# An Architectural Design for Autonomous and Networked Drones

Umut Can Cabuk*†, Mustafa Tosun *‡, Orhan Dagdeviren†, Yusuf Ozturk*

* Dept. of Electrical and Computer Eng., San Diego State University, CA, USA
† International Computer Institute, Ege University, Izmir Turkey
‡ Dept. of Computer Eng., Pamukkale University, Denizli, Turkey

***Abstract*—Drone technology is evolving rapidly, offering support for many new applications every day. Off-the-shelf drone products accessible at reasonable costs are enabling new applications and research directions from crowd management, product delivery, and smart cities to security and military applications. The potential of autonomous swarm missions opens up new research avenues; however, off-the-shelf drone products, as well as most commercial projects, lack features that are required by autonomous collective multi drone missions. One potential reason is that there is no consensus on how to develop drones with cooperative operation capacity systemically. This study suggests a generalized device architecture concerning both hardware and software organization to aid developers of cooperative drones designed for multi-drone missions. The proposed architecture grants autonomy and cooperation within a swarm. An additional solution is also provided to enable ad-hoc networking for drones when the drone platform does not natively support it. The architecture is developed and deployed on ModalAI VOXL m500 developer drone to achieve cooperative swarm sensing and is validated through use cases.**

***Index Terms*—Ad-hoc networking, Architectural design, Drone, Swarm, UAV.**

## I. Introduction

Modern drones cannot merely be treated as remote-controlled aeronautical vehicles; rather, they are powerful flying computers with a significant degree of autonomy and decent networking capacity. Although most commercial drones do not offer built-in swarming capacity and provide only rudimentary autonomy (limited to predefined paths), both features are already under development by researchers.

The wide availability and low cost of open-source drone hardware and software continue to enable drone research in both core technology areas and drone applications [1], [2]. Flight controllers and operating systems are now available from multiple vendors. For instance, how the motors and the peripherals must be controlled in the air can now be easily done through specialized hardware called flight controllers, such as PixHawk 4 (PX4), ArduPilot, etc [3]. The body frame has become a matter of taste, as there are plenty of modular options, like S500. Such advancements paved the way for more complicated devices that contain real computers running their own user-programmable operating systems, including Linux distributions or Robot Operating System (Robot OS - ROS).

User-programmable drones escalated the research and development processes to another level by enabling artificial intelligence, edge computing on the air, advanced autonomy, and autonomous swarming [4]–[6]. Although there are several research groups working on swarm intelligence and swarm applications, there is no consensus on the drone hardware and software architecture for enabling cooperating swarms.

In our research and development activities concerning autonomous swarm operations, we could not find a suitable reference design in the literature tailored for the hardware-software architecture of a swarm-enabled drone. This was the main motivation for designing our own model. Although the architecture, as well as other problems analyzed and addressed in this paper, can be generalized to many types of unmanned aerial vehicle/system (UAV/UAS) or even to terrestrial robots, we encourage the readers to focus on electric-powered rotary-wing vehicles as we limited our understanding of a drone to those.

### A. Related Works

The literature includes plenty of research introducing and discussing system architectures concerning drones. However, current research is mostly focused on either the organization of the flight controller and its peripherals [7], [8] or an entire wide area network ecosystem that consists of cellular stations, satellites, and other devices [9] in addition to the drones. What is less studied is how a flight computer should be implemented on top of the flight controller, as the ”brain” of a swarm-enabled drone.

Spica et al. [7] and Julie et al. [8] were two of the earlier works that proposed a hardware/software architecture for UAVs. Both proposals were detailed but primitive in the sense of what drones are capable of today. Their designs lead to advanced flight controller schemes rather than a programmable computer. Even more recent works continued to focus on the controller and its peripherals. Kakamoukas et al. [10] somewhat superficially discussed an architecture centered on the flight controller and a ground control station. Ladeira et al. [11] provided more details in their architecture concept, but again there is the flight controller, peripherals, and the connections between them. Campion et al. [9] emphasizes

The authors acknowledge funding from the Scientific and Technological Research Council of Turkey (TUBITAK), grant numbers 1059B142000439, 1059B142000564, and 121E500.

UAV-to-UAV communications but is highly superficial and ignores a flight computer that can potentially help manage such communication.

Bigazzi et al. [12] is a rare example of a comprehensive drone system architecture that emphasizes the role of a flight computer. The flight computer is based on NVIDIA Jetson Nano and benefits from multi-threading. While the design presented in this study allows a custom (so-called high-level) software to manage the drone and lets the drone to interact with peripherals, the scheme is not optimized for networking and swarm applications. Although there is a communication thread that handles remote control signals, it is built for coordinating the data flow between the high-level software and the flight controller, not for handling inter-drone communications or other kinds of wireless networking.

### B. Problem Definition and Contributions

This study addresses one major design challenge and a minor, more practical issue. The former is to provide a layered architecture for drone developers who work on autonomous swarms to follow, whereas the latter is an extension that allows drones to achieve ad-hoc networking in case it is not allowed by their original configuration.

The literature lacks a detailed (but also generalizable) architectural design for drones to be used in swarm missions covering both hardware and software modules, as well as inter-component communication. Existing works either lack the implementation details that may guide the developers (like how to handle the concurrency), leaving too much obscurity, or are hyper-focused on the flight controller and its relation with the device peripherals. On the other hand, our architecture design has the distinction of being a guiding light for developers who work on autonomous swarm operations.

Nevertheless, not all drone systems are perfectly built for swarming. In fact, most off-the-shelf products lack ad-hoc and mesh networking capabilities, even those targeting developers. They either lack the required hardware module or miss the firmware/driver support for the included module. Our development efforts rely on a particular drone system that is commercially available on the market but are applicable to a large variety of other systems with minor modification.

## II. Proposed Architecture

The proposed architectural design considers both hardware and software elements, as well as interconnections. The proposal considers the following design goals:

- A layered structure with high modularity to enable a flexible design that fits most applications.
- Parallelism with multiple pipelines and minimal resource usage to allow efficient cooperation and commanding.
- Ad-hoc networking to enable swarms without any infrastructure.
- Compatibility with existing or common sub-systems to achieve widespread support.

A comprehensive drawing of the proposed system architecture is given in Figure 1, where round shapes represent software components and sharp-cornered ones represent hardware components. Each drone in the swarm has exactly the same architecture.

### A. Hardware Components

ModalAI VOXL m500 development drone [13] is a rare example of commercial devices that are programmable and customizable to a large extent, with its own limitations like lack of ad-hoc networking capabilities or missing Bluetooth support, etc.

Illustrated in Figure 1, the hardware components of the system can be broadly categorized as the following (the actual hardware used in our implementation is given in the parentheses): the drone platform (Qualcomm Flight RB5), the computing platform (VOXL Flight) which is a single board that combines the flight computer (VOXL) and the flight controller (PX4) [14], and the external network interface (ESP32). Each drone is equipped with an ESP32 module to provide ad-hoc networking capacity through the ESP-NOW protocol. Motors, sensors, and other peripherals that are immediately driven by the flight controller are agnostic to this architectural design, yet their organization was already extensively studied [11].

In our test and development environment, namely on the m500 drone, most of those components were already present, except for the very essential ESP-32 module, which was integrated by the authors and programmed exclusively for this study. The long-haul radio (e.g., LTE or Microhard) and peripherals were optional.

### B. Software Components

The software components of the system can be broken down as follows: the flight computer OS and its services, MAVLink Router, Docker container, MAVSDK Server, the custom user program (our swarm mission program) consisting of a main thread and a listening thread; where the former also hosts two asynchronous event loops for mission and order handling functions. The flight controller also runs a customized OS, which is usually ignored for high-level designs. On the other hand, the external network interface also has its own software running.

The flight computer of VOXL m500 runs a customized Linux distribution labeled Yocto Jethro (Kernel 3.18). This "base" layer OS is responsible for keeping the device alive and connected as all the device drivers, fundamental OS services, and network connectivity (regardless of the network technology) are maintained directly by that layer. What is also running on this layer as a background service is the MAVLink Router program that forwards the MAVLink messages coming from upper layers to the flight controller (PX4) and vice versa.

However, since the MAVSDK Server application (another background service enabling casting, sending, and receiving MAVLink messages) is not implemented on the aforementioned base layer, it is not possible to govern the flight controller from there, which is an intentional design decision made

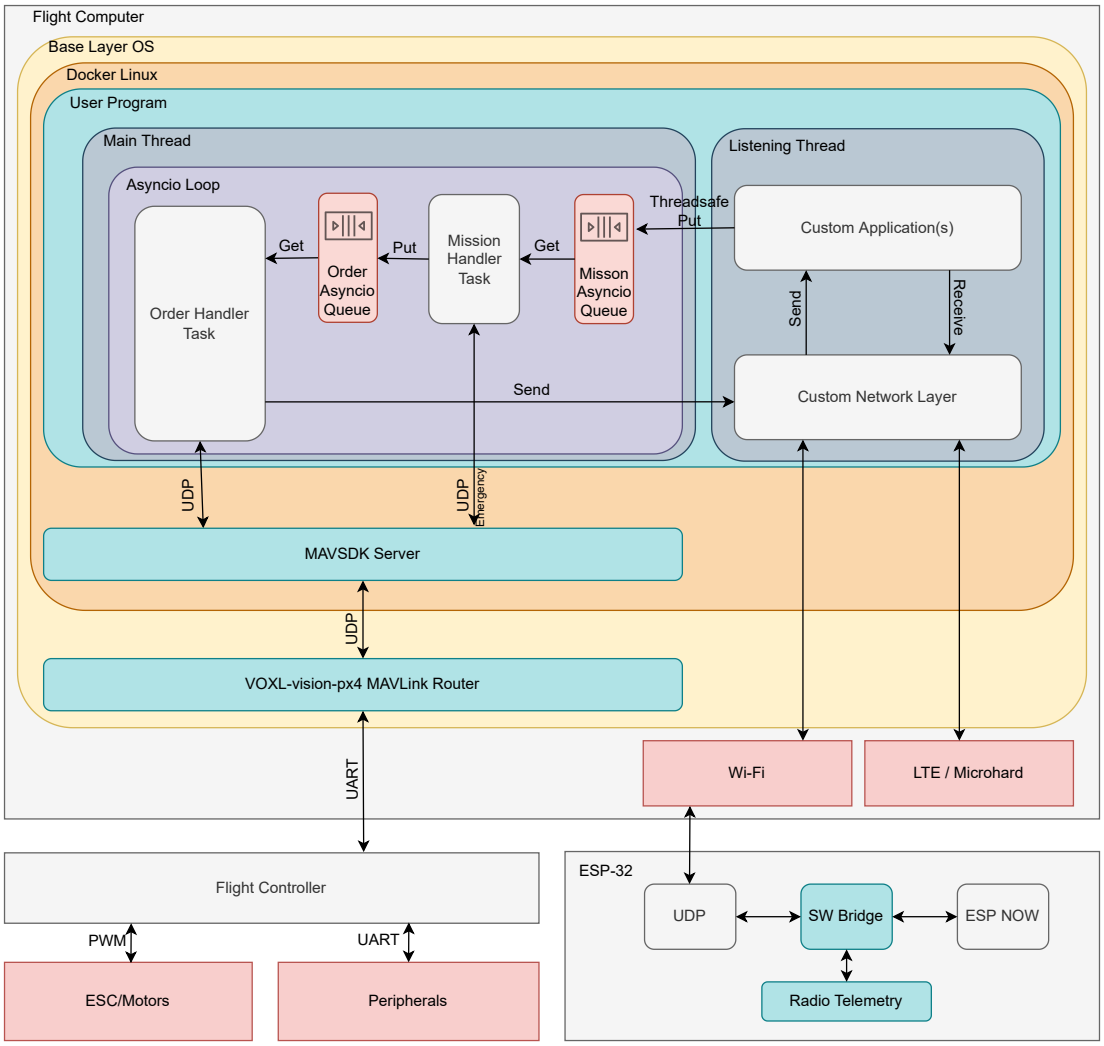


Fig. 1. Illustration of the proposed architectural design.

by the vendor that we would like to keep for device safety purposes. On top of the base, there is an abstraction layer made possible through Docker. A Docker container that runs a full-service Ubuntu OS is considered for running custom user programs, controlling the drone, and executing the missions. A custom program running in this layer (e.g., a Python program) must import the correct version of the MAVSDK library to enable communication between the MAVSDK Server application running in the background and the program functions.

The user program is where a swarm is built, missions are executed, and autonomy is provided. How this is made possible through concurrent agents is explained in the following section. Through the user program, more precisely, the custom network layer hosted by the listening thread, the architecture provides the developers to enforce their own network topologies and create mesh formations at their discretion. Yet, this can also be handled fully autonomously if needed. As most radio technology comes with decent medium access control (MAC) layer that is suitable for ad-hoc networking, there are not many reasons to rebuild a MAC layer; however, the network layer needs to be rewritten for most cases. This piece of software was completely developed by the authors, including the threads, loops, queues, etc.

Another essential software (per the minor problem discussed earlier) is the one controlling the external network interface, ESP32 in our case. This program runs exclusively on ESP32 on a continuous (i.e., loop) basis. Its mere functionality can be summarized as building a gateway connecting all the drones within its radio range. This program is mostly transparent to the custom user program that is running in the Docker container on the flight computer (but not to the base-layer OS). Each drone carries an ESP32 module and is connected to it via Wi-Fi. Yet, the drones do not directly communicate with each other; all the inter-drone communication is handled by their ESP32 modules through the ESP-NOW protocol. So, the network making the swarm is essentially a network of external interfaces, which are individually connected to their hosts (i.e., carrier drone).

### C. Inter-component Communications

Communication between threads is handled through thread-safe asynchronous queues (from the Queue function built into the asyncio library in Python) synchronous functions must be transformed into asynchronous coroutines to proceed. The communication between the handler functions in the main thread and the MAVSDK server is done through User Datagram Protocol (UDP) using local ports. MAVSDK reaches the MAVLink Router also via UDP. Under the hood, there is Google's remote procedure calls (gRPC) mechanism. This connection bridges the Docker container and the base OS. The flight computer sends and receives messages to and from the flight controller again via UDP, whereas the flight controller controls the motors using pulse width modulation (PWM) and finds peripherals over the serial port, Universal Asynchronous Receiver/Transmitter (UART) connection.

Drone flight computer may communicate with its external network interface through Wi-Fi in either of the station or access point modes only if there is support for it on both ends, which is very likely but not necessary. In our case, ESP32 has the support, alternatively universal serial bus (USB) connection could be used. Moreover, ESP32 modules communicate via ESP-NOW protocol, a custom broadcast protocol developed by Espressif. ESP-NOW runs on top of the physical layer and the MAC layer of Wi-Fi (802.11 family) but does not implement most parts of its network layer. The inter-node communication is done using MAC addresses as the IP stack is not implemented.

It is currently not known to us if any of the means of inter-component communications is prone to excessive delays or data losses, obviously, except for the Wi-Fi communication between ESP32 and the drone's Wi-Fi interface. This question is worth researching but left for future works due to the time and space constraints of this study.

## III. Autonomy and Multi-tasking

### A. Autonomy

The autonomy of an individual drone, as well as the entire swarm, is implemented through the concept of missions. A mission is a set of commands that should be executed when the drone is turned on. It may contain an arbitrary array of tasks and commands involving movement, computation, and communication. Typical commands include taking off, moving to a GPS-defined position, landing, sending heartbeat messages to potential neighbors, etc. A drone may run more than one mission consecutively but not in parallel. This limitation was considered intentionally in order to reduce the implementation complexity. The only exception to this would be the fail-safe behavior that could be triggered upon the reception of an emergency message or escalation of a predefined event. Nevertheless, a mission may contain tasks that need to be executed in parallel, such as sending a message while moving toward a destination. This is completely possible with the current scheme as long as command functions are implemented as asynchronous and atomic. Yet, there are more cases in the lower levels of the design that require multi-tasking.

### B. Multi-tasking

Our system design allows drones to form a swarm and execute cooperative missions that include both flight movements (e.g., taking off, hovering still, proceeding to a GPS position, landing, etc.) and passing executive orders to other drones in the fleet. As there is such communication, which may require the message-receiving drones to take action per the context of the incoming message, concurrency is inevitable in the architecture.

Using Python, and many other modern programming languages, different degrees of parallelism can be provided with any of these three common approaches (or a combination thereof): multi-processing, multi-threading, and asynchronous functions (also depending on the OS limitations). As all have their use and distinctive merits, the choice could not be made inattentively. Multi-processing is completely left out as an option for two reasons: first, it may not be suitable for many of the low-cost hardware (i.e., processors or micro-controllers) frequently used in cheaper drones; second, it requires more complex management for the memory and message passing operations without providing significant benefits, which may also result in higher energy consumption. A combination of multi-threading and asynchronous programming was preferred instead.

Figure 2 provides a general understanding of how concurrency is provided through the use of threads, asynchronous programming, and other mechanisms. Multi-threading is mainly used to separate the listening function (that handles the orders delivered by other drones) from the main workflow that dictates what the drone will do on its own. The main thread and the listener thread start running in parallel when the program is started on the drone. If the drone has a default list of actions (like taking off and flying somewhere), those actions are forwarded to the mission queue upon start. Regardless of the availability of default actions, the listener thread constantly checks the socket to find out if an order has been received by the radio (e.g., Wi-Fi). Likewise, whenever a new order is received, it is immediately pushed to the same mission queue, although it is hosted on the main thread. Other threads may have been implemented for various peripherals, like a camera, but omitted for the sake of simplicity.

How the queues are organized is shown in Figure 1. Both queues, as well as the mission and order handler functions, are implemented on the main thread. Mission handler and order handler functions are designed as asynchronous functions (built by the asyncio library of Python) and, thus, also run in parallel, although they are hosted in the same thread. Whenever the mission queue receives a set of orders, it parses them first and then evaluates them individually. If the order in consideration involves flight-related actions (like flying to a position or landing), the order is pushed into the order queue. If the order is related to communication (like sending a message to another drone or station), it is executed by calling

the relevant functions (like the message sending function). Additionally, the emergency orders (orders labeled with a flag indicating the emergency situation, such as an emergency landing) bypass the order queue and are executed immediately. Unlike the mission handler, the order handler does not fetch multiple orders at once but reads from the queue one by one. It executes the orders and manages any waiting time to prevent consecutive orders from being overwritten. Without such protection, a drone may attempt to land without reaching its destination in case it is required to go somewhere and then land. If it is intended to interrupt what a drone is currently doing and force it to do something else, such as landing, the emergency flag must be issued.

By the way, what we mean by executing a flight-related order in this section is actually limited to invoking the MAVSDK server program by using appropriate predefined functions, as the actual execution of the flight-related actions solely takes place on the flight controller (e.g., PX4), which also runs in parallel to the flight computer (on a different layer).

## IV. Validation

The proposed architecture was validated through the successful implementation of a use-case scenario on an experimental setup. According to the scenario, a swarm consisting of five drones was deployed, as demonstrated in Figure 3, in order to maintain continuous area coverage on some parts of a field. Such coverage may target sensing some features of the area, live imagery/surveillance, or provision of a temporal service (e.g., Wi-Fi network, cellular network, lightning, etc.). While we preferred to use Wi-Fi network provision to emphasize the connectivity, the actual scenario is agnostic to the architecture, and a ground station is also optional.

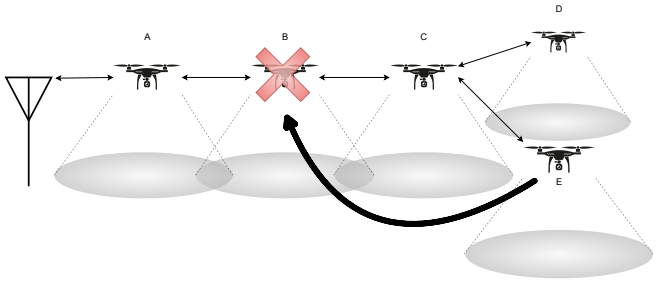


Fig. 3. Use case scenario where drone B fails and drone E is then ordered to take its place by drone C.

As per the scenario, one of the drones vital for the connectivity of the swarm and the persistence of the service (i.e., drone B in Figure 3) fails and loses its connection to the rest of the swarm. What happens after this moment is tightly dependent on the actually implemented topology control and routing algorithms on the custom network layer introduced in Figure 1. For the sake of example, a simple approach that lets one ”master” drone (i.e., drone C) to command another drone (i.e., drone E) to move to drone B’s last known position is presumed. The potential message traffic and the actions taken in such a case are illustrated in Figure 4.

Without using the proposed architecture and the developed pieces of software, it would not be possible to form a custom ad-hoc network, let drones command each other autonomously,

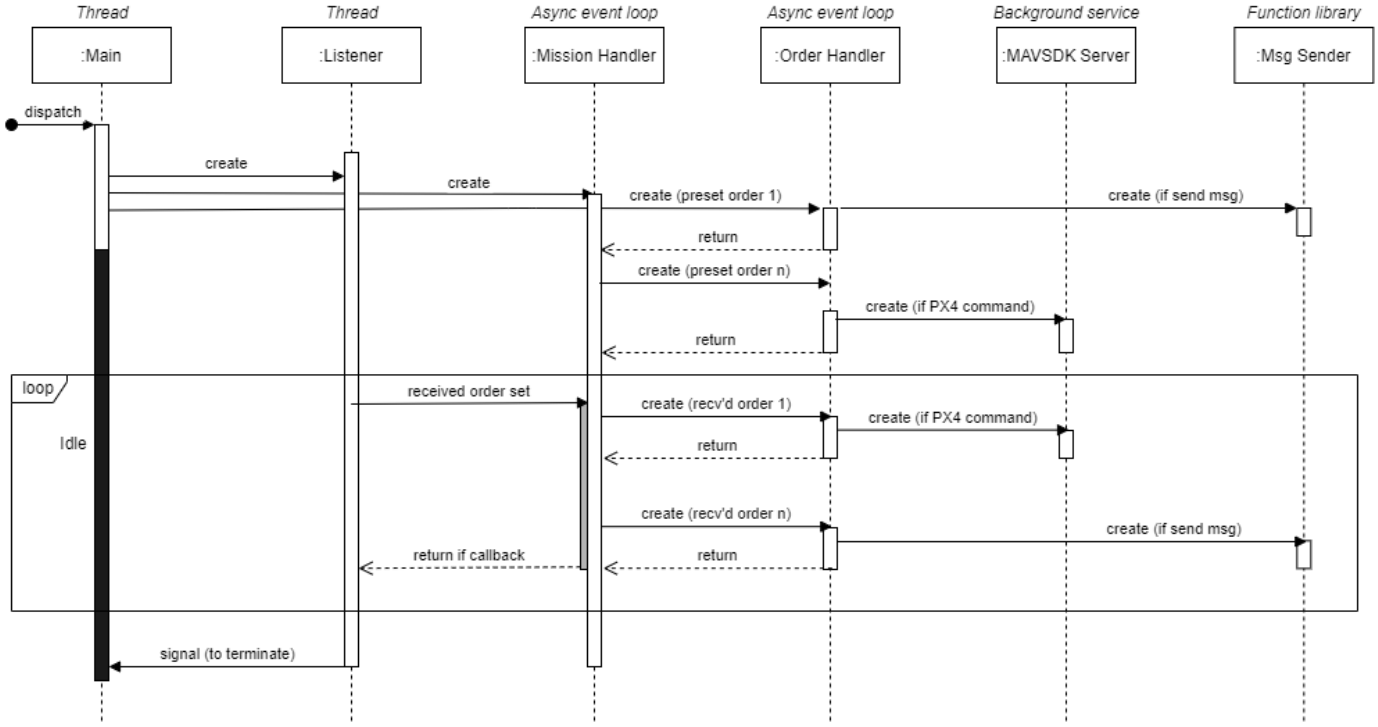


Fig. 2. Diagram showing how the proposed architecture handles concurrency.

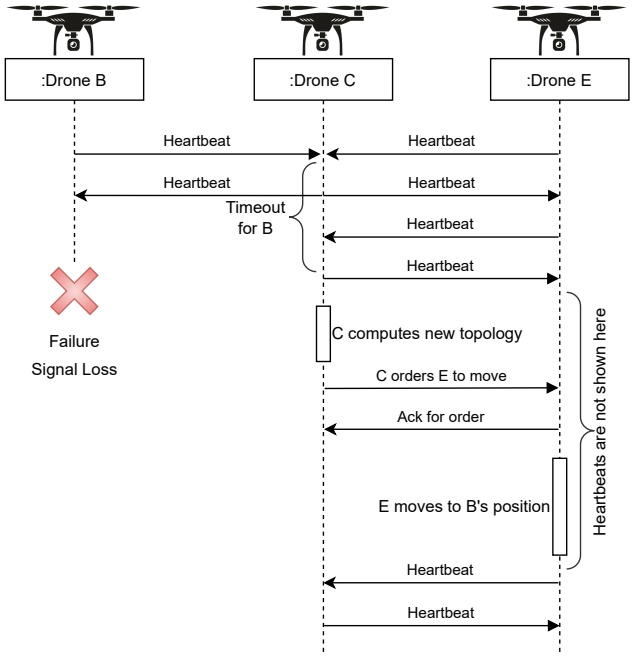


Fig. 4. Drone B fails, and drone E is then ordered to take its place by drone C.

and let drones interrupt other drones' ongoing tasks while enforcing them new higher priority tasks, with the testbed's standard configuration.

## V. Conclusion

This paper proposes a brand-new architectural design that is tailored for networked drones to be used in autonomous swarm operations. This design is intended to be a blueprint guide for the researchers and developers to follow when working on swarm missions. The architecture differentiates between a flight computer and a flight controller, making it unique among previous efforts. It addresses concurrency and autonomy requirements that may arise from swarming scenarios very well. Functional validation was made via the successful implementation of a use-case scenario. Our future research shall include topology construction and maintenance mechanisms and integration of other communication media, such as satellite and cellular networks.

## References


[1] Matt Schmittle, Anna Lukina, Lukas Vacek, Jnaneshwar Das, Christopher P. Buskirk, Stephen Rees, Janos Sztipanovits, Radu Grosu, and Vijay Kumar. Openuav: A uav testbed for the cps and robotics community. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 130–139, 2018.

[2] Na Lin, Luwei Fu, Liang Zhao, Geyong Min, Ahmed Al-Dubai, and Haris Gacanin. A novel multimodal collaborative drone-assisted vanet networking model. *IEEE Transactions on Wireless Communications*, 19(7):4919–4933, 2020.

[3] Anis Koubâa, Azza Allouch, Maram Alajlan, Yasir Javed, Abdelfettah Belghith, and Mohamed Khalgui. Micro air vehicle link (mavlink) in a nutshell: A survey. *IEEE Access*, 7:87658–87680, 2019.

[4] Wu Chen, Jiajia Liu, Hongzhi Guo, and Nei Kato. Toward robust and intelligent drone swarm: Challenges and future directions. *IEEE Network*, 34(4):278–283, 2020.

[5] Umut Can Cabuk, Gokhan Dalkilic, and Orhan Dagdeviren. Comad: Context-aware mutual authentication protocol for drone networks. *IEEE Access*, 9:78400–78414, 2021.

[6] Fawaz Alsolami, Fahad A. Alqurashi, Mohammad Kamrul Hasan, Rashid A. Saeed, S. Abdel-Khalek, and Anis Ben Ishak. Development of self-synchronized drones' network using cluster-based swarm intelligence approach. *IEEE Access*, 9:48010–48022, 2021.

[7] Riccardo Spica, Paolo Robuffo Giordano, Markus Ryll, Heinrich H. Bülthoff, and Antonio Franchi. An open-source hardware/software architecture for quadrotor uavs. *IFAC Proceedings Volumes*, 46(30):198–205, 2013. 2nd IFAC Workshop on Research, Education and Development of Unmanned Aerial Systems.

[8] Jean-Jacques Julie, Stéphane Kemkemian, and Julien Lafaix. Multi-sensors onboard uav, what is the most suitable processing machine architecture? In *2014 International Radar Conference*, pages 1–5, 2014.

[9] Mitch Campion, Prakash Ranganathan, and Saleh Faruque. Uav swarm communication and control architectures: a review. *Journal of Unmanned Vehicle Systems*, 7(2):93–106, 2019.

[10] Georgios Kakamoukas, Panagiotis Sarigiannidis, and Ioannis Moscholios. High level drone application enabler: An open source architecture. In *2020 12th International Symposium on Communication Systems, Networks and Digital Signal Processing (CSNDSP)*, pages 1–4, 2020.

[11] Matheus Ladeira, Yassine Ouhammou, and Emmanuel Grolleau. Towards a modular and customisable model-based architecture for autonomous drones. In *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1127–1128, 2020.

[12] Luca Bigazzi, Michele Basso, Enrico Boni, Giacomo Innocenti, and Massimiliano Pieraccini. A multilevel architecture for autonomous uavs. *Drones*, 5(3), 2021.

[13] ModalAI. *VOXL m500 Datasheet*, 2019. Available at: https://docs.modalai.com/m500-datasheet/.

[14] PX4. Px4 user guide: Architectural overview, 2021. Available at: https://docs.px4.io/master/en/concept/architecture.html.