Distributed Serverless Function Scheduling in Ad-Hoc Drone Networks

Giuseppe De Palma[a,b], Saverio Giallorenzo[a,b], Alexandre Heideker[a], Matteo Trentin[a,b,c], Angelo Trotta[a], Gianluigi Zavattaro[a,b]

[a] *Alma Mater Studiorum - Università di Bologna, Italy*
[b] *INRIA, Sophia Antipolis, France*
[c] *University of Southern Denmark, Sophia Antipolis, Denmark*

Abstract

The increasing use of Unmanned Aerial Vehicles (UAVs) in critical applications, such as disaster response, compels efficient communication and computation frameworks for highly dynamic ad-hoc networks. We present an interpretation of Function-as-a-Service serverless computing within the distributed settings of drone swarms to address their peculiar challenges in functionality deployment, resource allocation, and mission adaptability. We propose a novel two-layer network overlay architecture, combining a gossip-based topology management layer with a function scheduling layer to support distributed function scheduling. Our system introduces a declarative language of Ad-Hoc Allocation Priority Policies (AHAPP), tailored for ad-hoc drone networks, enabling flexible function deployment based on resource constraints and operational needs. The resulting combination addresses the volatility of UAVs networks by supporting execution semantics for stable- and dynamic-topology scenarios, function offloading, and resilience to network disruptions. We present experiments confirming that the features provided by our proposal support the efficient execution of serverless functions in ad-hoc drone networks, effectively handling their dynamic and heterogeneous nature, while achieving strong performance in terms of reliability, scheduling

Email addresses: giuseppe.depalma2@unibo.it (Giuseppe De Palma), saverio.giallorenzo2@unibo.it (Saverio Giallorenzo), alexandre.heideker@unibo.it (Alexandre Heideker), matteo.trentin2@unibo.it (Matteo Trentin), angelo.trotta5@unibo.it (Angelo Trotta), gianluigi.zavattaro@unibo.it (Gianluigi Zavattaro)

time, and communication overhead.

1. Introduction

The proliferation of unmanned aerial vehicles (UAVs), or drones, in civilian and commercial applications has sparked significant interest in developing efficient communication and computation frameworks for such decentralised, ad-hoc wireless networks, where each node participates in routing by forwarding data for others [1]. As these networks become increasingly complex, traditional computing architectures struggle to meet the demanding requirements of real-time data processing, dynamic resource allocation, and mission-critical operations [2].

Case Study and Motivating Example. The case study we present in this section is one of the possible use cases that our proposal can host. While we use the case throughout the article to explain our approach, in Section 6, we present evaluations regarding generic simulations with different topologies and number of elements, besides the ones that simulate the case study.

We illustrate an example of such highly dynamic systems with a running case study, modelling a multiphase Fire Detection and Response scenario. We represent the phases in Figure 1, and comment on them below.

As part of its responsibilities, the agency managing a natural area actively monitors various potential threats to the environment. Among these, fire detection is a critical component of their surveillance efforts. As show in panel a) of Figure 1, the drones fly across a monitored area, using cameras and sensors to conduct continuous aerial surveillance.

To ensure early detection and a swift response, the agency conducts hourly fire monitoring routines using drones, performing real-time heat signature analysis, detecting and precisely geolocating fire origins. Once a drone detects a fire, it transmits comprehensive geospatial data to the command centre, providing critical first-response information that enables rapid and targeted intervention.

Following initial fire identification, the agency commands the drones in fire locations to execute a survivor detection routine, depicted in panel b) of Figure 1. Equipped with multi-spectral imaging sensors and using machine learning algorithms, they conduct scans through smoke, debris, and challenging terrain [3].

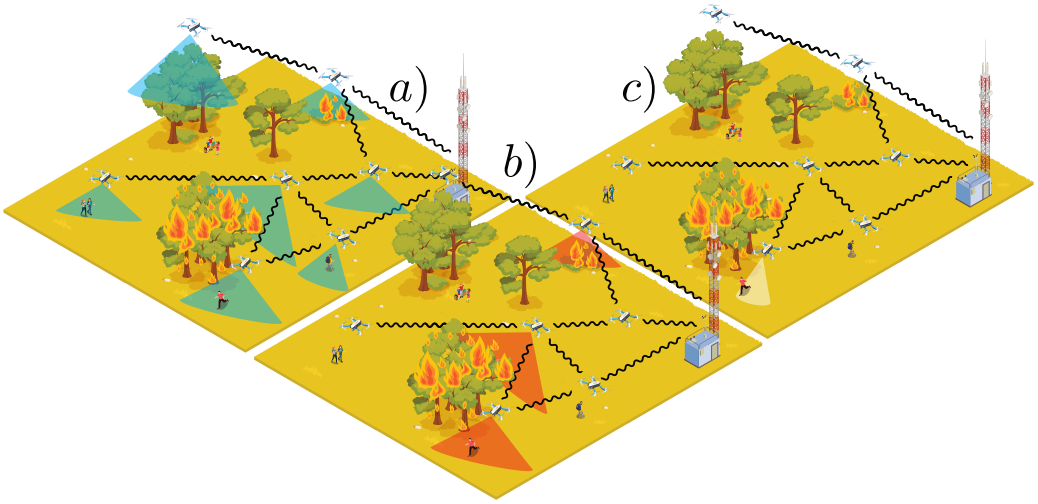

Figure 1: Depiction of the Fire Detection and Response case study main phases: a) Fire Detection, b) Survivor Location, and c) Real-Time Emergency Response Coordination.

The last phase of the case is that of survivor tracking, shown in panel c) of Figure 1. Given the information on the identified groups of survivors, the agency instructs the drones in the survivors' locations to follow a continuous tracking and information transmission routine [4]. Precision GPS equipment and real-time video streaming enable persistent monitoring of survivor locations. The drones can generate comprehensive data reports that include exact coordinates, visual tracking information, estimated survivor conditions, and recommended access routes for rescue teams.

Ad-Hoc Networks Serverless Computing. In contexts like the one described in the above case study, Function-as-a-Service (FaaS) Serverless Computing [5] emerges as a promising paradigm to address the challenges of lightweight, scalable, and distributed management of computational workloads in drone ad-hoc networks — we provide background information on FaaS for the unfamiliar reader in Section 2, along with a general introduction to ad-hoc networks and issues with workload distribution, drone swarms in particular.

FaaS is a cloud computing execution model where cloud providers automatically manage the infrastructure required to run application code. This paradigm shifts from traditional server-based architectures to event-driven, stateless functions that the platform runs in response to user invocations.

We argue that scenarios that require rapid deployment, minimal overhead,

and efficient resource usage like the one presented above, hardly fit into the existing, "traditional" application deployments practices, where all the drones already preload all functionalities that one can trigger.

Indeed, we see at least two main limitations with the preloading approach. First, the drones have limited storage capacity, hence, loading many disparate software modules would unlikely be a viable course. Second, and more important, it is unrealistic that the designers of the system can foresee and program all the functionalities that one can need deployed on the drones. In the preload setting, missing functionalities require the drones to waste precious time to go back to base, update their firmware/programming, and return to the interested areas.

To make the above observations more concrete, let us consider our case study. While one might preset the drone swarm for detecting fire alerts, this specialisation would limit the capabilities of the swarm to only that kind of emergency. However, emergency scenarios are unpredictable and may require rapid deployment of new capabilities. For instance, if the fire happens near in an industrial area, the emergency protocol could require specific chemical detection algorithms, another example is extreme weather conditions, which would require specific flight patterns and sensing protocols. Focusing even more specifically on the case study, e.g., the machine learning models for heat signature analysis and survivor detection might need frequent updates to improve accuracy and adapt to new conditions. With preloaded functions, updating these models would require taking drones out of service for reprogramming, creating dangerous gaps in coverage. Moreover, from an efficiency perspective, the multiphase nature of the operation (detection, survivor location, tracking) means that most functions are not needed continuously. Preloading all functions would waste valuable onboard resources. Of course, considering circumstances much different from fires, like floods and earthquakes, would require the drones to host even more heterogeneous sets of functionalities.

Hence, preloading all possible functionalities for these events hardly fits the exceptionality and time pressure of each situation.

Similarly, more sophisticated machanisms, like dynamic loading approaches, while an improvement over preloading, face important limitations when compared a distributed serverless architecture. The requirement to pre-allocate storage space for potential modules remains a constraint [6], whereas a serverless approach enables drones to share computational resources and dynamically offload functions. The update process is also cumbersome with dynamic

loading, requiring individual updates to each drone, while we propose an interpretation of serverless that can make functions available across the network of valid drones. Furthermore, dynamic loading systems must still anticipate which modules might be needed and include their dependencies, limiting true flexibility. Serverless architectures overcome these limitations by allowing deployment of arbitrary functions without pre-planning module dependencies. Moreover, most critically, in centralised approaches, when network partitions occur drones that cannot communicate with the central orchestrator cannot obtain the new functionalities. In contrast, our distributed serverless architecture makes function scheduling available through peer-to-peer sharing even when disconnected from a central infrastructure, making it more resilient for ad-hoc drone networks operating in challenging environments — indeed, the inherent scalability of the serverless approach facilitates the dynamic addition/removal of network nodes, a common requirement in drone swarm operations.

The idea of marrying ad-hoc drone networks and serverless comes from recent advancements in serverless technologies, which allows the efficient execution of both the functions on edge devices and, more importantly for the ad-hoc context, the components of the serverless platform itself, like the scheduler of the functions. For instance, FunLess [7, 8] is a recent serverless platform tailored for private edge cloud systems that leverages WebAssembly[1] (Wasm) as its runtime environment. This feature offers several advantages, among which portability and consistent development and deployment across heterogeneous devices (as found in ad-hoc networks) and a reduced memory footprint.

While technologies like FunLess provide support for building a serverless platform for ad-hoc drone networks, its implementation presents peculiar challenges that we address in this article.

Specifically, we look at both the architecture of the serverless platform and the protocols that support its functionalities in an environment characterised by the latency and intermittent connectivity of mobile aerial communication. In doing so, we strive to provide reliable function execution and enhance resource allocation across heterogeneous drone configurations.

Supporting Ad-Hoc Serverless Computing. In Section 3, we present our approach, hinged on a two-layer network overlay system designed for drone

[1] `https://webassembly.org/`.

networks. The architecture we propose uses a "gossiping" layer, built on top of the physical plane of drones and base station, to manage the network topology and drone status tracking. The gossiping layer plays a crucial role in disseminating information about drone status, available resources, and network topology. On top of this layer, we find a "function scheduling" one, which handles function placement and routing. The system implements both *hierarchical* and *progressive* scheduling approaches, which capture the high volatility of ad-hoc drone networks. Specifically, under the hierarchical approach, the base station maintains complete network information and directly assigns functions, useful in more static situations, with broadband, low-latency communication. In the progressive approach, the base station and the drones collaborate in making routing decisions to identify the viable targets for function execution, based on local information — this second approach fits scenarios with higher latencies and narrower bandwidth, which prevent the base station from having an up-to-date and comprehensive knowledge of the status of the network.

Moreover, while traditional serverless settings consider a 1:1 ratio between a user invocation and the execution of the related function — logically, the invocations existentially quantify the functions — we note that, in ad-hoc network settings, the user might require the execution of a function on all *viable* nodes (e.g., the fire detection routine found in our case study). For this reason, we introduce a *universal quantifier* that allows users to indicate the execution of the same function on all valid drones. Recognising that the fluid nature of ad-hoc drone networks presents challenges in assuring the correct execution of functions — invocations could more likely fail than classical serverless deployment due to network latencies and loss of connectivity — we also introduce two types of execution semantics for functions, a "*strong*" and a "*weak*" one, with their related interpretation of the above quantifiers. Briefly, the strong semantics requires the exact execution of a function on either one specific drone or all viable drones, while the weak semantics offers more flexibility by allowing execution on *at least one* drone (so the function can run on a set amount of alternative nodes, while the user receives the successful response from the drones) or *almost all* drones (setting a timeout after which the user receives all provided responses). Since drones have limited hardware, we also introduce the possibility of *offloading* the execution of functions to external computation nodes, like cloud and edge nodes, when necessary and indicated by the user.

Governing Decentralised Function Scheduling. A hallmark of classical serverless computing is that all nodes that make up the computation units available to the platform are functionally equivalent, which allows the users to ignore the underlying topology and leave to the discretion of the platform the task of deciding where to allocate function instances. While all computation nodes are functionally equivalent, performance-wise, they might differ depending on e.g., their geographical position or their status. For instance, effects like *code locality* [9] — due to latencies in loading function code and runtimes — or *session locality* [9] — due to the need to authenticate and open new sessions to interact with other services — can substantially increase the run time of functions. This observation sparked extensive research into strategies [10, 11, 12, 13, 14, 15] to enhance function execution efficiency. However, using a single, universal scheduling policy is inadequate for addressing diverse requirements, which led to the development of Allocation Priority Policies (APP) [16], a declarative language that for the definition of custom scheduling policies, so that the serverless platform can support multiple scheduling approaches simultaneously, each tailored for specific function groups.

In the UAVs context, nodes are not only performance- but also functionally-wise different, e.g., if we need to run a video scan of a specific area, only the drones equipping a camera and hovering that area are valid execution nodes.

For this reason, we introduce and integrate in our platform a new language, inspired by APP for Ad-Hoc Networks, called AHAPP (Ad-Hoc APP), which we present in Section 4. We deem AHAPP a new language because it mainly takes inspiration from APP regarding the definition of policies. However, in APP, the serverless scheduler holds a copy of a script that describes the scheduling of all functions; on the contrary, since in our platform all nodes can act as a scheduler, we pair a function execution invocation with its AHAPP script that describes its scheduling logic — i.e., the function "travels" with its scheduling script.

At its core, AHAPP allows users to specify whether functions should run on all drones or just one — with variations for the weak semantics. The language specifies resource-based targeting through its "attributes" system, which handles both physical properties, like geographical positions, and hardware requirements, such as available energy, memory, and storage but also access to rotors, cameras, or GPUs. AHAPP supports workflow management through the expression of *affinity* among functions (inspired by a recent dialect of APP [17]), allowing functions to specify dependencies on other functions that must (have) run on the same drone. Since more than

one drone can qualify for executing the function, AHAPP adopts, from APP, the concept of selection *strategy*, where the user can specify the priorities for choosing among alternative nodes. This feature allows users to refine scheduling based on factors like geographical proximity, load balancing, and energy reserves.

In Section 5, we delve into the details of the network protocol underlying our system. This protocol is essential for maintaining an accurate and up-to-date view of both the network topology and the status of individual drones within the distributed system and support the scheduling of functions. Relying on periodic message exchanges, the protocol addresses key challenges in mobile ad-hoc networks, particularly within drone swarms, where dynamic topologies, intermittent connectivity, and diverse hardware capabilities require robust distributed communication mechanisms.

Following the above design, we devise experiments to validate our approach, presenting them in Section 6. To assess the performance of distributed function deployment algorithms, we conduct extensive simulations across various application scenarios. Our evaluation highlights how the proposed decentralised scheme improves upon centralised scheduling models, maintaining a high service deployment ratio close to the optimal value while reducing reliance on global network state knowledge. Moreover, the interaction between knowledge dissemination, scheduling decisions, and deployment success will be carefully analysed, as stale or incomplete network information can significantly impact function placement and execution efficiency. The experiments confirm that the functionalities provided by our proposal support the efficient execution of serverless functions in ad-hoc drone networks, effectively adapting to their dynamic and heterogeneous nature. Our approach demonstrates strong performance in terms of reliability, scheduling time, and communication overhead, showing that decentralised function scheduling is both scalable and resilient under varying network conditions.

We conclude, in Section 8, by drawing final remarks and discussing future development.

2. Background

Before presenting our approach, in Section 3, we provide context for the key technologies and concepts underlying this research: serverless computing and ad-hoc (drone) networks.

2.1. Background on Serverless

Serverless computing has emerged as a response to the growing complexity and management challenges in modern cloud applications [5]. The core of the Serverless model is a Function-as-a-Service (FaaS) platform, which enables developers to build applications as compositions of stateless, event-driven functions that automatically scale according to demand. This model proves particularly cost-effective for applications with variable traffic patterns, as developers pay only for actual function execution time rather than maintaining constantly running servers. The key characteristic of these platforms is their infrastructure transparency. The underlying resources that power function execution (concretely, e.g., the cluster of nodes) are abstracted away, and developers need to only upload functions in any (supported) programming language and connect them to a triggering event, e.g., HTTP requests. When triggered by such events, the platform executes the relative function by selecting one of the available resources designed for execution and transmitting the invocation request to it. The function is then executed in ephemeral environments that can be instantiated and removed on demand. These environments, which may be implemented using virtual machines, containers, or specialised runtimes, handle all aspects of resource allocation and execution. Our research leverages this infrastructure transparency property of FaaS platforms in a novel context: ad-hoc networks.

2.2. Background on Ad-Hoc Networks

Ad-hoc networks enable decentralised communication without fixed infrastructure, making them suitable for dynamic environments, such as disaster response, military operations, and remote sensing [18]. Unlike conventional networks, these systems continuously adapt to changing conditions, requiring robust mechanisms for routing, resource allocation, and fault tolerance. Aerial ad-hoc networks, composed of drones, introduce additional complexities due to their three-dimensional mobility, limited energy resources, and highly dynamic topology [1]. Unlike ground-based networks, drone-based systems must continuously adapt to environmental conditions, network disruptions, and intermittent connectivity.

Traditional ad-hoc drone networks rely on pre-programmed behaviours and functions loaded at a base station, requiring drones to return for reprogramming when new functionality is needed [19]. This approach proves especially problematic in emergency scenarios like disaster response, where returning to base interrupts critical operations and wastes valuable response

time. Each round trip consumes significant battery power that could otherwise support mission-critical tasks, while the travel time reduces effective coverage and delays deploying new capabilities [20].

A more efficient approach would enable remote reprogramming by sending new functions directly to drones in the field, maintaining continuous operation, preserving battery life, and allowing rapid adaptation to changing mission requirements without physical recall. This remote function deployment, which aligns with our interpretation of serverless for ad-hoc networks, also enables dynamic load balancing and task redistribution across the network, critical for maintaining service when drones may fail or become unreachable in emergency situations.

However, bringing serverless computing to ad-hoc networks presents peculiar challenges. Traditional cloud serverless platforms rely on centralised schedulers found in complete topologies, i.e., the centralised scheduler can directly reach all nodes, and with complete system visibility, enabling precise, globally-controlled function allocation and resource management. In contrast, it is difficult to efficiently impose global coordination on ad-hoc networks, which frequently take (e.g., routing) decisions relying on (bandwidth-efficient but partial) local, aggregated knowledge.

Hence, implementing an ad-hoc network version of serverless requires scheduling to be lightweight, adaptive, and resilient to outdated or partial information. In particular, when dealing with drones, the resource constraints impose strict limits on function execution, demanding efficient scheduling routines that optimise energy and computational costs. Knowledge dissemination further complicates function deployment. In a cloud environment, function scheduling benefits from stable, high-bandwidth communication. In an ad-hoc network, global state propagation is impractical due to bandwidth and energy constraints, requiring decentralised approaches such as gossip-based information exchange. Function execution also faces unpredictability, as network disruptions can lead to execution failures, necessitating adaptive mechanisms.

Despite these challenges, serverless computing aligns well with the event-driven nature of drone networks. Modern, lightweight serverless execution environments, e.g., based on Wasm [8, 7], offer the possibility to both run functions and the serverless platform services across heterogeneous devices. By integrating decentralised scheduling, adaptive knowledge dissemination, and fault-tolerant execution, serverless computing can enhance the flexibility and efficiency of aerial ad-hoc networks. This work introduces a decentralised

function scheduling framework tailored to these constraints.

3. An Approach for Ad-hoc Serverless Computing

The approach that we propose consists of a two-layer network overlay among the drones and the Base Station, built on top of the physical plane, where the connections among the nodes in the network depend on the range and strength of the wireless signal and position of the drones w.r.t. each other and the Base Station.

In this section, we present the design and principles behind our approach. We concretely evaluate our proposal, e.g., the application of the different scheduling modalities, in Section 6.

We draw a schema of the overlay network we propose in Figure 2. In the figure, we depict, at the top, the physical plane, i.e., that of the physical drones and Base Station, under which we find the "gossiping" layer, dedicated to keeping track of the topology of the ad-hoc network, as well as the status of each drone. Specifically, this kind of information includes knowledge about the neighbours of each element (i.e., immediately reachable nodes) for each drone, as well as status information such as its available memory and storage. The drones spread information about their and their neighbours' status, e.g., whether their camera or GPU are available for use, and their current workload. In the figure, we denote these exchanges as $g(D_1, \cdots, D_n)$ to represent a gossip g about the drones with identifiers $D_1, \cdots, D_n$ — to illustrate the key mechanics of our approach protocol without too much detail, in the figure, we sketch a simplified sequence of gossiping exchanges among the drones and describe the protocol in full detail in Section 5. Gossip-based protocols are widely used in mobile ad-hoc networks thanks to their ability to minimise network congestion and provide robust, scalable communication. Unlike explicit point-to-point communications, which often require extensive control messages to maintain routing and scheduling state, gossiping relies on periodic, neighbour-only exchanges of compact status updates. Each node transmits its local state to immediate neighbours, who in turn disseminate this information further, enabling the network to converge toward a consistent, global view over time. This decentralised mechanism eliminates the need for coordinated broadcasts or central controllers, making it highly suitable for dynamic and infrastructure-less environments. As demonstrated by Shah [21], gossiping effectively reduces redundant message transmissions, lowering the likelihood of network congestion. More recent work by Fang

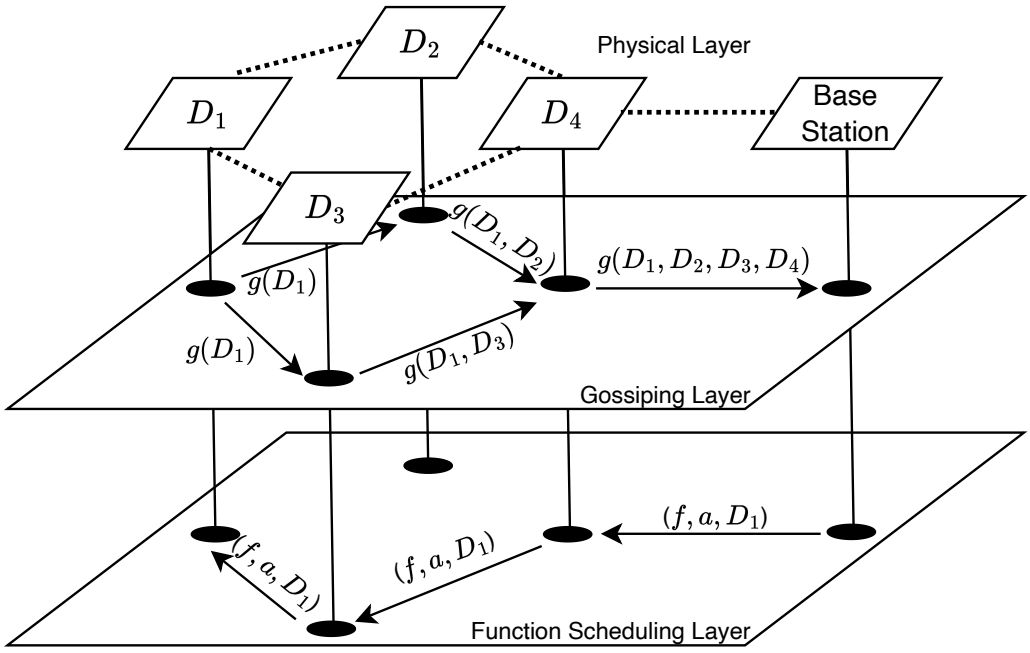

Figure 2: An example of the proposed overlay network.

et al. [22] reinforced this observation, showing that asynchronous gossip-based algorithms enable scalable, low-overhead coordination in distributed optimisation problems. Thus, in the context of ad-hoc networks serverless computing, gossiping can provide a lightweight and fault-tolerant substrate for disseminating state information, enabling each node to make informed scheduling decisions without relying on global knowledge or infrastructure.

In Figure 2, we find, at the bottom, the "function scheduling" layer, dedicated to the placement of functions and concerning the messages on the routing and allocation of functions, along with complementary messages such as the response to the user after the execution of a function. In the Figure, we represent these exchanges with the message (f, a, D_1), which we read as "deliver function f on drone D_1 following the scheduling policy a". As an example, we depict in the figure an illustrative flow that, starting from the Base Station, goes to drone D_4, which decides to forward the message to drone D_3, which finally relays the message to the addressee, D_1, that will execute the function. To inform these placement and routing decisions, the components at this layer use the information gathered from the gossiping one.

Closing the example, when the target drone (D_1) terminates the execution of the function, it sends back to the Base Station the result of the execution as a response. Similarly to the routing of the execution request message, the response message from the drone to the Base Station uses information from the gossiping layer to determine the path of the delivery. Since the response-relay part is standard for ad-hoc networks, we omit its description in the rest of our specification and focus on the traits distinctive of our approach.

As noticeable by the pairing of function execution requests with their scheduling policy, the function scheduling layer supports the definition of user-defined, per-function scheduling information expressed in the AHAPP language, discussed in Section 4. On the contrary, the gossiping layer runs a predefined behaviour on all nodes to disseminate status information within the network.

The schema in Figure 2 also illustrates another distinctive feature of our proposal. The functions that execute on the drones are not already present on the drones themselves. On the contrary, the functions "travel" with the request to run them. From a programming perspective, the creator and maintainer of the functionalities implemented by the functions is any user who has access to the platform and can issue the request to execute them at the Base Station. More practically, the user directly interacts only with the Base Station, by sending application-level messages, e.g., using REST or other protocols (the user-Base Station communication is an orthogonal detail to our proposal) and they do not directly interact with the function scheduling layer.

The gossiping and function-scheduling layers run in parallel and without coordination. At the gossiping level, the drones regularly exchange unsolicited messages about their status and that of their neighbourhood. At the function-scheduling level, users trigger the execution of functions on drones by sending requests to the Base Stations. The function-scheduling layer uses the information gathered from the gossiping layer available at the moment of scheduling to determine the allocation of functions on drones. Hence, the gossiping layer just informs the scheduling at each node (Base Station and drones), but it does not interact directly with the function-scheduling layer to take decision.

3.1. Rationale behind our two-layer approach

The reasoning behind our two-layer approach comes from the fact that we want to separate the scheduling concerns from the operations for network

connectivity, which are automatic and invisible to the users.

The gossiping layer builds directly on the physical plane, to construct a view of all the drones, performing distributed monitoring of their status. Given the low level of abstraction of this protocol and its relative independence w.r.t. the task of scheduling functions, we avoid embedding scheduling requests information into the messages exchanged at this level.

In the design of the scheduling layer, we capture the fact that different functions can require different amounts and kinds of resources, depending on the nature of the task they need to accomplish. This characteristic is a strong departing point between classical serverless computing and our proposal. Indeed, most serverless platforms define an arbitrary, hard-coded logic for the scheduling of functions. This approach assumes that all the nodes that make up the cluster of available targets for the execution of functions are functionally equal. We argue that, in ad-hoc networks, this assumption is unrealistic, since elements such as the position of the drone and the hardware it carries (e.g., some functions might require a GPU while others might need camera access, etc.) can determine the successful execution of a function. Therefore, we associate each function with a specific policy that informs the Base Station and the intervening drones on the requirements of the function under scheduling.

3.2. Hierarchical vs Progressive Scheduling

While associating a function with its scheduling policy helps users target the correct drones for executing the functions, lowering errors and increasing efficiency, we notice that, depending on the state of the network, the system can decide on which drone(s) to place a function in two main ways: a *hierarchical* and a *progressive* one.

In this section, we discuss the main trade-offs between these two scheduling modalities. We empirically explore the conjectures presented in this section in Section 6, through simulations.

In the first case, hierarchical, one can assume that the network topology is relatively stable — e.g., drones move slowly w.r.t. the propagation of messages — so that the Base Station has up-to-date information about the status of the drones to specify which drone(s) shall run the function under scheduling, leaving to the other nodes that connect the target drone to the Base Station (if any) the task of (efficiently) delivering this request. We represent this case in Figure 3. In the figure, we use the notation (f, a, D) to indicate a message for scheduling function f on drone D, following the policy

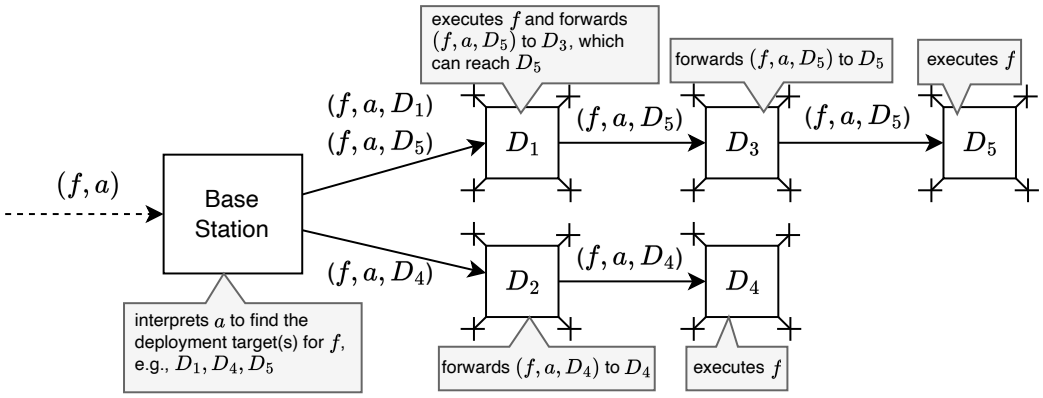

Figure 3: Example of hierarchical scheduling, scheduling layer.

a. Notice that the request sent by the platform's user — marked, in this and the subsequent figures in this section, using a dashed line to indicate the usage of a different communication protocol stack than the one among the Base Station and the Drones — that reaches the Base Station (from the left, in Figure 3) does not indicate a target. Indeed, while the user is unaware of the underlying composition and configuration of the drone network, it defines properties in the companion policy useful to identify valid executors of the function. As part of the hierarchical scheduling, it is the Base Station that decides on which drone to schedule the function. More precisely, while the typical use case for function execution is individual (one function, one execution node), in ad-hoc networks the users can require the execution of the same function on multiple drones, like, as presented in our case study from Section 1, when scheduling a function that needs to check for fires on all the areas covered by the drones of the network. As discussed in Section 3.3, the language for policies that we provide to users allows them to express the possibility of targeting multiple drones.

For simplicity, above and in the reminder of the section, we preserve the same notation f for functions in messages. However, we assume that the messages between the Base Station and the drones not only carry the code of the function as sent by the user but also a function instance identifier f_{id} defined by the Base Station to keep track of the function's lifecycle.

In the second case, progressive, one can assume that the network is highly dynamic, so that the Base Station cannot construct a reliable representation

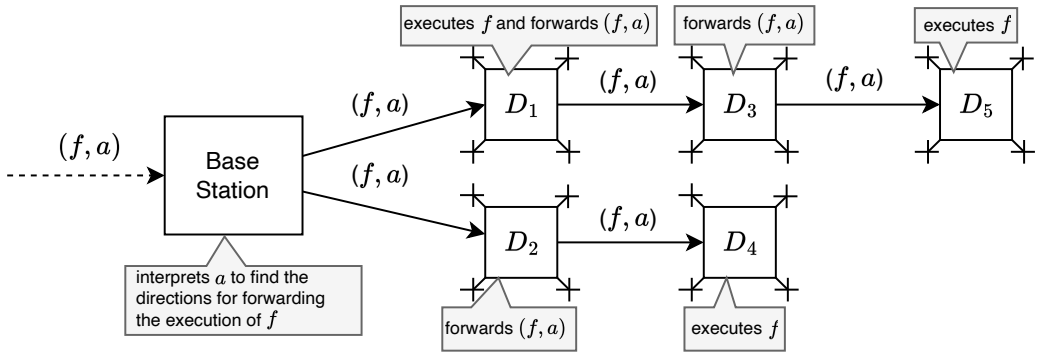

Figure 4: Example of progressive scheduling, scheduling layer.

of the status of the network based on the gossiping information. Since the Base Station cannot reliably identify the targets to execute the function, it uses the information it has to "push" the request in a *direction* that likely includes a target able to run the function. Hence, the Base Station does not indicate which drone shall run the function and rather forwards the request to the drones it can directly contact that lay in the target direction. In turn, the drones that receive the progressive scheduling consider whether they can execute the function or forward it to other drones in their neighbourhood that might satisfy the scheduling requirements. We illustrate the logic of progressive scheduling in Figure 4, reproducing the same function scheduling result of Figure 3. In particular, notice that, differently from Figure 3, the message transmitted by the Base Station and the other nodes has the same structure of the one sent by the user, i.e., it does not include an indication of a specific target drone, but rather uses the generic (f, a) form.

3.3. Execution Targets: Strong vs Weak Semantics

As mentioned, in traditional serverless platforms, users invoke the piecemeal execution of functions and ignore which target runs it[2]. However, in our context, users might need to run the same function on multiple targets, e.g., to gather information about the different areas they are in.

[2]This abstraction over the topology of execution nodes constitutes one of the hallmark traits of classical serverless computing, i.e., since nodes are all functionally equivalent, users can safely ignore which node executes their function.

Strong Semantics: "one" and "all". When our users specify a scheduling policy, they can indicate whether to run the function on "one" valid target — which captures the binding with an existential quantifier, i.e., we select one drone among those that can execute the function — or on "all" valid targets — which captures the binding with a universal quantifier, i.e., we select all drones among those that can execute the function.

Notably, the two invocation modalities do not only imply different targeting behaviours of the Base Station (and the intervening drones, in the progressive case), but also different timings for forwarding the response back to the user. Indeed, under the "one" scheduling request modality, the Base Station directly forwards back to the user the response it receives from the target executor (possibly dispatched through the other drones that connect them). On the contrary, under the "all" scheduling request modality, the Base Station implements a scatter-gather pattern, where it responds to the user only after it received all responses to the requests it sent to the target drones.

Of course, failures can happen. For instance, in the "one" case, the Base Station might not receive a response from the target drone, e.g., because the target correctly received the request but could not send back a response (e.g., because it became disconnected or due to malfunction). To prevent the Base Station from indefinitely waiting for a response, we pair a timeout to each scheduling request and, if the Base Station received no response after the given timeout, it reports the failed execution back to the user. By extension, under the "all" execution modality, we consider failed the execution of a function invocation when at least one of the requests did not return its response within the timeout. These issues include events such as communication problems (e.g., lost messages during both the scheduling and the response message deliveries) and node failures (e.g., drones that become irresponsive due to hardware malfunction). In all these cases, the only component that tracks the status of function executions is the Base Station, which follows the above failure-handling logic to make sure that user requests eventually terminate and report back to the user a response.

Since, in the above scheduling semantics, we impose strict requirements on the scheduling of functions, we call it *strong*, i.e., where the "one" scheduling modality entails the execution on exactly one target and the "all" modality requires all targets to return a response. Practically, a system can support this strong semantics if the underlying gossiping network can provide all the information about the drones, i.e., if the network enjoys a broad bandwidth

so that the drones exchange all the necessary information with the Base Station and the network can support the efficient and timely delivery of the coordination messages.

Weak Semantics: "at least one" and "almost all". When the connections among the drones are weaker, the network experiences narrow-bandwidth communications, which challenge the timely delivery of coordination messages and the effective propagation of the full amount of information about the status of the drones. In these cases, imposing the strong semantics presented above could be counterproductive, since its strict requirements would likely fail many invocations, due to network latencies. For this reason, we introduce a second type of semantics, called *weak*, that modifies the interpretation of the "one" and "all" scheduling modalities. Under the weak semantics, we soften the exactly "one" requirement to an "at least one", where the Base Station (and intervening drones) sends a set number of concurrent requests (defined by the user, in the policy) to possible targets and responds with a positive answer as soon as it receives the first response from one of the targets. Similarly, we weaken the "all" universality requirement to an "almost all" interpretation that specifies a timeout for the gathering of the responses. If all responses reach the Base Station within the set timeout (by the user, in the policy), it behaves as in the "all" modality; otherwise, the Base Station sends back to the user as many responses as it received at the triggering of the timeout (and discards all possible successive ones).

The above logic defines how our proposal handles fault tolerance in the highly dynamic context of drone networks. Essentially, fault tolerance is implicitly and effectively managed at the software-component level through the continuous operation of the gossiping knowledge dissemination protocol. Specifically, nodes dynamically react based on the latest disseminated information. If a node fails to send periodic updates or responses (due to component failures or intermittent connectivity), other nodes inherently detect this occurrence by the absence of fresh data and consequently ignore or remove this node from their local views. Additionally, nodes actively monitor their software components and, upon detecting local component failures, immediately stop advertising their resource availability through the gossip-based heartbeat messages. This rapid dissemination of status changes ensures that the entire network quickly adapts to node-level failures and maintains robust scheduling decisions. While all elements maintain a view of the status of the network, the system does not need these views to be consistent since each

drone determines whether it can execute a function for which it receives an allocation request. Regarding the handling of failures in function scheduling, the Base Station is the only component that maintains the status of the ongoing executions and registers an execution failure when a given function allocation fails to respond within a set timeout. The system does not implement any recovery/reallocation policies to handle faults and rather leaves to the user the decision on what to do in response to a failed function execution attempt.

Strong ang Weak Semantics, Exemplified. To better illustrate how each execution target presented in this section works in practice, let us consider a situation where we have several drones around an arbitrary point in space p, and we want to schedule a function f on them.

In the strong semantics case, using the "all" target, we would send the function to each suitable drone in the area (whether the delivery of the execution message happens in a single hop or through multiple hops depends on the topology of the network), and wait for all of their responses. If just one of these executions happens to fail, or the response does not reach the Base Station within a given timeout, the entire invocation is considered unsuccessful. The user receives no result, except for a notification of the failed execution attempt.

Still in the strong semantics, using the "one" target, we would send the function to the "most suitable" (as determined by the selection policy, discussed in Section 4) drone available, e.g., the one with the most available resources, and wait for its response. Similarly to "all", if the drone fails to respond, the invocation is considered unsuccessful, except the failure is only imputable to the single targeted drone.

Under the weak semantics, the "all" target defaults to its "almost all" counterpart. We would still send the functions to all the suitable drones in the area. However, failure handling changes significantly: after a set timeout, the Base Station sends back to the user all the responses it received for this execution request, even if some target drones have not responded yet. In other words, unlike with the 'all' target, we tolerate partial results and consider it a successful invocation.

The weak counterpart of the "one" target is "at least one", which would send invocation requests to a set number of suitable drones in the area. In this case, we consider the invocation successful unless all targeted drones fail to respond within a given timeout. The first response the Base Station receives

is the one sent back to the user and all others are discarded. Compared to the "one" target, we lose the guarantee that the function is only run once; however, the use of redundant requests increases the invocation's tolerance to failure.

3.4. Mixing Scheduling Modalities and Network Semantics

The attentive reader might have already spotter the fact that the hierarchical and progressive scheduling modalities and the weak and strong semantics are orthogonal ingredients that one can mix to obtain different scheduling logics.

While this observation is sound, we notice that, e.g., pairing the hierarchical scheduling with the weak semantics, could produce an inefficient configuration, i.e., one where the Base Station does not have enough information to choose a target node for executing the function or the network does not have enough capacity to support the effective completion of the flow of function execution.

Hence, the interplay between the scheduling modalities and the network semantics generates a design/combination space that deserves dedicated work to explore.

Since this proposal regards the general presentation of our approach, in the reminder, we fix a combination of these dimensions and delegate their proper exploration to future work (e.g., with specific case studies and experiments).

Concretely, we fix the combination of the strong semantics with the hierarchical scheduling modality and the weak semantics with the progressive modality, which, we conjecture, are the most reasonable pairings of this space w.r.t. the QoS of ad-hoc networks. Our rationale for this choice is that the strong semantics requires broad network bandwidth and timely updates, which is also an important requirement of the hierarchical modality. Similarly, pairing the weak semantics with the progressive scheduling modality mediates the limitations of narrow-bandwidth, lagging networks.

Given that the decision on whether to follow the strong or weak semantics is a system-level choice — specifically, taken by the Base Station depending on the information it receives from the drones — when the users define the policy for their functions, we ask them to indicate which strong semantics scheduling configuration they want for their function ("one" or "all"), followed by the parameter of the corresponding weak interpretation (the number of repetitions for the "at least one" and the timeout for the "almost all"). In

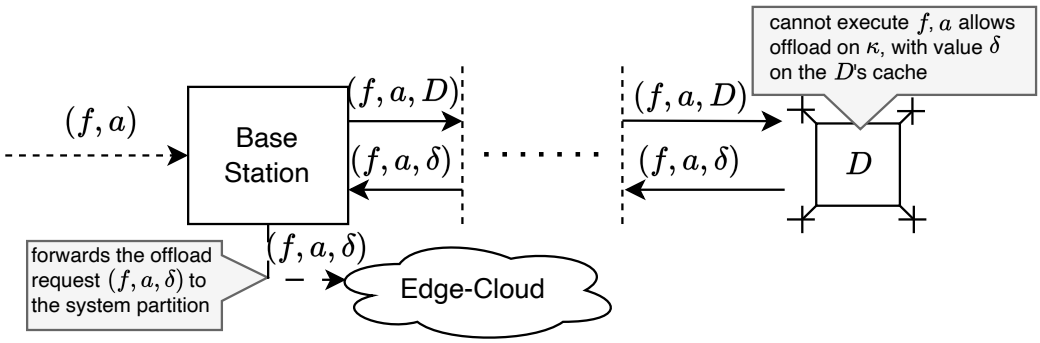

Figure 5: Example of function execution offloading, scheduling layer.

this way, depending on the current configuration of the network, the Base Station follows the corresponding interpretation and can enact the related scheduling modality.

Function Execution Offloading. Another important trait of our proposal is that users can indicate whether a drone can offload the execution of a function to some computation node outside the drone network. When a user specifies that the target drone can offload function execution, they can define a piece of data that might be on the drone and that the function requires for its execution — this behaviour is typical of workflows where chains of functions manipulate the data found in some repository, which, in this context, is the local cache of drones. Thus, if the drone decides to offload the execution of a function, e.g., due to limited execution capacity, it retrieves the data indicated by the user (e.g., querying its local key-value data storage) and sends an "offload" message to the Base Station that carries both the code of the function and the data it needs to run on.

As an example, depicted in Figure 5, consider function f, which needs to run within a certain location to gather some sensory data, like humidity and light irradiation. In the figure, using hierarchical scheduling, the Base Station determines that D is the target drone, issuing the placement of that function with the message (f, a, D). However, when the message reaches D, the drone realises it cannot run the function, e.g., because its current workload level does not afford it. Since the scheduling policy a specifies that D can offload the execution of this function, D sends back to the Base Station

a message (f, a, δ)[3], where δ carries the possible data found on D useful to run f, as indicated by the offload parameter in a — of course, one can also introduce refinements of the offloading logic, e.g., having the Base Station preserve a copy of f and a so that the drone can just send back a message (f_{id}, δ) carrying the function identifier and the data.

Notice that, in Figure 5, we do not indicate which device runs f outside the ad-hoc network (and belonging to the Edge-Cloud domain). Since the execution of the offloaded functions is an external ingredient to our proposal — i.e., the ad-hoc drone serverless network works irrespective of how the Base Station decides to handle the execution of these functions — we abstract away the specification of the execution of offloaded functions, and we focus only on the specification of the offload functionality.

3.5. Software Components: the Base Station and the Drones

We close this section by presenting the software components that support function scheduling, routing, execution, offloading, and response in the elements that make up the ad-hoc network: the Base Station and the drones.

Starting from the Base Station, depicted in Figure 6, we find three main software components: the function scheduler, the state sentinel, and the function execution tracker. The first two modules deal respectively with the function scheduling and the gossiping layers.

The function scheduler is the component that handles the request for function execution from users (represented in the figure with the inbound message (f, a) on the left of the component), their allocation on the proper drones (exemplified with the outbound message (f, a, D) on the right of the component), and the relay of responses (e.g., (f_{id}, r)) and offloading (e.g., (f, a, δ)).

The state sentinel module deals with the status information of the drones disseminated through gossiping. In the Base Station, this component acts as a "sink" of information, used by the function scheduler to gather information on the status of the drones present in the system.

The last component, the function execution tracker, records the status of the execution of functions, including their cardinalities (e.g., a function can

[3]From brevity, we assume that drones identifiers $D_1, ..., D_n$ and data, as contained in δ, belong to distinct domains, so that the drones and Base Station can distinguish the nature of the message they receive. In implementations, there is no need for such assumption because messages usually carry a label to categorise them.

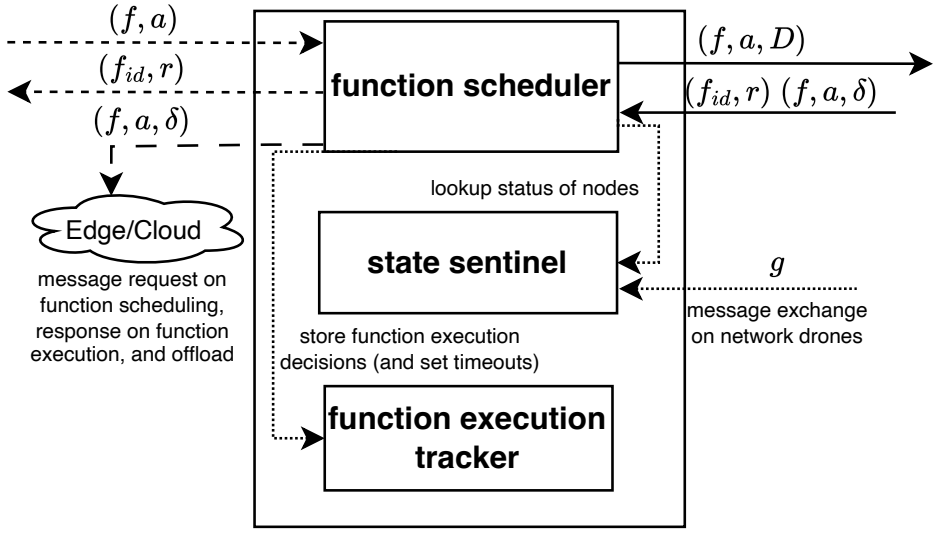

Figure 6: Software components found within the Base Station in our approach.

have a cardinality equal to the number of nodes if the scheduler issued its execution on "all" nodes) and their timeouts.

While slightly more complex, note how the architecture of the drones, depicted in Figure 7, recalls that of the Base Station. Indeed, while the drones are execution nodes for the functions, they also work as routers/schedulers of the requests, which requires the inclusion of a dedicated function scheduler and a state sentinel.

The main difference between the function scheduler found in the Base Station and the one found in the drones is in their behaviour w.r.t. the two scheduling modalities, hierarchical and progressive. When dealing with a hierarchical request, the drone's function scheduler merely works as an executor, if the receiver of the message is the target drone, or forwarder, in charge of choosing the next hop of the message. Under the progressive scheduling, the function scheduler assumes a much more "active" role, by evaluating both whether its host can run the function and/or if it needs to select which other drones to forward the execution message to.

To inform these decisions, the function scheduler uses the information about the other nodes in the network gathered by the state sentinel (similarly to its Base Station counterpart) along with the status of the drone's hardware and the current function execution workload (provided by the func-

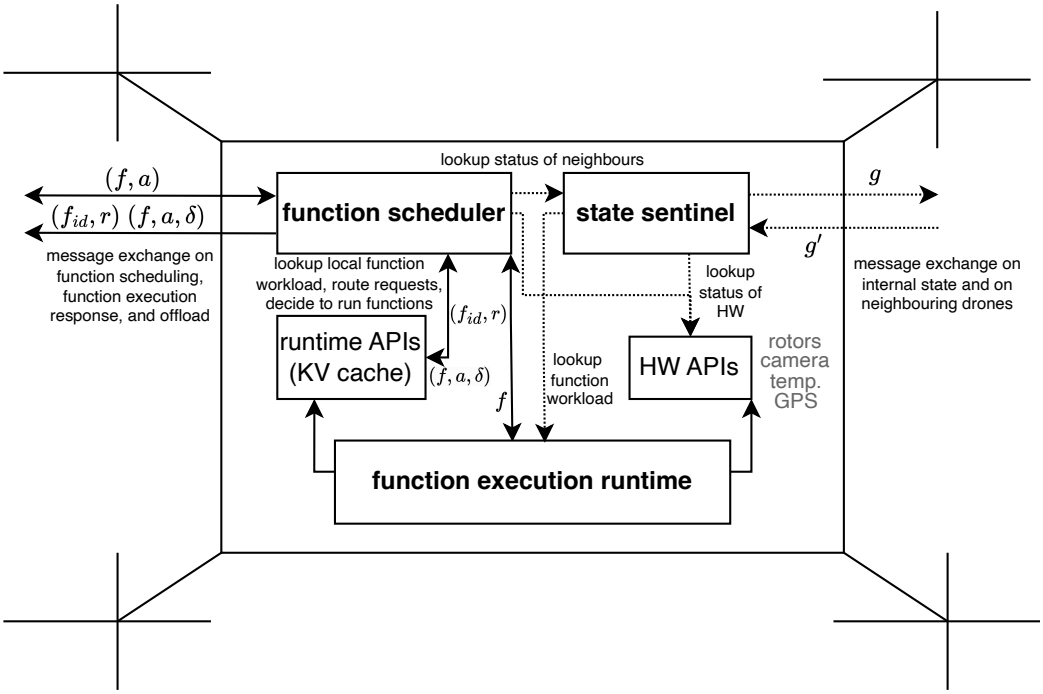

Figure 7: Software components found within a drone in our approach.

tion execution runtime).

As mentioned, the state sentinel exchanges messages with the nodes in direct connection with the host drone to advertise its status information (gathered from the hardware and function execution runtime modules) and assemble an overall picture of the configuration of the network.

The function execution runtime is the component that, embedding a WebAssembly engine, can run the functions. Note that, through the execution runtime, the functions can both access the hardware of the drone (e.g., camera, rotors, etc.) and its local runtime APIs, among which its key-value storage, useful when running function workflows where some function "leaves" some data on the drone and a successive one retrieves it for computation. These APIs are particularly useful when implementing the offloading feature — where the policy that accompanies the function can specify some value that the function uses and the offloading command pairs with the request of running the function on some external component.

Regarding failures, which mainly concentrate on the drones components,

we note that, since the network topology is highly dynamic, the software components react w.r.t. the information that reaches each node, so, if a node fails to send updates or responses, the other nodes just ignore that node. In particular, the Base Station, following its failure-handling logic (cf. Section 3.3) handles the absence of responses from targeted drones through timeouts. If a drone node realises that some of its components fail (e.g., a malfunctioning camera), it stops advertising the availability of those components in the gossiping phase, which updates all nodes about its changed state.

4. Ad-Hoc Allocation Priority Policies

We now present the language, called Ad-Hoc Allocation Priority Policies (AHAPP), that we provide to users for expressing the scheduling policies of their functions. As seen in Section 3, these policies inhabit the term a in message exchanges, like (f, a) and (f, a, D), for the scheduling of functions, and they are interpreted by the function scheduling modules of the Base Station and the drones to implement the scheduling logic of each function.

We start by introducing the constructs and parameters of the language and then apply them for capturing the functionalities of our case study (cf. Section 1).

4.1. The AHAPP language

The AHAPP language takes inspiration from the APP language and its dialects [17, 23, 24, 25, 26, 27, 28, 16]. As such, AHAPP is a subset of YAML [29], a widespread configuration language used in popular software for Cloud Computing like Docker and Kubernetes.

Unlike APP and its variants, AHAPP introduces a fundamental architectural shift. In previous approaches, a single script governed the scheduling of all functions, using tags to specify different policies for functions (i.e., one tag can specify the scheduling of many functions, each carrying that tag) with their scheduling logic. An important novelty introduced by AHAPP is that function scheduling requests see the function under scheduling paired with its policy — essentially the policy "travels" with the function within the system. This design innovation stems directly from the peculiar characteristics of our drone network, where individual drones are capable of taking scheduling decisions. Consequently, the messages transmitted must not only carry the function but also embed the logic determining the scheduling of that function. While, in AHAPP, we do not use tags to associate functions

```
targets:
 all: { almost_all: n } | one: { at_least_one: n }
offload: id | b
affinity: { id_1, ..., id_j }
attributes:
 position: { lat: d, lon: d, alt: d, range: n }
 loadavg: { max: n }
 energy: { min: n }
 memory: { min: n }
 storage: { min: n }
 device: { h_1,...,h_l }
  where h_{i,...,l} ∈ {
    rotors: { lock: b },
    camera: { lock: b, quality: id },
    gpu: { lock: b }
  }
strategy: [ r_1, ..., r_k ] where r_{1,...,k} ∈ {position, energy, ...}
```

Figure 8: The AHAPP language grammar.

to policies, we retain the functionality of tagging functions, useful to express affinity constraints (detailed below).

In Figure 8, we show the structure of AHAPP policies and the values that each option can assume, using | to indicate alternatives, *id* for identifiers (tags), *b* for Booleans, *n* for natural numbers, and *d* for decimals. We comment on the constructs from top to bottom. In Figure 8, we slightly stylise the code for readability, in particular, we represent YAML inline sets as {...} rather than using the `!!set`-prefix notation.

Targets. The `targets` key encodes the execution modality of the function, i.e., either on "all"/"almost all" drones or on "one"/"at least one" of them, as introduced in Section 3.3. Recalling that section, we pair the configurations of the strong and weak semantics so that selecting the `all` option requires the user to also indicate the timeout of the `almost_all` clause; similarly, selecting the `one` option requires the specification of the number of requests the system is allowed to replicate under the `at_least_one` modality.

In case a policy selects the `all`-`almost_all` option, the Base Station interprets the instruction by sending to all drones the request to run its companion function. The difference between the two modalities is that, under the strong semantics, the Base Station considers as failed the scheduling of a function when at least one of the requests to the drones did not return its response within a set timeout (defined by the platform). Under the weak semantics, the Base Station returns all responses received within the timeout indicated by the user as parameter of the `almost_all` option — i.e., receiving at least one response within the timeout configures as a successful execution of the function. On the drones' scheduler, this option entails the execution of the function (if the drone can run it) and/or the forwarding of the request.

When the policy selects the `one`-`at_least_one` option, the Base Station interprets `one` by selecting a specific drone that shall run the function, while, under `at_least_one`, the Base Station sends multiple requests and returns to the user the first response it receives. To send these requests, the Base Station splits the number of available requests n among the nodes that it can contact directly (in particular, those in the direction where some nodes can run the function). Practically, the Base Station achieves this "splitting" by sending out requests that carry the same policy it received but where the `at_least_one` parameter indicates a partition of the function instances granted by the initial request. As an example, consider a Base Station that receives a policy with `at_least_one` equal to 5, and it decides to send out two requests. These requests would carry the same policy the Base Station received but would differ in the value of the `at_least_one` option, which could be resp. e.g., 2 and 3 (or any non-zero 2-partition that sums is 5). In turn, the drones that forward this request reduce the value of the `at_least_one` by one if they run the function and split that value according to the number of messages it forwards for function execution.

When `targets` is omitted, we consider `one` and `at_least_one`: 2 as defaults.

Offload. The `offload` key defines whether a drone supposed to run a function (e.g., when directly selected by the Base Station) and unable to do it, e.g., due to a lack of energy or memory capacity (cf. Section 3.3) can redirect the execution of that function to the "edge-cloud" part of the platform. Notice that users can specify two values for this option, either an identifier or a Boolean value. The boolean values simply indicate whether the `offload` option is enabled or not. When providing an identifier, the user indicates

that the `offload` option is enabled and the offload request shall carry the piece of data (if any) pointed by that identifier as a key in the drone's local key-value cache. When omitted, the `offload` option is disabled.

Affinity. The `affinity` clause specifies a list of tags of functions that are required to be in execution or have run (within a fixed timeout) on the drone to run the function under scheduling. This option is particularly useful when implementing workflows of functions, e.g., where function f runs on a certain drone and then g has to run on the same drone because f left some data that g needs for its execution. The default value of this option is the empty list. Note that our interpretation of `affinity` does not imply synchronous execution of functions, e.g., a function can express affinity to indicate that it needs to run on a drone where some function ran and left some data for the former to work on. To implement this mechanism, we assume that the platform sets an "execution window" where the nodes keep track of the function tags they have executed. In this window, nodes maintain a list of tags of completed functions, and after a timeout period, they remove these "stale" tags. This timeout mechanism works in tandem with the drone's cache memory eviction policy, ensuring proper garbage collection of data left behind on the drone.

Attributes. The `attributes` key encodes properties and physical resources required by the functions to run on a target drone.

Properties include `position` and `loadavg`. The `position`, with the indication of the `lat`itude, `lon`gitude, `alt`itude, expressed as decimals, and a `range` that specifies a sphere of "tolerance" from the coordinates with radius n metres. The `loadavg` specifies the `max`imal percentage of capacity occupation (e.g., CPU) within a given period (set by the platform, e.g., 1 minute). When omitted, the default value of the `position` sets all coordinates to 0 and the radius to its maximum definable value; the default value of `loadavg` is 50.

We divide the overview of the resources into two groups. The first group, that of `energy`, `memory`, and `storage` specifies the `min`imal amount of resources necessary for effectively hosting the computation of the function. Specifically, the percentage of `energy` (default to 50), the amount of `memory` and `storage` capacity, in MB (resp. default to 8 and 32). The second group, that of `device`s, indicate a set of hardware resources found on the drone that the function requires accessing for its execution. The default value for this

option is the empty set. The possible elements in the set are (self-descriptive) `rotors`, `camera`, and `gpu`. When indicating these requirements, the user can express whether the function has to `lock` — Boolean and default to false — that hardware exclusively during its execution, e.g., to move the drone or to run compute-intensive parallel algorithms. During scheduling, if some other function is "locking" the specified hardware, the drone cannot run that function. The `camera` option can also specify a `quality` literal, like `FHD` and `4K`; the default value is `HD`.

Strategies. The last option, that of the `strategy`, specifies an ordering among the level of the `attributes` of the drones — except for the `device`s, which are binary — that the scheduler can use to choose a target when it identifies multiple valid ones. In practice, the `strategy` option allows the users to refine the deployment of their functions by specifying the prioritisation logic the scheduler should follow when choosing among equally valid targets. In this way, users can orient the deployment of their functions according to custom rules, e.g., by prioritising low-load or high-energy devices or minimising latency through geographically-aware scheduling.

The values for this option are `position` (closer), `loadavg` (lower) and available (higher) `energy`, `memory`, and `storage`. When defining the `strategy`, the user provides an exclusion order for ranking the possible candidates. The option also has as its trailing, default value the parameter `random`. Hence, if the user omits the definition of the `strategy` option, the scheduler `random`ly chooses among the possible candidates, otherwise it first filters out the candidates considering the options in the list in order of appearance and then chooses `random`ly among the remaining candidates, if any.

4.2. Modelling the Case Study with AHAPP

To illustrate the usage of AHAPP, we model the case study presented in Section 1 as a serverless architecture of four functions and their related AHAPP policies. For clarity, in the description below, we call the functions by their associated tag. In the policies, we omit to specify the `at_least_one` and `almost_all` options, which would require us to further characterise the usage and topological context of the ad-hoc network. Thus, for brevity, we omit their definition, which makes the policies use their respective default values.

We call the first function `ScanFireAlert`. This function runs on all drones and processes real-time sensor data for fire aerial surveillance. To this aim,

Listing 1: `ScanFireAlert` policy.

```
targets: all
attributes:
 device: { camera, gpu }
```

Listing 2: `AnalysePeopleScan` policy.

```
targets: all
affinity: { ScanPeople }
offload: ScanPeopleData
```

Listing 3: `ScanPeople` policy.

```
targets: all
attributes:
 position: { range: 100,
  lat: X, lon: Y, alt: Z }
 device: {
  camera
  gpu
  rotors: { lock: true }
 }
```

the function analyses heat signatures and geospatial information to detect potential fire outbreaks. When a fire is identified, the function runs an AI model for fire recognition and categorisation, working on heat intensity and smoke patterns. The response it produces is a fire alert, containing the geolocation data, fire characteristics, and severity assessment. We show in Listing 1 the code of the AHAPP policy for this function. In the policy, we target `all` drones and impose the execution of the function on only those nodes that equip a `camera` to acquire visual data on the territory hovered by the drone and a `gpu` for the efficient execution of the fire-detection AI model.

After the response from the `ScanFireAlert` function(s), we invoke the execution of the second function, called `ScanPeople`. The function runs on drones in areas where `ScanFireAlert` detected fires. `ScanPeople` processes data from the drone's camera to detect human heat signatures and movement patterns. This analysis requires the repeated acquisition of visual data from the same area, which implies that the function must have (exclusive) access to the drone's rotors to move it according to its detection routine. Like `ScanFireAlert`, also this function uses an AI model for categorisation. This function does not provide a contextual response to the user — it just confirms its successful execution — but it rather streams a continuous feed of data which other functions can use to refine the detection of potential survivors. We provide, in Listing 3, the policy for this function. In the policy, we specify that the function shall run on `all` drones within a `range` of `100`m from the `X`, `Y`, `Z` `position` (we use these symbols as placeholders for the actual coordinates) where `ScanFireAlert` found fire. Besides requiring drones with a `camera` and `gpu`, the function needs to acquire a `lock` on the drone's `rotors`

to implement its detection routine.

Launched in close succession after `ScanPeople`, `AnalysePeopleScan` performs advanced analysis on the data produced and stored on the drones by `ScanPeople`, to confirm the presence of survivors. The function employs machine learning algorithms to distinguish human signatures from other heat sources and movement patterns. The function evaluates the confidence level of each detection and generates, as a response, detailed information about potential survivor locations, including their estimated condition. The `AnalysePeopleScan` allows us to illustrate two other features of the AHAPP language: `affinity` and `offload`ing. Indeed, rather than giving a location of the drones that should run the `AnalysePeopleScan` function, we indicate that they shall run on (`all`) drones where the `ScanPeople` function is running (or has run). The reason for this indication is that `AnalysePeopleScan` runs analyses on the data produced by `ScanPeople`, stored in the local cache of the drone. Since this piece of data is fundamental for its execution, but the function can otherwise run on the Edge-Cloud, we set the policy to allow the `offload` of the function using the data stored, by `ScanPeople`, under the key `ScanPeopleData` in the local cache of the drone. This configuration allows us to handle the case where the target drone appears available to the Base Station, but it is unable to host the function because of lack of resources, which might happen if, e.g., the Base Station is relying on stale status data and the drone is hosting more functions than what the Base Station knows about. With the `offload` option, the drone can react to this situation by sending the request back to the Base Station — along with the data the function is supposted to work on, found in the local cache of the drone, indicated as the value of the `offload` clause. Once the Base Station receives an offloading message, it forwards a request, carrying the data the function has to work on, to a companion edge-cloud platform that acts as a "computational fallback". When offloading the execution to the companion cloud-edge platform, the Base Station takes care of interacting with the latter to also receive the response after the execution of the function and notify the user. Since the function runs in a domain not directly controller by ours, it is the companion one that determines the execution of the function, including the handling of possible errors that arise during its execution.

The last function, called `TrackSurvivor`, maintains continuous monitoring of confirmed survivors. It processes real-time GPS coordinates and video feeds to track survivor movements and status. The function generates a stream of data that include precise location updates, visual confirmation

Listing 4: `TrackSurvivor` policy.

```
targets: one
attributes:
 position: { range: 25,
  lat: X, lon: Y, alt: Z }
 device: {
  camera: { lock: true }
  gpu: { lock: true }
  rotors: { lock: true }
 }
strategy: [ energy, position ]
```

data, and dynamic access route calculations for rescue teams [30]. Since, now, we have precise information about the survivors, we issue the individual (one) execution of a `TrackSurvivor` function instance for each group of survivors in a given location. The tracking routine requires the exclusive access to the drone's rotors and camera to follow the survivors and of the drone's gpu for the efficient processing of data for streaming. Since more than one drone can be valid for the execution of the function, we specify the strategy for choosing among multiple alternatives, i.e., first, we favour the drones with the highest energy level (since the function needs to continuously track the survivors for as long as possible), second, we favour the drones closer to the required position (within the 25m range, imposed by the policy) — as per definition (cf. Section 4.1) if these two selection steps fail to weed down the options to one drone, we default to randomly select one of the alternatives.

On the Complexity of AHAPP Scheduling. Computationally, the complexity of scheduling functions interpreting APP scripts is generally linear in the size of the nodes that execute the functions [17]. AHAPP is not an exception.

We break down the above statement considering the weak and strong semantics, which we pair respectively with the hierarchical and progressive scheduling — hence, when referring to the scheduling approach we also implicitly refer to the related semantics.

Under the hierarchical scheduling, the Base Station has precise knowledge of all the drones in the network. Irrespective of the targets value, the Base

Station considers the application of the policy against all drones. Indeed, under the `all` option, the Base Station needs to evaluate the policy against the complete set of available nodes to select the ones that can execute the function. Similarly, under the `one` option, the Base Station has to evaluate the policy against all nodes to find the highest-scoring one which can execute the function (or randomly choose among the tied highest-scoring ones, if any). In the case of multi-hop function request messages, the intervening drones act as forwarders of the message. This task is computable in constant time, assuming the drones maintain routing tables computed when they receive information from the gossiping protocol.

When running under the progressive scheduling, the Base Station does not have a complete view of the elements in the network, but rather a local view limited to the one-hop drones it can directly reach. Similarly to the hierarchical scheduling (and, as argued above, irrespective of the `target` option), the Base Station has to compare the policy against all drones, although limited to the one-hop ones in its neighbourhood, making the complexity of the task linear in the number of drones in the network, in the worst-case scenario — a degenerate case where all drones are in the neighbourhood of the Base Station, which can directly reach them. In the general case, the Base Station considers the applicability of the policy against a fraction of the drones in the network, to decide to which one-hop drones to forward the request to. Once a drone in the network receives a progressive function execution request, it essentially follows the same (linear) algorithm of the Base Station, considering whether it can execute the function itself and whether it shall forward the execution to other one-hop nodes in its neighbourhood.

5. Scheduling and Gossiping Layer Network Protocol

After having described the specification of the programming and behaviour of the function scheduling layer, we present the details of the underlying network protocol, useful to maintain an accurate and up-to-date view of both the network topology and individual drone status across the distributed system and support function scheduling. The protocol addresses key challenges in mobile ad-hoc networks, particularly drone swarms, where dynamic topologies, intermittent connectivity, and varied hardware capabilities require robust distributed communication mechanisms. Since this protocol both provides information to the upper layer of function scheduling and supports the efficient relay of function allocation messages, we discuss the main

technical details that characterise how the scheduler chooses its allocation targets (the nodes where functions shall run) and how the network delivers the related messages to destination. In addition to supporting efficient function scheduling, the proposed gossip-based knowledge dissemination inherently provides robust fault tolerance. Due to the highly dynamic nature of drone swarm topologies, the protocol is designed to react immediately to changes in network conditions. Nodes detect failures implicitly through the absence of periodic updates, promptly removing unreachable nodes from their local views. Additionally, if a failure occurs, each neighbour node instantly ceases advertising its availability through the gossiping mechanism, ensuring rapid propagation of updated state information. Consequently, the function allocation module can dynamically adapt to network disruptions, maintaining robust and resilient scheduling decisions.

Elements of the Network Protocol. We model the system as a set of n_U mobile or static nodes forming a multihop mobile ad-hoc network. To enable serverless function scheduling in such environments, we propose a *function-allocation-aware protocol* that abstracts the complexity of the underlying network from the scheduling layer. This protocol ensures that function allocation requests are handled efficiently by disseminating network knowledge and facilitating function scheduling decisions. The protocol consists of two main modules: *Knowledge Dissemination*, which spreads network information, including topology, hardware capabilities, and node positions (cf. Section 5.1); and *Function Allocation Module*, which determines how to allocate function execution requests based on scheduling (AHAPP) instructions and the available knowledge (cf. Section 5.2). These two modules allow a clear separation between knowledge dissemination and function allocation. Knowledge dissemination operates as a background process, proactively updating all nodes with real-time system status information like network topology and node hardware availability. In contrast, function allocation is triggered on demand when a function request arises. This dual-phase approach ensures that the system remains continuously updated and capable of immediate function allocation, boosting both responsiveness and resource efficiency.

5.1. Knowledge Dissemination

Efficient function scheduling in mobile ad-hoc networks depends on the ability of nodes to exchange real-time information regarding network topology, resource availability, and mobility patterns. To address this challenge,

we propose a gossip-based knowledge dissemination protocol that propagates relevant information through periodic *heartbeat* messages. The primary objective of this protocol is to maintain an up-to-date network view while minimizing communication overhead, which is essential in dynamic environments where excessive message exchange can degrade performance. The protocol supports three levels of knowledge dissemination, each representing a trade-off between information completeness and communication efficiency: full dissemination, full dissemination with change detection, and aggregated dissemination.

Each node periodically transmits a `Heartbeat` message to its one-hop neighbours at fixed intervals of t_H seconds. This message consists of two components: local information and global information. The local information includes all parameters necessary for function scheduling, as detailed in Section 4. Specifically, let s_H byte be the size of the local information packet where each node reports data like its position, CPU load, residual energy, available memory and storage, camera quality, and device locks.

The global information component varies depending on the selected knowledge dissemination method, which determines how network-wide data is shared. This aspect, along with specific periodicities of the `Heartbeat` message, differentiates the three dissemination strategies discussed below.

Full Dissemination. In the full dissemination model, each node periodically transmits its own status along with the status of all its known neighbours. Over time, this approach ensures that every node acquires a complete global view of the network. The primary advantage of this method is its ability to support global decision-making for hierarchical scheduling, allowing nodes to make globally-informed placement and routing decisions. Indeed, similarly to the BATMAN protocol [31], this technique enables each node to discover and store the best next-hop toward a given destination based on received broadcast messages. The protocol leverages hop-count metrics to determine the optimal next-hop, ensuring efficient packet forwarding. Since only local next-hop information is maintained rather than full routing tables, the approach remains highly adaptive to mobility and topology changes. If a next-hop node becomes unreachable, the system dynamically selects an alternative best next-hop based on newly received `Heartbeat` message, allowing for rapid adaptation to network disruptions. In a static network, repeated iterations eventually disseminate each node's information to all others, ensuring network-wide synchronization. However, in a dynamic environment,

frequent topology changes can result in the propagation of stale information, rendering it obsolete before reaching all nodes. Consequently, nodes may base scheduling decisions on outdated network views, leading to suboptimal or incorrect function placements, which ultimately degrades system performance.

Furthermore, as each `Heartbeat` message eventually reaches a size of $n_U \cdot s_H$ bytes, this approach introduces significant scalability challenges. The global information contained in the message includes data from $n_U - 1$ nodes, increasing the transmission burden. As a result, the overhead scales proportionally with the number of nodes, leading to excessive communication costs. In large and highly dynamic networks, continuously propagating exhaustive state information can cause network congestion, increase latency, and degrade overall system efficiency.

Full Dissemination with Change Detection. To reduce unnecessary transmissions while preserving a global network view, we introduce an optimised variant of full dissemination in which nodes transmit updated information only when a state change occurs. Even though this technique uses the same periodic `Heartbeat` message in each interval of t_H seconds, the message will only be sent if there are changes to be reported. Moreover, only the changed attribute is sent, reducing unnecessary transmissions while preserving essential network-wide synchronization.

This approach retains the benefits of complete dissemination while minimizing redundant transmissions, thereby reducing communication overhead. The effectiveness of this method depends on the rate of topology and system changes: in highly dynamic scenarios, frequent updates may still result in significant overhead, whereas in more stable conditions, the protocol achieves a near-optimal balance between completeness and efficiency. To refine update propagation, we introduce threshold-based triggers for non-binary node attributes such as position, CPU load, residual energy, available memory, and storage. Specifically, an offset threshold is defined for each parameter, specifying the minimum value change required to report it. These thresholds are denoted as follows: th_p for the position, th_c for the CPU load, th_e for the residual energy, th_m for available memory and th_s for the available storage.

Another improvement is an alternative `Heartbeat` comprising a set of change blocks. Each block contains the essential node identification, a sequence number, an attribute identifier, and its new value. Despite these optimizations, similar to the full dissemination model, the `Heartbeat` message

will eventually reach a size of $n_U \cdot s_H$ bytes, introducing scalability concerns in large networks.

Aggregated Knowledge Dissemination. In contrast to the previous methods, the aggregated knowledge model restricts information exchange to local node states and summarised network metrics. Each node transmits every t_H seconds its own status, reduced to only position information, along with an aggregated representation of its neighbourhood, such as minimum, maximum, or average resource availability. This approach significantly reduces communication overhead by preventing the propagation of detailed node-level data across the network. Specifically, the network overhead is drastically minimised, as each `Heartbeat` message does not exceed s_H bytes. However, this method introduces a trade-off between efficiency and decision-making accuracy, as nodes make scheduling choices based on partial and abstracted network knowledge rather than complete global awareness. Consequently, the aggregated knowledge model is most suitable for progressive allocation strategies, where function scheduling relies on localised rather than global decision-making.

More in detail, the global information component of the `Heartbeat` message starts from the own status information; the approach chooses the best metrics of each neighbour to compose the `Heartbeat` message, intending to represent its potential. Less CPU load from the neighbour nodes is used instead of its own, as well as greater residual energy, available memory, and storage to compose the `Heartbeat`. Regarding non-binary node attributes, like camera and GPU availability, the aggregated information considers the node's and its neighbours' availability.

Note that position information is not included in the global information. This omission is intentional, as only one-hop location data is used to forward function allocation requests based on positional constraints (more detail in Section 5.2). This approach is analogous to geographic routing, as employed in GPSR (Greedy Perimeter Stateless Routing) [32], where GPS coordinates are leveraged to forward packets toward a specific destination.

Table 1 outlines the advantages and limitations of the different dissemination strategies, highlighting their impact on communication overhead and their suitability for various network dynamics, i.e., the expected rate of topological changes over time. The effectiveness of each method is context-dependent, influenced by network size, stability, and the computational requirements of the deployed functions. In our evaluation (cf. Section 6), we

Dissemination Method	Knowledge Scope	Communication Overhead	Application Context
Full	Global (all nodes)	High	Stable topologies
Full w/ Change	Global (updated states)	Medium	Stable and Moderate dynamics
Aggregated	Local + aggregated metrics	Low	Dynamic topologies

Table 1: Comparison of Knowledge Dissemination Strategies.

analyse the performance trade-offs among these approaches.

5.2. Function Allocation Module

Assuming the dissemination of network knowledge, when the system receives the request to execute a function, it must determine the most suitable node where to allocate the function, taking into account network constraints, resource availability, and scheduling efficiency. The function allocation module governs this process, ensuring that function allocation works based on scheduling (AHAPP) instructors and the available knowledge at each node.

As described in Section 3, the system supports two scheduling strategies: *hierarchical scheduling*, which relies on a global network view, and *progressive scheduling*, which operates under localised knowledge. We remind that, in this work, we make these two approaches respectively correspond to strong and weak execution semantics (cf. Section 3.4). The hierarchical scheduling assumes complete network knowledge, enabling it to achieve efficiency under the strong semantics, where function allocation is directly decided by the Base Station and the intervening nodes work are routers. Conversely, progressive scheduling is paired with the weak semantics, where a best-effort approach governs scheduling decisions based on neighbourhood-level information — due to incomplete network knowledge. In the following, we assume the hierarchical and progressive scheduling modalities paired with their respective strong and weak semantics.

Hierarchical Scheduling. The hierarchical scheduling strategy assumes that nodes have access to global knowledge obtained through either the *Full* or *Full with Change* dissemination methods. When a function allocation request for function f with scheduling policy a is initiated, the scheduling node (i.e., the Base Station) has a complete view of the network and can determine the optimal execution target. The request is then transmitted directly to the selected node using multi-hop forwarding. This approach ensures precise

function placement and avoids allocation overhead — w.r.t. the time/redundancy thresholds of the weak semantics. However, its efficiency depends on the accuracy and timeliness of the global knowledge, making it more suitable for stable topologies with low-latency communication (and broadband connectivity).

The scheduling procedure depends on the `target` value, which, in the hierarchical methodology, can be set to either `all` or `one`.

- if `all` is selected, the system identifies all nodes that satisfy the scheduling constraints and allocates the function execution on each of them;
- if `one` is selected, a selection process determines a single "suitable" node for execution.

To facilitate the selection process, driven by the `strategy` clause, we introduce ranking classes for requested resources, categorising them into three discrete levels: high, medium, and low. This categorisation simplifies the selection process, as a ranking based on continuous numerical values may be impractical when distinguishing between groups of nodes with comparable capabilities. To achieve this categorisation, we define two threshold values for each parameter: l1, which differentiates between low and medium, and l2, which separates medium from high. The thresholds are defined as follows: $l1_c$ and $l2_c$ for CPU load, $l1_e$ and $l2_e$ for residual energy, $l1_m$ and $l2_m$ for available memory, and $l1_s$ and $l2_s$ for storage.

Unlike these real-valued parameters, the binary constraints, such as `locks` and `position`, do not require threshold-based categorisation. Locks are either present or absent, directly determining whether a node is eligible for execution. Similarly, position-based constraints are treated as boolean conditions, where a node is considered inside or outside the requested deployment area.

Progressive Scheduling. The progressive scheduling modality is designed for dynamic networks, where the nodes have only localised and aggregated knowledge. Instead of making a global decision, the scheduling process follows a best-effort heuristic, progressively forwarding the request toward regions with successful execution capability. Each node receiving the request, evaluates its own resources and those of its direct neighbours, determining whether to execute the function locally or forward the request further — except the Base Station, which can choose only the second option. This method introduces

greater adaptability and reduces overhead, as nodes do not need to maintain an exhaustive global state. However, suboptimal placements may occur due to the incomplete network view available to each node. Unlike the hierarchical scheduling, the progressive approach relies solely on local information, meaning that each node knows only its direct one-hop neighbours and aggregated global information. Consequently, the scheduling cannot be globally assigned (like in the hierarchical scheduling, by the Base Station); instead, the scheduling follows a best-effort approach, dynamically deciding the next best hop (if needed) at each step.

To implement the progressive scheduling logic, we employ a combination of ranking and probabilistic selection. Let $\mathcal{N} = \{n, n_1, n_2, \dots\}$ be the set of potential allocation destinations, that includes node n and its one-hop neighbours. For each $\hat{n} \in \mathcal{N}$, we define a score function $v(\hat{n})$ as follows:

$$v(\hat{n}) = \alpha \cdot d(\hat{n})_\beta + (1 - \alpha) \cdot h(\hat{n}) \tag{1}$$

where $\alpha \in [0..1]$ is a pre-defined weight factor, $d(\hat{n})_\beta$ represents the direction factor, which depends on the parameter β that defines the maximum angular displacement, and the relative angular direction of $\hat{n}$ (having position $(x_{\hat{n}}, y_{\hat{n}})$) with respect to the allocation destination `position\X` and `position\Y`, if specified in the AHAPP request. This value is computed as follows:

$$\vec{nd} = (\texttt{position\X} - x_n,\ \texttt{position\Y} - y_n), \quad \vec{n\hat{n}} = (x_{\hat{n}} - x_n,\ y_{\hat{n}} - y_n)$$

$$\theta = \arccos\left(\frac{\vec{nd} \cdot \vec{n\hat{n}}}{\|\vec{nd}\|\ \|\vec{n\hat{n}}\|}\right)$$

$$d(\hat{n})_\beta = \begin{cases} 1, & \theta \leq 0°, \\ 1 - \dfrac{\theta}{\beta}, & 0° < \theta < \beta, \\ 0, & \theta \geq \beta. \end{cases} \tag{2}$$

In turn, $h(\hat{n})$ captures attribute constraints and is defined as:

$$h(\hat{n}) = \frac{\texttt{lock} + \sum_{a \in \text{ATT}} \min(1, (\frac{\hat{n}_\texttt{a}}{\texttt{a}})^2)}{|\text{ATT}| + 1} \tag{3}$$

where ATT = {`loadavg`, `energy`, `memory`, `storage`} is the set of attributes, $\hat{n}_\texttt{a}$ represents the aggregated value of attribute `a` received via the `Heartbeat`

message from node $\hat{n}$, and `lock` is 1 if the aggregated `device` constraints in $\hat{n}$ fulfil the request, and 0 otherwise.

After computing $v(\hat{n})$ for each node in $\mathcal{N}$, node n forwards the function request to node $\hat{n}$ with probability:

$$p_\gamma(v(\hat{n})) = \min\left(1, \left(\frac{v(\hat{n})}{\gamma}\right)^2\right) \tag{4}$$

where γ is determined based on the `targets` value, with specific parameters $\gamma_{\texttt{almost\_all}}$ and $\gamma_{\texttt{at\_least\_one}}$ — we remind that in the progressive scheduling, due to its best-effort nature, the `all` and `one` targets of the hierarchical scheduling are respectively mapped to their `almost_all` and `at_least_one` counterparts, reflecting the system's inability to guarantee a strong-semantics scheduling. If $\hat{n} \equiv n$, the function is executed locally at node n.

Scheduling Strategy	Knowledge	Scheduling	Semantic	Best Suited For
Hierarchical	Global (Full / Full w/ Change)	Direct selection optimal placement	Strong	Stable topologies low-latency networks
Progressive	Local (Aggregated)	Forwarding-based adaptive heuristic	Weak	Highly dynamic networks bandwidth-constrained systems

Table 2: Comparison of Function Allocation Strategies.

Semantics/Scheduling Selection. The selection of an appropriate function allocation semantics depends on multiple factors, including network stability, resource constraints, and application requirements. Table 1 provides an overview of the advantages and drawbacks of the two function allocation semantics. The observations align with those in the previous table on knowledge dissemination, as the effectiveness of scheduling methods is inherently tied to the availability and accuracy of network state information.

In Section 6, we conduct quantitative analyses that compare the above selection w.r.t. different network conditions.

In the future, we plan to explore more sophisticated approaches on a hybrid model that dynamically transitions between hierarchical and progressive strategies based on network conditions and resource availability. One such hybrid, adaptive mechanisms would autonomously switch among different dissemination and allocation methods to optimise allocation efficiency while minimizing communication overhead.

6. System Evaluation

We present an evaluation of the proposed serverless function scheduling framework for ad-hoc drone networks through extensive simulations. Note that, although we use the case study introduced in Section 1 throughout the article to illustrate the potential and practical applicability of our approach, our evaluation is more general and comprehensive. In addition to simulations based on scenarios from the case study, we conduct extensive generic simulations varying topologies, mobility patterns, and numbers of nodes, to thoroughly assess the performance and robustness of our proposed system under diverse network conditions.

6.1. Simulation Setup

To ensure a comprehensive evaluation, we conduct simulations in a controlled environment using predefined network configurations. For the simulation we use OMNeT++[4] coupled with the INET library[5]. We run the simulations with the default parameters defined in Table 3, which remain constant unless otherwise specified. In the following, we use dissemination method names and scheduling modalities interchangeably due to their tight coupling.

Table 3: Default Simulation Parameters

Parameter	**Value**
Number of Nodes n_U	12
Function request generation period n_U	20s
Node Speed (dynamic scenario)	15mps
Message Transmission Interval (t_H)	5s
Local Information Size (s_H)	176B
Ranking Weights (α, β)	$\alpha = 0.7$, $\beta = 90°$
Probability Factors `Heartbeat` (p_H)	0.1
Almost all probability ($\gamma_{\texttt{almost\_all}}$)	0.7
At least one probability ($\gamma_{\texttt{at\_least\_one}}$)	0.9

The simulation considers both static and mobile scenarios as well as a realistic case study based on the fire detection and survivor tracking scenario

[4]https://omnetpp.org/
[5]https://inet.omnetpp.org/

presented in Section 1. We generate random function requests, associated with scheduling policies compliant with the AHAPP language (cf. Figure 8). Additionally, we randomly assign the parameter values within each function request, ensuring a diverse set of test cases for evaluating system performance.

6.2. Performance Metrics

We evaluate our framework based on the following key performance indicators (KPI):

- **Average Function Deployment Delay**: The average time taken for a function to be successfully deployed.
- **Average Function Deployment Number**: The number of successfully deployed functions for each request generated in the system.
- **Network Overhead**: The size/amount of control messages exchanged in the network to achieve service deployment.

These metrics provide insights into the efficiency, reliability, and scalability of the proposed approach under different conditions.

6.3. Simulation Scenarios

We consider three simulation scenarios to evaluate the system under diverse network conditions:

- Static Grid-Based Deployment: in this scenario, nodes are statically placed in a grid topology. This setup allows us to assess performance in a well-structured network, where topology remains constant.
- Dynamic Mobile Nodes: where nodes move dynamically following a predefined mobility model. This scenario simulates a realistic UAV network, where nodes frequently change positions, impacting connectivity and deployment decisions. In this scenario, the nodes move at a fixed speed of 15mps.
- Case Study: Fire Detection and Rescue: this scenario follows the real-world use case described in Section 1, where a drone swarm is deployed for fire detection, survivor localization, and tracking. The goal is to evaluate system performance in a multi-phase, real-world environment.

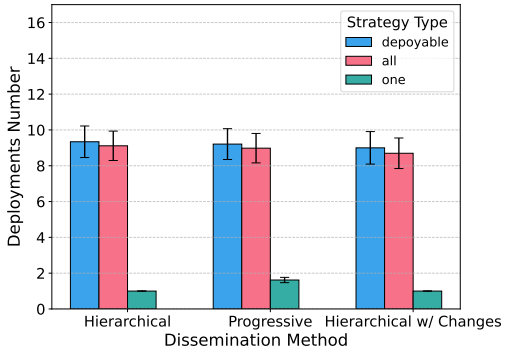

Figure 9: Number of deployed functions in the static scenario.

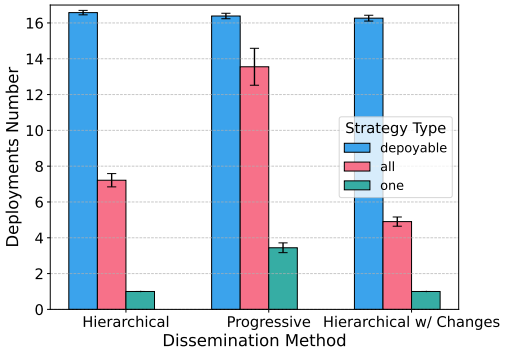

Figure 10: Number of deployed functions in the dynamic scenario.

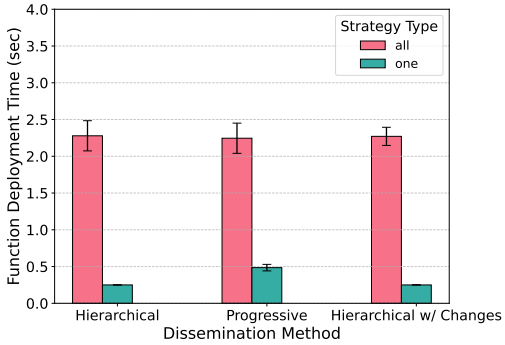

Figure 11: Deployment delay in the static scenario.

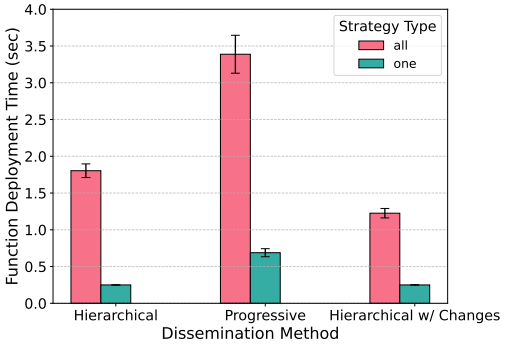

Figure 12: Deployment delay in the dynamic scenario.

6.4. Evaluation Results

We analyse the system behaviour under different deployment modalities and network configurations, focusing on the defined KPIs. The experiments are structured to evaluate the impact of scheduling policies, knowledge dissemination strategies, and network scalability.

Deployment Delay and Number of Deployments. To assess the efficiency of function execution, we compare the number of successfully deployed functions across different configurations. The results in Figure 9 and Figure 10 highlight the differences between the two main scheduling modalities, `all` and `one`, under various knowledge dissemination methods. In these figures, the blue bar represents the number of deployable nodes at function generation

time, providing a reference to evaluate the appropriateness of each modality w.r.t. the different scenarios.

In the static scenario, the `all` target modality achieves near-optimal results across all knowledge dissemination methods. This outcome is expected, as the static nature of the system ensures that all dissemination approaches effectively spread network information. The `one` modality also performs well, except in the progressive approach, where its distributed and probabilistic nature leads to function allocations on more than one node, increasing resource usage. Conversely, in the dynamic scenario, the hierarchical modality exhibits significantly lower performance. Both the Hierarchical and Hierarchical w/ Changes methods deploy less than half of the possible functions. This performance drop is primarily due to outdated network information at the Base Station, which prevents accurate function placement. Additionally, during function deployment, nodes move rapidly, meaning that some initially-selected nodes may no longer be in the correct area, while others, suitable ones are not considered. This limitation is effectively mitigated by the progressive method, which achieves near-optimal deployment. Its hop-by-hop, locally adaptive decision-making allows the system to dynamically respond to network topology changes, significantly improving function placement.

Figure 11 and Figure 12 illustrate the deployment delay, which follows the same trend observed in the function deployment analysis. The delay is inherently tied to the number of deployed functions—as more functions are successfully deployed, the average delay increases. Consequently, while the progressive method exhibits higher delay values, this is a direct result of its greater number of successful deployments, rather than an inefficiency in the scheduling mechanism.

Network Overhead Analysis. We evaluate the impact of the scheduling and knowledge dissemination modalities on network overhead by measuring the total number of control messages exchanged. The primary contributor to network load is knowledge dissemination, while function deployment incurs minimal overhead. We report the results in Figure 13 and Figure 14.

Full knowledge dissemination generates significant communication overhead, as each node periodically shares its entire network state with its neighbours. While this approach ensures high decision accuracy in static scenarios, it results in substantial bandwidth consumption. The Full with Change method improves performance by transmitting updates only when a node's state changes, providing a better trade-off between accuracy and efficiency.

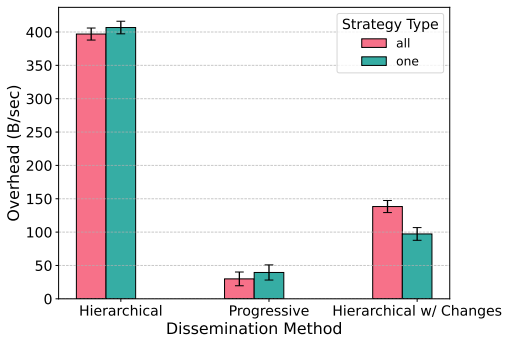

Figure 13: Network overhead delay in the static scenario.

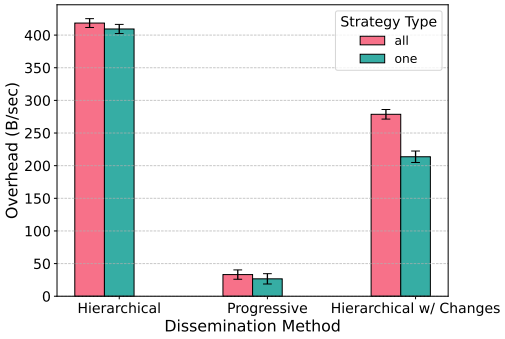

Figure 14: Network overhead delay in the dynamic scenario.

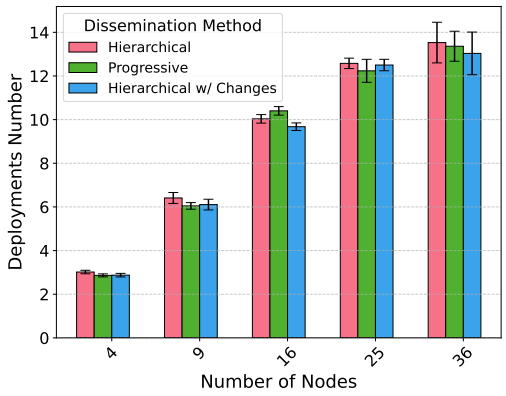

Figure 15: Number of deployed functions in the static scenario varying the number of nodes.

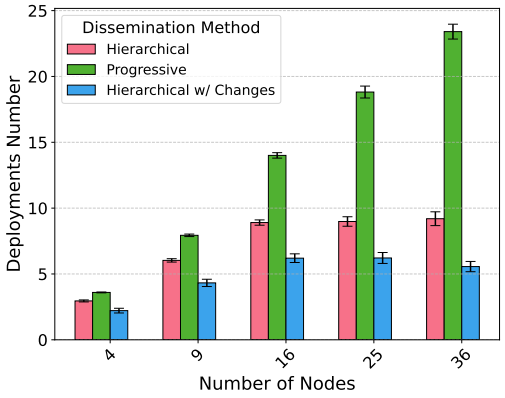

Figure 16: Number of deployed functions in the dynamic scenario varying the number of mobile nodes.

The Aggregated method, which shares only summary metrics, achieves the lowest overhead but at the cost of reduced scheduling precision. When comparing deployment modalities, the `all` and `one` approaches exhibit similar overhead levels due to the fixed periodic nature of knowledge dissemination. However, in the Full with Change method, the `all` targetting results in higher overhead. This increase is likely caused by network congestion, leading to packet losses and retransmissions, which ultimately inflate the total number of control messages exchanged.

Scalability: Impact of Network Size on Deployment Efficiency. To evaluate the scalability of the system, we analyse how the number of deployed func-

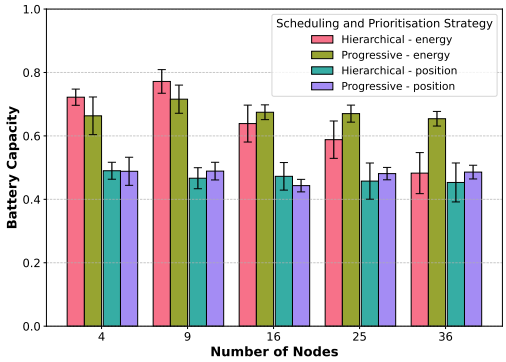

Figure 17: Battery Capacity for energy and position prioritisation `strategy` as the number of nodes increases. Higher values indicate better energy-aware deployments.

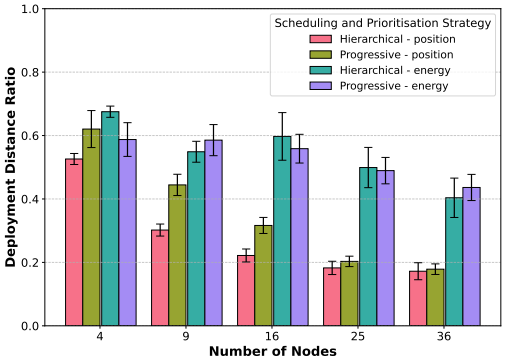

Figure 18: Deployment Distance Ratio for energy and position prioritisation `strategy` as the number of nodes increases. Lower values indicate better distance-aware deployments.

tions evolves as the number of nodes increases. The results are shown in Figure 15 and Figure 16, where we present data exclusively for the `all` targetting.

In the static scenario, all methods achieve optimal results, successfully deploying functions to all eligible nodes. However, in the dynamic scenario, the increasing number of nodes leads to a continuously changing network topology, introducing challenges for hierarchical approaches. In this case, both Hierarchical and Hierarchical with Changes exhibit limitations. The Hierarchical method struggles with outdated information and larger `Heartbeat` messages, increasing scheduling errors. The Hierarchical with Changes method, which benefits from selective updates in static scenarios, loses its advantage in dynamic networks due to the continuous need for updates, leading to frequent state inconsistencies.

In contrast, the Progressive approach scales effectively with the number of nodes. Its local and distributed decision-making allows it to adapt to mobility, ensuring a steady increase in the number of successfully deployed functions as more nodes become available. The results indicate that the Progressive method remains largely unaffected by network size and mobility, maintaining deployment efficiency across different scales.

Prioritisation Strategy Analysis. To assess how effectively the hierarchical and progressive scheduling strategies follow the user-defined prioritisation

`strategy`, we analyse two metrics: the *Battery Capacity* (Figure 17) and the *Deployment Distance Ratio* (Figure 18). The Battery Capacity measures how closely the selected nodes align with the goal of maximising residual *energy*, whereas the Deployment Distance Ratio evaluates *position* optimisation, indicating whether the selected nodes minimise the distance to a given target. Results show that both scheduling methods effectively adhere to the specified prioritisation strategies.

Specifically, in Figure 17, we report the Battery Capacity for both the energy and position prioritisation cases, using both hierarchical and progressive scheduling methods. When energy is prioritised, both strategies are able to select nodes with higher residual energy, achieving higher Battery Capacity values. However, the hierarchical scheduler shows some limitations as the number of nodes increases, mainly due to the difficulties in maintaining always up-to-date information in a dynamic scenario. Additionally, the Figure shows that when position is prioritised, the Battery Capacity tends to stabilize around 0.5 — confirming that energy is not a selection criterion in this configuration and is therefore not significantly affected.

Conversely, when distance is prioritised through the *position* `strategy`, both scheduling methods effectively select nodes closer to the target location. As shown in Figure 18, the Deployment Distance Ratio decreases with the increasing number of nodes, as more nodes become available near the target point. This feature allows the schedulers to better optimise the deployment with respect to distance, clearly improving that metric compared to the other `strategy` where position is not prioritised.

Case Study: a realistic serverless workflow. This section presents the results for the fire detection and rescue scenario, the realistic running case study we introduced in Section 1 and detailed in Section 4. The experiment simulates a 3×3 km^2 area where a swarm of drones is deployed. The simulation proceeds as follows:

1. At intervals of 1 minute, the Base Station (BS) issues the execution of the `ScanFireAlert` function, which is deployed to all drones equipped with a camera.
2. 5 minutes from the beginning of the simulation, we simulate a fire outbreak at specific spots in the area, triggering the alert by the previous functions and the request, by the Base Station, to execute the `ScanPeople` function on all drones near the fires.

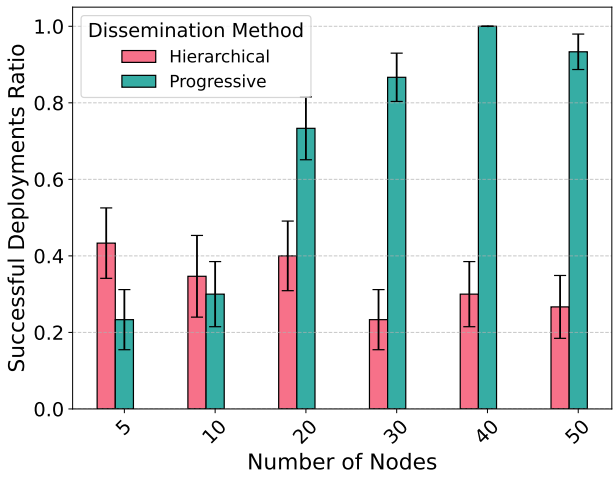

Figure 19: Successful deployment ratio in the fire detection and rescue scenario.

3. Similarly, we simulate the presence of survivors, some near the fire areas, which triggers the request to execute individual (one) `Track-Survivor` function instances to track the survivors.

For this experiment, all drones are assumed to be equipped with cameras and GPUs. As a performance metric, we calculate the *successful deployment ratio* across the three consecutive functions, i.e., the cases when all the three functions are successfully deployed.

Figure 19 presents the performance results as the number of drones varies. When the number of drones is low, only a small percentage of cases achieves successful deployment of the full function sequence, primarily due to limited area coverage. In this scenario, the Hierarchical method outperforms the Progressive method because the data exchange overhead is small in small-scale networks. However, as the number of drones increases, the Progressive method surpasses the Hierarchical approach, reaching a near-optimal success rate. Conversely, the Hierarchical method struggles with high network dynamism and an increasing number of nodes, causing its performance to stabilise around 0.25.

Overall, the experiments confirm that hierarchical scheduling is better for small or static networks, whereas progressive scheduling scales better in large and dynamic environments.

7. Related Work

Recent advancements in serverless computing have extended its capabilities to handle other environments than the cloud one, where the paradigm originated. The integration of serverless computing in mobile ad-hoc networks (MANETs) is a relatively new research direction, with most prior work focusing on edge, fog, and cloud computing methods for improving task offloading and resource allocation. In this section, we discuss related work from the literature.

One work close to ours is by Singh and Adhikari [33], who propose a hierarchical federated learning framework designed for edge-fog-cloud environments. Their solution introduces a pop-up ad-hoc network for dynamic communication between edge devices and fog servers, integrated with a serverless data pipeline to improve processing efficiency. To enhance participation and resource usage at the edge, the platform incorporates a game model that rewards edge devices based on their contributions. Patterson et al. [34] also present work close to ours, introducing a platform for swarm coordination using a centralised controller able to deploy serverless functions on both the cloud and edge devices, showing improved performance predictability, lower network traffic, and battery efficiency compared to existing alternatives.

Improving function execution in serverless platforms remains a focal point of research for serverless for edge computing. Jindal et al. [35] introduce the Function Delivery Network (FDN), an innovative extension of FaaS that enables function execution across heterogeneous clusters. Their Function-Delivery-as-a-Service model selects the most suitable execution platform based on platform characteristics, potential collaboration with other platforms, and data localisation. Jindal et al.'s evaluation shows that scheduling functions on edge platforms can substantially reduce energy consumption while maintaining adherence to Service Level Objective requirements. Garbugli et al. [36] tackle Quality of Service (QoS) management in IoT applications leveraging edge cloud FaaS through TEMPOS, a middleware that ensures performance reliability in environments with high stochastic variability. Their approach coordinates QoS monitoring, control, and management across the Cloud-to-Things continuum, tackling challenges such as bandwidth fluctuations and shared virtualised resources to provide a more stable and efficient execution environment.

Several studies have explored how edge, fog, and cloud computing can improve computation offloading in ad-hoc networks, particularly in UAV-

enabled systems. For instance, Grasso et al. [37] propose a reinforcement learning-based job offloading mechanism in a flying ad-hoc network (FANET), where UAVs with mobile edge computing (MEC) capabilities collaborate to balance computational loads and reduce delay. Similarly, You et al. [38] present a joint optimisation strategy for task scheduling, resource allocation, and UAV trajectory planning to enhance computing performance in FANETs. Other approaches leverage multi-tier architectures to enhance computation efficiency. For example, Luo et al. [39] introduce an adaptive scheduling model that dynamically distributes workloads across UAV-based edge servers, optimising energy consumption and execution latency. In the same direction, Niu et al. [40] focus on task scheduling in emergency scenarios, where UAVs act as mobile computing nodes to process data and maintain network connectivity in disaster-affected regions, while Khan et al. [41] integrate MEC with secure identity-based signcryption to ensure confidentiality and authentication in FANETs.

The integration of serverless computing in mobile ad-hoc networks is still in its early stages, with most efforts focusing on making serverless frameworks adaptable to edge computing environments. One of the key challenges in this domain is function scheduling, as traditional serverless models rely on centralised cloud infrastructure, whereas ad-hoc networks require decentralised, self-organising strategies. Several studies have examined orchestration mechanisms for UAV-based ad-hoc networks. For example, Grigoropoulos and Lalis [42] propose a hierarchical orchestration model that spans edge, fog, and cloud layers to improve function deployment in UAV-based networks. Similarly, Keshavamurthy et al. [43] introduce a distributed control framework for UAV relay networks, addressing key challenges in network topology management and function execution. Beyond terrestrial and aerial MEC systems, space-assisted computing has gained attention. Lin et al. [44] explore a game-theoretic framework for task offloading in tactical ad-hoc networks that integrate low-Earth orbit (LEO) satellites and UAV-based MEC nodes. Lin et al.'s work highlights the challenges of resource allocation and pricing strategies in heterogeneous environments where UAVs and satellites provide computation offloading capabilities. In terms of task scheduling, several studies propose methods based on heuristic and machine learning to improve execution efficiency. Qin et al. [45] introduce a time-sensitive task scheduling algorithm for UAV-based MEC systems, prioritising reconnaissance operations based on dynamic environmental conditions. Similarly, Secinti et al. [46] explores fog computing techniques for UAV-based software-defined networks,

enabling adaptive task allocation and network-aware function placement.

In ad-hoc serverless computing, where network topology is highly dynamic and nodes frequently join or leave the network, service discovery is essential to ensure that function invocations can reliably locate and communicate with appropriate execution nodes. Ververidis Polyzos [47] provide a comprehensive survey of service discovery mechanisms for MANETs, categorising different strategies based on centralised, decentralised, and hybrid architectures. Beyond individual studies, Ferrer et al. [48] provide a comprehensive survey of mobile cloud, edge computing, and mobile ad-hoc computing, identifying opportunities and challenges in moving from centralised cloud models to fully distributed cloud paradigms. The study highlights ad-hoc cloud computing as a critical evolution in distributed computing, emphasizing the need for self-organising, fault-tolerant resource allocation mechanisms.

Comparison with Our Work. Besides the works by Singh and Adhikari and Patterson et al., which are the closest to ours but concentrate on a centralised interpretation of serverless function distribution, the aforementioned studies focus primarily on computation offloading, task scheduling, and orchestration in MANET and FANET networks. Hence, none specifically address the design of a fully decentralised, topology-aware serverless execution model that integrates function scheduling with the constraints of a mobile ad-hoc environment. Our approach introduces a serverless deployment-aware protocol that abstracts the complexity of MANETs while ensuring efficient function placement and execution through a combination of gossip-based knowledge dissemination and adaptive function allocation.

Furthermore, our work introduces scheduling mechanisms that support both hierarchical and progressive deployment models, enabling flexible adaptation to different network conditions. By integrating these techniques, we present a first-of-its-kind solution for serverless computing in ad-hoc networks, addressing key challenges in network topology awareness, function invocation reliability, and execution efficiency. Unlike previous approaches, our work is the first to apply serverless computing to an ad-hoc context where nodes function both as execution hosts and as schedulers, enabling a fully decentralised and self-organising computing paradigm.

To provide a broader overview of the positioning of our work, we present a comparative analysis with the main related studies discussed in the literature. We organise the comparison into two complementary tables.

The first table, reported in Table 4, focuses on the serverless comput-

Works	Decentralised Scheduling	Multi-node Execution	Execution Offloading	Custom Policies
Grasso et al. [37]	●	○	●	○
Grigoropoulos and Lalis [42]	○	○	○	●
Khan et al. [41]	○	○	◐	○
Luo et al. [39]	○	○	○	○
Niu et al. [40]	○	○	○	○
Patterson et al. [34]	○	○	○	●
You et al. [38]	○	○	●	○
This Work	●	●	●	●

Table 4: Comparison of our work (last row) with related one from the literature on serverless.

ing perspective, summarising the key contributions and limitations of the works targeting function scheduling and orchestration in distributed environments. The comparison highlights the lack of solutions that specifically address the ad-hoc and decentralised execution context that characterises our proposal. Specifically, starting from the left-most column, we find "Decentralised Scheduling", which is the main characteristic of our system, i.e., the fact that scheduling happens in a decentralised way, following the multi-hop protocol described in Section 3.4. Among these works, only one adopts a decentralised scheduling model, which is the solution presented by Grasso et al. [37]. Another attribute that distinguishes our work is "Multi-node Execution", which corresponds to a single user request triggering the execution of a function across multiple nodes. Ours is the only alternative that supports one such feature. However, Khan et al. [41] partially cover this feature by allowing a task to be split into multiple subtasks processed in parallel by different nodes. "Execution Offloading" regards the possibility for the execution nodes (the drones) to delegate function executions to other systems. The proposals by You et al. [38] and Grasso et al. [37] also support this feature, while Khan et al. [41] implement a restricted version of it, limited to offloading to the edge. Finally, "Custom Policy" is the support for user-defined, custom scheduling policies (which our system offers through AHAPP). Patterson et al. [34] support this feature via HiveMind and Grigoropoulos and Lalis [42] enable users to specify custom policies through their Kubernetes-based setup and the custom Fractus scheduler (both centralised). In summary, our system is the only one to support all five capabilities in a cohesive, decentralised architecture.

Works	Ad-Hoc Network Support	Worker-Agnostic Deployment	Support for Dynamic Topology	Generic Resource Awareness
Grasso et al. [37]	◐	○	◐	◐
Grigoropoulos and Lalis [42]	◐	◐	◐	◐
Keshavamurthy et al. [43]	◐	○	◐	◐
Khan et al. [41]	◐	○	◐	○
Lin et al. [44]	◐	○	◐	◐
Luo et al. [39]	◐	○	◐	◐
Niu et al. [40]	◐	○	◐	◐
Qin et al. [45]	◐	○	◐	●
Secinti et al. [46]	◐	○	◐	◐
You et al. [38]	◐	○	◐	◐
This Work	●	●	●	●

Table 5: Comparison of our work (last row) with related one from the literature on network and ad-hoc systems.

The second table, reported in Table 5, provides a comparative analysis from the network and ad-hoc system perspective. We consider key characteristics that are crucial in the context of mobile ad-hoc networks and UAV-based systems: "Ad-hoc Network Support" indicates whether the system is designed to operate over a decentralised, infrastructure-less network; "Worker-Agnostic Deployment" refers to the ability of the scheduling strategy to operate without requiring prior knowledge of the specific execution nodes at scheduling time — an essential property for progressive, fully decentralised deployments; "Support for Dynamic Topology" denotes the capability of the system to adapt to topology changes, node mobility, and unpredictable network dynamics; finally, "Generic Resource Awareness" evaluates whether the system can take into account generic node characteristics like energy, position, and hardware capabilities during scheduling decisions, besides the traditional ones like load and memory occupancy. We deem all the considered related work only partially supporting ad-hoc networks (first column) because none of them supports multi-hop deployments (this aspect is complementary to the decentralised scheduling approach found in Table 4) — from the intersection between the first columns of the two tables, we note that, while Grasso et al. [37] support decentralised scheduling, they rely on 5G connectivity and they do not (need to) support multi-hop communications. Regarding "Worker-Agnostic Deployment", the only proposal that partially addresses this concern is by Grigoropoulos and Lalis [42], who, by parti-

tioning workers into homogeneous pools can abstract away from an exact knowledge of the workers, following a hierarchical scheduling logic that first selects the pool and then selects one member in that pool based on a companion worker-selection policy. None of the considered proposals have "Support for Dynamic Topology" because they take scheduling decisions based on a centralised view of the topology, which might substantially differ from the actual one — the topology might change drastically during a scheduling session in highly dynamic scenarios. Lastly, only Qin et al. [45] give adequate support for "Generic Resource Awareness" since they handle task scheduling also considering energy and application-specific sensor values.

8. Conclusion

We present a novel proposal for serverless computing tailored for ad-hoc drone networks, specifically designed to address the peculiar challenges posed by such dynamic and resource-constrained environments. Our approach leverages a two-layer network overlay architecture, combining a gossiping layer for efficient network management with a function scheduling layer that adapts to the varying conditions of drone swarm operations. Through the integration of hierarchical and progressive scheduling approaches, we ensure robust and flexible function execution, even in the face of intermittent connectivity and high latencies.

Our experimental results demonstrate the effectiveness of our proposal, highlighting its ability to efficiently manage the execution of serverless functions while reducing communication overhead and maintain high reliability. Specifically, our simulations show that progressive scheduling achieves nearly optimal deployment success (close to 100%) in highly dynamic scenarios with up to 50 drones, compared to approximately 25% for hierarchical scheduling under similar conditions. Moreover, progressive scheduling significantly reduces network overhead, approximately halving the communication cost observed with hierarchical methods. These findings validate the potential of serverless computing to enhance the performance and adaptability of drone swarms, offering a promising avenue for future development in both civilian and commercial applications.

Looking ahead, there are several exciting opportunities for further improving the system. For instance, we envision working on refining the AHAPP language to capture function-specific traits, e.g., linked to the performance of function execution on the disparate hardware of a given drone swarm. This

idea comes from a dialect of APP, called cAPP [27, 23], which uses static analysis to derive cost equations from the function's code and select the best (cloud) node for executing that function given, e.g., recorded latencies to call external services. In our settings, one can incorporate these cost estimations w.r.t. the (recorded performance of the) different hardware found among the components of the network and provide refined rankings for selecting the most suitable allocation targets.

Beyond the refinement of the scheduling language, further research in ad-hoc networks can enhance function deployment. One avenue is adaptive knowledge dissemination, where the system dynamically selects the most suitable method—Full, Full with Changes, or Aggregated—based on topology stability and network congestion. The current deployment-aware protocol considers only node capabilities, neglecting network conditions and link quality. These aspects are important in multi-hop networks, especially for functions that need to frequently push data outside the drone network (e.g., to report continuous updates to the user). Ignoring link reliability can lead to high-latency interactions and execution failures, highlighting the need for network-aware function placement. Another avenue is improving aggregation methods for Progressive deployment. The existing approach uses summary metrics, but further optimisation could involve probabilistic modelling, weighted aggregation based on node stability, or adaptive techniques that adjust dynamically to network variations. These refinements would enhance scheduling accuracy and efficiency in highly dynamic environments. Finally, improving resilience in dynamic topologies remains essential. While Progressive scheduling adapts to mobility, predictive models leveraging reinforcement learning or distributed heuristics could anticipate topology changes, enabling pre-emptive function placement and reducing failures caused by connectivity disruptions.

Acknowledgements

The work has been partially supported by the PRIN 2022 project RAIN4C: "Reliable Aerial and satellIte Networks: joint Communication, Computation, Caching for Critical scenarios" (Id: 20227N3SPN, CUP: J53D23007020001), the research project FREEDA (CUP: I53D23003550006) funded by the framework PRIN 2022 (MUR, Italy), the French ANR project SmartCloud ANR-23-CE25-0012, and PNRR CN HPC - SPOKE 9 - Innovation Grant LEONARDO

- TASI - RTMER funded by the NextGenerationEU European initiative through the MUR, Italy (CUP: J33C22001170001).

References

[1] L. Gupta, R. Jain, G. Vaszkun, Survey of important issues in UAV communication networks, IEEE Commun. Surv. Tutorials 18 (2) (2016) 1123–1152. doi:10.1109/COMST.2015.2495297.
URL https://doi.org/10.1109/COMST.2015.2495297

[2] Y. Zeng, R. Zhang, T. J. Lim, Wireless communications with unmanned aerial vehicles: opportunities and challenges, IEEE Commun. Mag. 54 (5) (2016) 36–42. doi:10.1109/MCOM.2016.7470933.
URL https://doi.org/10.1109/MCOM.2016.7470933

[3] P. Doherty, P. Rudol, A UAV search and rescue scenario with human body detection and geolocalization, in: M. A. Orgun, J. Thornton (Eds.), AI 2007: Advances in Artificial Intelligence, 20th Australian Joint Conference on Artificial Intelligence, Gold Coast, Australia, December 2-6, 2007, Proceedings, Vol. 4830 of Lecture Notes in Computer Science, Springer, 2007, pp. 1–13. doi:10.1007/978-3-540-76928-6_1.
URL https://doi.org/10.1007/978-3-540-76928-6_1

[4] S. A. F. Manssor, S. Sun, M. A. G. Al-Sadoon, S. Ali, Real-time human detection in thermal infrared imaging at night using enhanced tiny-yolov3 network, J. Real Time Image Process. 19 (2) (2022) 261–274. doi:10.1007/S11554-021-01182-Z.
URL https://doi.org/10.1007/s11554-021-01182-z

[5] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Menezes Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. A. Popa, I. Stoica, D. A. Patterson, Cloud programming simplified: A berkeley view on serverless computing, Tech. Rep. UCB/EECS-2019-3, EECS Department, University of California, Berkeley (02 2019).

[6] A. E. Yaacoub, L. Mottola, T. Voigt, P. Rümmer, Nerta: Enabling dynamic software updates in mobile robotics, in: A. Ferscha, M. C. Chan, S. S. Kanhere, R. R. V. Prasad (Eds.), Proceedings of the 2022

International Conference on Embedded Wireless Systems and Networks, EWSN 2022, Linz, Austria, October 3-5, 2022, Junction Publishing / ACM, 2022, pp. 120–125. doi:10.5555/3578948.3578959.
URL https://dl.acm.org/doi/10.5555/3578948.3578959

[7] G. De Palma, S. Giallorenzo, J. Mauro, M. Trentin, G. Zavattaro, Webassembly at the edge: Benchmarking a serverless platform for private edge cloud systems, IEEE Internet Computing (01) (2024) 1–8. doi:10.1109/MIC.2024.3513035.

[8] G. De Palma, S. Giallorenzo, J. Mauro, M. Trentin, G. Zavattaro, Funless: Functions-as-a-service for private edge cloud systems, in: IEEE International Conference on Web Services, ICWS 2024, Shenzhen, China, July 7-13, 2024, IEEE, 2024, pp. 961–967. doi:10.1109/ICWS62655.2024.00114.

[9] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Serverless computation with openlambda, in: 8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16), 2016.

[10] J. Sampé, M. Sánchez-Artigas, P. García-López, G. París, Data-driven serverless functions for object storage, in: Middleware, Middleware '17, ACM, 2017, pp. 121–133. doi:10.1145/3135974.3135980.
URL https://doi.org/10.1145/3135974.3135980

[11] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, R. Arpaci-Dusseau, {SOCK}: Rapid task provisioning with {Serverless-Optimized} containers, in: 2018 USENIX Annual Technical Conference (USENIX ATC 18), 2018, pp. 57–70.

[12] M. Shahrad, J. Balkind, D. Wentzlaff, Architectural implications of function-as-a-service computing, in: Proc. of MICRO, 2019, pp. 1063–1075.

[13] K. Solaiman, M. A. Adnan, Wlec: A not so cold architecture to mitigate cold start problem in serverless computing, in: 2020 IEEE International Conference on Cloud Engineering (IC2E), 2020, pp. 144–153. doi:10.1109/IC2E48712.2020.00022.

[14] A. Banaei, M. Sharifi, Etas: predictive scheduling of functions on worker nodes of apache openwhisk platform, The Journal of Supercomputing (9 2021). doi:10.1007/s11227-021-04057-z.

[15] C. P. Smith, A. Jindal, M. Chadha, M. Gerndt, S. Benedict, Fado: Faas functions and data orchestrator for multiple serverless edge-cloud clusters, in: 2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC), IEEE, 2022, pp. 17–25.

[16] G. D. Palma, S. Giallorenzo, J. Mauro, G. Zavattaro, Allocation priority policies for serverless function-execution scheduling optimisation, in: E. Kafeza, B. Benatallah, F. Martinelli, H. Hacid, A. Bouguettaya, H. Motahari (Eds.), Service-Oriented Computing - 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14-17, 2020, Proceedings, Vol. 12571 of Lecture Notes in Computer Science, Springer, 2020, pp. 416–430. doi:10.1007/978-3-030-65310-1_29.

[17] G. De Palma, S. Giallorenzo, J. Mauro, M. Trentin, G. Zavattaro, Affinity-aware serverless function scheduling, in: 22nd IEEE International Conference on Software Architecture, ICSA 2025, Odense, Denmark, March 31-April 4, 2025, IEEE, 2025.

[18] C. E. Perkins, Ad hoc networking, Pearson Education India, 2008.

[19] F. Al-Turjman, M. Abujubbeh, A. Malekloo, L. Mostarda, Uavs assessment in software-defined iot networks: An overview, Comput. Commun. 150 (2020) 519–536. doi:10.1016/J.COMCOM.2019.12.004.
URL https://doi.org/10.1016/j.comcom.2019.12.004

[20] M. Mozaffari, W. Saad, M. Bennis, M. Debbah, Mobile unmanned aerial vehicles (uavs) for energy-efficient internet of things communications, IEEE Trans. Wirel. Commun. 16 (11) (2017) 7574–7589. doi:10.1109/TWC.2017.2751045.
URL https://doi.org/10.1109/TWC.2017.2751045

[21] D. Shah, Network gossip algorithms, in: 2009 IEEE International Conference on Acoustics, Speech and Signal Processing, IEEE, 2009, pp. 3673–3676.

[22] X. Fang, B. Zhang, D. Yuan, Gossip-based asynchronous algorithms for distributed composite optimization, Neurocomputing 616 (2025) 128952.

[23] G. De Palma, S. Giallorenzo, J. Mauro, M. Trentin, G. Zavattaro, Function-as-a-service allocation policies made formal, in: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation. REoCAS Colloquium in Honor of Rocco De Nicola - 12th International Symposium, ISoLA 2024, Crete, Greece, October 27-31, 2024, Proceedings, Part I, Vol. 15219 of Lecture Notes in Computer Science, Springer, 2024, pp. 306–321. doi:10.1007/978-3-031-73709-1_19.

[24] G. De Palma, S. Giallorenzo, J. Mauro, M. Trentin, G. Zavattaro, An openwhisk extension for topology-aware allocation priority policies, in: I. Castellani, F. Tiezzi (Eds.), Coordination Models and Languages - 26th IFIP WG 6.1 International Conference, COORDINATION 2024, Held as Part of the 19th International Federated Conference on Distributed Computing Techniques, DisCoTec 2024, Groningen, The Netherlands, June 17-21, 2024, Proceedings, Vol. 14676 of Lecture Notes in Computer Science, Springer, 2024, pp. 201–218. doi:10.1007/978-3-031-62697-5_11.

[25] G. De Palma, S. Giallorenzo, J. Mauro, M. Trentin, G. Zavattaro, Formally verifying function scheduling properties in serverless applications, IT Prof. 25 (6) (2023) 94–99. doi:10.1109/MITP.2023.3333071.

[26] G. De Palma, S. Giallorenzo, J. Mauro, M. Trentin, G. Zavattaro, Custom serverless function scheduling policies: An APP tutorial, in: G. Dorai, M. Gabbrielli, G. Manzonetto, A. Osmani, M. Prandini, G. Zavattaro, O. Zimmermann (Eds.), Joint Post-proceedings of the Third and Fourth International Conference on Microservices, Microservices 2020/2022, May 10-12, 2022, Paris, France, Vol. 111 of OASIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 5:1–5:16. doi:10.4230/OASICS.MICROSERVICES.2020-2022.5.

[27] G. De Palma, S. Giallorenzo, C. Laneve, J. Mauro, M. Trentin, G. Zavattaro, Serverless scheduling policies based on cost analysis, in: M. H. ter Beek, C. Dubslaff (Eds.), Proceedings of the First Workshop on Trends in Configurable Systems Analysis, TiCSA@ETAPS 2023, Paris, France, 23rd April 2023, Vol. 392 of EPTCS, 2023, pp. 40–52. doi:10.4204/EPTCS.392.3.

[28] G. De Palma, S. Giallorenzo, J. Mauro, M. Trentin, G. Zavattaro, A declarative approach to topology-aware serverless function-execution scheduling, in: C. A. Ardagna, N. L. Atukorala, B. Benatallah, A. Bouguettaya, F. Casati, C. K. Chang, R. N. Chang, E. Damiani, C. G. Guegan, R. Ward, F. Xhafa, X. Xu, J. Zhang (Eds.), IEEE International Conference on Web Services, ICWS 2022, Barcelona, Spain, July 10-16, 2022, IEEE, 2022, pp. 337–342. doi:10.1109/ICWS55610.2022.00056.

[29] YAML, YAML specification, https://yaml.org/spec/ (11 2022).

[30] N. Kumar, S. Misra, M. S. Obaidat, Collaborative learning automata-based routing for rescue operations in dense urban regions using vehicular sensor networks, IEEE Syst. J. 9 (3) (2015) 1081–1090. doi:10.1109/JSYST.2014.2335451.
URL https://doi.org/10.1109/JSYST.2014.2335451

[31] S. Liu, C. Dong, X. Zhu, L. Zhang, et al., Performance evaluation of batman-adv protocol on convergecast communication in uav networks, in: GLOBECOM 2022-2022 IEEE Global Communications Conference, IEEE, 2022, pp. 5105–5110.

[32] B. Karp, H.-T. Kung, Gpsr: Greedy perimeter stateless routing for wireless networks, in: Proceedings of the 6th annual international conference on Mobile computing and networking, 2000, pp. 243–254.

[33] N. Singh, M. Adhikari, Popfl: A scalable federated learning model in serverless edge computing integrating with dynamic pop-up network, Ad Hoc Networks 169 (2025) 103728.

[34] L. Patterson, D. Pigorovsky, B. Dempsey, N. Lazarev, A. Shah, C. Steinhoff, A. Bruno, J. Hu, C. Delimitrou, Hivemind: a hardware-software system stack for serverless edge swarms, in: V. Salapura, M. Zahran, F. Chong, L. Tang (Eds.), ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022, ACM, 2022, pp. 800–816. doi:10.1145/3470496.3527407.
URL https://doi.org/10.1145/3470496.3527407

[35] A. Jindal, M. Gerndt, M. Chadha, V. Podolskiy, P. Chen, Function delivery network: Extending serverless computing for heterogeneous platforms, Software: Practice and Experience 51 (9) (2021) 1936–1963.

[36] A. Garbugli, A. Sabbioni, A. Corradi, P. Bellavista, Tempos: Qos management middleware for edge cloud computing faas in the internet of things, IEEE Access 10 (2022) 49114–49127. doi:10.1109/ACCESS.2022.3173434.

[37] C. Grasso, R. Raftopoulos, G. Schembra, Deep q-learning for job offloading orchestration in a fleet of mec uavs in 5g environments, in: 2021 IEEE 7th International Conference on Network Softwarization (NetSoft), IEEE, 2021, pp. 186–190.

[38] W. You, C. Dong, Q. Wu, Y. Qu, Y. Wu, R. He, Joint task scheduling, resource allocation, and uav trajectory under clustering for fanets, China Communications 19 (1) (2022) 104–118.

[39] Y. Luo, W. Ding, B. Zhang, Optimization of task scheduling and dynamic service strategy for multi-uav-enabled mobile-edge computing system, IEEE Transactions on Cognitive Communications and Networking 7 (3) (2021) 970–984.

[40] Z. Niu, H. Liu, X. Lin, J. Du, Task scheduling with uav-assisted dispersed computing for disaster scenario, IEEE Systems Journal 16 (4) (2022) 6429–6440.

[41] M. A. Khan, I. Ullah, S. Nisar, F. Noor, I. M. Qureshi, F. Khanzada, H. Khattak, M. A. Aziz, Multiaccess edge computing empowered flying ad hoc networks with secure deployment using identity-based generalized signcryption, Mobile Information Systems 2020 (1) (2020) 8861947.

[42] N. Grigoropoulos, S. Lalis, Fractus: Orchestration of distributed applications in the drone-edge-cloud continuum, in: 2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC), IEEE, 2022, pp. 838–848.

[43] B. Keshavamurthy, M. A. Bliss, N. Michelusi, Maestro-x: Distributed orchestration of rotary-wing uav-relay swarms, IEEE Transactions on Cognitive Communications and Networking 9 (3) (2023) 794–810.

[44] X. Lin, A. Liu, C. Han, X. Liang, K. Pan, Z. Gao, Leo satellite and uavs assisted mobile edge computing for tactical ad-hoc network: A game theory approach, IEEE Internet of Things Journal (2023).

[45] Z. Qin, H. Wang, Z. Wei, Y. Qu, F. Xiong, H. Dai, T. Wu, Task selection and scheduling in uav-enabled mec for reconnaissance with time-varying priorities, IEEE Internet of Things Journal 8 (24) (2021) 17290–17307.

[46] G. Secinti, A. Trotta, S. Mohanti, M. Di Felice, K. R. Chowdhury, Focus: Fog computing in uas software-defined mesh networks, IEEE Transactions on Intelligent Transportation Systems 21 (6) (2019) 2664–2674.

[47] C. N. Ververidis, G. C. Polyzos, Service discovery for mobile ad hoc networks: a survey of issues and techniques, IEEE Communications Surveys & Tutorials 10 (3) (2008) 30–45.

[48] A. J. Ferrer, J. M. Marquès, J. Jorba, Towards the decentralised cloud: Survey on approaches and challenges for mobile, ad hoc, and edge computing, ACM Computing Surveys (CSUR) 51 (6) (2019) 1–36.