Transparent Handover of Automated Drone Missions between Edge-based Control Stations

Theodoros Aslanidis
School of Computer Science
University College Dublin
theodoros.aslanidis@ucdconnect.ie

Manos Koutsoubelias and Spyros Lalis
Electrical and Computer Engineering Department
University of Thessaly
{emkouts, lalis}@uth.gr

Abstract

Drones are an attractive platform for carrying sensor/actuator payloads for different civilian applications. Thanks to modern autopilots, it is now possible to run missions in a fully automated way, using suitable mission control programs. To minimize latency, such programs can run at the edge and have direct wireless connectivity with the drone. However, edge machines can have a limited wireless range, which may not suffice to support the mission at hand. To address this problem, we propose a protocol and algorithm for the transparent handover of mission execution between different edge-based mission controller stations that collectively cover the full mission area. We provide a detailed description of the proposed protocol. We also discuss a prototype implementation and evaluate its performance using a realistic testbed, showing that the protocol overhead is sufficiently small to support a wide range of applications.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed systems; I.2.9 [**Robotics**]: Autonomous vehicles; J.7 [**Computers in Other Systems**]: Command and Control

General Terms

Design, Experimentation, Measurement, Performance

Keywords

Drones, Mission control, Automated mission execution, Ground control stations, Control handover, Edge computing

1 Introduction

Drones are used in an ever-increasing number of applications, such as smart cities [13], smart agriculture [10], and surveillance [15]. The cornerstone of this wide usage are modern autopilots, which handle complex flight control tasks while offering high-level commands to users. In fact, efforts are made to fully automate mission execution through suitable programming abstractions and middleware [3, 5, 8].

Such software typically relies on some telemetry protocol in order to receive status information and send commands to the drone. The general assumption is that the control station has a stable and fast wireless link to the drone. However, this may not hold if the drone needs to travel long distances. While 5G could provide a solution, the deployment of the required infrastructure is still ongoing, and certain regions may never have good connectivity. Also, satellite links are typically too expensive for low-budget systems.

To address the problem, one can exploit edge computing by deploying the mission controller on several edge nodes placed at different locations, so that the mission can be executed without any periods of disconnection. However, a suitable mechanism is required to implement the required handover between the mission controllers.

In this paper, we propose such a solution. The main contributions are as follows: (i) We propose a distributed protocol for the transparent handover of mission control between edge-based controllers. (ii) We provide a detailed, structured description of the protocol. (iii) We evaluate the performance of the proposed approach using a prototype implementation in a realistic testbed. Notably, our mechanism is orthogonal to the policy used to trigger such handovers. The investigation of handover policies is beyond the scope of this paper.

The rest of the paper is structured as follows. Section 2 presents the system model and proposed approach. Section 3 describes the handover protocol, while Section 4 provides a model for estimating the worst-case delay. Section 5 presents the evaluation. Section 6 discusses related work. Section 7 concludes the paper.

2 System Model & Approach

We assume drones with onboard navigation and obstacle-avoidance capability, where critical real-time control operations are executed locally by an autopilot system. The high-level coordination of the mission is performed by a separate entity, the mission controller, which collects information from the drone, takes decisions, and sends commands.

The actual mission logic is given in the form of a program, which invokes the services provided by the drone through a remote API. As illustrated in Figure 1, the mission program runs in the mission controller on top of middleware, which performs these invocations under the hood through a request-

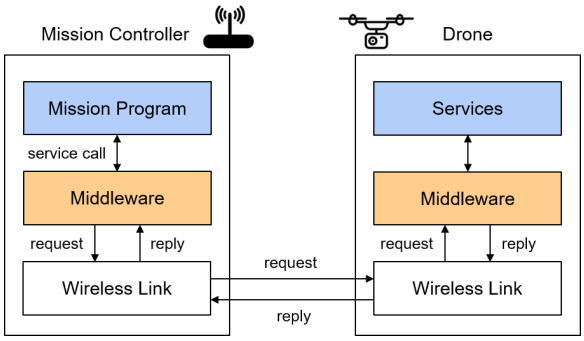

Figure 1. Software architecture.

reply protocol. On the drone side, the middleware intercepts such requests, invokes the local service, and sends the result back to the controller as a reply. The middleware recognizes duplicate requests to avoid performing the same invocation more than once. If a request has already been serviced, the produced result is sent again to the mission controller.

To cover larger geographical areas, one may run the mission controller logic on multiple edge nodes, picked so as to have uninterrupted connectivity with the drone during the entire mission, in the spirit of Figure 2. The requirement is for at least one node to be able to communicate with the drone at any point in time, but in the general case, several nodes may be able to communicate with the drone at the same time. The drone picks the controller deemed most suitable for the smooth execution of the mission, e.g., the one that is closer to its current position. We refer to the selected mission controller as the "master". When the drone decides to switch controllers, it informs the current master to perform a handover and accepts requests only from the new master.

In order to support such a handover of mission execution, we propose a protocol between the mission controllers (Section 3). The approach is transparent for the mission program, which has the illusion of running on top of a single mission controller with stable connectivity to the drone during the entire mission. The approach is also (largely) transparent for the drone-side middleware. The only change is to reject requests from a mission controller if this is not the selected master. Notably, the proposed approach is designed for deterministic mission programs, which take decisions based on information that is received via service invocations through the mission controller environment.

3 Handover Protocol

We assume that N edge nodes are selected to host the instances of the mission controller n_i, $0 \leq i \leq N-1$, so as to cover the entire path of the drone for the mission at hand. We arrange the controllers in a directed ring overlay, illustrated in Figure 2. Initially, the role of the master is assigned to the controller on the node closest to the point from where the drone takes off to start its mission, let this be n_0. The drone knows the location of the edge nodes, their communication range, and the initial assignment of the master role.

During execution, the master controller logs the invocations that have been performed between the mission program and the drone. The log is periodically flushed and circulated

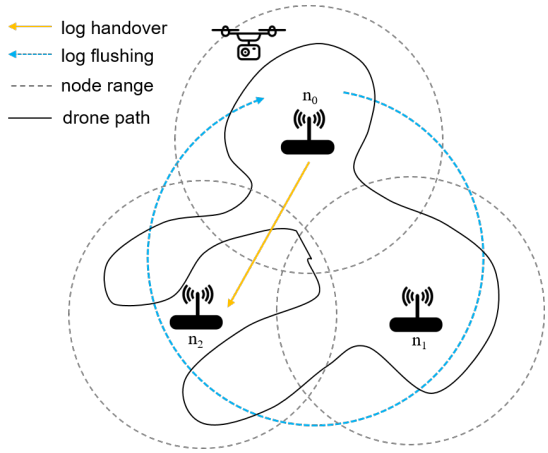

Figure 2. Multiple controllers for increased coverage.

along a ring, allowing all other controllers to fast-forward their execution concurrently to the actual execution. The ring topology is chosen so that the master controller only needs to perform 1 instead of $N-1$ log transmissions.

An algorithmic description of the handover protocol between the controllers, in conjunction with the handling of the drone invocations that are performed by the mission program, is given in Algorithm 1. For a more structured illustration, the logic is presented in an event-oriented manner.

Initially, each controller receives as parameters its own identifier, the number of controllers, and the log flush limit. Based on its identifier, the controller decides if it is the current master, setting its state to ACTIVE or PASSIVE accordingly. If ACTIVE, the controller establishes a wireless transport connection to the drone, used to invoke its services[1].

When the mission program invokes a service and the controller is in the ACTIVE state, the controller performs the request-reply interaction with the drone, writes the request sequence number (variable *reqnr* in Algorithm 1) and the received reply in the log, and increases the sequence number. Before returning the reply to the mission program, it is checked whether the log has reached the flush limit, in which case the log is sent to the next controller in the ring with the handover flag set to false (blue arrow in Figure 2). Note that the transmission of the log message can be performed concurrently with the execution of the mission program.

If the drone triggers a handover[2], the mission controller sets its state to PASSIVE and its master flag to false. Then, the log is sent directly to the new master with the handover flag set to true (orange arrow in Figure 2). When a controller receives a log message that does not concern a handover, it forwards it to the next controller in the ring (unless it has completed the circle). If the message arrives in proper order

[1] Depending on the implementation, the mission controller may be able to perform invocations without the prior establishment of a connection. Still, for reasons of generality, here we assume that a connection is needed before invoking the services of the drone.

[2] The mission controller may receive a handover message while waiting for a reply to a previously sent request. In this case, too, the handover message is handled as usual. To focus on the essence, this exception is not shown in Algorithm 1.

Algorithm 1 Mission execution and handover algorithm.

```
 1: upon initialization for id, N, flush do
 2:     reqnr, lognr ← 0, 0
 3:     log, logQ ← ∅, ∅
 4:     prvid, nxtid ← (id − 1) mod N, (id + 1) mod N
 5:     if id = 0 then
 6:         state, master ← ACTIVE, true
 7:         connectToDrone()
 8:     else
 9:         state, master ← PASSIVE, false
10:     end if
11: end

12: upon invocation req when state = ACTIVE do
13:     sendDrone(⟨REQUEST, id, reqnr + 1, req⟩)
14:     recvDrone(⟨REPLY, rpl⟩)
15:     append(log, reqnr + 1, rpl)
16:     reqnr ← reqnr + 1
17:     if size(log) ≥ flush then
18:         lognr ← lognr + 1
19:         send(nxtid, ⟨LOG, lognr, id, log, false⟩)
20:         log ← ∅
21:     end if
22:     return rpl to mission program
23: end

24: upon recvDrone(⟨HANDOVER, mid⟩) do
25:     state, master ← PASSIVE, false
26:     lognr ← lognr + 1
27:     sendLog(mid, ⟨LOG, lognr, mid, log, true⟩)
28:     log ← ∅
29: end

30: upon recvLog(⟨LOG, mlognr, mid, mlog, handover⟩) do
31:     if handover ∨ mid ≠ nxtid then
32:         sendLog(nxtid, ⟨LOG, mlognr, mid, mlog, false⟩)
33:     end if
34:     if mlognr = lognr + 1 ∧ state = PASSIVE then
35:         lognr ← mlognr
36:         log ← mlog
37:         master ← handover
38:         if master ∧ log = ∅ then
39:             state = ACTIVE
40:             connectToDrone()
41:         else if log ≠ ∅
42:             state ← REPLAY
43:         end if
44:     else if mlognr > lognr + 1
45:         insert(logQ, ⟨mlognr, mid, mlog, handover⟩)
46:     end if
47: end

48: upon invocation req when state = REPLAY do
49:     rpl ← getReplyFromLog(log, reqnr + 1)
50:     reqnr ← reqnr + 1
51:     if reqnr = getLastReqNr(log) then
52:         log ← ∅
53:         state ← PASSIVE
54:         if logQ = ∅ ∧ master then
55:             state ← ACTIVE
56:             connectToDrone()
57:         else if logQ ≠ ∅ ∧ head(logQ).mlognr = lognr + 1
58:             logmsg ← rmvHead(logQ)
59:             recvLog(logmsg) without forwarding
60:         end if
61:     end if
62:     return rpl to mission program
63: end
```

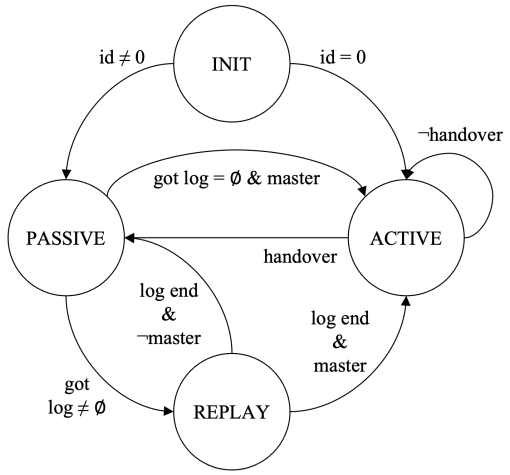

Figure 3. State diagram of the mission controller.

and the state is PASSIVE, the controller copies the log locally and sets its state to REPLAY to adopt a special mode of mission program execution (see below). If the log message concerns a handover, the controller sets an internal flag to become the new master. Then, a transition is made to the ACTIVE or REPLAY state, depending on whether the log is empty or not. The controller also forwards the log message (with the handover flag set to false) along the ring, so that all other controllers receive it and perform the replay as needed[3]. If the log message is out of order or the controller is not PASSIVE, it is placed in a queue to be handled later.

When the mission program invokes the drone and the controller is in the REPLAY state, the corresponding reply is fetched from the log (based on the request sequence number), without any interaction with the drone. If the end of the log is reached, the log is cleared and the controller makes a transition to the ACTIVE or PASSIVE state, depending on the setting of its master flag. If PASSIVE, the next log message in the queue (if any) is handled (without forwarding since this has already been done when it was first received). If the queue is empty, the controller remains in the PASSIVE state, and program execution is suspended.

Figure 3 shows the state diagram of the mission controller. Note that, depending on the flush limit and log transmission delay, there can be several log messages circulating along the ring one after the other, leading to several transitions between the PASSIVE and the REPLAY state. While a handover message (sent directly to the new master) can overtake normal flush messages (circulated along the ring), the orderly handling of log messages is ensured based on their sequence numbers (variable *lognr* in Algorithm 1).

4 Performance Model

We use a simple analytical model to estimate the delay of the proposed handover protocol, based on the following

[3]This log message will also be received by the old master, which will forward it along the ring but will ignore it based on its sequence number. This extra transmission can be eliminated by circumventing the old master in this particular circulation of the log message along the ring. To keep our description and implementation simple, we do not include this optimization in Algorithm 1.

assumptions: (i) mission execution in the REPLAY state has negligible delay; (ii) the transmission of log messages occurs concurrently with local processing and mission execution. Since these assumptions are idealistic, the formula provides a rough estimate for the handover delay.

Let l and b denote the communication latency and bandwidth between two mission controllers. Then, the time that is needed to send a log message of size s from one mission controller to another is $T_{log}(s) = l + \frac{s}{b}$, taking into account the time needed for the first bit of information to cross the network and the time that is needed to transfer the amount of information in the message. Also, let T_{conn} be the time it takes for the master controller to connect to the drone before resuming the proper execution of the mission program.

Then, the worst-case handover delay can be approximated as $T^{max}_{handover} = T_{log}(0) + h \times T_{log}(f+e) + T_{conn}$, where f is the log flush limit, e is the size of the largest possible log entry, and h is the number of hops along the ring between the old and the new master. This captures the scenario where the handover is triggered right after the master flushes the log with the largest possible size[4]. Even though the log in the handover message is empty (as it was flushed right before) the new master has to wait until it receives/handles the previous log flush message being circulated around the ring.

5 Evaluation

5.1 Implementation

We have implemented the proposed handover protocol in a Python-based middleware prototype. The drone environment supports two services: the *mobility service* with methods for sending navigation commands to the drone and getting state information, and the *camera service* with methods for controlling the resolution of the onboard camera and taking pictures. Furthermore, the drone environment contains the logic for the selection of the master mission controller and the notification of the old master about the handover. The controller environment maintains a proxy object for the drone with API bindings that can be used by the mission program to remotely invoke the available services. It also implements the handover protocol that was discussed in the previous section. The log is implemented as a list object and is serialized over the network using Python's pickle module.

The communication between the controller and the drone environments occurs over WiFi and TCP/IP. The connection is set up each time the controller becomes ACTIVE (the connection is closed when the controller becomes PASSIVE). The communication among the controllers is over Ethernet and TCP/IP. The corresponding connections are set up once, as part of the initialization. The endpoint information for all connections is included in a configuration file, which is read by these environments upon initialization.

5.2 Testbed Setup

We perform our measurements in a lab-based testbed, shown in Figure 4, with a system configuration that is practically identical to that of a real deployment in the field. A

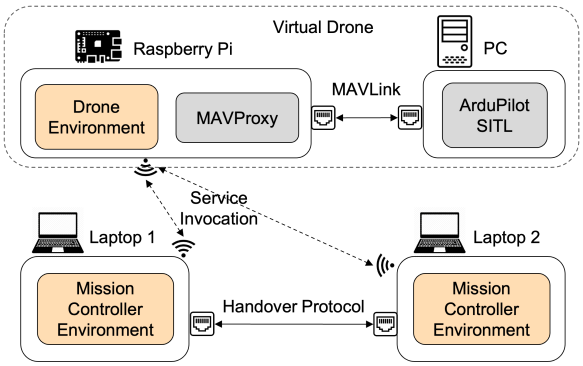

Figure 4. Testbed used for measurements.

Raspberry Pi (RPi) running the drone environment is used as a typical embedded companion computer for small drones. The instances of the mission controller run on two laptops, which communicate with the RPi over WiFi in ad-hoc mode. The laptops communicate over Ethernet-LAN. To reproduce typical Internet connectivity at the edge, we artificially slow down the communication between the mission controllers at a latency of 20 ms and a bandwidth of 25 Mbps. To focus on the core aspects of the handover process, we do not simulate physical link behavior as a function of mobility.

The mobility service of the drone environment on the RPi communicates with the drone's autopilot with MAVLink messages [11] via MAVProxy [12]. For the autopilot, we use the official Ardupilot software-in-the-loop (SITL) configuration [1] running on a PC. This is connected to the RPi over Ethernet and the MAVLink messages are exchanged over TCP/IP. Notably, the only difference from a real drone is that the autopilot runs on a separate onboard platform that is connected to the RPi over serial. We stress that the physics model in the SITL simulator reproduces real flight behavior with high accuracy. For convenience, the drone's camera service is configured to read and return a static 1 MB image.

5.3 Mission Program

To evaluate our handover mechanism, we use a simple mission program running in the above testbed setup, which directs the drone to visit specific waypoints. The program continuously polls the drone to retrieve its current position and when the next waypoint is reached it instructs the drone to take a picture. We run experiments for 25 waypoints arranged in a 5×5 grid, spaced every 50 meters. The mission altitude is 10 meters and the flying speed is set to 5 m/s.

To generate a large number of invocations, we let the mission program poll the drone every 1 second. This results in roughly 360 invocations during the core part of the mission (excluding the initial take-off and final landing phase). Notably, the mission program should not wait between invocations when executed in the REPLAY state. To this end, the mission program uses a *sleep()* method of the controller environment, which implements the proper behavior depending on the internal state of the mission controller.

To run tests in a controlled way, the handover points are specified in a configuration file of the drone environment. Each line has the form $\langle reqnr, mid \rangle$, where *reqnr* is the se-

[4]The log may grow (substantially) beyond the flush limit, as follows. Assume that the log has nearly reached the flush limit f. Then, as a side effect of the next invocation, the controller writes in the log a large entry of size e, which then causes the log to be flushed with total size $f+e$.

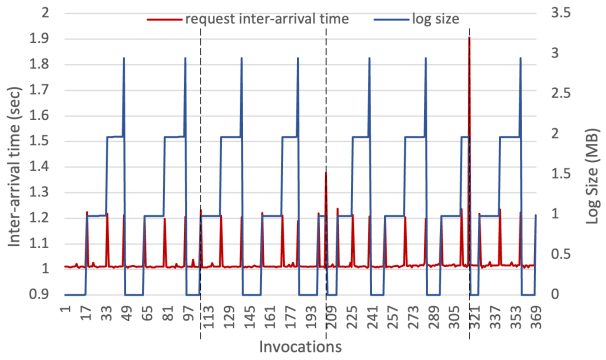

Figure 5. Request inter-arrival time and log growth for execution with flush limit 2 MB and three handovers.

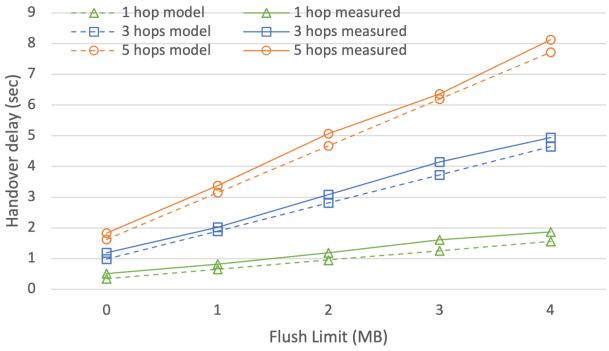

Figure 6. Worst-case handover delay with six mission controllers as a function of the log flush limit, for different hop distances between the old and the new master.

quence number of the invocation request after which the drone should trigger a handover, and *mid* is the identifier of the new master controller.

5.4 Experimental Results

Figure 5 shows the inter-arrival time (red line, first y-axis) for the invocation requests that reach the drone (RPi), for an indicative mission execution scenario where the log flush limit is set to 2 MB and three handovers are performed between two controllers during the core part of the mission (dashed vertical lines). The figure also shows the growth of the log at the master controller (blue line, second y-axis). The observed pattern is similar for multiple executions.

It can be seen that the request inter-arrival time is close to the invocation/polling period (1 second), as most invocations to the drone's services are fast. Larger delays occur periodically, when the mission program invokes the drone's camera service at each waypoint, due to the time needed to transfer the image (about 1 MB) from the RPi to the mission controller over WiFi. The extra delay due to the handovers can be seen indirectly via the peaks in the request inter-arrival time at the respective points of the execution. Note that this can vary significantly, depending on the size of the log at the handover point, from being practically negligible (first handover where the log is practically empty) to being comparable or even higher than the invocation of the drone's camera service (in the second and third handover where the log is 1 MB and 2 MB, respectively). Still, even in the last case, the handover cost of about 0.9 sec is perfectly acceptable for most applications. Further, periodic log flushing does not increase the inter-arrival time beyond its expected value. Therefore, one may adopt a more aggressive flushing policy to ensure that the size of the log remains relatively small at all times, in anticipation of possible handovers. This is even more important since the log may grow significantly beyond its flush limit (in our case, +1 MB) as explained in Section 4.

To study the worst-case handover delay for a wide range of scenarios, we use a configuration with six controllers and vary the number of hops between the old and the new master in the ring. To ensure that log message transmissions occur over the network, we place all even controllers on one host and all odd ones on the other (we only have two laptops), and run handover scenarios for 1, 3, and 5 hops, always between an even and an odd controller. To reproduce the worst case as discussed in Section 4, each handover is introduced right after the old master has flushed the log along the ring with the largest possible size. We also test different flush limits 0, 1, 2, 3 and 4 MB (at 0 MB, the log is flushed after each invocation). Figure 6 shows the average handover delay over multiple experiments versus the estimation of the model.

We observe that the delay grows as the flush limit increases, as expected, due to the longer transmission time of the log messages between the controllers. The delay also increases for a larger number of hops between the old and the new master, because the new master waits for a longer amount of time for the flushed log to arrive before it can resume the normal execution of the mission program. Note that the estimations of the analytical model are close to the measured handover delay. The reason the measured values are higher is that our testbed introduces artificial overhead due to the contention between several instances of the mission controller running on the same physical host.

From the above results, it is clear that the best strategy is to use a small flush limit. While frequent log flushing leads to a larger number of log messages (e.g., about 360 log messages are generated with a flush limit of 0 MB since the log is flushed after each invocation), this has no noticeable impact on the actual mission execution time. The reason is that the circulation of log messages along the ring and the replay of the mission program on the controllers is done in parallel to the execution of the mission program on the master controller. Also, frequent flushing reduces the size (and transmission time) of the individual log messages.

6 Related Work

The programming of robotic/drone applications has been extensively researched. As a result, different programming models, libraries, and middleware environments have been proposed. The Proto language [3] provides multi-robot programming support using parallel computing logic. Each robot acts as a computing device and all of them form a so-called amorphous medium used to perform the operations of the mission. Also, a function-oriented configuration method is used to map spatial operations into commands for each device and finally into instructions for each mobile

robot. Karma [5] is a system for programming and managing micro-aerial vehicle swarms that do not have any long-range communication capability. A central entity is responsible for coordinating the mission by receiving data from the drones and giving orders to them according to the mission objectives. However, this interaction occurs when the drones return to their so-called hive. TeCoLa [8] uses a high-level coordinated approach where a distinguished entity, the mission controller, runs a program that controls the mission by monitoring the state of the nodes, taking decisions, and sending commands to the vehicles. Each vehicle provides several services depending on its resources, while the mission program can invoke these services in a transparent way through RPCs targeting a single vehicle or a team/group of vehicles.

The objective of the above works is to offer convenient and powerful abstractions to the mission developer in order to simplify the implementation of mission programs. Our work is complementary to such efforts, as it addresses the handover between different mission execution controllers/managers covering different geographical areas.

There is a large body of work exploring handover heuristics and strategies at the level of the wireless network infrastructure. The authors of [7] introduce a deep reinforcement learning framework for the handover decision, considering multiple parameters such as speed, direction, and position of the drone, to prevent unnecessary handovers while maximizing the base station RSSI. Similar work is presented in [9], using a fuzzy inference system to decide the handover. [4] proposes a route-aware handover mechanism for drones operating in cellular networks, using the drone's location and velocity information to predict its future path and select the best base station. In [2], the authors use reinforcement learning to manage handover and resource allocation in cellular networks that serve both drones and terrestrial users, focusing on the challenges due to the interference of drones on user uplinks and the frequent handovers required for drones.

An approach utilizing UAVs as on-demand forwarding switches in SDN-based networks is studied in [16], leading to more efficient management and faster handovers, with reduced signaling overheads, end-to-end delay, and handover latency. The work in [14] presents an approach for managing handovers in UAV-based wireless networks in three-dimensional space, where drones act as base stations for other (terrestrial) users. The proposed mechanism adjusts the height and distance between drones and evaluates the optimal coverage decision algorithm using seamless handover success probability and false handover initiation probability. An intelligent handover control method for UAV-based cellular networks is also studied in [6]. It utilizes a deep learning model to predict trajectories and analyze positional relations, aiming at improving communication quality and reducing communication interruptions for the end-users.

All these works focus on the low-level handover management of wireless communication. In contrast, our system model does not rely on a cellular wireless network, but employs an edge computing approach for the mission execution itself, transferring control between different controllers with local wireless capability. To the best of our knowledge, our work is the first to address this problem. To this end, our handover protocol primarily addresses the continuity and consistency of mission execution, ensuring the necessary synchronization between the controllers. Notably, our mechanism is orthogonal to the heuristic for taking handover decisions, and it can be used in a straightforward way to support different strategies, e.g., based on the quality of connectivity, anticipated interference, or path planning information.

7 Conclusion

We have developed and evaluated a protocol for the transparent handover of mission execution between multiple mission controllers at the edge. Our evaluation shows that, when using a small log flush limit, the proposed approach can scale to a large number of mission controllers at a small overhead that is perfectly acceptable for applications that do not have tight coordination requirements.

Acknowledgments

This work has received funding from the Horizon Europe research and innovation program of the European Union, under grant agreement no 101092912, project MLSysOps.

8 References

[1] ArduPilot SITL Simulator. https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html.

[2] A. Azari, F. Ghavimi, M. Ozger, R. Jantti, and C. Cavdar. Machine learning assisted handover and resource management for cellular connected drones. In *IEEE Vehicular Technology Conference*, pages 1–7, 2020.

[3] J. Bachrach, J. Beal, and J. McLurkin. Composable continuous-space programs for robotic swarms. *Neural Computing and Applications*, 19(6):825–847, 2010.

[4] J. Bai, S.-p. Yeh, F. Xue, and S. Talwar. Route-aware handover enhancement for drones in cellular networks. In *IEEE Global Communications Conference*, pages 1–6, 2019.

[5] K. Dantu, B. Kate, J. Waterman, P. Bailis, and M. Welsh. Programming micro-aerial vehicle swarms with karma. In *Proc. ACM Conference on Embedded Networked Sensor Systems*, pages 121–134, 2011.

[6] B. Hu, H. Yang, L. Wang, and S. Chen. A trajectory prediction based intelligent handover control method in uav cellular networks. *China Communications*, 16(1):1–14, 2019.

[7] Y. Jang, S. M. Raza, M. Kim, and H. Choo. Proactive handover decision for uavs with deep reinforcement learning. *Sensors*, 22(3), 2022.

[8] M. Koutsoubelias and S. Lalis. Tecola: A programming framework for dynamic and heterogeneous robotic teams. In *Proc. Intl Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, pages 115–124, 2016.

[9] E. Lee, C. Choi, and P. Kim. Intelligent handover scheme for drone using fuzzy inference systems. *IEEE Access*, 5:13712–13719, 2017.

[10] P. K. R. Maddikunta, S. Hakak, M. Alazab, S. Bhattacharya, T. R. Gadekallu, W. Z. Khan, and Q.-V. Pham. Unmanned aerial vehicles in smart agriculture: Applications, requirements, and challenges. *IEEE Sensors Journal*, 21(16):17608–17619, 2021.

[11] MAVLink. https://mavlink.io/en/.

[12] MAVProxy. https://ardupilot.org/mavproxy/.

[13] F. Mohammed, A. Idries, N. Mohamed, J. Al-Jaroodi, and I. Jawhar. Uavs for smart cities: Opportunities and challenges. In *Proc. Intl Conference on Unmanned Aircraft Systems*, pages 267–273, 2014.

[14] K.-N. Park, J.-H. Kang, B.-M. Cho, K.-J. Park, and H. Kim. Handover management of net-drones for future internet platforms. *International Journal of Distributed Sensor Networks*, 12(3):5760245, 2016.

[15] E. Semsch, M. Jakob, D. Pavlicek, and M. Pechoucek. Autonomous uav surveillance in complex urban environments. In *Proc. IEEE/WIC/ACM Intl Joint Conference on Web Intelligence and Intelligent Agent Technology*, pages 82–85, 2009.

[16] V. Sharma, F. Song, I. You, and H.-C. Chao. Efficient management and fast handovers in software defined wireless networks using uavs. *IEEE Network*, 31(6):78–85, 2017.