Flying Courier

[EECS 149/249A Class Project]

David C Tsai
dtsai@berkeley.edu

Alex Arron Kashi
akashi@berkeley.edu

Department of Electrical Engineering and Computer Science
University of California
Berkeley, CA

ABSTRACT

Applications of aerial robotics, particularly quadcopters, are being explored increasingly as the prevalence of task automation and robotics spreads. Two notable areas of investigation are freight and indoor navigation. Conventionally, the freight problem is investigated in the context of open, outdoor settings with various environmental hazards, while research in indoor maneuvering assumes an extensive infrastructure supporting both the quadcopter and the surrounding environment. In this project, we made substantial progress on developing an indoor package delivery drone system that is both precise and lightweight. To approach this problem, we adapted common practices for indoor quadcopter flight in research settings to a simplified, more accessible setup. We abstracted away the details of low-level flight by using an onboard flight control unit and built our system on a high-level control scheme that defines basic modal behavior, assisted by a novel, all-visual motion capture system we developed. Aside from being unable to finish due to issues with the flight control firmware and getting our quadcopter in the air, our system worked well and supports the idea that further work on our system could open the door for a wider body of engineers and researchers to make use of quadcopters.

1. INTRODUCTION

We determined that to accomplish our goal of indoor package delivery, our implementation would be composed of three main components: the quadcopter, the motion capture system, and the software.

1.1 Quadcopter Design

We wanted our quadcopter to support both a powerful onboard computer, as well as a large payload. Modifying a ready-to-fly qudcopter for this task would not be ideal, given that modifications are not usually supported by the manufacturer and may create compatibility issues. For this reason, we decided to go with a custom build with motors, propellors, and batteries specced to support a total of approximately 2.5 kilograms fully assembled and loaded. We also wanted our electrical system to be compact and reduce wire clutter, which is important for keeping it safe from spinning propellors and other hazards. To provide some stability, we also an off-the-shelf power distribution board to power components on 5 and 12 volt rails.

1.2 Motion Capture Design

To realize an accessible setup for indoor quadcopter flight, we needed a source of positioning information for the quadcopter that was both easy to find and inexpensive. By using a combination of computer vision and a spatial transformation model of the quadcopter within our motion capture system, we were able to implement a solution using only images printed on paper and a common webcam. Because our system is based solely on visual data, we added requirements for mitigating the effects of occlusion, distance, and classification error.

1.3 Software Design

We required that our system operate with modal behavior according to a state machine. We preferred composability and maintainability, so we also needed it to be easy to analyze and debug. Given these requirements, we used a mediator-style architecture in ROS, a pub-sub message-passing middleware. In our design, a base component node publishes the status of the system with respect to its modal behavior, and each subcomponent node is responsible for taking actions to enforce the conditions of the particular state reported by the base node. With this structure, subcomponents are able to operate independently of other subcomponents and provide transparency for the execution model, as opposed to a monolithic timed loop that attempts to encompass all of the system's functionality in a single, sequential chain of logic.

2. IMPLEMENTATION DETAILS

2.1 Packages and Libraries

We made extensive use of several packages and libraries in our system, including:

- ROS, a robotics-oriented IPC runtime
- MAVROS, a ROS package for interfacing with devices using the MAVLink communication protocol such as our flight controller
- ar_track_alvar, a ROS package for detecting the pose of fiducial markers known as AR tags
- usb_cam, a ROS package for interfacing with USB webcams on a Linux host
- tf, a ROS package that serves as a universal interface for reading and manipulating spatial transformations

- mraa, a C library with Python bindings that enables access to the I/O pins of our onboard computer

2.2 Building the Quadcopter

For the quadcopter, our list of components includes a set of 30A ESCs, 920KV motors, propellors, four-cell lithium polymer batteries, a carbon fiber frame, a power distribution board, and the Pixhawk flight control unit (FCU). On the computational side, we used an Intel Joule single-board computer (SBC), a powered USB hub, a USB webcam, and a metal gear servo. The hub connected the FCU and camera to the SBC. The SBC was chosen for its processing power and the fact that it could run Linux, and consequently, ROS. The FCU was chosen on the basis that it could have 2-way communication with the SBC using the MAVLink protocol. Our system is connected as in Figure 1.

2.3 Motion Capture Algorithm and Implementation

Rigid body transformations are composed of a translation and a rotation and can be represented by a 4x4 matrix using homogenous coordinates. They represent the relative displacement in position and orientation between two frames of reference. Using ar_track_alvar, we are able to obtain a translation and rotation from our camera to an AR tag (Figure 2) with images our camera takes. Because we are able to get this information for each AR tag in an image, we can model the position of our quadcopter as a series of rigid body transformations and provide motion capture data for the FCU to consume.

We printed out 100 19.5 cm AR tags, sequentially arranged them in a 10 by 10 array backed with foamboard, and specified one to be the origin tag. By measuring the distance between centers of tags, we computed the rigid body transformation from the translation and rotation between each tag and the origin tag. In our code, we cached these inter-array transformations, intending for them to be static.

The rigid body transformation of AR tag n to the origin tag is given by

$$g_{ot_n} = \begin{bmatrix} 1 & 0 & 0 & (\text{n mod } 10) * (width + padding) \\ 0 & 1 & 0 & \lfloor \frac{\text{n}}{10} \rfloor * (height + padding) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Another static value we cached was the transformation from the FCU's frame of reference to the webcam's frame of reference. We mounted the camera facing directly downward on the left-side boom of the quadcopter frame, and so the transformation was as defined by a tf broadcaster we wrote as a spatial representation our airframe in ROS. We specified that the camera pitch was 90 degrees from the frame of the flight controller and translated by a vector of $\begin{bmatrix} 0 & .11 & -.07 \end{bmatrix}^\top$ meters, so the resulting rigid body transformation is given by

$$g_{cf} = \begin{bmatrix} cos(\frac{\pi}{2}) & 0 & sin(\frac{\pi}{2}) & 0 \\ 0 & 1 & 0 & .11 \\ -sin(\frac{\pi}{2}) & 0 & cos(\frac{\pi}{2}) & -.07 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The rigid body transformation from the camera to AR tag n, reported by ar_track_alvar, is given by Figure 3. This is the only non-static transformation in our motion capture system. With this, we can obtain the position of the quadcopter relative to the origin tag (assuming AR tag n is in view of the camera), $g_{of} = g_{ot_n} g_{ct}^{-1} g_{cf}$

We also included a launchpad area with several tags small enough such that our quadcopter can get its position when grounded at that area using the same method. With the launchpad, we defined the lateral displacement similarly to the main array, but manually tuned the Z-axis translation to account for the fact that ar_track_alvar believes that the tag is 19.5 cm wide like the other tags.

To improve the robustness of this system, we implemented a simple averaging step over all the tag transformations within a short period of time. By doing this, we mitigate the effects of outliers due to ar_track_alvar errors and prevent the system from failing if some of the tags within the camera's view are occluded. To implement this, we introduced two lists to simplify bookkeeping. Each list is 100 elements, with each index corresponding to a tag on our grid. One list contains the timestamp of the last time a tag was seen, and another contains the transformation from the FCU to the origin corresponding to that time. On every tf message, we lock our lists with a mutex, update the lists with timestamps and transformations, and then unlock it. Our localization loop, running at 10 Hz by default, acquires the lock and filters the timestamp list for indices that have been updated within the last 0.2 seconds. Then it creates a copy of the valid transformations and releases the lock for our update callback to continue updating the list. Locking these lists prevents valid transformations from being overwritten by the update callback to be something that doesn't correspond to the correct timestamp. Then, the loop proceeds to average the translations to obtain a final result, which it sends to the FCU.

The current iteration of this algorithm works well due to the fact that we are dealing with only 100 tags. Bottlenecks at scale could include memory usage limitations, as well as slowdowns in numpy's filtering method that we use to find transformations with valid timestamps. In the future, the system could also be augmented in a number of ways to further increase robustness and performance and will be discussed in a later section.

2.4 What is ROS?

Robot Operating System, or ROS, is not a operating system in the traditional sense. Instead, it provides communication infrastructure for groups of computational units, a pattern commonly encountered in complex robotic systems. It primarily serves as a platform for developers to create easily reuseable robotic software, and reflects this goal in its design. The two primary constructs in ROS are nodes and topics. Nodes represent separate processes, while topics represent communication channels between nodes. Functionally, ROS operates on a publish-subscribe basis where a node can broadcast information and other nodes can subscribe to incoming broadcasts and act according to the contents of the broadcast message. Under the hood, ROS is a distributed peer-to-peer system that coordinates message passing through a name-service running in a master node.

2.5 Software Architecture

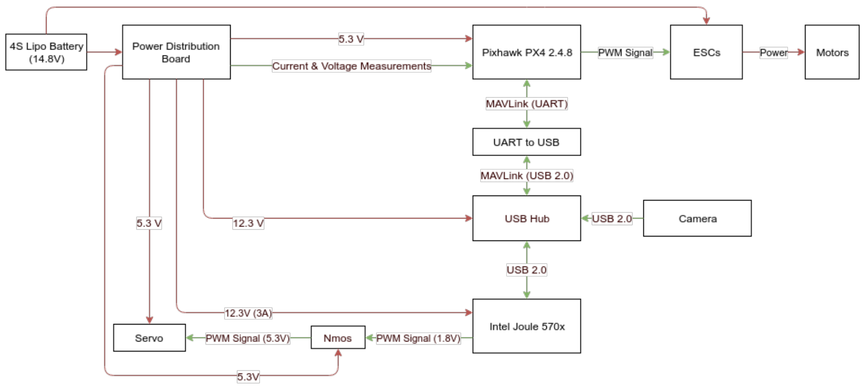

Figure 1: All of the quadcopter components wired up.

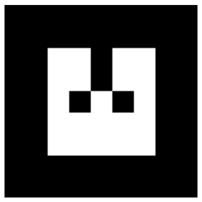

Figure 2: ar_track_alvar recognizes this fiducial marker, known as an AR tag, as ar_marker_0.

$$g_{ct_n} = \begin{bmatrix} \cos(\alpha)\cos(\beta) & \cos(\alpha)\sin(\beta)sin(\gamma) - \sin(\alpha)\cos(\gamma) & \cos(\alpha)\sin(\beta)\cos(\gamma) + \sin(\alpha)\sin(\gamma) & T_x \\ \sin(\alpha)\cos(\beta) & \sin(\alpha)\sin(\beta)\sin(\gamma) + \cos(\alpha)\cos(\gamma) & \sin(\alpha)\sin(\beta)\cos(\gamma) - \cos(\alpha)\sin(\gamma) & T_y \\ -\sin(\beta) & \cos(\beta)\sin(\gamma) & \cos(\beta)\cos(\gamma) & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where the roll, pitch, and yaw of AR tag n to the camera is given by $\begin{bmatrix}\gamma & \beta & \alpha\end{bmatrix}^\top$
and the translation from AR tag n to the camera is given by $\begin{bmatrix}T_x & T_y & T_z\end{bmatrix}^\top$

Figure 3: The quadcopter's position, g_{ct_n}, given that tag n is in view of the camera.

Code specific to the operation of our quadcopter was placed in a ROS package called ”flyco”, an abbreviation of ”Flying Courier”. This implementation manifests itself in ROS in the computation graph on Figure 4.

The primary control mechanism of the system is the main status that flyco_base_node, the main node in our package, publishes to the topic ”/flyco/main_status” at 50 Hz. The main status encompasses both the FCU status from MAVROS, as well as the modal behavior that we would like to achieve in our system from the state machine in Figure 5.

The main status message contains a few key fields. It reports the system's status, the FCU's status, the current pose, and the position of the quadcopter's current destination. Besides changes in the FCU status, the main status also changes when it is issued a command on the topic ”/flyco/cmd”. As mentioned in the software design description, flyco_base_node serves as a dispatcher for the subcomponents subscribed to the topic ”/flyco/main_status”.

There are currently four subcomponents necessary to achieve our desired modal behavior:

2.5.1 flyco_flight_mode_manager

This node is in charge of making sure that the FCU is in the correct flight mode according to the main status. In general, radio-controlled aircraft have several different flight modes that represent varying levels of control over the aircraft. For example, a self-level mode is assistive and attempts to level the aircraft when the pilot releases the controls, whereas a rate mode would require the pilot to manually level the aircraft. In our case, the relevant flight modes for our FCU would be ”MANUAL”, ”POSCTL”, ”AUTO.LAND”, and ”OFFBOARD”. The FCU boots into ”MANUAL” mode and expects pilot input on an RC channel. ”POSCTL” and ”AUTO.LAND” are automatic modes in which the FCU attempts to hold the aircraft at its current position and land the aircraft, respectively. ”OFFBOARD” is the flight mode that allows the FCU to respond to setpoint commands that we can send from our SBC over a serial connection. flyco_flight_mode_manager uses a proxy to a MAVROS service at the topic ”/mavros/set_mode” that requests a change in flight modes from the FCU. Using this service, this node checks every main status update and makes sure that the flight mode reported by the FCU is appropriate for the current main status. If it isn't then it requests the change until it gets made. Aside from the FCU's internal failsafe system and our safety monitor, the flight mode manager is the only agent in the system that is capable of requesting a change in flight modes.

2.5.2 flyco_setpoint_manager

This node is in charge of sending setpoint messages to the FCU through the topic ”/mavros/setpoint_position/local”. It makes sure that the setpoint that the FCU receives always matches the destination listed in the main status as long as the main status says it should be navigating by setpoints. Another key role that this node plays is keeping the FCU in OFFBOARD mode for as long as it is required to navigate to a setpoint. The FCU has a safety feature that exits OFFBOARD mode and sets the flight mode to a failsafe mode if it has not received a setpoint message in 0.5 seconds. flyco_setpoint_manager not only gives the FCU directions, but also acts as a heartbeat to let the FCU know that things are fine.

2.5.3 flyco_indoor_safety_monitor

This node is in charge of enforcing safety constraints in our system. There are currently two safety checks that the monitor performs on every status update. It checks to see if the reported local position exceeds some boundary and also if the reported position differential exceeds a bound determined by an exponential moving average of past position differentials (Figure 6). If it detects some violation, it sends a command to set the system into a fault state from which no additional commands can be sent to the base node. The safety monitor also attempts to change the flight mode to the failsafe mode when a violation occurs.

2.5.4 flyco_path_manager

This node is in charge of navigating the quadcopter through a given path of waypoints. It executes a 10 Hz loop that attempts to set the main status to navigate to a setpoint if it has received a path. It listens for a path to follow on the topic ”/flyco/path” in the form of a list of positions and overrides any existing path that it was following and attempts to follow the new one if it receives one. At every status update, it checks if the reported position is at its current destination. If it is, then it either sets the next waypoint in the path as its destination if there are more waypoints remaining, or drops the payload by sending a PWM signal to the servo if there are no more.

3. RESULTS

The high level planning component of our project was very successful. The motion capture solution was accurate, the path navigation issued the correct commands, including the payload drop, at the correct locations, and the safety monitor, while on the conservative side, executed the failsafe procedures in the right conditions. Our software interface to the quadcopter was also successful, as we were able to take off and land with MAVROS, albeit with lateral drift. Where we fell short was getting the system to fly while using our motion capture system.

4. ANALYSIS

Our software architecture allowed us to analyze and test subcomponents individually to get the desired system behavior. Because the main loop of each subcomponent interfaces with the rest of the system via a single source of truth, the main status, we could look at each one and verify its logic and behavior as a single-input, variable-output system. Because each subcomponent's either outputs a request to change the main status or a request to MAVROS and the FCU, which we've abstracted, the requirements of each system, with respect to its direct impact on the physical world, are sparse and easy to analyze. The weakness of our architecture is that due to its high-level, multi-threaded nature, as well as the abstraction layer of MAVROS and the FCU, modeling the behavior of the subcomponents concurrently running in the physical world becomes more complex by orders of magnitude.

With regards to technical problems, we had some doubts about the effectiveness of the camera in tracking AR markers through takeoff. Our camera did not have an interface to control or monitor the focus and also used a rolling shutter. The consequences are that during liftoff, the loss of tag

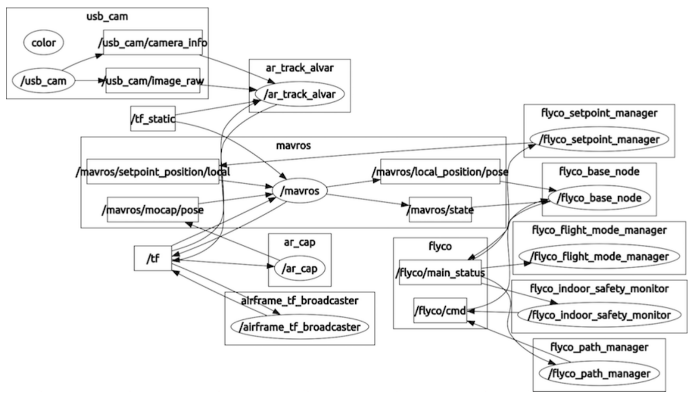

Figure 4: The compuation graph of our system. Nodes are represented as ovals while topics are represented as small rectangles. The ”/flyco/path” topic is not shown because no running nodes are publishing to it.

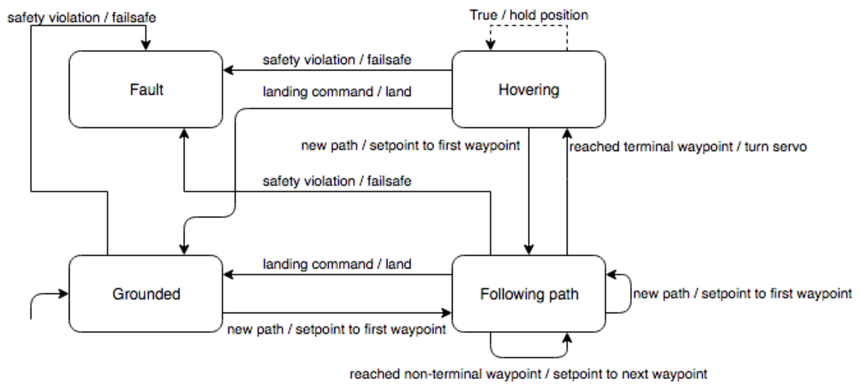
safety violation / failsafe
True / hold position
Fault
safety violation / failsafe
Hovering
landing command / land
new path / setpoint to first waypoint
reached terminal waypoint / turn servo
safety violation / failsafe
landing command / land
Grounded
Following path
new path / setpoint to first waypoint
new path / setpoint to first waypoint
reached non-terminal waypoint / setpoint to next waypoint

Figure 5: The state machine representing our system’s modal behavior.

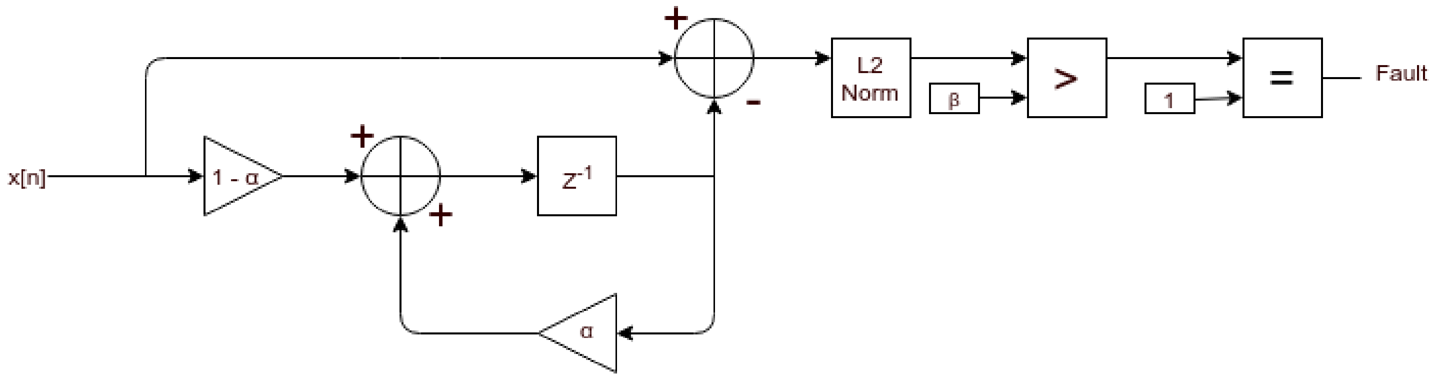

Figure 6: The model of the position differential check used in the safety monitor.

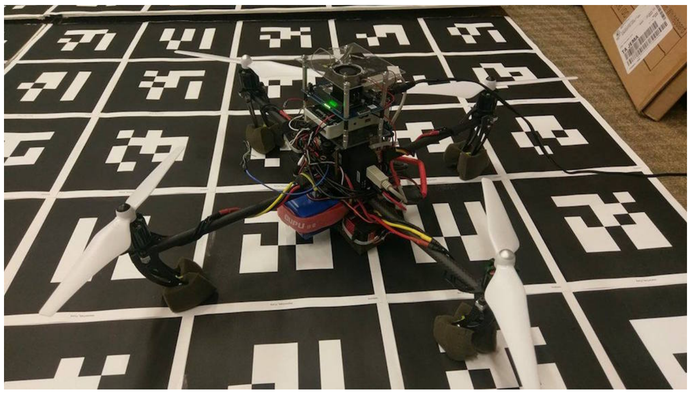

Figure 7: The completed quadcopter build, resting on top of our AR tag array.

tracking is essentially non-deterministic due to unknown focus and shutter distortion behaviors. This concern, however, was ultimately overshadowed by another problem. According to the MAVROS console log, the FCU immediately entered a fault state with some unlabeled behavior as soon as it was armed. Because our system design had abstracted away FCU operations and this issue fell outside of our model, we were unable to adequately analyze and remedy it. Despite this, we are confident that the system is close to correct and functional.

5. CONCLUDING THOUGHTS

Aside from getting our system in the air, we achieved good results. The quadcopter build was clean and also very powerful, and the software was easy to reason about and refine. We were able to cover and test much of our expected modal behavior. Most of our difficulties were due to being unfamiliar with the FCU and how it interacts with MAVROS and the rest of our system. We improperly assumed some things about the FCU, such as its capabilities and behaviors out of the box, which consumed a lot of debugging time. Not being familiar with the FCU was likely also what kept us from finishing the project as we described in the results.

If we had more time, we would more closely investigate the cause of the FCU fault state and hopefully be able to resolve the issue within the firmware. We would also like to experiment with different camera setups, varying on the number, type, and positioning of cameras. Having multiple global shutter cameras with varying fixed focus levels may improve tracking.

In terms of expanding on this project, we would like to improve our motion capture system. The current implementation of our motion capture would not scale well, as the amount of memory required would increase linearly with the number of tags in use. It would be good to refactor our code so that it scales better. One approach would be to intelligently cache tag transformations based on the reported position so that far-away tags would not have to stay in memory. With regards to performance, we could further mitigate the effects of outliers from tracking errors by dropping out transformations with high variance from being averaged, or by using a random sampling technique such as RANSAC, or both. We could also explore the option of extending ar_track_alvar's bundle tracking implementation. We initially tried to treat our AR tag array as one large bundle, but the output was very erratic and unreliable.

Though we failed to use the quadcopter in the final iteration, our motion capture system was more successful than we had anticipated it would be. Given the opportunity, we would like to continue working on this project to deliver not only a finished product, but a polished product.

Our code can be found at github.com/tasilb/FlyingCourier

6. ACKNOWLEDGMENTS

We would like to give a big thanks to the wonderful course staff for all the help, and for a great semester!

7. RESOURCES

Below is a list of we resources used for this project.

- Basic quadcopter information (https://myrcdrones.wordpress.com/2015/05/23/hello-world/)
- Hardware configuration calculation (http://www.ecalc.ch/xcoptercalc.php?ecalc&lang=en)
- Thrust and component selection (https://www.rcgroups.com/forums/showthread.php?2376436-Multistar-Elite-2216-920kv-3S-and-4S-Prop-Data)
- SBC datasheet (http://www.intel.com/content/dam/support/us/en/documents/joule-products/intel-joule-module-datasheet.pdf)
- SBC pin diagram (http://www.intel.com/content/www/us/en/support/boards-and-kits/000022494.html)
- SBC user forum (https://communities.intel.com/community/tech/intel-joule/)
- PX4 developer's guide (http://dev.px4.io/)
- Getting started with MAVROS (https://404warehouse.net/2015/12/20/autopilot-offboard-control-using-mavros-package-on-ros/)