SoftwarePilot: An Open Source Middleware for Fully Autonomous Aerial Systems

Jayson Boubin, Zichen Zhang, Shiqi Zhang, Christopher Stewart
Department of Computer Scienc and Engineering, The Ohio State University

I. Introduction

Fully autonomous aerial systems (FAAS) execute dynamic missions based on data sensed at runtime, achieving high level goals with no human involvement. FAAS include unmanned aerial vehicles (UAV) for data sensing, and edge and cloud systems for data processing and decision making. FAAS are a considerable improvement upon unmanned aerial systems (UAS), which require human piloting and decision making. Autonomous execution allows FAAS to operate continuously, intelligently, and in environments that may be unsafe for humans.

FAAS application domains are many, including crop scouting, search and rescue, and battlefield surveillance. Each domain and application requires different configurations (e.g # of UAV, compute resources, search algorithms, classifiers, autonomy policies) [1]. Furthermore, FAAS operate in resource constrained environments, creating power concerns for both the UAV and edge.

FAAS designers require software that can be configured and can execute on configurable hardware. Many software suites exist for managing UAS [2]–[4] but these suites do not provide features that FAAS designers require. We present SoftwarePilot, a middleware that provides configurable software and hardware support to FAAS.

SoftwarePilot is a programmable middleware for FAAS. It uses a microservice architecture to allow programmers to easily extend existing system by adding support for custom recognition models, pathfinding algorithms, autonomy policies or other FAAS software components on most modern edge devices.

In this paper, we detail the basic design of SoftwarePilot, along with the implementation of representative FAAS routines using SoftwarePilot. SoftwarePilot is completely open source, with all code hosted on Github [5].

II. Design

SoftwarePilot is a middleware, facilitating communication between software on the edge and UAV in the sky. It provides a Java API for edge systems to control UAV flight actions and capture and retrieve sensed data. Currently, SoftwarePilot allows programmable Java routines written at the edge to communicate with DJI drones through the DJI SDK.

The SoftwarePilot middleware is built on a microservice architecture, allowing programmers to add and remove components as necessary. Microservices come in two forms, drivers and routines.

Drivers are Java programs that accept API requests over CoAP as remote procedure calls. Drivers function as sub-components of the API, with each driver implementing a series of similar API commands. For example, one driver may control UAV flight actions, while another provides access to recognition models.

Routines interface with drivers to create FAAS applications. Routines are programs that combine logic written in Java with API requests to create cohesive FAAS applications.

SoftwarePilot implements this microservice design using four logical layers: hardware, runtime management, API, application.

- Hardware layer: UAV, edge systems, networking devices, and hardware accelerators. FAAS have many applications, each requiring different architectural configurations. SoftwarePilot has a runtime management layer to allow for interface with most modern hardware devices.
- Runtime management layer: Containers and Virtual machines supporting a Java interface for loading microservices and a CoAP client to the API layer.
- API layer: Drivers and external applications that communicate through remote procedure calls over COAP. Drivers have the ability to invoke external applications to satisfy API requests, such as Python or bash scripts for specialty applications. SoftwarePilot provides the ability to invoke arbitrary commands to call these external applications at the driver level.
- Application layer: Routines combine driver requests to control UAV flight, gather and process sensed data, recognize objects, and make pathing decisions. The series of API calls and surrounding control logic can create a cohesive FAAS application.

Using this design, we detail the implementation of two characteristic FAAS applications, target recognition

and optimization (TRO), and target recognition and avoidance (TRA), in the next section.

III. Implementation

Using SoftwarePilot, we have implemented two characteristic FAAS applications, TRO, and TRA. Many FAAS applications can be decomposed into a few core kernels. FAAS are often sent into environments to collect data on objects of interest to operators. Objects could include diseased crops in a field or survivors in a disaster area. Often, recognition of the target is not enough, and operators desire a high quality image of the target, so an FAAS may wish to optimize its position to capture better images of the target. FAAS may also want to avoid certain objects during their search. For example, FAAS searching for diseased corn may want to avoid soy bean fields they may encounter. These tasks constitute common autonomy kernels that separate FAAS from UAS. Using drivers, routines, and external applications, we were to create two applications that performed TRO and TRA.

To support these routines, we constructed a driver that could move the UAV within its environment and capture images using the UAV camera and send them to the edge for classification. We decided to use facial recognition for object detection and avoidance, as high quality facial recognition models are readily available.

We constructed two routines to perform TRO and TRA. Both routines constrain the drone to flight area of 3 cubic feet, and support 3 gimbal pitch angles. both routines explore the area using an A* based pathfinding algorithm from prior work [6].

TRO searches for faces in the flight area. TRO chooses the path with the highest corresponding value output from the pathfinding algorithm. This results in an FAAS that moves steadily through its environment towards better pictures of the target, a human face.

TRA avoids faces in the flight area. TRA chooses the path which leads it to the worst picture of a face in its environment. TRA moves slowly away from the face to the point where its images no longer contain faces.

These two routines can be executed using our middleware on any machine that can run Docker and VirtualBox. We have tested them on Mac, Linux, and Windows laptops using facial recognition from OpenCV and DLIB. These routines, like the rest of our middleware, are open source, and hosted on Github.

References

[1] B. Boroujerdian, H. Genc, S. Krishnan, W. Cui, A. Faust, and V. Reddi, "Mavbench: Micro aerial vehicle benchmarking," in *MICRO*, 2018.

[2] J. L. Sanchez-Lopez, R. A. S. Fernández, H. Bavle, C. Sampedro, M. Molina, J. Pestana, and P. Campoy, "Aerostack: An architecture and open-source software framework for aerial robotics," in *International Conference on Unmanned Aircraft Systems*, 2016.

[3] L. Meier, P. Tanskanen, L. Heng, G. H. Lee, F. Fraundorfer, and M. Pollefeys, "Pixhawk: A micro aerial vehicle design for autonomous flight using onboard computer vision," *Autonomous Robots*, vol. 33, no. 1-2, 2012.

[4] DJI, "Prerequisites-dji mobile sdk documentation." https://developer.dji.com/, 2018.

[5] J. Boubin, C. Stewart, S. Zhang, N. T. Babu, and Z. Zhang, "Softwarepilot." http://github.com/boubinjg/softwarepilot, 2019.

[6] J. Boubin, N. T.R. Babu, C. Stewart, J. Chumley, and S. Zhang, "Managing edge resources for fully autonomous aerial systems," in *2019 IEEE/ACM Symposium on Edge Computing (SEC)*, IEEE, 2019.