command/topic should be published to the flight controller and publish it all in the same code. This newer and more involved technology could potentially give greater control to the developer on what commands are published to the flight controller or it may add an unnecessary layer of complexity. Due to time constraints, this was not able to be tested and is left to future teams.

ROS 2

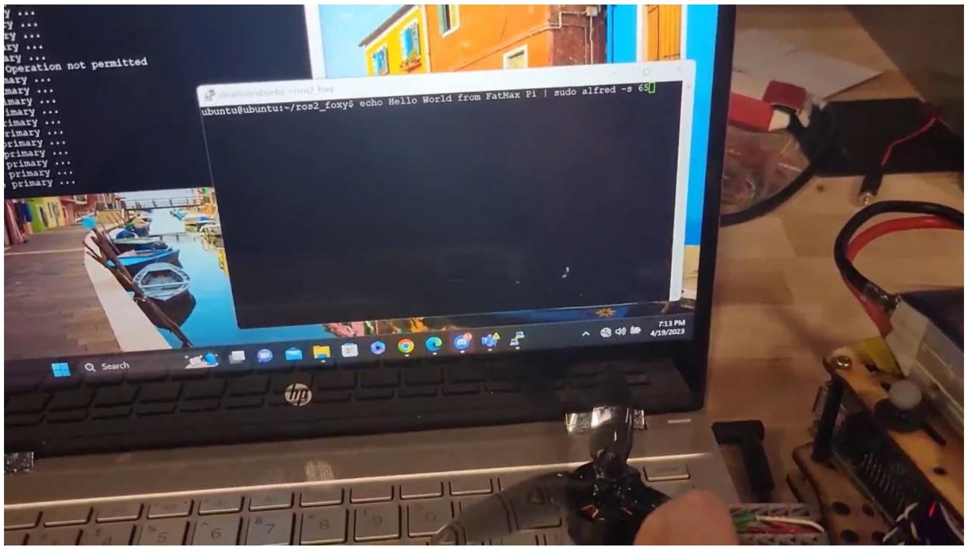
Robot Operating System version 2 (ROS 2) is an open-source collection of tools used for robot applications. ROS 2 forms the middleware between different ROS processes. In this case: the optitrack API and the Pixhawk flight controller where the Raspberry Pi acts as the companion computer where ROS is installed. ROS uses a publish/subscribe mechanism to topics in order to pass messages between these end nodes. The rigid bodies from Optitrack form individual topics in ROS for example. The streaming data of the rigid body would be the message that could be subscribed to be another ROS node like the Pixhawk controller. ROS has two client libraries that are useful for developing code to subscribe and publish to topics: rclcpp (C++) and rclpy (Python). ROS 2 only works with Python 3 (while ROS only works with Python 2) and targets C++ version 14.


```
    sec: 1681501709
    nanosec: 780255794
  frame_id: world
pose:
  position:
    x: -0.6901768445968628
    y: 4.195038318634033
    z: 1.3767873048782349
  orientation:
    x: -0.8945271372795105
    y: -0.4026074707508087
    z: 0.0835422649795914
    w: 0.17535409331321716
---
header:
  stamp:
    sec: 1681501709
    nanosec: 788589239
  frame_id: world
pose:
  position:
    x: -0.6817666292190552
    y: 4.194458484649658
    z: 1.3808883428573608
  orientation:
    x: -0.8940422534942627
    y: -0.4031167030334472 7
    z: 0.0833471491932869
    w: 0.17674444615840912
---
header:
  stamp:
    sec: 1681501709
    nanosec: 796922445
  frame_id: world
pose:
  position:
    x: -0.6732980608940125
    y: 4.193647384643555
    z: 1.3850630521774292
  orientation:
    x: -0.8940907120704651
    y: -0.403530091047287
    z: 0.0830389559268951 4
    w: 0.17569799721240997
---
header:
  stamp:
    sec: 1681501709
    nanosec: 805255889
  frame_id: world
pose:
  position:
    x: -0.6645170450210571
    y: 4.192952632904053
    z: 1.3893651962280273
  orientation:
    x: -0.8947750926017761
    y: -0.40265679359436035
    z: 0.0863198712468147 3
    w: 0.17260946333400356
---
header:
```

This is a screenshot of a demo presenting the ROS2 topic being received from Optitrack. As shown in the picture, each frame presents some header information, such as frame id and timestamp. After that, is the pose data. It first shows the positional data with its XYZ coordinates, and the second half shows the orientation in pitch, yaw, and roll. One should keep in mind that how Optitrack interprets coordinates and how the Pixhawk interprets coordinates is different and must be changed accordingly. It’s important to note that MAVROS already handles this discrepancy automatically.

B.A.T.M.A.N. and A.L.F.R.E.D.

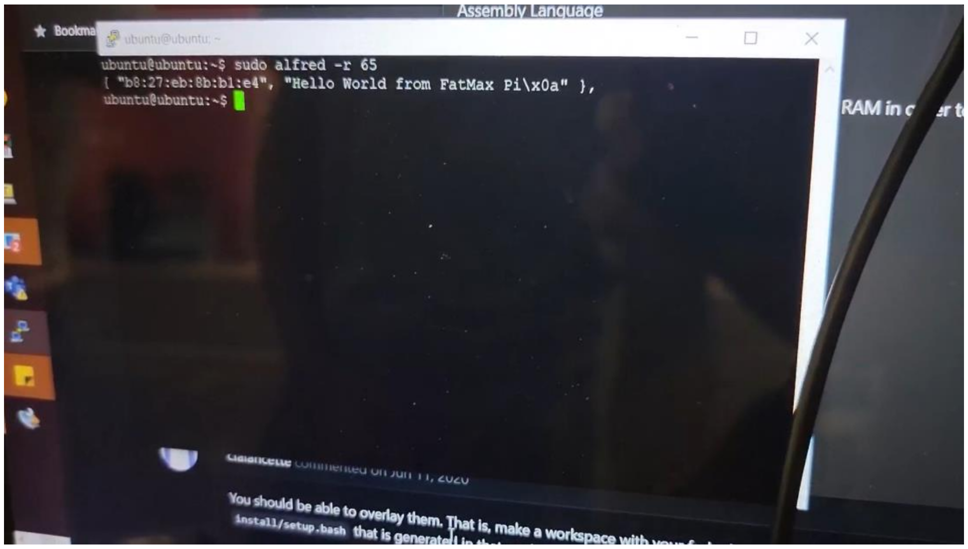
The networking component between drones was designed by a previous capstone team and is implemented on the Raspberry Pi in conjunction with the rest of the system. The Better Approach to Mobile Ad-hoc Networking (B.A.T.M.A.N.) and the Almighty Lightweight Fact Remote Exchange Daemon (A.L.F.R.E.D.) are the tools used to establish the ad-hoc network. This network configuration would allow drones to add and drop from the network and initiate/repeat messages to nearby drones without any centralization or single point of failure. Each drone would keep a n-nearest neighbors table which would designate which drones it should forward messages to. As a result, any message a drone sends would cascade through the network such that every drone receives the message. This is important so that every drone in the network can remain up to date with all critical information about the hive. Messages could also be configured to be sent only to the drones near it and not repeated over the entire network.



```
Operation not permitted
```

```
ubuntu@ubuntu:~/ros2_foxy$ echo Hello World from FatMax Pi | sudo alfred -s 65
```

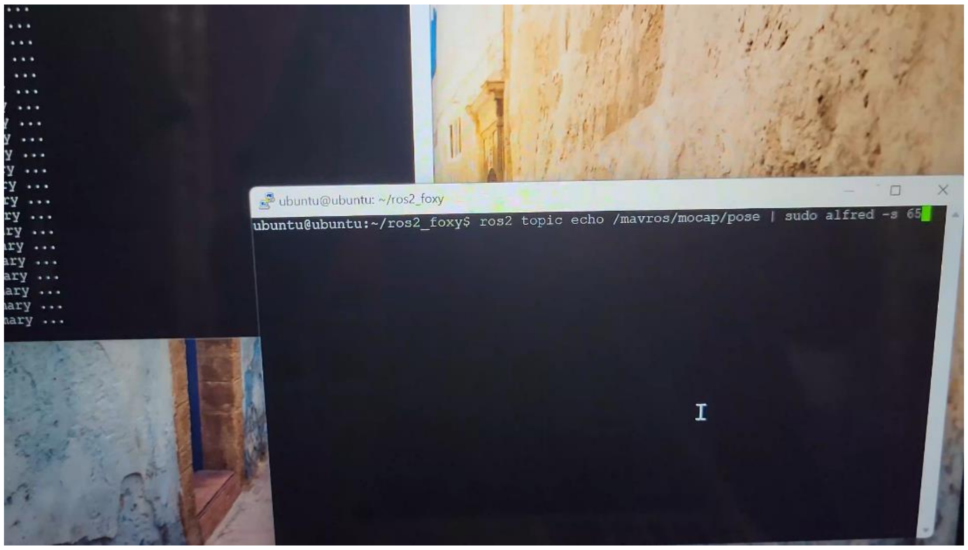
The images here show B.A.T.M.A.N. working on two different Raspberry Pis. The first image shows a Windows computer with an SSH going into the onboard computer of the FatMax drone. The message it is sending is “Hello World from FatMax Pi”. This message is currently in a B.A.T.M.A.N. network channel and can be received by any other computer that can read from this channel.



```
ubuntu@ubuntu:~$ sudo alfred -r 65
{ "b8:27:eb:8b:b1:e4", "Hello World from FatMax Pi\x0a" },
ubuntu@ubuntu:~$
```

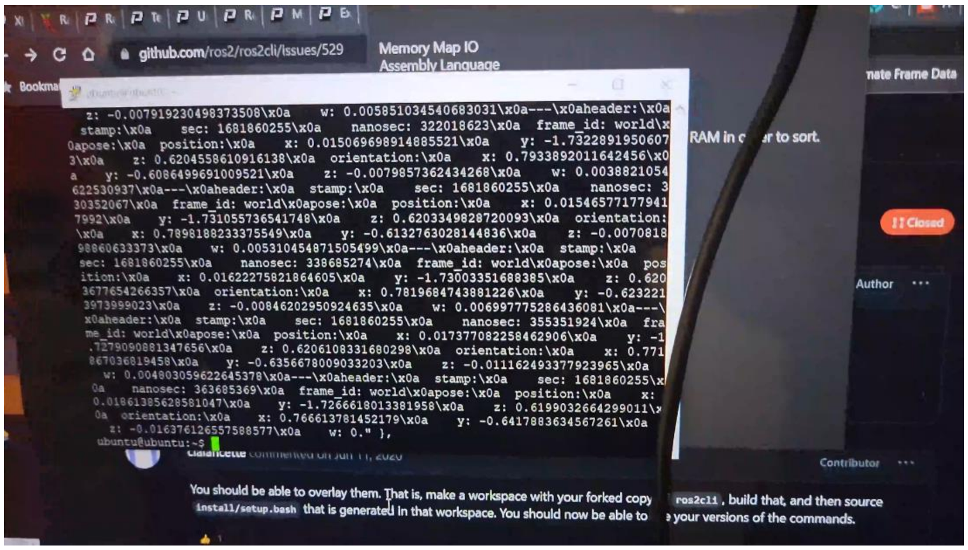
The second image shows another Windows computer with an SSH going into another Raspberry Pi with B.A.T.M.A.N. installed. This Pi is subscribing to the channel in order to receive the message. As shown above, the Pi was successful in receiving the message from the other Pi.

Here, we have Christopher “Kiely” Moore standing with the FatMax in the Opitrack motion capture cage. Currently, a Windows computer operating Motive API is taking in positional data from the markers on the FatMax and sending it to the onboard computer in ROS 2 topics. The current setup is to simulate drones communicating with one another where in a hypothetical situation, the drones would need information from one another in order to make decisions akin to a swarm intelligence.



```
ubuntu@ubuntu:~/ros2_foxy$ ros2 topic echo /mavros/mocap/pose | sudo alfred -s 65
```

Here, on the Windows computer with an SSH to the FatMax onboard computer, it is using a command to receive the pose data from the Motive API, and taking the message it receives to send it through B.A.T.M.A.N. In a hypothetical situation, a neighboring drone may take the pose data and use it to perhaps swap with its neighbor. Other forms of data may also be sent through the B.A.T.M.A.N. network.



```
  z: -0.007919230498373508\x0a    w: 0.005851034540683031\x0a---\x0aheader:\x0a
 stamp:\x0a    sec: 1681860255\x0a    nanosec: 322018623\x0a  frame_id: world\x
0apose:\x0a  position:\x0a    x: 0.015069698914885521\x0a    y: -1.7322891950607
3\x0a    z: 0.6204558610916138\x0a  orientation:\x0a    x: 0.7933892011642456\x0
a    y: -0.6086499691009521\x0a    z: -0.007985736243426B\x0a    w: 0.0038821054
622530937\x0a---\x0aheader:\x0a  stamp:\x0a    sec: 1681860255\x0a    nanosec: 3
30352067\x0a  frame_id: world\x0apose:\x0a  position:\x0a    x: 0.01546577177941
7992\x0a    y: -1.731055736541748\x0a    z: 0.6203349828720093\x0a  orientation:
\x0a    x: 0.7898188233375549\x0a    y: -0.6132763028144836\x0a    z: -0.0070818
98860633373\x0a    w: 0.005310454871505499\x0a---\x0aheader:\x0a  stamp:\x0a
 sec: 1681860255\x0a    nanosec: 338685274\x0a  frame_id: world\x0apose:\x0a  pos
ition:\x0a    x: 0.01622275821864605\x0a    y: -1.73003351688385\x0a    z: 0.620
3677654266357\x0a  orientation:\x0a    x: 0.7819684743881226\x0a    y: -0.623221
3973999023\x0a    z: -0.00846202950924635\x0a    w: 0.006997775286436081\x0a---\
x0aheader:\x0a  stamp:\x0a    sec: 1681860255\x0a    nanosec: 355351924\x0a  fra
me_id: world\x0apose:\x0a  position:\x0a    x: 0.017377082258462906\x0a    y: -1
.7279090881347656\x0a    z: 0.6206108331680298\x0a  orientation:\x0a    x: 0.771
867036819458\x0a    y: -0.6356678009033203\x0a    z: -0.011162493377923965\x0a
  w: 0.004803059622645378\x0a---\x0aheader:\x0a  stamp:\x0a    sec: 1681860255\x
0a    nanosec: 363685369\x0a  frame_id: world\x0apose:\x0a  position:\x0a    x:
 0.01861385628581047\x0a    y: -1.7266618013381958\x0a    z: 0.619903266429901l\x
0a  orientation:\x0a    x: 0.766613781452179\x0a    y: -0.6417883634567261\x0a
  z: -0.01637612655758857\x0a    w: 0." },
ubuntu@ubuntu:~$
```

github.com/ros2/ros2cli/issues/529
Memory Map IO
Assembly Language
You should be able to overlay them. That is, make a workspace with your forked copy ros2cli, build that, and then source install/setup.bash that is generated in that workspace. You should now be able to your versions of the commands.

The final image shows the second Windows PC with the SSH to the second Raspberry Pi. As shown, currently, the message sends the pose data in chunks, rather than individual frames. This will need to be resolved in the future in order to better make use of the pose data being sent, as with this implementation, the data is in a format that is not feasible to parse through.

Societal Impact

FAA Regulations

The Federal Aviation Administration has several regulations regarding the flight of unmanned aircraft systems (UAS). Code of Federal Regulations Chapter 14 Part 107 is specific to the flight of small, unmanned aircraft systems (sUAS). When testing indoors the regulations aren't necessary to adhere to. However, when implementing a drone swam for use in public airspace, it is paramount that the proper actions are taken to follow the law placed by the FAA. A few key regulations that affect the project are:

- Visual line-of-sight (VLOS) only; the unmanned aircraft must remain within VLOS of the operator or visual observer.
- Small, unmanned aircraft may not operate over any persons not directly involved in the operation.
- A person may not manipulate flight controls or act as a remote pilot in command or visual observer in the operation of more than one unmanned aircraft at the same time.

To operate an autonomous drone swarm that adheres to FAA regulations requires a few things. A remote pilot in command needs to be assigned to each drone and able to take manual control of their specific drone in case of loss of autonomous function. The swarm cannot operate in overpopulated areas. The drones cannot leave visual line of sight of the pilots or visual observers. These restrictions limit the possibilities of autonomous drone swarm technology but are in place for safety.

The FAA has waivers for situations that can approve an operation that breaks certain regulations. The regulations are also able to be changed and have been modified over time. It could be possible to receive a waiver from the FAA allowing the use of autonomous drone swarm technology. Waivers have been offered in the past approving the operation of drone light shows that require the use of multiple autonomous drones that fly in a pre-defined path. As the technology develops, it is likely that the FAA will create new regulations or modify existing ones to accommodate the use of this technology.

Ethical Considerations

AI and Autonomous vehicles are a new and burgeoning industry that has seen rapid development over the last decade. Given the final implementation of this project is to be a fully autonomous drone swarm, some low-level AI will need to be implemented onto each drone in the swarm for the drone to control its own pathing. It is important to implement constraints into the drone swarm to ensure the protection and safety of any individual near the operational area of the swarm.

Autonomous drone swarms are a tool that, depending on the peripherals attached to the individual drones in the swam, can perform several different tasks. As with any tool, improper use or malicious intent can lead to drastic ramifications. It is important that there is oversight regarding the application of this technology. The FAA already provides oversight into the operation of sUAS and there is no doubt that the FAA will continue to provide this service. Engineers need to be mindful of the ramifications of their work and ensure that the public is informed on the technology they develop.

One thing that must be mentioned about the efficient scanning of areas utilizing this drone swarm technology is its application for security and surveillance. It is an inevitability for a technology like this to be used for a particularly potent form of enforcement, much more drastic than current implementations and may even rival the amount of privacy concerns that the current organization of the internet and its many social services enforce today, most notoriously social media. Other forms of this research are already especially overt about this aspect, such as the Drone Chasing project presented in another Capstone project. It is especially important to understand that despite the current presentation of this project as being mainly used for a case such as search and rescue, it is not and will not be the only form that the project will take.

To briefly describe the Drone Chasing project, it involves a drone that utilizes a camera fitted with image/object recognition that then follows said object around. It is currently advertised as a mechanism to help police track down alleged suspects with an automated approach. This is already a rather drastic form of technology to implement in the United States police's already increasing arsenal. If one were to imagine this project combined with the Drone Swarm project, the increase of the effects of the Drone Chasing project would be exponential. The amount of influence that a policing force could have on a society would go beyond concerning and would present major invasions of privacy arguably past what the NSA currently does in the digital world. This *must* be one of the main considerations when working on this project.

Benefits to Society

The biggest benefit of autonomous drone swarms is the rapid coverage of a large section of airspace. Search and Rescue operations would benefit most from this technology, were the difference in a few seconds could be life or death, being able to rapidly deploy drones over an area to map the terrain is a huge given the alternative manned options. Being able to minimize the time it takes to get airspace coverage streamlines airborne operations and cuts down on costs. If sustained complete coverage is needed, drone could automatically be swapped out to account for the drone's battery life.

Engineering Standards, Constraints, and Security

Software Life Cycle Process (IEEE Std 12207)

The team met with our sponsor, Dr. Lauf, typically on Monday afternoons. During this time, our team would ask Dr. Lauf questions regarding what software would be appropriate as well as give him updates on our findings. Dr. Lauf would also give us updates on the direction he thought this project should take. A few weeks into the semester, the school got access to the Optitrack Motion Capture System and had it installed around the drone cage. At this point, Dr. Lauf suggested we adjust our project goals to focus on closing the gap between the Optitrack Mocap API data and the PX4 flight controller using the Raspberry Pi as the companion computer between them. We then had to adjust our milestones and weekly schedule to include this change.

We then had to adjust the schedule again when the ROS 2 implementation took longer than expected. This was caused by conflicting supported versions. Mocap_optitrack, which is a package used to stream the rigid body data from Optitrack, was only supported through ROS version 1 with Ubuntu version 16.04 while the PX4 flight controller recommended using ROS version 2 with Ubuntu version 20.04. Both versions were tested in parallel on two Raspberry Pis, but this set back forced us to rework our schedule appropriately.

Quality Assurance Plan (IEEE Std 730)

The only requirements for this semester’s work on Drone Swarm was that the Raspberry Pi must be able to receive rigid body data from the Optitrack motion capture system and translate those messages into flight commands and/or informative topics that the PX4 flight controller could use. There were no specifications from our sponsor Dr. Lauf for operating system requirements or software requirements, however, he did offer suggestions in both these areas.

For this reason, we had to decide what operating system would be best for the project in the long term. First, we had to choose between Raspbian and Linux. Generally, Raspbian is the suggested operating system to be used on Raspberry Pis, and for that reason it offers several advantages. However, after attempting to install ROS on Raspbian version 11 (Bullseye), there were a lot of issues. ROS was only designed for older Raspbian versions like Jessie and the newer ROS 2 was designed primarily for Linux. For this reason, along with the fact that Linux can be run on any suitable micro-controller (in case a different companion is used in the future in place of the raspberry pi) we chose to run a Linux operating system.

The version and flavor of Linux had to be chosen next. This was done by installing ROS 2 on two different versions of Ubuntu: 16.04, which was supported by Optitrack, and 20.04, which was supported by PX4. Both were tested to determine which would be the better long-term solution. The 20.04 with ROS 2 was newer and had more updated functionality. For example, it uses Python 3 and targets C++14. However, if this version was not stable for all system components, then it would fail quality assurance. Similarly, if the version was too antiquated it may make it difficult to continue to build new technologies on it.

The process of choosing the right operating system and software version caused system testing to take longer but ensuring that future teams had the best foundation to build Drone

Swarm on was necessary considering the length of this project. Ultimately, we were able to successfully connect Optitrack with the newer ROS 2 and Ubuntu 20.04 using the original Optitrack_mocap package with some slight modification to how it is called. There was a GitHub for a new Optitrack_mocap package designed to be used with ROS 2, however, this proved to be unreliable at the present time. Perhaps with more testing this package can be used, but currently the original Optitrack_mocap package works very well connecting Optitrack and ROS 2 and is more reliable.

Testing was conducted in a results-based format. The requirement was that the Raspberry Pi must be able to receive rigid body data from the Optitrack motion capture system and translate those messages into flight commands for that drone. It was necessary that each Raspberry Pi on a drone would be able to subscribe to only its rigid body data and not receive data for all rigid bodies and then have to parse it before it is usable. For this reason, ROS 2 was used rather than using Optitrack's NatNet SDK directly which was tested first.

Verification and Validation (IEEE Std 1012)

Seeing that our semester project scope changed slightly part way through the semester, it was important to ensure that our work was keeping in line with these goals. This was done by remaining in contact with our sponsor, Dr. Lauf, about our progress and keeping him updated on what steps we planned to take next in case our plans were different than his expectations. We also continuously verified that our work was in line with our project goals. At the end of the semester before the showcase, we gave Dr. Lauf a final update on our work to ensure he was satisfied with it before we presented it at showcase.

System Requirements Specification (IEEE Std 1233)

Our requirements were:

(1) Use the Optitrack motion capture system installed in LARRI at the University of Louisville for locational data in place of GPS data.

(2) Use the PX4 flight controller to command the drone.

(3) Use a Raspberry Pi as the companion computer placed on-board of the drone with the PX4 flight controller.

(4) Implement a system with the capability to collect data from Optitrack onto the raspberry pi and forward this data and any relevant flight commands to the PX4 flight controller.

Software Design Document (IEEE Std 1016)

Research was the primary focus of this project. The work we completed this semester should act as the system design that future capstone teams will use. For this reason, this documentation along with the system diagrams included and the installation instructions in the appendices serve as the design documentation for this project. A formal design document was not created because the full design was not known until the end of the semester.

Constraints

Availability of Raspberry Pis to test with was a concern for this project. We did not get access to a Raspberry Pi until several weeks into the semester. Our work around was to use virtual machines until we got a Raspberry Pi. While difficult, there was a learning opportunity to understand the added complexity at the network layer in virtual machines. Time was another constraint to this project. There was a lot of research necessary to find the best long-term software to use in Drone Swarm, which left little time for development. Moreover, it was necessary to research previous capstone teams' work to be able to implement their portions into the design.

Another constraint, which was not a significant limitation this semester, but could be for future work, is that we only had access to one drone. It will be necessary to acquire more drones to test communication and flying formations. Space would also be an eventual concern as flying formations need to be tested with horizontal movement such as would be the case as the drone swarm is flying to its destination. However, safety is the primary concern because rogue drones could cause damage or injury. Therefore, it is necessary that testing be conducted in the small, confined space at LARRI to limit the risk of losing control of a drone. Before flight testing can begin outdoors, it will be necessary to include an override option so that someone can take control of the drone if something goes wrong.

Security

There are currently no security measures connected to the B.A.T.M.A.N. ad-hoc network. Therefore, it is exposed to attack. There are some security options available to add using batman-adv and these should be considered as the Drone Swarm system grows.

The B.A.T.M.A.N network setup allows for anything to join the network with no form of verification. Steps could be taken to create some form of handshake with the network before a node can gain access to it. This can start as rudimentary password entry, but as the project is used in real world applications, the security should be scaled accordingly as chances for attack are increased.

Another form of security is message encryption. While in testing, there would be little need for this sort of security measure to be implemented so long as no real, potentially sensitive information is involved. However, when the project is deployed in real-life scenarios, it is important to implement some sort of message encryption in order to prevent easy access of sensitive data from malicious third parties.

Recommendations for Future Work

Continuation Off Current Progress and ROS 2 Humble

The next capstone team to work on Drone Swarm should start by completing the connection between the Raspberry Pi and the PX4 flight controller. The MAVROS package is already installed and MAVLink is already set as the default communication link. Further configuration options can be found through the link listed in the appendices. After this communication has been tested, the team can work to separate messages from Optitrack's "Wide Max" rigid body data stream. Currently, every message sent from an Optitrack topic can be sent as one message over the B.A.T.M.A.N. network on termination, but this is not practical. One method for separating messages is to use the rclcpp or rclpy libraries. This is the method we attempted but were unable to complete it in time because it needed access to the rigid body message structure. Another solution would be to upgrade ROS 2 from Foxy to Humble. Humble is a newer, not fully supported version of ROS 2 which includes the option to send messages one at a time. This could be very useful for testing purposes but will almost certainly bottleneck the system if this is used in place of the streamed data. Next, the team should work on connecting the previous capstone team's flight pattern algorithm with flight commands based on locational data received through Optitrack.

ROS Kinetic and Ubuntu 16.04

Another option for the future capstone teams to take on would be to downgrade to ROS Kinetic Kame using Ubuntu Mate 16.04. External positional data is well documented with this environment and is better supported compared to utilizing ROS 2 setups. This may better guarantee flight of the drone and allow for testing using the Optitrack-sourced pose data. However, this would also mean having to work with more outdated features compared to what is provided with ROS 2, though this is mostly inconsequential from what the team has experienced.

The Difference Between Binary and Source Installations

For many Linux packages, there will be multiple ways to install certain packages. One through Binary, and one through Source. This is common amongst the ROS packages and will be relevant to both the setup process and future design changes to the setup in the future. Both have their merits and must be taken into serious consideration.

The most commonly known method of package installation is Binary. This is the method of installation often used with a package manager such as “apt” or “apt-get”. This is a rather simplified method of installing packages to a machine, as the files provided are already built with the said environment already in mind. This can greatly streamline the process of getting a desired package up and running on a machine without having to compile the source code into machine code. However, as the name of the process suggests, the code being binary will inevitably make certain packages unable to function in environments that it is not supported for, nor would it be possible to configure the files to be able to work on more specialized environments.

Source installations, on the other hand, are generally more uncommon. This involves taking the source code from a repository and compiling and installing it from scratch. This is a much lengthier process with extremely particular steps when not followed, which would cause the building to fail, not to mention that building processes may take a very long time to complete. Essentially, the source installation process is much more difficult than binary installations. That being said, source installations are also much more flexible to configure. There is much more freedom to configure the files to specific environments that aren't initially supported for the source code. This can make it so that packages not initially supported for perhaps a specific OS can be reconfigured to function properly. Furthermore, any custom changes to the package can also be easily made by modifying the source code directly and rebuilding the packages. This can resolve shortcomings/glitches in the original package, and even allow for the creation of new functionality not initially provided by the project. One such example is the "--once" tag for the "ros2 topic echo" command.

Appendices

Project Time Table

Drone Swarm Spring 2023 Schedule	
Sprint 1 (Week 2 – Week 3)	Project Overview, Research of Previous Work, and Initial Presentation
Sprint 2 (Week 4 – Week 6)	Create B.A.T.M.A.N. and A.L.F.R.E.D. network in virtual machines. (Research other implementations like Docker)
Sprint 3 (Week 7 – Week 9)	Collect streaming data of rigid bodies from Optitrack, Midterm Presentation
Sprint 4 (Week 10 – Week 13)	Install ROS 2 and use it subscribe to message data of Optitrack rigid bodies
Sprint 5 (Week 14)	Research and start set-up of ROS 2 to Pixhawk flight controller connection
Sprint 6 (Week 15)	Create start-up script and image, Showcase

Software Installation Instructions

Build from Image

An image of the OS has been created at the end of the semester. This image can be used to flash a new version of the OS in the state it was in at the end of the semesters work. The image has been uploaded to the projects GitHub for future teams to use.

Manual installation

Start with imaging Ubuntu 20.04 (LTS at the time) onto a micro SD card for the microcomputer (Raspberry pi). To get ROS2 onto the machine, install it with the Debian packages by adding the ROS2 apt repository to the sources list.

```
sudo apt install software-properties-common
sudo add-apt-repository universe
```

Need to get a GPG key for authenticating with ROS servers to get the sources list.

```
sudo apt update && sudo apt install curl -y
sudo curl -sSL
https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -
o /usr/share/keyrings/ros-archive-keyring.gpg

echo "deb [arch=$(dpkg --print-architecture) signed-
by=/usr/share/keyrings/ros-archive-keyring.gpg]
```

```
http://packages.ros.org/ros2/ubuntu $(. /etc/os-release && echo
$UBUNTU_CODENAME) main" | sudo tee
/etc/apt/sources.list.d/ros2.list > /dev/null
```

Now that the sources list is updated, use apt to update and upgrade all the packages to ensure all software is up to date. Using apt install the ROS foxy (desktop) and the ROS developer tools. The mavros and mavros-extras packages should also be installed. When installing any ROS packages using apt, the naming convention is ros-<distro>-<package name>

```
sudo apt update && sudo apt upgrade
sudo apt install ros-foxy-desktop ros-dev-tools python3-
argcomplete
sudo apt install ros-foxy-mavros ros-foxy-mavros-extras
```

Create the director for the ROS workspace. This workspace will hold all the packages that can't install via Debian packages. This workspace will serve as an overlay to the base ROS installation. The chmod command gives read, write, and execute privileges for the workspace to all users so that colcon can build the packages in the directory.

```
sudo mkdir -p ros_ws/src
cd ros_ws
sudo chmod 777 -R .
```

Clone the following ros2 packages into the source folder of the workstation.

```
cd src
sudo git clone https://github.com/ros-
drivers/mocap_optitrack.git
sudo git clone https://github.com/PX4/px4_ros_com.git
sudo git clone https://github.com/PX4/px4_msgs.git
```

navigate into the mocap_optitrack package. In the config folder for the package, edit the mocap.yaml file to work with the current network setup and mocap streaming setup. The mocap.yaml file sets up the ROS parameters for the mocap node. The rigid bodies parameter sets up the topics that will be published that correspond to the rigid body data in motive, the number is equal to the rigid body identifier in motive. The only topic that needs to be published is the pose topic; listing the other is optional and not listing them just means those topics won't be published. In the optitrack_config parameters, the multicast address needs to be the same as the multicast interface in the motive streaming options (the default being 239.255.42.99). The command port and data port also need to be set respectively.

```yaml
mocap_node:
    ros__parameters:
        rigid_bodies:
            1:
                pose: "Robot1/pose"
                pose2d: "Robot1/ground_pose"
                child_frame_id: "Robot1/base_link"
                parent_frame_id: "world"
            2:
                pose: "Robot2/pose"
                pose2d: "Robot2/ground_pose"
                child_frame_id: "Robot2/base_link"
                parent_frame_id: "world"
        optitrack_config:
                multicast_address: "239.255.42.99"
                command_port: 1510
                data_port: 1511
```

Before building the workspace, it is important to source the underlay ROS installation so that the built in ROS commands can be used. After sourcing the underlay, rosdep is used to install any missing dependencies of the packages in our src folder. The Colcon build command builds the ROS packages in the src folder.

```
cd ~/ros_ws
source /opt/ros/foxy/setup.sh
rosdep install --from-paths src -i
colcon build --symlink-install
```

After successfully building the workspace, the overlay needs to be sourced so the ROS commands can see the installed packages in the workspace. The command is similar to the previous source except for the location of the setup file, which is in the install folder of the workspace.

Instead of launching each ROS node one by one, A launch file can be used to launch all our needed nodes simultaneously. In the ROS workspace create a file done.launch.py. The launch file should look like such:

```
import os
from ament_index_python.packages import
get_package_share_directory
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='mocap_optitrack',
            executable='mocap_node',
            name='mocap',
            remappings=[
                ('/Robot1/pose', '/mavros/mocap/pose'),
            ]
            parameters=[
                os.path.join(

get_package_share_directory('mocap_optitrack'),
                    'config',
                    'mocap.yaml'
                )
            ]
        ),
        Node(
            ...
        ),
    ])
```

The drone.lauch.py launches the mocap_optitrack package. The remappings argument remaps topics from the packages to other topics that might exits. The The parameters argument sets the ROS parameters of the node to launch, the mocap.yaml file has the parameter settings for the node and here it is used to set the parameters as well. As many nodes as needed can be added to the LaunchDescription. The launch file can be used with the following command

```
ros2 launch drone.launch.py
```

Batman commands:

```
sudo apt-get install -y batctl
sudo batctl if add wlan0
sudo ifconfig bat0 mtu 1468

sudo ifconfig wlan0 up
sudo ifconfig bat0 up
sudo batctl if add wlan0
```

Edit the contents of interfaces.d. It may be necessary to create this directory if it does not exist. The net-tools package is also required.

```
sudo nano /etc/network/interfaces.d/wlan0

auto wlan0
iface wlan0 inet manual
    wireless-channel 1
    wireless-essid call-code-mesh
    wireless-mode ad-hoc
```

Alfred Setup:

```
git clone git://git.open-mesh.org/alfred.git
sudo apt install libnl-3-dev
sudo apt install libnl-genl-3-dev
sudo apt install libcap-dev
sudo apt install libgps-dev
make
make install
sudo alfred -i wlan0 -m
```

Send Messages on port n (65 is used as an example):

```
echo Hello World! | sudo alfred -s 65
```

Read Messages on port n

```
sudo alfred -r 65
```

For more information on B.A.T.M.A.N. and A.L.F.R.E.D. see the GitHub repo created by the Spring 2022 networking capstone team:

https://github.com/bytemaster0/batcloud-networking/tree/1993643dc72440dc9bae285d0a6f3a38469e8b37

To start the drone, a remote connection can be established, such as SSH, in order to run the commands necessary. PuTTY was the SSH software of choice for the team to utilize, though, other forms of SSH such as Tera Term can function just as well.

Sources

# *Ros 2 user guide*. ROS 2 User Guide | PX4 User Guide. (2023, April 14). Retrieved April 25, 2023, from https://docs.px4.io/main/en/ros/ros2_comm.html

Ermakov, V. (2016, June 23). *Wiki*. ros.org. Retrieved April 25, 2023, from http://wiki.ros.org/mavros_extras

Ermakov, V. (2017, May 12). *Wiki*. ros.org. Retrieved April 25, 2023, from http://wiki.ros.org/mavros_msgs

Ermakov, V. (2018, March 3). *Wiki*. ros.org. Retrieved April 25, 2023, from http://wiki.ros.org/mavros

Gräve, K., Bencz, A., Baltovski, T., & Yamokoski, J. D. (2014, February 26). *Wiki*. ros.org. Retrieved April 25, 2023, from http://wiki.ros.org/mocap_optitrack

S. Macenski, T. Foote, B. Gerkey, C. Lalancette, W. Woodall, “Robot Operating System 2: Design, architecture, and uses in the wild,” Science Robotics vol. 7, May 2022.