Addressing Problems Facing Unmanned Aerial System Scheduling Systems in Urban Environments

by

Daniel D'Souza

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved March 2018 by the
Graduate Supervisory Committee:

Sethuraman Panchanathan, Chair
Spring Berman
Yu Zhang

ARIZONA STATE UNIVERSITY

May 2018

ABSTRACT

Research literature was reviewed to find recommended tools and technologies for operating Unmanned Aerial Systems (UAS) fleets in an urban environment. However, restrictive legislation prohibits fully autonomous flight without an operator. Existing literature covers considerations for operating UAS fleets in a controlled environment, with an emphasis on the effect different networking approaches have on the topology of the UAS network. The primary network topology used to implement UAS communications is 802.11 protocols, which can transmit telemetry and a video stream using off the shelf hardware. Other implementations use low-frequency radios for long distance communication, or higher latency 4G LTE modems to access existing network infrastructure. However, a gap remains testing different network topologies outside of a controlled environment.

With the correct permits in place, further research can explore how different UAS network topologies behave in an urban environment when implemented with off the shelf UAS hardware. In addition to testing different network topologies, this thesis covers the implementation of building a secure, scalable system using modern cloud computation tools and services capable of supporting a variable number of UAS. The system also supports the end-to-end simulation of the system considering factors such as battery life and realistic UAS kinematics. The implementation of the system leads to new findings needed to deploy UAS fleets in urban environments.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

MOTIVATION

Currently, there exists a gap in research for the considerations of deploying tasks to a UAS fleet in an urban environment. While existing research gives plenty of considerations to techniques for deploying UAS fleets and swarms, restrictive legislation has made it difficult to test these techniques in an urban environment.

My goal for this thesis is to explore different design options for building a scalable fleet management system for drones. The system should will use modern technologies for building secure, scalable systems and remain upgradable as UAS capabilities and legislation change. The system should also be transferable to other campuses and similar urban environments. Further, having secured the necessary permissions, this thesis will test different hypotheses flying an UAS in an urban environment.

UAS DESIGN

I have participated in remote control aviation since I was in elementary school. At the time, AM transmitters were still considered sophisticated, and brushed motors with NiMH batteries were the dominant technology for all electric planes. The power-to-weight ratios were much tighter, limiting the options for electric models. Thankfully I never had to fly a model on Nickel Cadmium (NiCad) batteries.

Model aviation has come a long way since then. Airframes composed of Styrofoam strengthened with carbon fiber are commonplace, and custom models with 3D printed parts or fiberglass are readily accessible for the amateur hobbyist. Advances in battery technology and brushless motors make choosing a power system as simple as opening an online calculator and your wallet. The combination of light airframes and powerful drive

systems makes it easy to build models with a high power-to-weight ratio. And advances in radio, sensor, and microcontrollers has been the driving force behind increasingly complicated models, capable of fully autonomous flight from takeoff to landing. Currently, the biggest limitation to amateur model aviation is government legislation, designed to prevent model aviation from interfering with commercial operations. If more companies keep refining their technology, and legislation remains a barrier, the number of viable options for UAS components will increase.

Choice Of Autopilot And Flight Stack

The autopilot takes input from its sensors and generates outputs to motor controllers. It is the most critical component of a modern UAS. For our UAS, we choose to use the Pixhawk 2.1, herby references “the Cube.” Why did we choose the Cube? The Cube, the latest iteration in the Pixhawk family, is one of the most advanced autopilots available to amateur pilots. Capable autopilots can be compared by two qualities, how good the sensors are, and how reliable is the system. In the case of the Cube, the combination of gyroscopes and accelerometers allows the autopilot to track the motion of the UAS in 6 degrees of freedom. The Cube has other sensors, including a barometer and Global Positioning System (GPS), which enable it to track its absolute position. As far as reliability, the Cube has a redundant microcontroller and triple redundant sensors. In the case that one set of sensors does not report the same as the others, the Cube will compensate and raise an error.

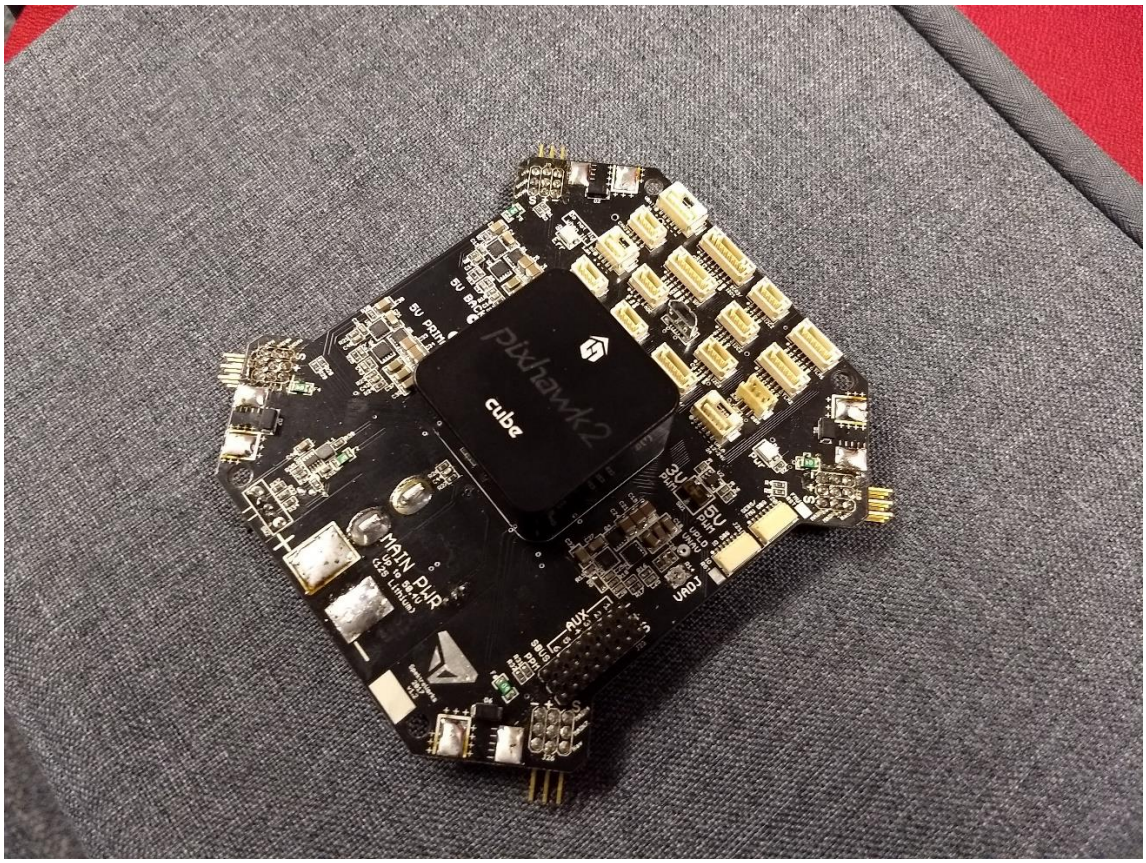

Figure 1: Pixhawk Cube Mounted Onboard a Spektreworks Carrier Board.

An alternative to the Cube would be Intel's Aero dev kit. This board combines a PX4 based autopilot like the Pixhawk with an onboard computer. However, Intel's dev kit does not have the same redundancy of the Cube and given that most of the software and connectors are proprietary, it is not a good choice for a custom platform. Intel has also been discontinuing its old hardware. Considering that, it would be better to choose the Cube, which is certain to be supported.

Choice OF ONBOARD COMPUTER

A drone's onboard computer is the second most important consideration in designing a fully autonomous UAS. Three considerations must be taken when choosing a board.

First, the board must be powerful enough to support the tasks required. Second, the board must have enough connectivity to handle a variety of networking and sensor configurations. Third, the board should not draw a large amount of power, wasting the precious battery power as heat.

Raspberry PI

The first UAS prototype only had to update its position and point its gimbal at a target's location, as given by GPS coordinates. The quad-core CPU and one gigabyte of RAM on a Raspberry Pi model 3 was sufficient for the task. For networking, the UAS received instructions and sent back diagnostics and video to the ground station through 802.11 based communication. A USB camera was housed inside the gimbal. The GPIO and USB ports on the Raspberry Pi model 3 handled this as well. Finally, the Raspberry Pi could be powered with as little as 500 milliamps – a power efficient solution.

Figure 2: Raspberry Pi Mounted Onboard the First UAS Prototype

However, when it came to extend the system, the Raspberry Pi fell short. Video encoding was done on the Raspberry Pi's CPU. Decoding the video stream and object-recognition was below our target framerate of 30 frames per second. When the CPU usage hit 100 percent, the software that provided instructions to the autopilot stalled as well, because the implementation of networking on the Raspberry Pi is directly tied to the CPU through USB. For the second revision, we decided to upgrade our onboard computer.

With System on a Chip (SOC) platforms, originally designed for mobile phones, becoming available as Single Board Computers (SBC), there are many options to choose from. Although the Raspberry Pi is the most popular due its low cost, well documented

hardware, and large number of software packages, there are many options with more powerful CPU and GPU options, addition options for expansion through PCIe and FPGAs, and more advanced networking options.

Jetson TX1 AND TX2

The first platform we considered was Nvidia's Jetson platform, specifically its newest TX2 revision. The Jetson TX2 itself is a module – it needs a breakout board to connect peripherals before it can be used. The Jetson TX2 module provides a quad-core CPU, and an Nvidia GPU. This means that it can run advanced computer vision and deep inferencing workloads on the GPU, much faster than any other board. The Jetson TX2 is closer to a laptop than a mobile phone – and its connectivity is limited by how much you are willing to pay for a carrier board. It features a PCIe bus, mainly to support advanced networking options, like high powered 802.11 modems or cellular modems. For computer vision, the Jetson TX2 has enough CSI lanes to attach multiple cameras, and dedicated hardware to support video encoding and image processing. Of course, USB cameras will still work provided that the carrier board has enough USB ports, or an external hub is used. Finally, power considerations. Given a 19V input, the Jetson TX2 draws a maximum of two amps under full load. While we didn't test with peripherals, we could expect it to draw close to three amps. I would consider this midrange power utilization as far as onboard drone computers, although the Jetson TX2 sets the bar for performance to power ratio among comparable systems. So, is using the Jetson TX2 is as simple as picking a carrier board?

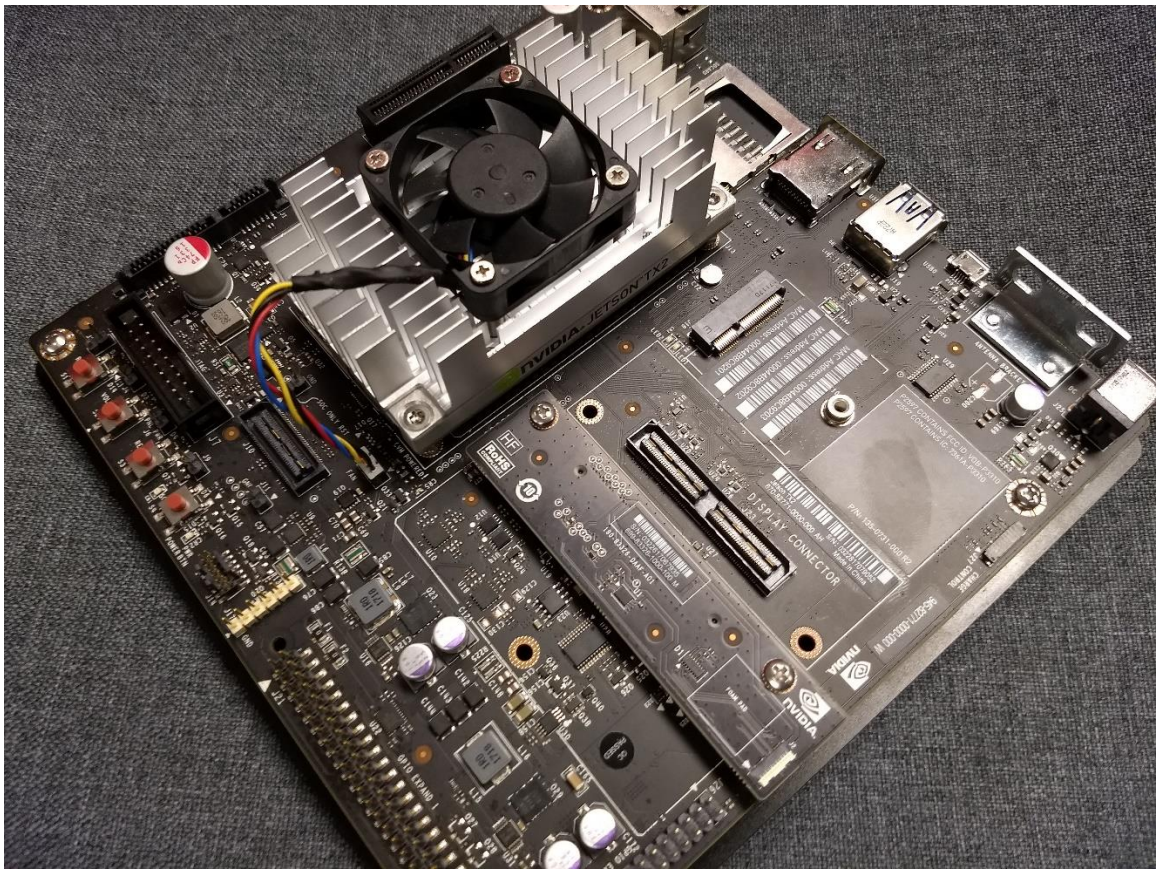

Figure 3: Nvidia's Jetson TX2 Mounted on a Development Carrier Board.

Unfortunately not. While working with the Jetson TX2, we discovered that there were too many problems for us to solve. And given how poorly documented base software was – I doubt that I could solved any of the them in a timely manner. As evidence against using the Jetson TX2 as the onboard computer mounted, the decision was made for us when someone incorrectly powered on the Jetson TX2. The carrier board was unsalvageable. Rather than replace it, we decided to pursue other options.

Rock64

The next SBC we pursued was the Rock64, a more powerful version of the Raspberry Pi. At a similar price point to the Raspberry Pi, the Rock 64 includes four gigabytes of RAM

and a RK3328 Quad-Core ARM Cortex A53 64-Bit Processor. Besides the additional processing power, the biggest difference between the Rock64 and the Raspberry Pi was the inclusion of a USB 3.0 port. At some point in time, we decided that a depth camera to serve as the primary sensor for obstacle avoidance. We decided on the Intel RealSense camera, mainly because of the small form factor and long range. More on that later. The Rock 64 consumes at most two amps under load from the 5V rail without peripherals. While it uses more power than a Raspberry Pi, the performance to power ration is similar. It still doesn't hold a candle to the Jetson TX2 comparing available computer power. The Rock 64 is pin compatible with the Raspberry Pi, so it still meets our connectivity needs. However, the Rock 64 uses a 64-bit CPU – and different distribution of Linux.

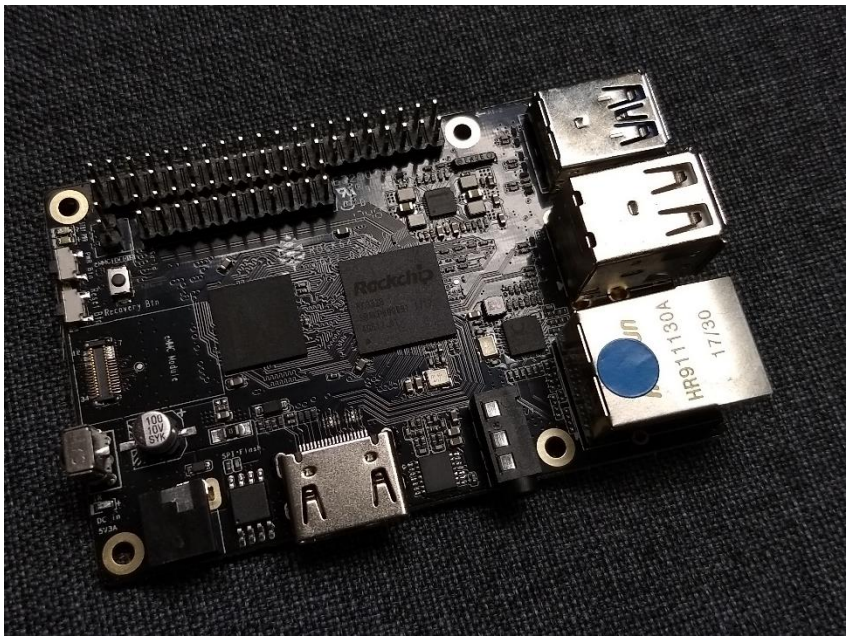

Figure 4: Rock64 Single Board Computer

The Rock64 uses the same architecture as the Jetson TX2 – arm64, so our code was easily compatible. Intel's RealSense library, however, was not. Unsurprisingly, Intel only

provides compatible packages for x86 systems, and all the build scripts are written assuming an x86 distribution of Linux. After recompiling the Rock64's kernel and trying several permutations of libraries to get modules to build correctly, we finally received data from the camera. But it was corrupted. Having spent a week troubleshooting instead of developing software – we decided to find a different SBC.

At this point in the project, the hardware and component layout on the UAS was close to finalized. The UAS communicates over 4G communication and 900 Mhz fallback sending instructions and receiving telemetry. The UAS also streams back video from an Intel Realsense and two additional cameras. These requirements started to restrict our options for SBCs.

UP Board and UP Squared

The UAS currently uses a Up Squared board. The Up Squared is based on a modern quad-core Intel Pentium chip. This means that we can run any x86 based operating system, and use any package compiled for x86, making the target environment the same as the development environment. As a result, our software, the Intel RealSense and the 4G modem worked perfectly on the first try. The Up Squared has similar connectivity to the Jetson TX2. There is a mini PCIe port for the 4G modem. For communication, there is a slurry of GPIO for connecting the Pixhawk autopilot and 900 MHz modem. Plenty of USB ports for the Intel RealSense and other cameras. The Up Squared even provides SATA connections for a full 2.5 inch SATA Drive, and a mSATA slot for SSDs. These forms of storage are faster and more reliable than the included eMMC flash memory or a microSD card. The UP Squared seems like the perfect board, right?

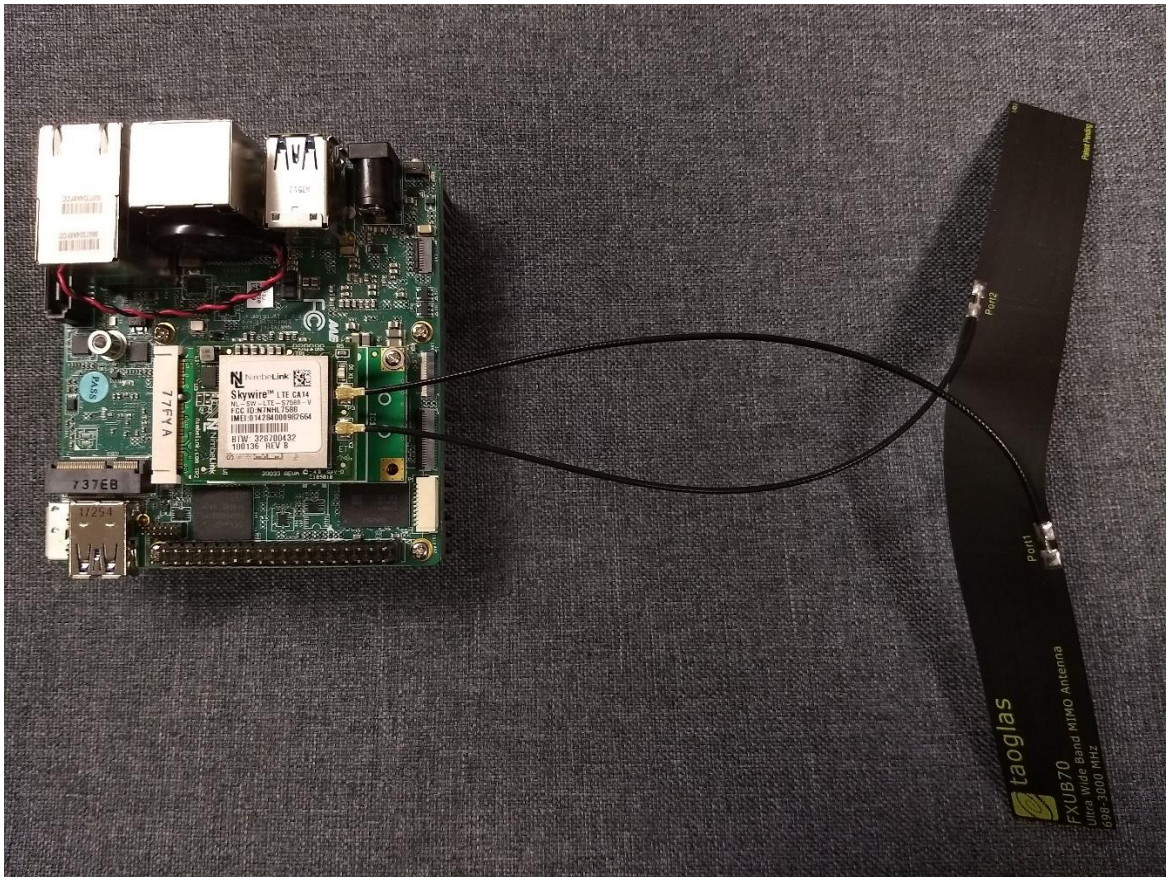

Figure 5: UP Squared Single Board Computer, With 4G LTE Module and Antenna

The drawback to the UP Squared is its power consumption. Under full load, it draws almost three amps, and close to four amps with all the peripherals. It draws this much power, yet the performance is comparable to the Rock64. Where does the extra power go? It is converted into heat – the Up squared requires a heatsink to function properly – which adds extra weight to the UAS. Although the extra weight and higher power draw reduces the flight time – it is a compromise we have to make in order to achieve our goals for the project.

Other Single Board Computers Considered

Another compute solution commonly used onboard UAS is the NXP family of microcontrollers. The advantage of using the NXP system, is that the hard work of selecting compatible components and writing compatible software has already been completed. This allows the customer to pick whichever combination of video encoders, wireless modems, cameras and other sensors they need to work together, and incorporate it as a production-ready single board computer. The advantage of using the NXP ecosystem over similar ecosystems, is that NXP offers high performance CPUs like the Rock64 capable of difficult image processing tasks. The 3DR Solo UAS utilizes NXP components to provide wireless video streaming from a GoPro action camera, and an interface to the onboard Pixhawk flight controller.

However, the customer must design their own carrier board to interface their chosen components together. For student developers and for the scope of this thesis, designing a carrier board would be too difficult. The second drawback is the cost. NXP solutions are designed to be used in commercial products and are priced accordingly; licensing their hardware is outside our budget.

Texas instruments makes another popular single board computer series, based on their OMAP processors. The most well-known is their BeagleBone series. However, BeagleBone single board computers with their extended GPIO are better suited for applications where direct access to hardware and sensors from the operating system is required. Their CPU performance is not comparable to the current Raspberry Pi. Because we use the Pixhawk Cube to interface with all sensors and other hardware, our UAS

would not benefit from extended GPIO onboard the UAS, making single board computers with more compute power a better choice.

Choice Of Camera Systems

The first iteration of the camera system onboard the UAS is a two axis gimbal. The gimbal oriented the camera with pitch and roll axes, and the UAS yaw axis. By controlling the gimbal axes, the UAS could observe any area under the UAS. This gimbal houses a 720p USB camera, provided a 16 by 9 aspect ratio with a 70-degree field of view. The gimbal also houses a LED for illuminating the vision target at night.

Figure 6: Two-Axis Gimbal from Previous UAS

A gimballed observation platform onboard a UAS is ideal for focusing on a capturing a specific target, however is not very useful for general observation. A 360-degree view around the UAS allows the operator to observer more of the surroundings. If the UAS rotates in place, it is possible to capture static images from 360 degrees and stitch them together. However, the UAS would take several second to generate a single frame – streaming a 360-degree video would be impossible with such a system.

Recently, 360-degree action cameras are starting to become popular. These cameras are equipped with two cameras behind wide angle lenses. Because the lenses and cameras are fixed relative to each other, dedicated hardware can handle a static image warp and stitch operation and pass the new frame to a hardware encoder. This pipeline can also be implemented in software, at the cost of needing a relatively powerful GPU to process 720p video in real time.

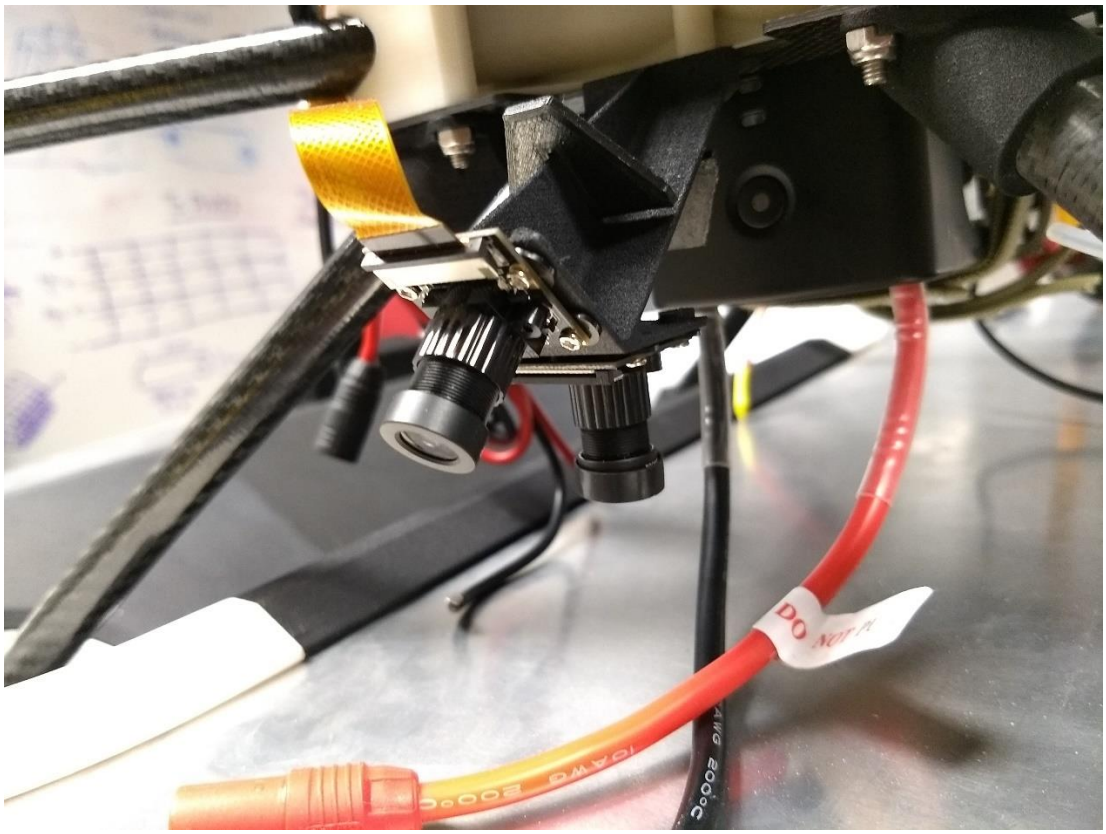

Figure 7: Front and Downward Facing Cameras Mounted On The UAS

Implementations of 360-degree surveillance can take advantage of the fact that the operator is more interested in what is below the UAS than what is above it. Rather than needing to capture a sphere of video, the cameras system can capture a 180-degree inverted dome instead. The reduced area can be covered with a single camera with a fisheye lens. However, our UAS uses two cameras with a less extreme lens to see around the UAS' landing gear.

SYSTEM NETWORKING

802.11

802.11 based communication is the first protocol we considered, as it presented the widest variety of options to structure our network. Hierarchical 802.11 networks are well documented, and there is plenty of existing hardware and software implementations that we can use off the shelf (Kumar et al., 2008). It would be trivial to attach our hardware to the existing Arizona State University campus network, making deployment simple (Parker, 2008). 802.11 has built in security options that would make it difficult to breach without physical access to our network infrastructure.

Campus Wireless Network

In our first attempt, we attempted to connect directly to the Arizona State University campus network. For our tests, we connected as a student. I am not aware of any difference between signing in as a student or signing in as a faculty member. As far as I can tell, students have the same access to any resources on the network, because we can access anything on the subnet. Our testing focused on areas while the drone would be flying, albeit from ground level. Testing was conducted by walking around campus with a mobile device, monitoring signal strength and ping times to a local server on the network.

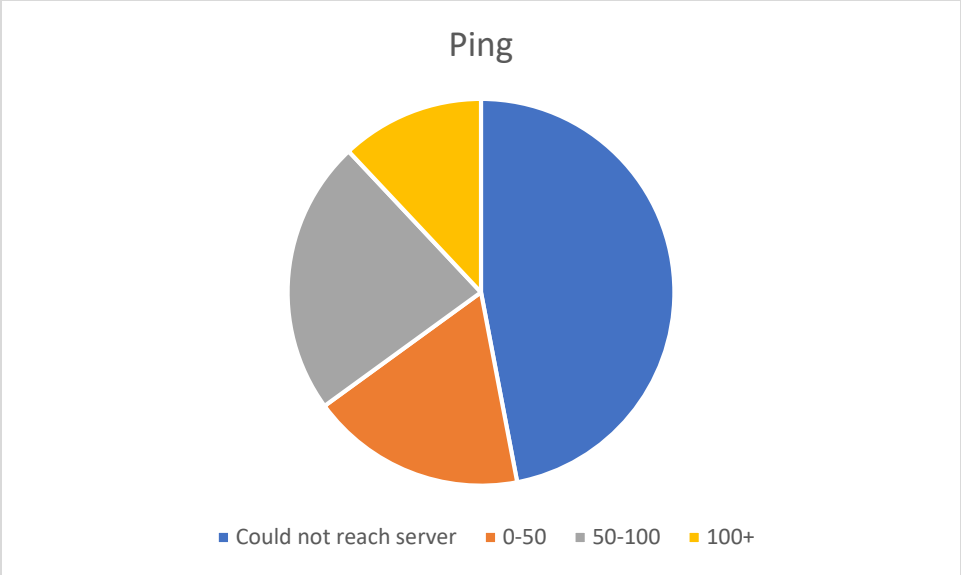

Figure 8: Result Of 100 Pings Over the Campus Wireless Network

Leg	Total Pings	Unsuccessful pings	Successful pings	Average Successful Ping	Reconnects
Leg 1	100	47	53	72	3
Leg 2	100	82	18	81	4
Leg 3	100	36	64	60	2
Leg 4	100	28	72	75	1
Leg 5	100	53	47	83	3

Table 1: Statistics for The Wireless Campus Network Test

As a result, I deemed that using the campus wireless network to be unfeasible. There are several explanations that contribute to our results. The first is access point switching. The campus network is set up with an overlapping set of access points. However, a mobile device can only connect to a single access point at a time. While your device can establish a connection from anywhere in range of any access point, the connection is

briefly lost between the time you leave the range of one access point and reconnect to the other. The result is that the device stops its session with the server, and the device must reestablish the session in software.

While reconnecting is easily doable, it is impossible to determine for how long the connection would be lost. The reconnection time is highly dependent on the load to the access point. Take for example, an access point near a crowded campus coffee shop. The mobile device will try to connect to the closest access point with the strongest signal. However, the handshake may take a few seconds to complete while the access point is handling the requests from dozens of other devices. The mobile device would only try to connect to a different access point in range if the first connection fails, and it could easily be 30 seconds before a successful connection is made and the mobile device can talk to the network. This unpredictably long delay is unacceptable, and not conducive to providing instructions and receiving constant telemetry to a UAS.

Of course, switching between access points implies that there are multiple access points to connect to. A bigger problem is locations where there is no wireless coverage provided by the campus network at all! The solution is to add another access point, right? Adding additional access points is easier said than done. That would require adding additional wired infrastructure for an access point or wireless repeaters to cover the affected location – a cost prohibited approach for large areas like the Arizona State University campus. Second, finding every dead spot in the campus is a daunting task. This approach does not scale very well, finding every dead spot over a large area like a city would take a considerable amount of time and manpower. Another UAS could be constructed to map

these spots, but that is beyond the scope of the thesis. Suffice to say, achieving full wireless coverage over an urban area is an unsolved problem compounded by the relatively short range of 802.11 systems.

Further problems for using the campus network to network the UAS lie in the design of the campus network, specifically in the layout for the access points. The first problem is that the location of the access points – they are all indoors. Given that our UAS will be flying outdoors, there will never be a direct line of sight to an access point, and the wireless signal must travel through a brick or concrete wall. These materials are not transparent to RF signals. This means that our drone will always have a suboptimal connection to an access point. This would not be a problem in itself – after all, mobile devices are still able to connect to the network. However, obstructions to the wireless signal give the coverage field an irregular shape. The irregular shape and consistently weak connection amplifies the access point disconnection point. Even though the mobile device could be moving through the theoretical range of an access point, a RF opaque material could disconnect the mobile device and force it to reconnect to a further access point.

Another reason the design of the campus network is infeasible for UAS control is the design and direction of the antenna. Modern access points employ beamforming – a technique that improves the connection quality to distant devices employing directional antenna. Beamforming allows access points to support many devices across a larger area – simply add more antenna. However, these antenna arrays are mounted on the ceilings of lecture halls and classrooms and are pointed *down*. Our testing of the campus wireless

network signal strengths is conducted at ground level – not the ideal place for a large UAS to fly. I suspect that if we carried out our testing at an altitude of 10 meters or higher, the wireless signal quality of the campus network would deteriorate further.

It is safe to conclude that utilizing an existing campus wireless network to control a UAS is a flawed idea. The system is optimized for stationary devices like laptop computers that can maintain a constant connection without worrying about switching access points. User devices will always be located beneath an access point, the optimal location for directional antennas used in beamforming devices. Finally, the design of the campus wireless network assumes that most devices will be located inside a building, and access points are placed accordingly.

Access Point Mode

One possible solution is to complement the existing wireless network with another set of access points targeted at UAS (Spanos et al., 2004) (Cortes et al., 2004). This implementation would consist of long range radios, possibly mounted on towers. Depending on the layout of the urban environment, directional antennas could be employed to improve signal connectivity. Compared to omni-directional antennas, directional antennas require planning. It is easy to have dead spots with no coverage if planners do not consider where antennas should be located. Given a topographical map of the environment and the specifications of the access point, it would be possible to design an algorithm that can calculate the best placement of access points and antenna orientations in order to ensure coverage of an urban environment, however that implementation is outside the scope of this thesis.

Besides the cost of setting up a new network, the interference to existing wireless networks in the 2.4 GHz band is a concern. The 2.4GHz band is spread into 14 overlapping channels that the access point can broadcast on. When two access points broadcast on close channels, the interference significantly impacts the signal quality, leading to a massive number of dropped packets. The same applies when to broadcasting on close bands. Conventional knowledge says that a good balance between coverage and signal quality is three overlapping access points. Introducing an additional access point will degrade the existing campus wireless network, and the drone wireless network will operate sub optimally due to wireless interference.

The prohibitive cost of scaling up a dedicated network and the harmful interference it gives off in the 2.4 GHz spectrum makes setting up such a network a hard sell for a scalable UAS system.

Mesh Network Through Ad-Hoc Construction

The opposite approach to a hierarchical network of access points would be a mesh network (De Gennaro et al., 2006). In such a mesh network, the different nodes would be the base stations and the drones. Given the construction of the base station, it is safe to assume that base stations would remain at fixed locations, while drones would remain mobile. Messages would originate from various locations, and travel to the addressed drone hopping from neighboring node to node.

Dedicated hardware for mesh networking is available. However, radios designed for mesh networking and suitable in size, power, and bandwidth requirements to allow mounting on a UAS are prohibitively expensive. We were able to borrow a pair of mesh

networking radios for use on ground vehicles. These radios were not open source, and were prohibitively expensive to configure, and we stopped pursuing hardware solutions to mesh networking.

A facsimile to a hardware mesh network can be constructed over an 802.11 network. In our “software-defined” mesh network, a single board computer with an 802.11 adapter broadcasts on a given SSID and wireless “cell,” acting as a node. Nodes are responsible for maintaining a list of reachable neighbors (Malpani et al., 2000). Before opening communication, the source node uses these nodes to search for a path to the destination node. However, the topology of the network can change (Olfati-Saber et al, 2004). If nodes are added or removed, or a mobile node moves out range of its neighbors, the topology of the network will need to be recalculated. Depending on the complexity of the software routing setup, the mesh network can be made robust to a changing topology (Michael et al., 2010).

For testing our ad-hoc mesh network, we need several computers with 802.11 capabilities. We selected an assortment of Raspberry Pi model 3s and Raspberry Pi Zero Ws for the task. Although we had a supply of Raspberry Pi waiting for other projects, these single board computers are cheap at around $35 to $40, and the Raspberry Pi Zero W boards available for $10. An alternative single board computer would be the Orange Pi versions of the model 3 and Zero W with similar CPU and networking capabilities, however we did not have them in stock. Each Raspberry Pi was configured with a fresh install of Raspbian, a Debian-based Linux distribution with packages designed to target GPIO and other hardware found on many single board computers. Each Raspberry Pi had

a static address assigned to its wireless interface, and the address of every other Raspberry Pi was explicitly written on each Raspberry Pi. While it was possible to have each Raspberry Pi enumerate itself automatically, we chose to manually enumerate each Raspberry Pi in order to solve a routing issue where Raspberry Pi A could talk to Raspberry Pi B, and Raspberry Pi B could talk to Raspberry Pi C, but Raspberry Pi A could not talk to Raspberry Pi C with Raspberry Pi B as an intermediate node. After that each Raspberry Pi was configured to bring up an unsecured wireless network with the same SSID. Attempts to secure the wireless ad-hoc network were met connection issues, so falling back to an unsecured wireless network was necessary.

Number of nodes	Latency (average ping)	Single camera stream	Latency under load
2	5 milliseconds	10 frames per second	20 milliseconds
3	8	5	74
4	12	2	93
5	13	0 (could not stream)	135

Table 2: Statistics for The Mesh Network Test

Testing our Raspberry Pi mesh network revealed two major problems with network latency and throughput. As expected, the latency increased as the number of nodes increased. What we did not expect was how degraded the Raspberry Pi's video streams were. With only one camera streaming over the mesh network, a 720p stream could

barely reach five frames per second. Adding addition streams killed the network throughput entirely. However, there is an obvious cause to the low throughput – the implementation of networking on the Raspberry Pi. The Raspberry Pi's networking abilities are directly tied to its CPU – if the latter is overload then the former slows down to a crawl. The solution would be to use a board with a dedicated network controller, but we did not have enough such computers with that capability.

Should we implement a mesh network in software, we would have to face the same problems with network latency. For sending instructions and telemetry, low network latency is not critical. However, throughput becomes a much bigger problem. In a mesh network, the video stream would likely travel over several nodes before it reached a hardline connection, a sink. In a network with few mobile nodes, the chance that multiple streams would be routed through a single node increases, which would overwhelm the throughput capacity of the node and add a huge amount of latency to all packets entering and leaving the node.

A solution to the throughput problem would be organize the network such that nodes are not overloaded and allow the source to choose the best route to the video sink by measuring throughput and latency in real time. Algorithms to solves this problem are well documented, however make a critical assumption. These algorithms only operate to solve a consensus or coverage problem, rather than specific tasks which must be accomplished by one agent in the network. Allowing agents to accomplish their tasks while maintaining such a graph is impossible, without adding additional UAS or base stations nodes as routers for the UAS working on a task.

So where does that leave mesh networks? Mesh networks will undoubtedly play a critical role in networking the autonomous fleets of the future. For current implementations of autonomous fleets, the reliability and value of off-the-shelf hardware and software for hierarchical networks offset the setup costs of a hierarchical network, compared to the unreliable performance of mesh networks. For mesh networks to be a compelling option, the cost of and usability of open-source radios become more reasonable, and algorithms to schedule tasks and maintain a stable network topography must be proven.

900 MHz

So far, we have only considered 802.11 protocols, which operate in at 2.4 GHz. Could there be a better solution in older technologies? 900 MHz radio modems have been commonly used as the telemetry link in radio control vehicles – there is enough bandwidth to pass a serial connection through.

One benefit to using 900 MHz radios is the broadcast range. With an inexpensive transmitter, receiver, and antenna, it is easy to push a kilometer in urban environments. Expand on hardware here.

However, 900 MHz is not without its downsides. The biggest limitation to communicating over 900 MHz is the throughput. In order to stream video at a minimum of 24 frames per seconds, the stream resolution cannot be higher than VGA.

The other downside to 900 MHz communication is the radio interference. Like technologies in the 2.4 GHz spectrum, 900 MHz is prone to interference from other electrical devices common to urban environments. While software can adapt for noise

with smarter protocols, retransmitted packets and protocol overhead further reduce the limited throughput of 900 MHz communication.

Figure 9: 900 Mhz Radio Used on The Ground Station

For this reason, 900 MHz is a good alternative to use as a fallback autopilot telemetry link. In the event the primary connection is lost, the autopilot can still communicate the status of the UAS over the fallback telemetry link. To this end, we stream autopilot data over the primary data link and fallback 900 MHz connection to maintain redundancy.

4G

With the advent of 3G came consumer development kits. However, 3G technology was designed to retrieve websites, not deliver a full multimedia experience. As a result, the

development kits focused on providing connectivity for embedded development boards to enable logging. Development kits for 3G are still available today – and are much cheaper than their 4G counterparts.

4G is broad description of several technologies – the latest being HSPA and CDMA.

Category	Downlink (Mbps)	Uplink (Mbps)	Common devices
CAT 0	1	1	IOT modems
CAT 1	10	5	
CAT 2	50	25	
CAT 3	100	50	
CAT 4	150	50	
CAT 5	300	75	
CAT 6	300	50	LG G4, Samsung Galaxy S6, Moto G5 Plus
CAT 9	450	150	Samsung Note 5, iPhone 7
CAT 12	600	150	iPhone X, Pixel 2
CAT 16	Gigabit	150	Galaxy Note 8, HTC U11, LG V30

Table 3: 4G LTE Modem Categories

Current State Of 4g Modem Support In Linux

Of course, just because 4G modem hardware exists – does not mean that software supports it. A device driver in the form of kernel support or a kernel module must be present. The available drivers differ between kernel versions, and new drivers are not backwards compatible with older kernels. So how do modem manufacturers make devices that work over multiple generations of kernels? The solution require flexibility on both the software (kernel) side, and the hardware (modem) side. In software, modern kernels ship with multiple drivers. How does the kernel know which driver to use? When the device is first connected, the operating system enumerates it as a blank device – and no driver is used. This triggers the software usb-modeswitch to automatically lookup the device according to the correct table for that kernel, and re-enumerate the device with the correct device ID. With the correct device ID, the kernel can now use the appropriate driver for the device. However this solution only works if the driver-modem match has been manually entered, and if the kernel has the correct driver. For additional compatibility, many modern 4G modems can have their interface mode switched onboard the modem. These modems provide a control channel, usually a serial port, and a wider data channel for the actual connection. Using the control channel, the communication protocol of the data channel can be manually changed, allowing the modem manufacturer to provide support for newer drivers through firmware updates.

The Point-to-Point protocol is the oldest driver and is still universally supported for modems. Rather than interacting directly with the modem hardware, the Point-to-Point protocol makes the modem handle the connection to the cellular service provider. The data stream is proxied to the embedded computer over a serial port. Although requiring

the modem to manage the connection is a very portable solution, it requires the modem to send all the information over a single serial port. This significantly reduces the available bandwidth and increases the latency.

The package in Linux to configure a Point-to-Point protocol is pppd. Configuring pppd requires a configuration file with AT commands – commands sent over the modem's serial control port to configure the modem prior to dialing the cellular service provider. After each boot, pppd configures the modem, asks the modem to dial the cellular service provider, and configures the serial port as new network interface.

Cellular modems which work over USB and have the option to act as a built in wireless access point can handle every aspect of the connection, even assigning an address to the client. On clients running a recent edition of Windows, these devices work out of the box. Simply plug in the device, configure account settings through the device web portal, and you have a connection. The Linux kernel enumerates the devices as a network card and connects to them using pppd. Although this solution incurs the latency cost of pppd, the throughput is unaffected thanks to higher bandwidth of the USB connection. In some cases, the Linux kernel does not enumerate the device correctly. Instead of connecting the device as a network card, it may get mounted as a USB drive or a miscellaneous serial device. If the device is incorrectly enumerated, the best solution is to configure Linux to re-enumerate the device using a compatible driver. Unfortunately, these drivers must have support for the network hardware inside the modem – it pays to make sure your device will work with the driver for your kernel version before purchasing.

CDC-ECM and CDC-NCM

When 3G and 4G cellular modems started becoming common additions to consumer laptops – consumers and cellular manufacturers wanted their devices to work out of the box. For Linux, this meant that users running generic Linux distributions like Ubuntu or Fedora should have hardware support after a fresh install – and not have to write their own scripts or search for instructions on a forum. To capitalize on this trend, a standard API was introduced for device drivers to interact with the modem. Broadcom implemented its proprietary QMI protocol, and the community proposed the MBIM protocol. For the purposes of this thesis, MBIM is considered an open source version of QMI.

On the user side, the libmbim and libqmi packages provide a set of utilities to control modems using the MBIM and QMI protocols respectively. The modemmanager package sits on top of either libmbim or libqmi and integrates with the operating system. Similar to the different variants of network manager packages in different distributions, the modem manager keeps tracks of the modem configuration, and provides a standard way for the different distributions of Linux to connect to a cellular service provider and map the device to standard network device – and to the user, the cellular modem looks no different than an ethernet controller or an 802.11 adapter.

Implementation Of 4G LTE

Near the beginning of the project, we wanted to test that a cellular modem was feasible for our project. From previous research, we knew that streaming telemetry over a 4G connection is not a novel idea and is commonly used in purpose built UAS. However, the systems we considered were not designed to be use in an urban environment, and no there

was no standard reference comparing airborne cellular modems to 802.11 systems. Therefore, we had to evaluate both systems on our own.

The first system we evaluated was a Hologram Nova development kit, based off a U-Blox modem. The Nova development kit is a USB cellular modem designed to prototype Internet of Things devices. The manufacturer says that the device provides instructions to setup the device with a Raspberry Pi single board computer, and a setup script to install the required tools and APIs to send and receive data with the modem. Hologram also provides cellular plans for their modems with pay-as-you-go pricing. These data plans are targeted at IOT devices like remote sensors – and are not expected to use much bandwidth. With our requirement to stream video, an application that requires a high bandwidth, Hologram's data plan would not work for us. Hologram's devices are compatible with Verizon networks, in fact their cellular data plans are rebranded from Verizon. Figuring that the device would work with a normal Verizon SIM, we bought a Verizon data plan and SIM. However, Hologram devices only work out of the box with their own SIM cards and I did not want to spend extra time to reconfigure a 3G modem which would only be used for a proof of concept.

Figure 10: UML 295 4G LTE Modem Attached to a Laptop

The next system we evaluated was a Verizon USB cellular modem, the UML295. The UML295 was released in 2012, so the documentation for it was somewhat dated. However, the UML295 handles the details of the connection by itself and worked out of the box on a Windows laptop. On my personal Linux machine, I was able to configure pppd to map a virtual network adapter to the cellular modem. On a Raspberry Pi, the UML295 would not enumerate as a USB modem. As I suspected, the Raspberry Pi was running an older kernel which did not know how to recognize the device. Adding a manual entry to switch the device to the correct mode fixed the problem.

At this point, we are able to test the stability of the 4G LTE network around the Arizona State University Campus. The test setup is a laptop with the UML295 plugged into a USB port, constantly monitoring the ping to a remote server. We selected the same route as the campus 802.11 wireless network tests, again with different areas with narrow corridors between buildings as well as open spaces.

Leg	Total Pings	Unsuccessful pings	Successful pings	Average Successful Ping	Reconnects
Leg 1	100	0	100	72	0
Leg 2	100	0	100	81	0
Leg 3	100	0	100	60	0
Leg 4	100	0	100	75	0
Leg 5	100	0	100	83	0

Table 4: 4G LTE Statistics Measured with The UML295 Modem

Having established that a 4G connection is the best solution available at the time, we proceeded to finalize the cellular modem hardware. Among the various 4G modem options, our options are divided among USB cellular modems similar to the UML295 but with more recent hardware, and cellular modems which connect over PCIe. USB cellular modems have the advantage of portability – the software and hardware configurations are the same whether they are plugged into a development workstation or a UAS. However, USB cellular modems are bulky. For a UAS where space is at a premium, it would be difficult to accommodate a mounting place among the other sensors and equipment on the drone. Cellular modems which operate over PCIe are designed to work on a target

platform. In our case this is the UP Squared single board computer onboard our drone. However, connecting to this board from development machines requires us to source more hardware. In our case, since our single board computer is powerful enough to act as a development platform, this was a non-issue. The biggest advantage of PCIe modems is their form-factor. Most PCIe cellular modems are of the mPCIe form factor. Excluding the antenna, a mPCIE board fits within the footprint of our single board computer, requiring no additional space onboard the drone.

Figure 11: 4G LTE CAT 3 Modem Mounted to a Raspberry Pi 3

The next system we evaluated was a NimbeLink Skywire development kit. Skywire development kits consist of two parts: the modem and the interface board. The modem

component is a full cellular modem, with a SIM card slot and external antenna connections. The interface board allows you to connect different modems to some device. For example, there are interface boards available as an Arduino shield, Raspberry Pi hat, BeagleBone capes, and mPCIe cards. In our tests, we used the Raspberry Pi hat for testing, and the mPCIe card onboard the UAS.

We ordered a Skywire cellular modems from NimbeLink. We chose a CAT3 modem because it was well documented, and their CAT4 modem based of the popular Sierra Airprime HL7588 to see if the theoretically higher download speed would translate into real world performance. Both modems were compatible with both interface boards, so when diagnosing problems with our setup, we could easily switch hardware setups to isolate the problem. On Linux, both cellular modems are designed to be used with the CDC-NCM driver, which enumerates the modem with serial ports for control, and exposed network interfaces. A script configures the modem (see appendix) and obtains an IP address, which is used to configure the modem.

I tried to enable different modes on the cellular modem, and different drivers. According to the documentation, these cellular modems also support CDC-ECM and MBIM modes. Changing modes is simple. Issue the command with the correction options to the modem's control port and reboot the modem. Changing the cellular modem back to a working configuration is much more difficult, and involves more trial and error than repeatable instructions, so I didn't try more than I needed. Sadly, the CDC-ECM mode performed similarly to the CDC-NCM mode, although requiring more configuration, and although the device correctly enumerated the MBIM driver, I could not get the tools from

the libmbim package to recognize the cellular modems. The implementation of the libmbim package is likely incomplete, and it is beyond the scope of my understanding and this thesis to fix the problem. Although a successful test of the MBIM driver would make the cellular modem and network configuration much simpler – a hacked together script is the only way the system works.

Leg	Total Pings	Unsuccessful pings	Successful pings	Average Successful Ping	Reconnects
Leg 1	100	0	100	82	0
Leg 2	100	0	100	64	0
Leg 3	100	0	100	63	0
Leg 4	100	0	100	76	0
Leg 5	100	0	100	85	0

Table 5: 4G LTE Test Statistics Measured Using the Skywire CAT4 Modem

	Indoors	Outdoors
Measured downlink speed (Mbps)	1	4
Measured uplink speed (Mbps)	4	12

Table 6: 4G LTE Speed Tests with the Skywire CAT4 Modem

Although the CAT4 cellular modem spec has a 50 percent faster download speed, we do not observe faster speeds in real world tests. Interestingly, the measured downlink speed is slower than the measured uplink speed. One possible cause is that our antenna is not configured correctly to receive multiple streams. Another possible cause is that the 4G

LTE network around campus is very congested. However, the CAT4 modem can maintain a more reliable connection. Therefore, it is the cellular modem of choice for our UAS.

The CAT4 modem is comparable to smartphone, opening new options for casually testing network functionality with only a smartphone. With some software to emulate drone communications, we can deploy a fleet of smartphones to profile the backend software responsiveness over a real 4G connection, rather than a virtual network inside the cloud.

Thoughts For Future Consideration

The mantra of the Internet of Things is to connect more devices to the Internet, gathering an increasing volume and variety of user data. With the advent of 5G technology, the number and variety of cellular modem kits will continue to increase.

5G technologies will also let us revisit mesh networking between devices. The network topology would look like the ad-hoc mesh network using the 802.11 network described previously, however with one major distinction. Besides the drones and the base stations acting as nodes, cellular towers would act as long range and high throughput nodes in the network. This would drastically reduce the latency in the network, because fewer hopes between nodes would be needed to send information from the source to the destination. In addition, the throughput would be a non-issue for routing data, as a single cellular tower would have a capacity higher than the theoretical sum of every source combined. Taking advantage of the mesh network for UAS would enable approaches to navigation and information previously impractical due to high latency, such as real-time map

building by sharing features. Information streams which require a high throughput could be shared between drones, allowing a UAS to draw directly from the sensors of other, networked UAS.

BACKEND TECHNOLOGIES

A system can work well in theory, but for a system this complicated needs to be tested in the real world. Developing software to run reliably at scale requires a different mindset from algorithm development. Cost and upkeep time is an important constraint.

The first possibility would be to run the scheduling backend on a local server, located geographically close to where the system would be deployed, possibly on the same network on which the UAS and its base stations are deployed. If we were to implement this on the Arizona State University Campus, ideal locations include the CIDSE server room and the HPC lab at Goldwater. The biggest advantage to a locally hosted server would be the low latency and high throughput between the devices on the same network. For example, the base stations located on the roofs of buildings would be a good place to download video and logs from the UAS, which consist of several gigabytes of data.

Although a local server makes sense from a network standpoint, the benefit of lower latency does not apply to the UAS, which are connected the system over a 4G connection. For a UAS to connect the local server, our cellular service provider Verizon would route the data through their servers, through the campus' Internet Service Provider Centurylink, and to our local server. Running traceroute a program to trace the history of packets, confirms that our packets pass through both Verizon and Centurylink servers

before reaching the local server. Although the additional latency is not problematic, a system that minimizes latency would be ideal.

Cloud Compute

Another option would be to host the scheduling application on a remote server running in the cloud. Would running this scheduling application in the cloud be better than running the application on a local server? Let's take another look at the latency between a UAS and cloud server. Running traceroute again reveals that although our packets travel the same number of hops between the UAS and the remote server, the round trip latency from the UAS to the remote server is lower. The most probable reason is that the connections between Verizon and our remote server are capable of a much higher throughput than the connections between Verizon servers to our campus. To minimize latency to the UAS, running the application would be the better choice. It is worth revisiting the latency discussion if the UAS were connected the same network as the local server, over 802.11 for example. In such a scenario, there would be a minimum number of intermediate hops for packets sent between the UAS and application on the local server, minimizing latency more so than running the application on a remote server.

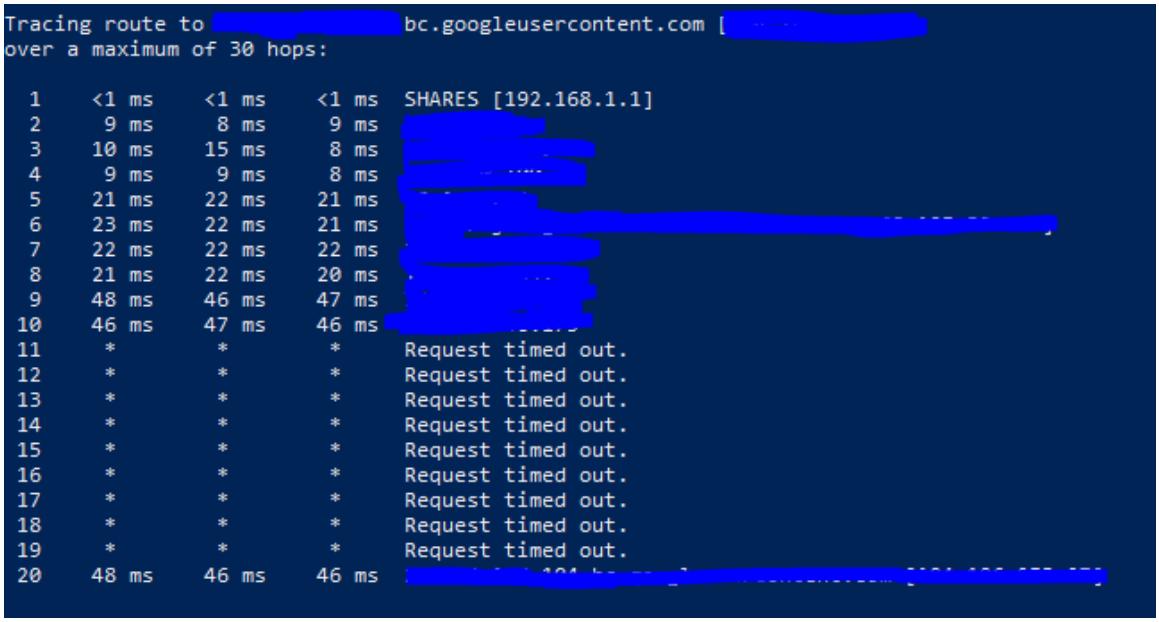

Figure 12: Traceroute to Remote Server.

Another advantage to hosting our application on remote servers running on the cloud is the ability to purchase compute time to scale the application with the needs of the system. Scaling local hardware is also an option. However, you have to make the decision to either buy more computing power in advance, or the application will be crash until its hardware can be upgraded. The biggest difference between scaling local hardware vs compute power from the cloud is the ease of use – a cloud application can scale with a short script from a system administrator, or automatically with a load balancer. How can cloud service providers sell compute time in such a granular way? Rather than selling compute time by selling a complete server, cloud providers sell virtual machines with limited resources. Employing this granular approach allows customers to customize their system in real time, increasing or decreasing the "size" of the virtual machine to match the application's needs.

A good application can be ruined if it has poor security. Luckily, there are many implementations of firewalls and encryption software to make sure that unsavory individuals cannot disrupt services or glean private information from a system. However, firewalls and encryption are useless if they are turned off or worse, misconfigured! On local hardware, a competent system administrator is responsible for managing and updating software configurations. Cloud service providers take a different approach, providing intuitive frontend interfaces which automatically configure every aspect of networks and firewalls, and virtual machine access and encryption. Having cloud service providers handle the system administration tasks allows developers to spend more time iterating on their application, rather than tweaking their system configuration.

There are many options for hosting cloud applications today, and there is no best choice as different service providers specialize in different areas.

Docker

Developers today face a growing problem developing and deploying applications. When adopting a new software stack, developers value how long it takes to implement a particular feature, and how portable that feature is the next time a developer needs to reuse or adapt it. Especially in frontend and other user interface projects, feature reusability and shorter feature development time comes in the form of third party packages. In contrast to packages built into the runtime environment which update together, third party packages have independent release cycles. Often times, the updates in one package break features from another, creating an endless cycle of bug fixes to resolve the problem. Currently the best solution to the problem is use a package manager.

Once a compatible set of package versions is established, the package manager keeps track of the packages and the same development environment can be easily recreated on a different computer.

Although package managers allow developers to vendor packages between different development machines, development and deployment platforms still differ enough that an application may behave differently between the development platform and the virtual machine it is deployed on. Developers typically require a fully-featured operating system, whereas applications are deployed in a spartan environment, with only the dependencies needed to run. For example, the networking stack on developer machines are more permissive, whereas the application's virtual machine will only open the required ports for that application.

One way to simulate a bare minimum deployment environment, is to test code inside a virtual machine. However, to properly test deploying an application requires the developer to build a new virtual machine for each test, which takes a long time. The solution to decrease testing time is to use container style virtual machines – particularly Docker containers. Docker containers are like onions - structured in layers. During a rebuild, only layers that change need to be rebuilt. In a typical container, a base image based on a Linux distribution like Ubuntu which comes with many libraries and a build environment or a base image based on a sparse distribution like Alpine is used as the starting point. Different steps inside a the Dockerfile, a scripting language for specifying Docker containers, are each their own layer. Steps could copy files from the host environment, compile the application, or run unit tests. The Dockerfile ends with a

command to start the application. Dockerfiles are portable, so developers with the same environment can use the same Dockerfile to build a Docker container.

Developers can also upload their Docker container to central server, where it will be deployed to the application's environment. How is building a Docker container different than compiling an application into one static binary, with no external dependencies? From a dependency standpoint, both Docker containers and static binaries prevent mismatching dependencies from crashing the application. But Docker containers take the next step and standardize the application's environment such that there is no difference between running a Docker container on developer machine and in the production environment.

Microservices

A simple application could be as simple as a single Docker container. However, the more features an application has, the more difficult it is to package all those features in one application in a manner conducive to easy debugging. Large applications often take a long time to start, and tracing errors will take much longer the more interconnected a feature is with other features. One solution is to break up the monolithic application, group features by functionality, and package that each functionality inside a microservice.

Packaging functionality into individual microservices increases code portability and reduces the amount of surrounding code during debugging. Each microservice is packaged with only the dependencies needed for its functionality. For example, a microservice that manages user data validation does need the same dependences that an

image processing microservice would require – both applications could be developed and debugged in parallel.

Although a microservices is typically run in its own virtual machine, a new variation called serverless microservices are quickly becoming more popular. The biggest draw to serverless microservices is the cost. Unlike a virtual machine which waits idle until it receives a request, a serverless microservice runs a trigger – meaning that compute time is billed based on the number of requests rather than uptime. Usage based billing is more granular than even dynamically scaling an application's infrastructure, making deploying an application written with serverless microservices extremely cost effective. In fact, the cost effectiveness of serverless microservice platforms like Amazon's AWS Lambda is the reason that Amazon's Alexa has so many third-party skills.

Message Passing

In a monolithic application, passing information is as simple as ensuring that an object is in scope, and passing information through the control flow in the code. However, microservices are different applications, and cannot directly pass control flow. Before any microservices are written, the development team must decide how to break the application down into different functionality into different microservices. Microservices are not useful by themselves. Rather they communicate with each other, processing information and sending the result down the pipeline. This is accomplished through message passing – and serves the dual purpose of standardizing the API by which microservices exchange information. At the same time developers break an application down by functionality, they must specify what information is passed between each

microservice. Specifying standard interfaces allows developers to develop dependent microservices in parallel and test individual microservices with expected input and output without dependent microservices.

One form of message passing is Remote Procedure Call, in which microservices pass information and the control flow by calling a different microservice and waiting for an answer. The most popular implementation of Remote Procedure Calls is Google's implementation, abbreviated as gRPC. For two microservices to implement gRPC, one microservice acts as a server, and the other acts as a client. In addition, the remote procedures are partially defined in terms of input and output so that developers can write the client without any knowledge of the server implementation. As far as implementing applications connected with gRPC, most common languages like Python, Go, and Javascript have an implementation of gRPC. Applications are not limited to using just one language – gRPC messages between microservices are serialized and deserialized using the Protobuf library. Using gRPC for message passing allows developers to build heterogeneous microservices according to a rigidly defined API – ensuring that the final application is fully modular as to simply debugging and testing.

Kubernetes

Compared to writing one monolithic application, developers now need to keep track of dozens or more microservices. And if a microservice has multiple replicas of itself, this number increases linearly. If multiple developers work on different microservices, even adhering to the rigid API set by message passing, a new version of a microservice could still be incompatible with an older microservice. Enter Kubernetes, a management system

for microservices. The smallest unit of control in Kubernetes is a pod, which is usually a docker container. An identical set of pods is a Replica. Kubernetes allows developers to describe the architecture of an application as a dependency of pods. If a pod in a replica has an error, it is automatically restarted. When updating pods from one version of a microservice to the next, Kubernetes can stager the upgrade to make sure that the new version will not cause any problems. Thanks to Kubernetes managing the health of its cluster of pods, developers can write a descriptive architecture, and spend more time developing microservices and less time maintaining the versioning and health of the cluster.

Language Choice – Why We Choose Go

The primary languages used in this project are Go, and Python.

Go, also known as Golang, is a language developed by Google engineers to replace C++ for extremely parallel applications which can scale without the significant overhead in languages like Java or Python. Google had a problem. A Google developer would write an application and leaves behind documentation. After that developer moves on to the next project, another developer would take a long time to get reacquainted with the codebase when a bug needs fixing. For complex, multi-threaded applications, this problem was amplified by how long the application took to compile. In order to minimize developer time wasted on understanding old codebases (hard to understand C++) and time wasted waiting for an application to compile Google developed Go – a language with a simple syntax similar to C to make code easier to understand, and a runtime which

can compile code quickly and is optimized for an arbitrarily large number of concurrent threads.

For this UAS scheduling application, there are two different types of microservices. Microservices with parallelizable operations, and microservices with complex logic. For the former type of micro service, choosing Go allows us to focus on application logic rather than concurrency. For the latter microservice, Python is a good choice to write descriptive code, and concurrency can be added by increasing the number of replicas of that service. In addition, choosing Python also gives access to the vast number of NumPy-based math libraries, although equivalents are available in Go.

We considered other popular languages for server application development like Ruby and PHP. However, although Ruby as a language is more ergonomic than Go, and PHP is well documented, neither languages' interpreter is as fast or memory efficient as Go's runtime.

We also considered other system's level languages like Rust and C++. Although Rust is a promising language which promises runtime safety through strict control of references, the language specification is still under development and therefore not suited for implementing a flight control system where all errata must be documented. In contrast, the errata in C++ compilers are very well documented. However, I am reluctant to use C++ for new projects unless the hardware support or existing libraries offer a compelling advantage to offset the unmaintainable nature of large C++ projects.

Latency And Protocols

Assuming the UAS can communicate with the flight scheduling system, minimizing latency is the next biggest challenge. Although we can minimize latency by switching different to different network architectures, the latency due to routing is inescapable. The software architecture of the application also adds latency, depending on the protocol used to communicate. Software latency is divided into two categories: latency establishing a connection and the latency maintaining a connection. When the UAS establishes a connection to the flight scheduling system, the flight control system must establish the authenticity of the UAS and allocate resources for the UAS. After the UAS has established a connection, it can send and receive messages with the flight control system with less overhead. However, if the UAS loses the connection and must reconnect – it re-incurs the overhead of establishing a connection. Unfortunately, there is no one solution with a low overhead to establish a connection and has a low latency to maintain the connection. The best protocol must be determined on a case by case basis.

Another item to consider is the consistency of the messaging protocol. Although a solution could have a low minimum latency in theory, in practice a poor connection could cause a large gap between the minimum and maximum observed latency, making the UAS actions unpredictable. Ideally, the standard deviation of the latency for sending or receiving a message should be relatively small. In a perfect world, the average latency would be closer to the minimum latency than the higher latency, but, is highly dependent on the network architecture. On a network with a stable connection, it would be ok to use a protocol which takes a relatively long time to establish a connection but sends and receives further messages over the same connection with low latency. However, the same

protocol would have a higher average latency on a network which occasionally loses connection. Every time the UAS looses connection to the flight scheduling system, it would incur the overhead of establishing a connection, increasing the average latency. To ensure more consistent latency, a protocol which takes a similar time to establish a connection and maintain the connection could be used, at the cost of a higher average latency.

We tested three different communication protocols, to find out which protocol gives the most consistent performance over our 4G communication network, and which protocols offered the lowest latency. All of our solutions were implemented in Go, using Echo http router written by Labstack.

HTTP Requests

To establish a benchmark, we first tested HTTP requests. HTTP requests are commonly used to request resources and send information across the internet in a secure manner.

Systems built around HTTP are structured such that all data is kept in the backend, which clients can access through an API. The API is structured to expose only the relevant information to the client, and underlying data and computation can be structured completely different to what the client sees.

HTTP requests can only be sent from a client to the server. A server cannot push information to a client over HTTP, like an interrupt. As such, systems built on HTTP requests embrace the client-server and structure the information flow accordingly.

HTTP requests have a fixed amount of latency. As each HTTP request is its own connection, the overhead of establishing a connection to the backend is counted every time.

If the UAS were to communicate with the backend using HTTP requests, it would have to regularly POST to the backend with its status and GET any updates. However, polling the backend would introduce latency to the system, as there may be some time between when the server stores information, and a client request it.

Long Polling

Long polling over HTTP requests is one solution to minimizing the latency do to polling. In a long poll, the client opens a request to the backend, and waits for a response. The backend keeps the connection open for a period and responds to the request if it has new information. If the backend has no new information after a set period, it closes the request.

The client waits for this information asynchronously, so client performance is unaffected. However, the backend implementation is inefficient as it must spend compute cycles checking for new information. Although long polling minimizes the latency between the client receiving information from the server, long polling scales poorly the more clients communicate with the backend.

Websocket API

So far, our implementations have avoided TCP socket-based communication. Although raw TCP sockets have the lowest overhead of any communication protocol safe to use over the Internet, interrupted communication will kill the connection without warning. If

the connection is killed, neither the server nor the client will be aware of the fact until one tries to send a message to the other. At that point, the connection will need to be renegotiated, adding latency to the queued message.

The WebSocket API was implemented to address the implementation hazards of raw TCP sockets, and add encryption. A WebSocket is opened over an HTTP request, and is upgraded to a TCP connection, providing full duplex communication between the client and the server. Further the WebSocket API monitors the state of the connection and will automatically raise an error if one side of the socket disappears, using keep-alive packets.

SYSTEM ARCHITECTURE

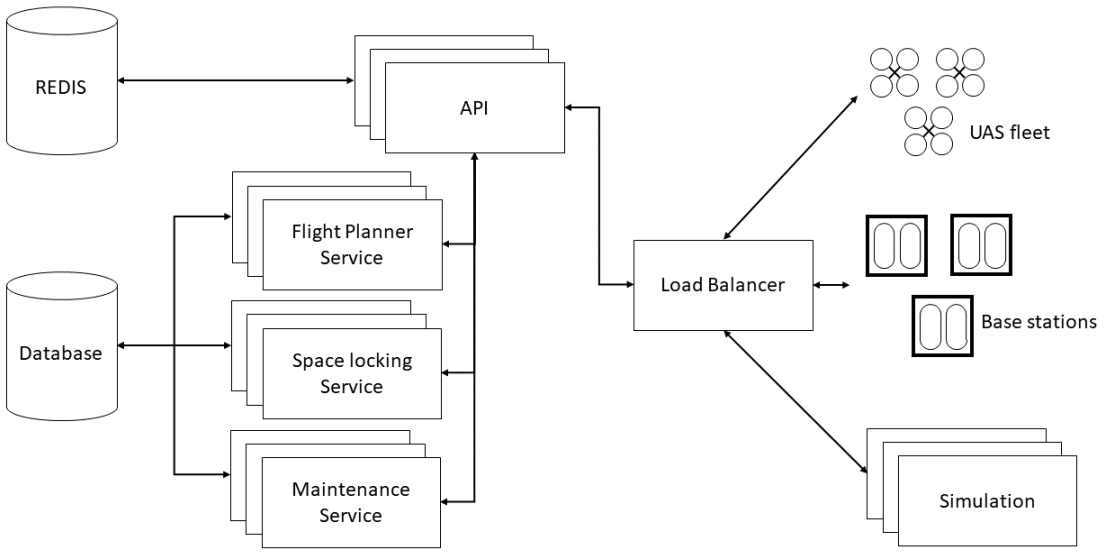

Figure 13: Information and Control Flow Through The Backend

Client devices like the base stations, UAS, and simulation UAS connect with the backend systems through a load balancer. We use Nginx as a load balancer. The load balancer distributes http requests and web socket connections to replicas of the API.

API

The API authenticates client devices, and establishes a WebSocket connection, a bi-directional data link. The API also passes messages between different clients, for example the battery voltage from a UAS to a charging station. However, two different clients may be connected to different replicas of the API. To communicate between two different replicas, a Redis instance is used for publish-subscribe communications between replicas. By publishing streams to the Redis instance, and subscribing to various streams, a connection can access other connections as if all connections were running in the same instance.

Flight Plan Service

When a new UAS task is generated, the flight plan service selects the best UAS for the task (Farinelli et al., 2004) (Gerkey et al., 2004) and calculates a flight plan for the UAS. The flight plan service interacts with the space locking service to avoid scheduling a flight plan through space which will be locked, by sharing a map through a MySQL database. Creating a flight plan does not lock spaces, allowing for multiple flight plans to be created in case of an emergency – if the UAS has a fault and needs to land immediately, it can choose a backup flight plan to the nearest base station or empty space.

Currently, the Flight Plan service uses A* (Bellmore et al. 1968) to calculate the most cost-effective path to the goal. For now, the cost function only considers distance to the goal, making the cost between any two adjacent grid spaces identical. However, the cost function should consider more data to generate more efficient flight plans. Adding wind

speed and direction to the cost would calculate flight plans with the least wind resistance, enabling longer flight times. Factoring in safety as a cost to the flight plan would yield safer flight plans, where safety would be calculated using the population density in the grid space at a given time.

Space Locking Service

As a UAS is active, it obtains a lock on space before it navigates through it (Bullo et al., 2009). The space locking service listens to requests from every UAS and synchronizes locked space between UAS to prevent collisions.

Spaces are tri-state variables. A state is either unlocked, locked red, or locked blue. The different colors represent the direction the UAS is flying though the space. Red spaces are for drones heading in a north or west direction. Blue spaces are for drones heading in a south or east direction. UAS flying in a locked, red state fly at a lower altitude than a UAS in a locked blue state to avoid collisions. A space is either locked or unlocked and can be locked both red and blue at the same time by different UAS.

When a UAS receives a flight plan, it attempts to lock the next flight leg. A flight leg is defined as a consecutive, linear set of spaces. The space locking service returns the result of whether the space was successfully locked. In the event of a conflict, the lock is given to the UAS nearest to the locked space. If a UAS cannot get the lock for the space, then it will fly to the adjacent space and periodically retry for a lock.

Maintenance Service

The maintenance service is responsible for processing UAS logs and storing a copy of every UAS logs in the database. This dataset will be used to build models for individual UAS components and construct a maintenance schedule.

UAS Software

Onboard the UAS, multiple services manage different aspects of the UAS. The video streaming service sends video to an ingestion server, where it served to other clients. Processed video is collected from Raspberry Pi cameras and stitched together into one image before it is uploaded to a video ingestion server.

The control service opens a WebSocket the backend and listens for locking information and flight plans. Once a flight plan is received, the UAS requests clearance from the flight plan service and starts to execute its plans. The maintenance service communicates the current UAS status to the backend, and uploads flight logs with detailed information after each task is completed. Both the control and maintenance services send data over a WebSocket connection to the backend.

Base Station Software

A Raspberry Pi onboard the base station reads sensor data from an Arduino. The base station sensors monitor temperature, moisture and humidity, and the voltage and amperage delivered to the charging rails. This data is streamed to the backend over a WebSocket and used to calculate base station health and availability. The base station receives data from any landed UAS to determine when to stop charging.

Simulation

ArduCopter comes with a simulation environment Software In The Loop, abbreviated SITL. SITL simulates the motion of a UAS and communicates using mavlink messages just like a hardware autopilot would. User software can send instructions to the UAS and receive status messages by sending messages over a TCP or UDP port, similar to how the UAS onboard computer communicates with the autopilot over a serial port.

To simulate an entire UAS, SITL is bundled with the UAS control software, and packaged in a Docker container. To Kubernetes, this container communicates with the backend over the WebSocket API in the same way as a physical UAS. We can create as many virtual UAS as needed to test the simulation.

MAPPING AND SCHEDULING

Introduction To The Scheduling Problem

To solve a scheduling problem, a scheduler must allocate resources to agents according within constraints to accomplish a goal. Constraints effect agent interaction, for example two UAS cannot occupy the same space at the same time. Constraints also affect individual agents, for example a UAS cannot receive a longer schedule than its flight time.

To solve a scheduling problem, a scheduler must allocate resources in an optimal way; there cannot be more optimal allocation of resources. Although optimality is a desirable feature for a scheduler, optimality places many limitations on the scheduler. An optimal scheduler assumes that all agents and goals are known at the time of scheduling. If a new agent or goal is added, the entire schedule must be recomputed. An optimal scheduler has

an exponential runtime. The more agents and goals need to be scheduled, the longer it will take to find the optimal solution.

An alternative to solving the scheduling problem is to find the best possible solution as each new goal arises. This greedy approach calculates the optimum resource allocation for an agent and modifies the allocation to fix any resource conflicts. Although this approach will rarely yield the optimal solution, the scheduler can run in polynomial time and will find a valid solution.

Designing The Space Resource

Before we could attempt the scheduling problem, we had to structure our resources such that a goal and solution could be specified in terms of our resources.

The scheduler allocates space to the UAS to ensure that multiple UAS never inhabit the same space, resulting in a collision. There are multiple approaches to ensuring that multiple agents do not occupy the same space, which can be distilled into ensuring that two trajectories do not intersect and assigning non-intersecting sets of space to each agent.

Generating non-intersecting trajectories is not resilient to dynamically adding plans. Therefore, we chose to segment the space into regular shapes, and allow the UAS to request a lock on an upcoming space.

We chose to use a two-dimensional grid over a three-dimensional grid to represent space in our scheduling system. If we were to implement a three-dimensional grid, we would have to implement a three-dimensional mapping system to pre-compute static obstacles.

There is no cost-effective method to compute a detailed three-dimensional grid over large areas. Storing and streaming three-dimensional data is more complicated as well. However, UAS move in three-dimensions. For this reason, we chose to use a 2-dimensional grid, where a grid space represents the column from ground level to the flight ceiling of 400 feet.

Why Use Hexagons?

The standard choice for a regular grid is to tessellate squares. Square grids systems are simple to implement and can be adapted for use in three-dimensional space by representing space as cubes. However, since we are limiting our navigation to two-dimensional space, we are open to other options. Enter hexagonal grids.

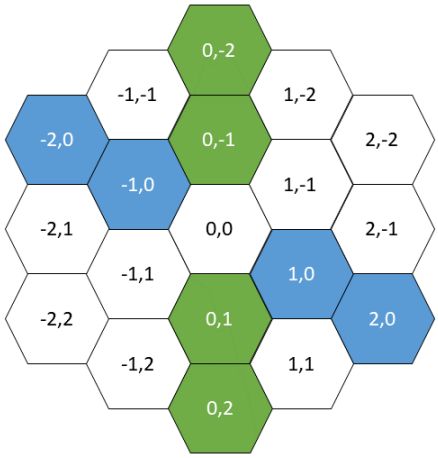

Figure 14: A Hexagonal Coordinate System

Each hexagonal grid space has 6 neighbors, vs the 4 neighbors of a square grid space. The extra neighbors allow the UAS to have more granular control over the space it takes

up. Hexagonal grids also simply the math involved for common heuristics. Rather than relying on Euclidian distance, which requires square roots, or the more inaccurate Manhattan distance, distance in a hexagonal grid can be calculated with simple operations and remain accurate. This is possible due to the askew axes.

Converting Between Absolute And Grid Coordinates.

The UAS utilizes its GPS sensors to report is condition according to its longitude and latitude. The backend however, allocates space according to the hexagonal coordinate system. Although approximations between the system can be made over short distances, the error accumulates over long distances. For areas over a square kilometer, calculating the distance between two coordinates could vary by a meter depending on the accuracy of the calculations.

The hexagonal grid presents another solution. Unlike calculating the distance between latitude and longitudinal points in terms of a two-dimensional offset, utilizing a heading and distance result yields a more precise result. By likening a hexagonal grid to an isometric circular grid, a hexagonal grid can be created such that the centers of each hexagon are equidistant according to latitude and longitude.

Hover, after projecting each hexagon onto the curved surface described by latitude and longitudinal points, the hexagonal edges no longer match up. This inconsistency makes determining to which hexagon an absolute point falls within a difficult prospect. A probabilistic approach is employed by means of a particle filter (Schwager et al., 2011), to find the most likely hexagon to which a point belongs. The alternative is to calculate the absolute distance to each hexagonal center – a computationally intensive approach.

SYSTEM RESULTS

The simulation environment is used to evaluate system performance. Unfortunately, technical and paperwork problems prevented us from deploying the system on the Arizona State University campus for now, however we will work towards resolving the hurdles.

The simulation environment contains three UAS with three base stations. Three UAS with three base stations were chosen with cost in mind, because spinning up that many virtual machines gets expensive. In practice, more UAS and base stations can be added to the simulation, provided the number of base stations equals or exceeds the number of UAS.

A 2000 x 2000 map with obstacles was randomly generated to simulate the map. This represents an area covering four square kilometers, which is roughly the area the system would service when deployed over ASU. Rather than creating a random map by independently assigning a grid space as an obstacle according to some distribution, a random center is chosen, and an obstacle is expanded as a ring a random distance from the center. The maximum radius for an obstacle is 100 grid spaces. A maximum of 20 obstacles were placed on the map, and obstacles that encompass a base station were discarded. Finally, a sample of random paths are examined to check the map for validity.

The UAS top speed is capped at three meters per second, and acceleration at half a meter per second. The UAS battery life is calculated by the SITL simulation, and the simulator is configured with the battery specification. The average flight time in the simulation is 45 minutes, which matches the upper end we observe in actual flight testing.

Scheduling Flight Plans Results

The system uses an admissible heuristic to calculate flight path costs, guaranteeing a reasonably optimal path. Further testing is required to validate if more sophisticated heuristics which factor wind speed and straight-line distance are admissible and will be tested once the appropriate data is gathered.

Given a reasonably optimal flight plan, the next most important metric is how long the system takes to calculate the plan. The time to generate a plan increases proportionally with the distance between the starting location and the goal. For this implementation, the map is fixed, along with the maximum number of grid spaces the graph will expand while searching for the goal. Devising a system to return plans within some time constraint regardless of the map size is left for future work.

For the experiment, flight plans were calculated on random maps, with variable distances between the starting location and the goal. The average time to compute 100 plans of each length is presented in a table.

Average plan length	Average time to compute plan
20	0.01 seconds
50	0.02 seconds
100	0.05 seconds
500	0.24 seconds
1000	0.47 seconds

Table 7: Average Time to Compute Plans

The time to compute a plan is directly proportional the length of the plan. For a map with one-meter resolution, covering the four square kilometers of the Arizona State University campus, a maximum half-second delay to calculate plans does not impede the overall system.

Space Locking Results

The space locking is an answer to several problems facing the scheduling system. It prevents conflicts in real time, solving the problem of how to implement multi-agent path planning when some goals are unknown at plan compile time, avoiding recompiling the entire solution. Space locking also solves the bandwidth problem which arises when a single UAS locks a large area of space because only one UAS can safely pass through the leg of space, like a one-way bridge on a two-way road. Thanks to the space locking service, the amount of navigable space is effectively doubled.

Testing the system on a reduced map size to force overlapping paths yielded no mid-air collisions, so the space locking service works as expected! However, in exchange for shorter overall paths, space locking increases the flight time. Would it be better to plan non-intersecting paths around existing flight paths, like light bikes paths in the game Tron?

For the experiment, flight plans were generated in a constrained 20 x 20 map to force intersecting paths. Obstacles were disabled for testing. Three intersecting paths were generated from three random starting and goal locations and executed with space locking active, and three non-intersecting paths were also generated.

Average plan length	Intersecting execution time	Non-intersecting execution time
5	43 seconds	38 seconds
10	54 seconds	135 seconds
20	67 seconds	253 seconds

Table 8: Average Time to Execute Intersecting and Non-Intersecting Paths

Even with pauses where the UAS is hovering while waiting to lock the next leg, executing the intersecting pat is faster than executing the non-intersecting path. This is

due the more frequent edge cases where UAS must fly around another path to reach their goal, in the worst case traveling at least three times as far as the shortest path. Consequently, space locking allows the system to make the best use of the UAS flight time.

FUTURE WORK

Improve Networking

The UAS currently uses a CAT 4 4G LTE modem which is sufficient for streaming video and telemetry. However, the current standard for 4G modems is CAT 16. The wireless communication will soon embrace 5G technologies, using a wider range of frequencies to connect devices. Adding these improvements would allow the UAS to operate in different types of environment and utilize the existing wireless infrastructure and allow an UAS to draw from richer data sources from ground stations and other UAS in real time.

Although 4G LTE modem is currently the best solution for reliably networking UAS in an urban environment, new innovations to existing 802.11 technology will enable better mesh networking solutions.

Obstacle Avoidance

Currently, the UAS relies on its knowledge of clear spaces, and cannot detect new obstacles, like a sign or an overgrown tree. To make the system more robust, the UAS should be able to navigate around unknown obstacles and ask for a new flight plan in the worst case.

Utilizing a depth camera to obtain three-dimensional depth maps of the environment and plan around obstacles in the flight path is one solution. Unfortunately, our Intel RealSense camera did not arrive in time to add it to current iteration of the UAS.

Another solution is to borrow the depth features produced in visual odometry solutions and look for features in the current flight path.

Peer To Peer Communication

Revisiting a mesh network of UAS and base stations with the appropriate hardware would increase the adaptability of the system to different situations. Consider an urban environment where the exiting network infrastructure is too damaged to support the UAS fleet. In such an environment, a system with robust communications is required for surveillance tasks. The deployment of the base station would be a separate logistics problem.

Two improvements are needed for mesh networks to be a viable solution – an increase in throughput and a constraint on scheduling to keep a balanced network topology. Achieving the first improvement is a function of how much money is spent on the transceivers. Achieving the scheduling constraint in real time while maintaining a balanced network topology is a subject for future research.

One To Many Control

No matter the degree of autonomy our system can reach, eventually a person will want to take manual control over the system. Given the standard controls of a UAS which map altitude and orientation to multiple axes on a controller, it is very difficult for a single

operator to control two, let alone a fleet of UAS. To this end, a manual fleet operation mode is needed to abstract the flight of UAS to simple controls.

A possible solution can be found in video game design, where single player must coordinate multiple units and multitask objectives to win an adversarial game. The typical control scheme allows the operator to select a single unit or a combination of units and assign a movement task or action task. The granular control over units allows operators to customize their playstyle and requires a significant amount of time to develop – essential aspects of designing enjoyable gameplay. Intentionally, this control scheme becomes unmanageable with many units.

Abstracting control to a higher level would allow the operator to manage an arbitrary number of UAS. At this level of abstraction, operator would specify tasks to the system rather than individual UAS and the system would generate tasks for individual UAS. Out of the multiple ways to generate the individual tasks, the best researched method is to the convert the overall tasks into a Boolean satisfiability problem and use a SAT solver to generate a plan of action for each UAS. However, addition constraints would be needed to ensure that the plan remains comprehensible to the operator, who may stop the system if it misbehaves.

What constitutes acceptable behavior? Research into the ability of humans to comprehend swarm behavior (Karavas et al., 2015) reveals that humans have more difficulty understanding swarm behavior the more noise is added to individual movement. More research is required to determine the threshold for human understanding of plans for systems with multiple objectives.

REFERENCES

Parker, L. E. (2008). Multiple mobile robot systems. In *Springer Handbook of Robotics* (pp. 921-941). Springer Berlin Heidelberg.

Kumar, V., Rus, D., & Sukhatme, G. S. (2008). Networked robots. In *Springer Handbook of Robotics* (pp. 943-958). Springer Berlin Heidelberg.

Michael, N., Fink, J., Loizou, S., & Kumar, V. (2010). Architecture, abstractions, and algorithms for controlling large teams of robots: Experimental testbed and results. In *Robotics Research* (pp. 409-419). Springer, Berlin, Heidelberg.

Malpani, N., Welch, J. L., & Vaidya, N. (2000, August). Leader election algorithms for mobile ad hoc networks. In *Proceedings of the 4th international workshop on Discrete algorithms and methods for mobile computing and communications* (pp. 96-103). ACM.

Farinelli, A., Iocchi, L., & Nardi, D. (2004). Multirobot systems: a classification focused on coordination. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, *34*(5), 2015-2028.

Gerkey, B. P., & Matarić, M. J. (2004). A formal analysis and taxonomy of task allocation in multi-robot systems. *The International Journal of Robotics Research*, *23*(9), 939-954.

Cortes, J., Martinez, S., Karatas, T., & Bullo, F. (2004). Coverage control for mobile sensing networks. *IEEE Transactions on robotics and Automation*, *20*(2), 243-255.

Olfati-Saber, R., & Murray, R. M. (2004). Consensus problems in networks of agents with switching topology and time-delays. *IEEE Transactions on automatic control*, *49*(9), 1520-1533.

Bullo, F., Cortes, J., & Martinez, S. (2009). *Distributed control of robotic networks: a mathematical approach to motion coordination algorithms*. Princeton University Press.

Schwager, M., Rus, D., & Slotine, J. J. (2011). Unifying geometric, probabilistic, and potential field approaches to multi-robot deployment. *The International Journal of Robotics Research*, *30*(3), 371-383.

De Gennaro, M. C., & Jadbabaie, A. (2006, December). Decentralized control of connectivity for multi-agent systems. In *Decision and Control, 2006 45th IEEE Conference on* (pp. 3628-3633). IEEE.

Spanos, D. P., & Murray, R. M. (2004, December). Robust connectivity of networked vehicles. In *Decision and Control, 2004. CDC. 43rd IEEE Conference on* (Vol. 3, pp. 2893-2898). IEEE.

Bellmore, M., & Nemhauser, G. L. (1968). The traveling salesman problem: a survey. *Operations Research*, *16*(3), 538-558.

Goerzen, C., Kong, Z., & Mettler, B. (2010). A survey of motion planning algorithms from the perspective of autonomous UAV guidance. *Journal of Intelligent and Robotic Systems*, *57*(1-4), 65.

Karaman, S., & Frazzoli, E. (2010). Incremental sampling-based algorithms for optimal motion planning. *Robotics Science and Systems VI*, *104*, 2.

Bellmore, M., & Nemhauser, G. L. (1968). The traveling salesman problem: a survey. *Operations Research*, *16*(3), 538-558.

Bellman, R. (1962). Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM (JACM)*, *9*(1), 61-63.

Van Den Berg, J., Guy, S. J., Lin, M., & Manocha, D. (2011). Reciprocal n-body collision avoidance. In *Robotics research* (pp. 3-19). Springer, Berlin, Heidelberg.

Fox, D., Burgard, W., & Thrun, S. (1997). The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, *4*(1), 23-33.

Thrun, S. (2002). Robotic mapping: A survey. *Exploring artificial intelligence in the new millennium*, *1*, 1-35.

Durrant-Whyte, H., & Bailey, T. (2006). *Simultaneous Localization and Mapping (SLAM): Part I The Essential Algorithms. Robotics and Automation Magazine 13 (2): 99–110. doi: 10.1109/MRA. 2006.1638022*. Retrieved 2008-04-08.

Forster, C., Lynen, S., Kneip, L., & Scaramuzza, D. (2013, November). Collaborative monocular slam with multiple micro aerial vehicles. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on* (pp. 3962-3970). IEEE.

Kumar, S., Gil, S., Katabi, D., & Rus, D. (2014, September). Accurate indoor localization with zero start-up cost. In *Proceedings of the 20th annual international conference on Mobile computing and networking* (pp. 483-494). ACM.

Beni, G. (2004, July). From swarm intelligence to swarm robotics. In *International Workshop on Swarm Robotics* (pp. 1-9). Springer, Berlin, Heidelberg.

Wilson, S., Pavlic, T. P., Kumar, G. P., Buffin, A., Pratt, S. C., & Berman, S. (2014). Design of ant-inspired stochastic control policies for collective transport by robotic swarms. *Swarm Intelligence*, *8*(4), 303-327.

Şahin, E. (2004, July). Swarm robotics: From sources of inspiration to domains of application. In *International workshop on swarm robotics* (pp. 10-20). Springer, Berlin, Heidelberg.

Pickem, D., Glotfelter, P., Wang, L., Mote, M., Ames, A., Feron, E., & Egerstedt, M. (2017, May). The Robotarium: A remotely accessible swarm robotics research testbed. In

Robotics and Automation (ICRA), 2017 IEEE International Conference on (pp. 1699-1706). IEEE.

Yim, M. (2007). Modular self-reconfigurable robot systems: Challenges and opportunities for the future. *IEEE Robotics Automat. Mag.*, *10*, 2-11.

Chen, I. M., & Burdick, J. W. (1995, May). Determining task optimal modular robot assembly configurations. In *Robotics and Automation, 1995. Proceedings., 1995 IEEE International Conference on* (Vol. 1, pp. 132-137). IEEE.

Qu, F., Wang, F. Y., & Yang, L. (2010). Intelligent transportation spaces: vehicles, traffic, communications, and beyond. *IEEE Communications Magazine*, *48*(11).

Papadimitratos, P., De La Fortelle, A., Evenssen, K., Brignolo, R., & Cosenza, S. (2009). Vehicular communication systems: Enabling technologies, applications, and future outlook on intelligent transportation. *IEEE communications magazine*, *47*(11).

Thrun, S., & Liu, Y. (2005). Multi-robot SLAM with sparse extended information filers. In *Robotics Research. The Eleventh International Symposium* (pp. 254-266). Springer, Berlin, Heidelberg.

Campbell, M., Egerstedt, M., How, J. P., & Murray, R. M. (2010). Autonomous driving in urban environments: approaches, lessons and challenges. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, *368*(1928), 4649-4672.

Underwood, C., Pellegrino, S., Lappas, V. J., Bridges, C. P., & Baker, J. (2015). Using CubeSat/micro-satellite technology to demonstrate the Autonomous Assembly of a Reconfigurable Space Telescope (AAReST). *Acta Astronautica*, *114*, 112-122.

Hadaegh, F. Y., Lu, W. M., & Wang, P. K. (1998). Adaptive control of formation flying spacecraft for interferometry. *IFAC Proceedings Volumes*, *31*(20), 117-122.

Dautenhahn, K. (2007). Socially intelligent robots: dimensions of human–robot interaction. *Philosophical Transactions of the Royal Society B: Biological Sciences*, *362*(1480), 679-704.

Karavas, G. K., & Artemiadis, P. (2015, September). On the effect of swarm collective behavior on human perception: Towards brain-swarm interfaces. In *Multisensor Fusion and Integration for Intelligent Systems (MFI), 2015 IEEE International Conference on* (pp. 172-177). IEEE.

Chung, T. H., Hollinger, G. A., & Isler, V. (2011). Search and pursuit-evasion in mobile robotics. *Autonomous robots*, *31*(4), 299.

Emery-Montemerlo, R., Gordon, G., Schneider, J., & Thrun, S. (2005, April). Game theoretic control for robot teams. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on* (pp. 1163-1169). IEEE.

López-Camacho, E., Ochoa, G., Terashima-Marín, H., & Burke, E. K. (2013). An effective heuristic for the two-dimensional irregular bin packing problem. *Annals of Operations Research*, *206*(1), 241-264.

Forster, C., Pizzoli, M., & Scaramuzza, D. (2014, May). SVO: Fast semi-direct monocular visual odometry. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on* (pp. 15-22). IEEE.

Estep, E. (2013). Mobile html5: Efficiency and performance of websockets and server-sent events.