System Architecture for Autonomous Drone-Based Remote Sensing

Manos Koutsoubelias(✉), Nasos Grigoropoulos, Giorgos Polychronis, Giannis Badakis, and Spyros Lalis

Electrical and Computer Engineering Department,
University of Thessaly, Volos, Greece
{emkouts,athgrigo,gpolychronis,badakis,lalis}@uth.gr

Abstract. Thanks to modern autopilot hardware and software, multi-rotor drones can fly and perform different maneuvers in a precise way, guided merely by high-level commands. This, in turn, opens the way towards fully automated drone-based systems whose operation can be driven by a computer program, without any human intervention. In this work, we present a modular architecture for such a system, which integrates a drone, a hangar, battery charger and a weather station with the necessary software components so as to provide an autonomous remote sensing service, which can operate at the edge while being interfaced as needed with external systems and applications. The proposed system architecture is described in detail, focusing on the core software components and the interaction between them. We also discuss the drone and ground station that is used to test our implementation in the field as well as a simulation environment which allows us to perform a wide range of experiments in a flexible and controlled way.

Keywords: System architecture · Software design · Drones · Automation · Remote monitoring · Edge computing

1 Introduction

Remote monitoring has long been used to gather data about geographical regions of interest. For many years this was done through advanced satellite technology. However, this comes at a very high cost and in numerous cases data for the area of interest is available only during specific times of the day depending on the location and field of view of the satellite carrying the required sensing equipment. For this reason, a large number of on-demand sensing tasks were conducted using manned aircraft equipped with the proper sensors.

This research has been co-finance by the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH - CREATE - INNOVATE, project PV-Auto-Scout, code T1EDK-02435.

T. Hara and H. Yamaguchi (Eds.): MobiQuitous 2021, LNICST 419, pp. 220–242, 2022.
https://doi.org/10.1007/978-3-030-94822-1_13

During the recent years, a far cheaper and more flexible approach is to use unmanned aerial vehicles, also referred to as drones. Indeed, drones have become very popular and are now being used in an increasing number of applications, such as agriculture, structural health monitoring and surveillance. What drives this widespread use is the fact that modern drones can practically fly on their own, thanks to advanced embedded autopilot subsystems which continuously gather data from various on-board sensors, process them in real-time, and take all the steering decisions necessary to keep the vehicle steady during flight. In particular, multi-rotor drones (polycopters) can perform very precise maneuvers and even hold completely still, hovering in their current position. As a result, these drones can be flown via simple high-level commands, even by laymen who have little or no piloting experience.

In the same manner, such drones can be flown by computer programs too. There are several efforts to provide suitable tools and programming abstractions that simplify the development of computer-driven missions. Moreover, using available precision landing sensor technologies, the drone can accurately land even on small target areas, such as the landing pad of a hangar that can provide shelter when the drone is not being used and serve as a battery recharging (or battery switching) station during missions.

By combining these technologies, it becomes possible to fully automate the entire cycle of drone operation, thereby opening the way to a new class of drone-based remote sensing systems. This is particularly attractive if the drone routinely needs to perform the same sensing tasks over an area of interest, as this is typically the case in several monitoring and surveillance applications.

In this paper, we present a complete architecture for such a system, which integrates a drone, its hangar and battery charging subsystem, a local weather station and all the software components that are needed in order to provide a complete and fully autonomous remote sensing service. The main contributions of the paper are: (i) We propose a modular architecture for autonomous drone-based remote sensing systems, which can operate at the edge in a standalone way or as part of a more complex ecosystem. (ii) We describe the proposed design in detail, focusing on the software components that are responsible for the core system operation and mission execution. (iii) We discuss the drone and ground station setup we use in our field tests as well as a simulation environment used to test the functionality of our system in the lab for a wide range of scenarios.

We wish to note that our work focuses on outdoor sensing scenarios, using drones with autonomous flight, navigation and landing capability – many such platforms exist, and nowadays it is also relatively straightforward to even have custom designs, such as the drone we use in our tests. While some elements of the proposed system could be potentially reused to support indoor scenarios, this direction is beyond the scope of this work.

The rest of the paper is structured as follows. Section 2 presents the high-level system architecture. Section 3 discusses the main management logic, while Sect. 4 focuses on the aspect of mission execution. Section 5 describes the hardware-based and simulation-based setup used to test our implementation. Section 6

gives an overview of related work. Finally, Sect. 7 concludes the paper and provides some directions for future work.

2 System Concept and Architecture

We assume a geographical area or specific points of interest where certain sensing tasks can be performed using a drone. On the one hand, there can be standard sensing tasks that have to be conducted on a periodic basis, without any explicit request. On the other hand, certain sensing tasks may be introduced on demand, triggered by requests coming from external systems, e.g., end-user applications.

To support such a remote sensing capability, we propose a system that can manage the full cycle of operation—not just the flight of the drone during a mission, but also all the take-off and landing, storage and battery charging procedures—with little or no human intervention. Of course, the sensing equipment that needs to be mounted on the drone, the data collected via these sensors and the processing that needs to be performed on this data, all depend on the application. Nevertheless, having a general system design and implementation, which properly addresses all the application-neutral aspects of such a drone-based remote sensing system, is of big value as it is then possible to support any specific application with minimal customization and debugging effort.

In the spirit of edge computing, we envision such a system to run on a ground station close to the area of operation. This way one can have a direct and low-latency communication with the drone via WiFi or long-range RF technology, enabling autonomous and robust operation without relying on fast and stable connectivity with the cloud—even in the era of 5G, good Internet connectivity is by no means guaranteed if the system needs to operate in remote areas. The basic system infrastructure includes a suitably designed hangar where the drone can land autonomously, a battery charging subsystem (inside the hangar) and a local weather station, with which the ground station can communicate over dedicated serial links or a local area network. We assume that standard GPS accuracy is sufficient for the navigation and sensing tasks of the drone. To support applications with higher accuracy requirements, the system infrastructure and the drone need to be equipped with more advanced positioning subsystems, such as GPS RTK and/or visual positioning systems (VPS), but this does not affect the proposed system design. Finally, we assume that the drone can land on the hangar with high accuracy, supported by an appropriate precision landing mechanism, and dock on the battery charging system without manual intervention.

From a software perspective, the proposed system is built in a modular way, being composed of micro-services which are combined to provide the overall functionality. The top-level system architecture is shown in Fig. 1. Some components play the role of high-level drivers and provide access to the physical parts of the system, while others are responsible for purely software-based functions. Next, we outline the role of each component.

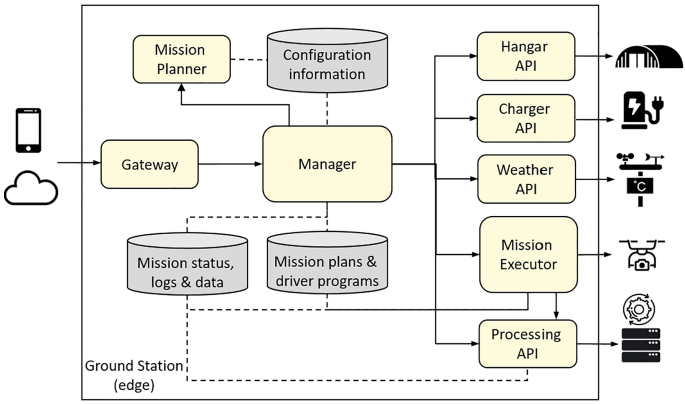


Fig. 1. Top-level system architecture.

The Gateway provides a technology-neutral interface towards external systems, including the end-user application through which the system can be managed remotely. Its role is to convey incoming requests to the Manager and send back the corresponding replies. Note that the interaction with external systems is not crucial for the core system operation; the system can execute monitoring tasks in an autonomous manner, even when having unstable/bad connectivity to external systems. Of course, access to the Gateway needs to be properly secured in order to avoid unauthorized access. This can be achieved using well-established technologies, such as VPNs in combination with more refined access control mechanisms (which is not the focus of this work).

The heart of the system is the Manager, which is responsible for the automation of the entire cycle of operation. Briefly, the Manager picks the mission plan for the sensing task at hand, makes the initial preparations and starts the execution of the plan. During execution, it monitors progress and updates the mission state. When the mission is completed, the Manager performs the necessary finalization actions so that the system becomes ready for the next sensing task. Section 3 describes the Manager in more detail.

The mission plans for the sensing tasks can be predefined or generated in a dynamic way. The former typically applies to standard tasks with fixed operational parameters. The latter is needed for tasks with open parameters that are determined at launch time (e.g., the specific points of interest are supplied as part of an on-demand sensing request), or to support replanning in case a sensing task is aborted. In these cases, the Mission Planner is invoked in order to produce a suitable plan. If the application does not have such requirements, the Mission Planner component becomes superfluous and is not included in the system configuration.

The Mission Executor is responsible for the actual execution of the mission plans. This is done using a so-called driver program, which parses the plan and performs the required interactions with the drone. It is important to note that the

system may support various types/formats of sensing missions, and each mission type typically comes with its own driver program. The internal operation of the Mission Executor is described in Sect. 4.

The rest of the components serve as front-ends for different peripheral subsystems. The Hangar API supports the interaction with the hangar where the drone is stored, while the Charger API is used to access the battery charging subsystem. Information about the current weather conditions can be retrieved via the Weather API that communicates with a local station near the area of operation. Finally, the Processing API is used to process the data collected by the drone by employing suitable services, which may run locally on the ground station or be deployed in more powerful nearby edge infrastructure. Data processing may take place during the mission or once the mission completes, depending on the application and system configuration. Of course, the Processing API is not needed for pure data collection applications.

Further, the system keeps certain information on persistent storage. This includes specific points of interest, no-fly zones and drone characteristics, e.g., estimated flight time as a function of average flight speed, which may need to be taken into account when generating a new mission plan. Also, the system stores the available (predefined or auto-generated) mission plans and the driver program(s) for their execution. Last but not least, it keeps a log for each sensing task and corresponding mission execution, including its status and the data that was collected (and possibly processed) in this context. The different system components create/access this information via a shared repository, e.g., using a database for structured information in conjunction with a file system for unstructured data. This information can also be inspected and updated by authorized external systems and end-user applications via the Gateway.

3 Manager

The Manager oversees the different phases of a full monitoring cycle, in a similar way this would be done by a human operator. Namely, it invokes other components to perform the necessary actions and decides when to move the system to the next state of operation.

The internal organization of the Manager is shown in Fig. 2. It includes an RPC Server that is invoked by the Gateway when external requests arrive, an event-driven finite state machine (FSM) that implements the coordination of the operation cycle, and a set of observers that monitor different processes or subsystems and issue an event when certain conditions apply. These entities are independent threads running concurrently to each other. Sensing tasks are kept in a queue, sorted according to the scheduled start times. The events generated by the observers are kept in a separate FIFO queue, from where they are retrieved and handled by the FSM in a sequential manner. The operation of the FSM and the observers is described in more detail in the following subsections.

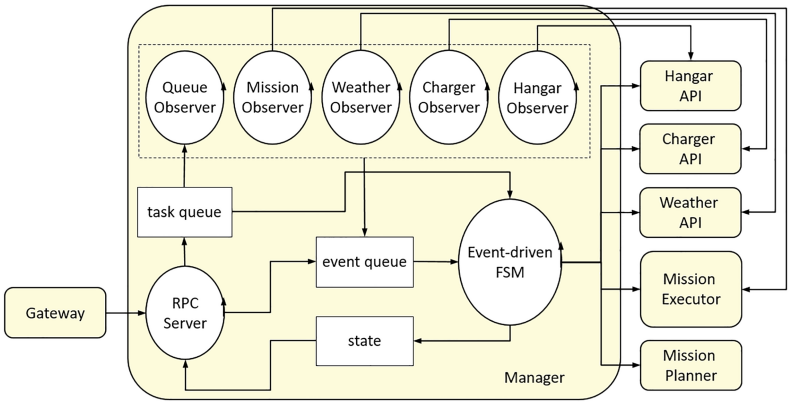


Fig. 2. Internal organization of the Manager.

3.1 Finite State Machine

The FSM runs in a loop. In each iteration, the next event is removed from the queue and the respective handler is invoked, which performs a state transition along with some actions. An event is dropped if it is not compatible with the current state or the last processed event was of the same type. If the queue is empty, the FSM blocks until an event is added to the queue.

The FSM has six states, representing the key phases of the operation cycle. The state diagram and the respective transitions are given in Fig. 3. The events that trigger the state transitions are indicated by the capitalized labels over the edges while the sources of each event are given in brackets. Figure 4 shows the core logic of the event handlers, where the dashed arrows denote the logical flow between handler invocations resulting as a side-effect of event generation.

Initially, the Manager is in the READY state where the drone is fully charged and can be used for the next sensing task. A new cycle is triggered via the START event, generated by the Queue Observer when it is time to start the next sensing task. As a result, the Manager invokes the Mission Planner to generate the mission plan for the task at hand, if such a plan is not already available. It also confirms, via the Weather API, that the weather conditions are good (else the task is postponed for a later point in time and the state remains READY). Then, the hangar is instructed to open its hatch via a call to the respective API. Since this operation may take some time, it is tracked by the Hangar Observer while the system moves in the OPENING state. When the hangar opens, the Hangar Observer generates an OPENED event. As a result, the Manager invokes the Mission Executor to start the execution of the mission plan, and moves to the RUNNING state.

Once the mission completes and the drone has properly landed, the Mission Observer generates the DONE event. The Manager then checks the mission status, and if the mission was aborted it may reschedule the remainder of the task (discussed in more detail in the sequel). In any case, the Manager instructs

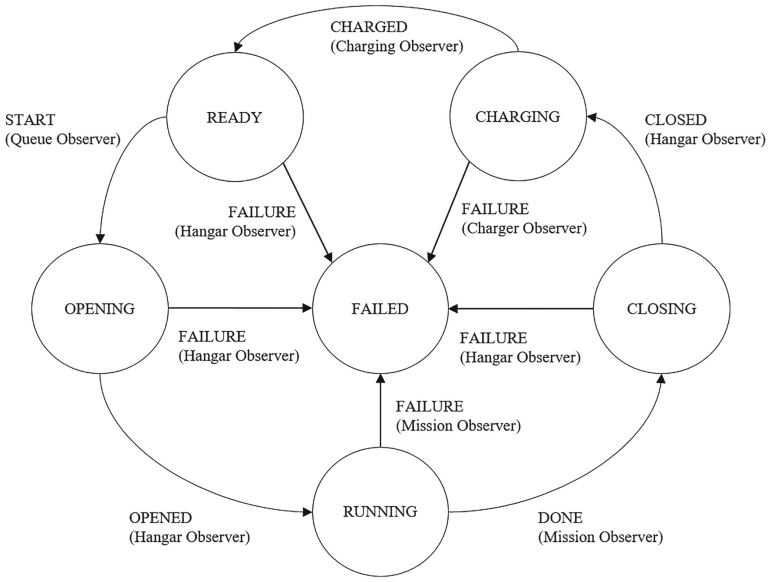


Fig. 3. State diagram of the FSM.

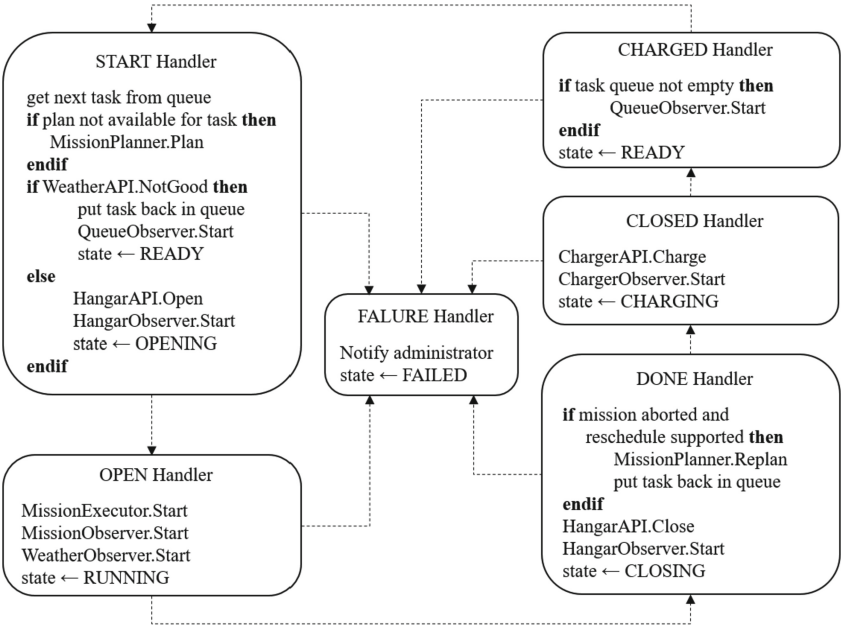


Fig. 4. Event handling logic of the FSM.

the hangar to close its hatch and moves to the CLOSING state, and when the Hangar Observer generates the CLOSED event the Manager starts charging the drone's batteries via the Charger API and enters the CHARGING state. Finally, when the Charger Observer generates a CHARGED event, the Manager starts the Queue Observer, if the task queue is not empty, and makes the transition back to the READY state from where the system can repeat the same procedure for the next sensing task. If there are no pending scheduled tasks, the system remains idle until a new task is submitted via the Gateway, in which case the RPC Server adds it in the queue and starts the Queue Observer.

During the above cycle of operation, some hard failures may occur which cannot be handled by the Manager in a graceful manner so that the system can continue its operation. More specifically, in the OPENING state, the hangar may not open; in the RUNNING state, the drone may not respond to critical commands or fail to land properly; in the CLOSING state, the hangar may not close; also, in the CHARGING state, the charger may not be able to properly charge the drone's batteries. In all these cases, a FAILURE event is generated by the respective observer. In turn, the Manager notifies the administrator and makes the transition to the FAILED state. The system remains in this state until it is reset manually; in some cases, it may be possible for the administrator to inspect and correct the problem remotely, via the Gateway, while other problems may require physical intervention. We note that a mission abort is not considered a hard failure. Even if a sensing task is stopped prematurely, it may be perfectly possible to reschedule it or proceed with the next task in the queue.

3.2 Observers

The observers are started on a need-to basis by the event handlers of the FSM, as indicated in Fig. 4. An observer is automatically stopped when it generates an event or the FSM makes a state transition.

The Queue Observer is responsible for triggering the execution of the next sensing task. To this end, it checks the task queue, determines when the next task needs to start and sets a timer accordingly in order to issue the START event. If a task is postponed or rescheduled, it is put back at the head of the queue so that its execution starts as early as possible.

The Hangar Observer uses the Hangar API to receive information about the position of its door/hatch. When this is fully opened or closed, it generates the OPENED and CLOSED event, respectively. If a problem occurs when trying to open/close the hatch, the FAILURE event is generated.

The Charger Observer uses the Charger API to check the progress of battery charging. When the drone's batteries are fully charged, it issues the CHARGED event. If charging does not work properly, it issues the FAILURE event.

The Weather Observer uses the Weather API to receive information about the weather conditions from the local station. If weather conditions deteriorate while a mission is already in progress, the Mission Executor is invoked to abort the mission. Note that the Weather Observer does not abort the mission itself nor

does it generate an FSM event; if the Mission Executor indeed aborts the mission, the corresponding DONE event will be generated by the Mission Observer, discussed next.

The Mission Observer periodically polls the Mission Executor in order to receive information about the mission execution status. If the mission was terminated in a graceful manner, either because it was completed or because it was aborted, the Mission Observer generates the DONE event. If the mission status indicates abnormal termination due to a hard problem, the observer generates a FAILURE event.

3.3 Task Rescheduling and Data Processing

As was mentioned above, if a sensing task is aborted, the Manager can reschedule it so that it is completed as planned. To this end, the driver program needs to record in the system repository additional information regarding the degree of mission completion. In turn, the Manager can pass this information to the Mission Planner in order to produce a new plan for the remaining part of the task. Furthermore, to reduce the number of take-offs and landings that are performed by the drone, the Mission Planner can be asked to combine the rescheduled task with the next one in the queue, provided both have the same type. Note, however, that it may not be possible to support such an automated rescheduling and replanning for all sensing tasks and mission types. If an aborted task cannot be rescheduled, it keeps this status so that the administrator can take further action as needed, e.g., by manually submitting a suitable follow-up sensing task.

The data that is collected by the drone is stored by the driver program in the system repository from where it can be inspected/retrieved via the Gateway. Moreover, in some application scenarios, this data may need to be processed before making it available to external systems. This can be performed through an independent process running in parallel to the core mission control loop, in order to retrieve the data from the repository, invoke the corresponding data processing services via the Processing API, and store back the results. We do not elaborate about this aspect here, as this is does not affect the core operation of the system. Finally, we note that the mission itself can be data-driven, in which case some data processing will be performed by the driver program, without involving the Manager; we discuss this aspect at the end of the next section.

3.4 System Failures

As mentioned above, if any subsystem fails (Hangar, Battery Charger, Mission Executor), the Manager will enter the FAILED state, and the user (system administrator) will be notified in order to take action. If the Manager itself fails (e.g., the host machine crashes), this will be detected by the user when attempting to interact with it through the Gateway. Upon restart after an abrupt crash, the Manager enters the FAILED state and the user is notified as usual.

Of course, a system failure may also occur during the execution of a sensing task. In this case, the wireless connection between the Manager and the drone will

break, and the drone will return and land autonomously, without requiring any external guidance. In principle, the Manager can resume such tasks, practically in the same way this is done for tasks that are aborted: by checking the logs and invoking the Mission Planner to produce a new plan for the part of the task that was not completed due to the system crash. In this particular case, however, we adopt a more conservative approach: the Manager does not attempt to automatically resume operation after a failure, and it has to be reset manually.

Finally, we note that it is possible to make the Manager fault-tolerant by adopting replication techniques, such as [16] or [24], so that system operation continues smoothly as long as at least one of the replicas remains operational. This discussion, however, is beyond the scope of this paper.

4 Mission Executor

The Mission Executor component is responsible for actually running a sensing task according to its specific mission plan, while communicating with the drone as needed. The interaction with the Manager takes places through an RPC Server, while the core functionality of mission execution is implemented in a separate environment that is invoked by the RPC Server, via a suitable front-end that hides technology-specific details. In our current prototype, the mission execution environment is implemented using the TeCoLa platform [23], as shown in Fig. 5. TeCoLa supports a flexible, service-oriented interaction with the drone in Python, and can work over different wireless technologies, including WiFi and long-range RF.

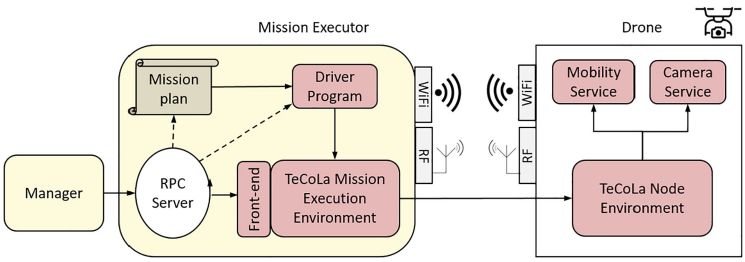


Fig. 5. Mission execution approach.

In a nutshell, the Mission Executor takes as input a mission plan and the driver program that will parse and execute the plan by invoking the drone's service as needed. The driver program runs on top of the Mission Execution Environment, which takes care of the underlying communication with the drone over a wireless link. The drone, in turn, runs the Node Environment, which processes the requests of the Mission Execution Environment, calls the corresponding local services and sends back the replies. In the basic system configuration used to test our system, the drone provides two services: the mobility service for the

navigation-related functions of the autopilot, and the camera service for taking pictures via the onboard camera.

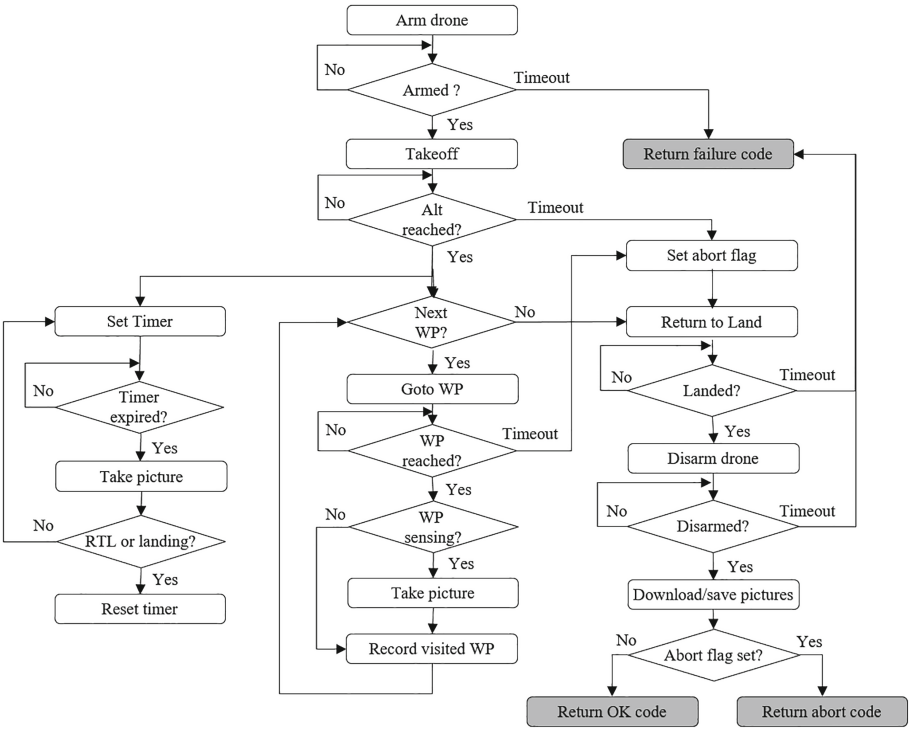


Fig. 6. Main logic of a driver program for taking aerial images over a target area following a waypoint-based mission plan.

As an example of a driver program, Fig. 6 illustrates the main logic of a program that is designed to execute waypoint-based mission plans while using the camera as a sensor. Pictures are taken periodically during the entire mission as the drone moves between the waypoints, as well as at certain waypoints specified in the plan. Initially, the program arms the drone and instructs it to take-off. It also starts a timer in order to perform periodic sensing, shown in the figure as an independent thread. Concurrently, in a loop, the program instructs the drone to move to the next waypoint. If a focused sensing action is required at that point, the drone is instructed to take a picture before moving to the next waypoint. When the drone successfully visits all waypoints, it is instructed to return and land. Finally, the drone is disarmed, the recorded pictures are downloaded from the drone and saved in the system repository, and the program returns a success code, leading to the generation of the DONE event by the Mission Observer.

If the drone does not reach the take-off altitude or a waypoint within a reasonable amount of time, this is considered as an indication of very localized

adverse weather conditions that are not detectable by the weather station. To be safe, the driver program decides to abort the mission, and instructs the drone to return to the hangar and land immediately. Similar checks can be performed for other things, such as the drone's battery level in order to abort the mission if this drops below a safety margin (not shown in Fig. 6 to avoid clutter). In any case, the fact that the plan was not carried out to full completion is reflected in the return code of the program and the mission completion status reported by the Mission Executor to the Manager, so that the latter can identify and reschedule aborted tasks. Also, in the above example, the driver program explicitly records each waypoint that is successfully visited by the drone. This information can be passed (along with the original plan) to the Mission Planner to produce a plan for the rest of the sensing task. Recall that even if the mission completion status indicates an abort rather than full completion, the Mission Observer will still generate a DONE event as the system can continue its operation.

If the drone fails to arm/disarm or does not land properly, is not possible for the system to continue operation without manual inspection and perhaps even hands-on maintenance. In these cases, the driver program terminates immediately with a failure code, which leads to the generation of a FAILURE event by the Mission Observer. Several additional safety checks, e.g., for smooth motor operation and lack of vibrations, can be performed along the same lines, in order to put the system in the FAILED state whenever human intervention is required (to focus on the essence, the example in Fig. 6 does not include such checks).

At any point in time during execution, the Manager may ask the Mission Executor to pause, continue or abort the mission, due to a corresponding external request received via the Gateway. Also, the Weather Observer may request an abort if it detects deteriorating weather conditions. In these cases, the Mission Executor up-calls special handlers of the driver program, which take the necessary actions. More specifically, on pause, the last command that was sent to the drone is recorded and the drone is instructed to hover in its current position, while the driver program is suspended. When requested to continue execution, the last command is re-sent to the drone and the driver program is resumed. Finally, a requested abort leads to the same behavior as when the driver program takes this decision by itself. The event handling logic of the driver program is not shown in Fig. 6 for brevity.

We wish to stress that the format of the mission plans as well as the driver program used for their execution can both be customized according to the application's requirements. It is also straightforward to introduce additional drone sensor (and actuator) services, provided the drone features the corresponding hardware. Furthermore, the system can employ different types of missions and corresponding driver programs, which may rely on platforms other than TeCoLa; one merely needs to a add a corresponding mission execution environment and front-end. This flexibility is important for sensing applications where there is no single size that fits all needs. In particular, it is possible to support dynamic data-driven missions. This can be done using a suitable driver program, which receives data from the drone during the mission (e.g., at specific waypoints),

processes this data and guides the drone based on the results. To minimize delays, heavyweight data transfers between the drone and the driver program on the ground station would need to be performed over sufficiently fast wireless links. The driver program can perform the required data processing by invoking in a direct way the appropriate services via the Processing API, without involving the Manager. Depending on the system configuration and available computing resources, the data processing services can run locally on the ground station or at a powerful edge infrastructure with good connectivity to the ground station.

5 Functional Testing

We have implemented a complete system prototype, where all the software components discussed in the previous sections are separate microservices, packaged as Docker containers in order to support a managed deployment on the ground station. The mission plans used to test our implementation are waypoint-based, and the logic of the driver program that executes such plans is similar to that in Fig. 6, with several additional checks that may lead to an abort or failure depending on the gravity of the detected issue. We have performed a wide range of tests using a custom drone and ground-station but also a suitable simulation environment. We describe the two setups in more detail below.

5.1 Field Setup

The setup used to perform tests in the field is shown in Fig. 7. The drone is a custom hexacopter with a CUAV V5 nano autopilot board [10], which provides all flight-related sensors and runs the popular ArduPilot [1] autopilot software for multicopters. The drone features as a separate companion board a Raspberry Pi 3 Model B [8] (RPi) with a quad-core ARM Cortex A53 processor at 1.2 GHz and 1 GB of RAM, running the Debian-based Raspberry Pi OS Buster, the officially supported Linux distribution for the RPi. The RPi is connected to the autopilot board over serial and runs the TeCoLa Node Environment. The mobility service employs the DroneKit library [3], which communicates with the autopilot through MAVProxy [6]. The camera service accesses an onboard 8 MP RPi Camera Module v2 [9] via the picamera library [7].

For the ground station we use an Intel NUC mini PC running a Ubuntu 16.0 LTS Linux distribution, powered via an external car battery. Since the field tests focus on the operation of the core system components, this configuration does not include a weather station, a hangar or battery charger subsystem. Instead, the Weather API is hardwired to report good weather conditions, the Hangar API produces open/close events right after receiving the corresponding commands, and the battery charging status reported by the Charger API is set manually when the drone is equipped with fresh batteries. Also, to have full control on the missions that are performed in the field, we do not use a Mission Planner component. All tests are based on manually prepared mission plans and no rescheduling takes place in case of an abort.

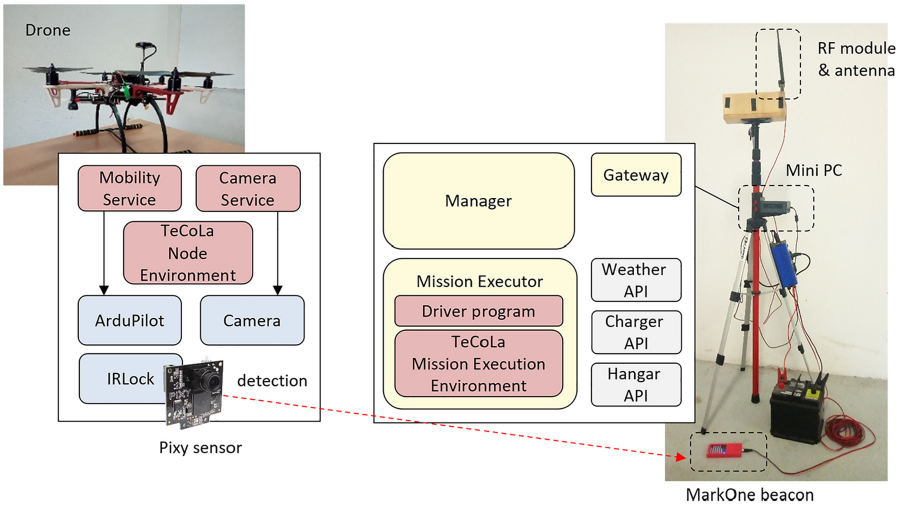


Fig. 7. System setup used in the field tests.

Precision landing is supported in the same way as this would be the case in a real deployment. More specifically, we use the IRLock mechanism [5], which is already supported by the off-the-shelf ArduPilot distribution. IRLock relies on the onboard Pixy camera sensor to track an IR beacon. To achieve robust tracking, we use the MarkOne beacon, which can be detected from a height of up to 15 m. The beacon is powered by the same external battery as the mini PC, using a long cable so that it can be placed at the desired landing position.

The communication between the ground station and the drone can be configured to go over WiFi or RF telemetry. Both the drone and the embedded computer have corresponding interfaces; the latter has an integrated WiFi antenna, while the RF module is externally connected to it via USB and is mounted on a pole for better coverage. The system can be configured to use only one of these interfaces, or both of them concurrently with the option of service-based binding. For instance, the RF which has a longer range can be used for the more critical mobility service, whereas the WiFi which can support higher throughput can be used for the camera service.

While field trials are important to verify the desired system operation under real conditions, they also come with several limitations, primarily due to safety rules and regulations in drone operation. Field trials are also very time consuming since they require extensive preparation and travel. For these reasons, a large number of more complex experiments are performed using a simulated setup described in the sequel.

5.2 Simulated Setup

To thoroughly test all system functions and several corner cases in a flexible and controlled way, we use a modular setup, which integrates different simulation technologies through virtual machine (VM) and container technology in order to support experimentation with different virtual unmanned vehicles and virtual ground stations. In this particular setup, we employ the Gazebo simulator [4], which is popular in the robotics community and has built-in support for the simulation of polycopters, and ns3 [27] for the simulation of wireless communication. The setup is illustrated in Fig. 8 and is described in more detail below.

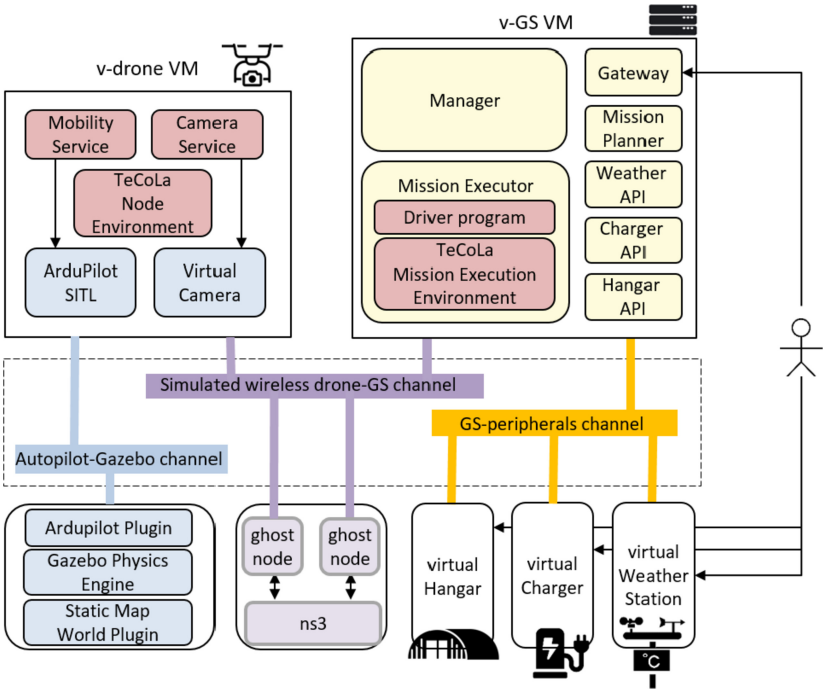


Fig. 8. System setup using the Gazebo simulator and ns3.

The drone functions are implemented within a virtual machine (v-drone VM), mimicking a drone configuration where the flight controller and the companion computer coexist on the same board. The autopilot software and the TeCoLa Node Environment with the mobility and the camera services run as containers within the v-drone VM. In this setup, we employ the official software-in-the-loop (SITL) configuration of Ardupilot [2], coupled to the physics engine of Gazebo that models the behavior of a typical quadcopter with high fidelity. Gazebo itself runs in a separate container, outside the v-drone VM, and communicates with the SITL ArduPilot via the Ardupilot plugin over a dedicated communication

channel of the simulation environment. In addition, Gazebo is configured to use the static map world plugin, which provides a ground plane with satellite imagery used to feed real images to a virtual camera. The mobility service in the v-drone VM communicates with the autopilot via Dronekit, in the same way this is done in the real drone. The camera service is modified to access a virtual camera container, which communicates with Gazebo to return images based on the drone's current position in the virtual world.

The ground station is also a virtual machine (v-GS VM) that includes the entire management system. The communication between the v-drone and the v-GS occurs over a virtual WiFi network. This is simulated using ns3, which is configured to operate according to the 802.11b standard, and is accessed by the VMs via proper (virtual) wireless interfaces that map into corresponding so-called ghost nodes in ns3. We note that the WiFi simulation also takes into account the distance between the ground station and the drone. The position of the ground station is fixed at startup, while the position of the drone is updated at runtime based on the reports of the autopilot that are sent to ns3 via a side channel (not shown in the figure to avoid clutter).

In this configuration, we use a simple Mission Planner component, which takes as input a sequence of waypoints and produces a plan in the format that is expected by the driver program. The plans are produced taking into account the drone's operational autonomy (battery capacity vs energy needed for the drone to fly at a certain speed). If a sensing task is too large, the planner will produce several smaller plans, which the Manager schedules in sequence via the task queue. The planner can also combine smaller plans into larger ones. This feature is used when the Manager reschedules a task after an abort, in order to merge it with the next one in the task queue (if any).

For the weather station, the hangar and the battery charger, we use mockups which run as independent containers and are accessed by the corresponding API components over a separate communication channel. These mockups can be pre-configured as well as interactively controlled by the user during the simulation to behave according to the test scenario. More specifically, the weather station can report different conditions, while the hangar can open/close its hatch and the charger appear to charge the drone's battery with a configurable delay or exhibit a malfunction.

5.3 User Interface

In our prototype, all system entities can be accessed and monitored via command-line tools and different logs. Furthermore, as an indicative interactive user interface, we have developed a simple GUI, shown in Fig. 9.

Through this GUI, the user can start, pause, continue and abort missions as well as view the current drone position on top of a google map along with some basic status information. There is also a panel, next to the map view and above the control buttons, for displaying the most recent photo taken by the drone (as soon as this is stored in the system repository).

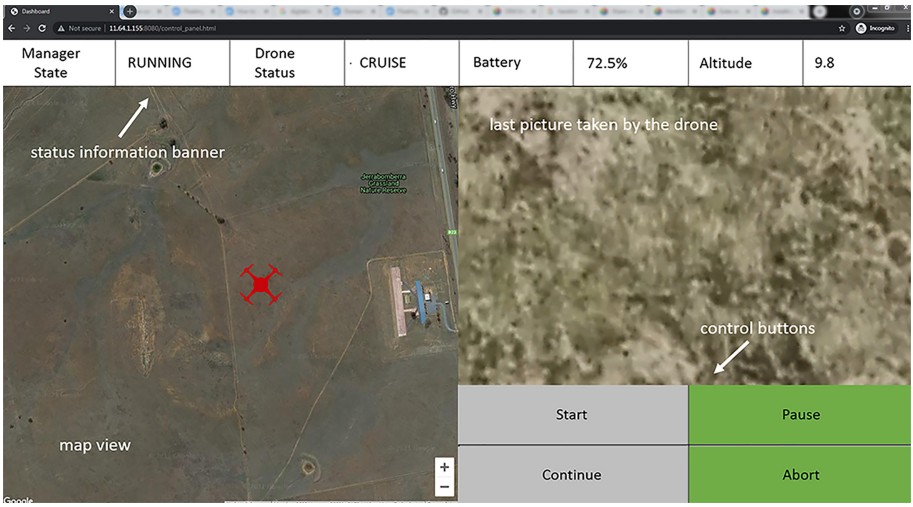


Fig. 9. Snapshot of user interface during system operation.

We note that the GUI is designed as an independent (web-based) application, which communicates with our system via the Gateway. It can be used to interact with the real ground station in field tests as well as with the virtual ground station when performing simulated tests.

5.4 Indicative Test Scenario

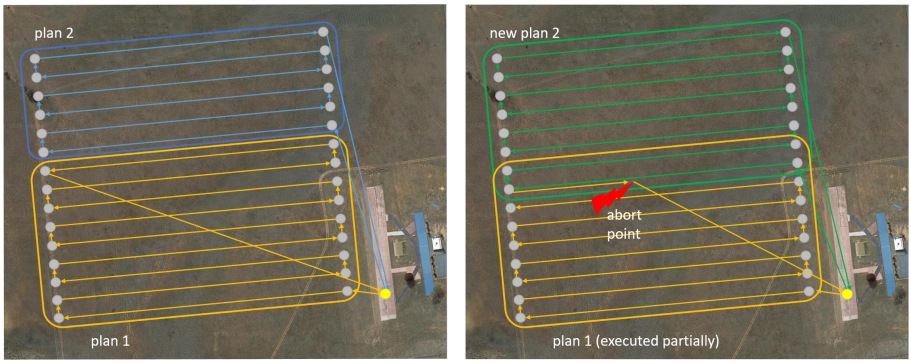
Using the simulated setup, we have tested our implementation extensively for a wide range of scenarios regarding different cases of user intervention (mission pause, continue and abort), dynamically changing weather conditions, failures of the hangar/chargers and disconnections between the ground station and the drone. To give an indicative example, assume the administrator has added a periodic sensing task for scanning a target area of 300×300 m through several passes at an altitude of 10 m. The desired path of the task is defined via a sequence of waypoints with a "horizontal" distance of 300 m and a "vertical" distance of 20 m, depicted in Fig. 10a by the gray bullets.

The total flight distance that needs to be covered is about 5 Km without taking into account take-off and landing, at a specified flying speed of 4 m/s. However, at that speed, the drone has an autonomy of only about 3.5 Km. As a consequence, this task is mapped to two smaller mission plans, illustrated in Fig. 10a in orange and blue color, respectively. Since this is a periodic task, these plans are produced offline and are stored in the system repository from where they are retrieved by the Manager when the task is started.

During the execution of the first plan (orange), the weather station is set to report bad conditions. As a result, the mission is aborted at the point marked in Fig. 10b and the drone returns back to base and lands. The remaining part

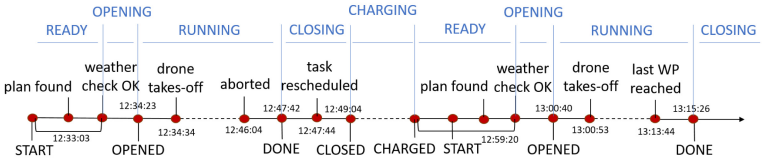
of the first plan is rescheduled and merged with the second plan (blue), thereby producing a new ad-hoc plan (green) that is executed as soon as the drone's batteries are fully charged.

Figure 11a shows the sequence of system events that are generated for/during the execution of this sensing task, in line with what was described in Sect. 3. To

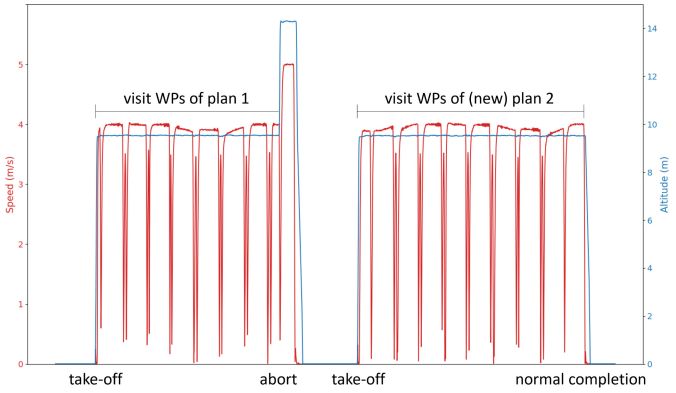


(a) Initial offline plans. (b) Modified ad-hoc second plan.

Fig. 10. Original and adapted plan due to abort. (Color figure online)



(a) Sequence of system events, actions and states (timeline not in scale).



(b) Flight speed and altitude of the drone.

Fig. 11. System events and drone flight data. (Color figure online)

display events that occur at widely different time scales, the timeline is not in scale but merely illustrates the logical sequence of events with their timestamps (given that granularity is in seconds, some events have the same timestamp; also, larger time periods are denoted by a dashed line). The state of the FSM at the various time intervals is indicated above the timeline (in blue). As can be seen by observing the event timestamps, the simulation environment is configured so that the time needed for the hangar to fully open/close its hatch is to 80 s, while the delay for charging the drone's batteries is set to 10 min (of course, the actual recharging time for the batteries of the real drone is much larger).

Figure 11b depicts the drone's speed and flight altitude relative to the home position, which are recorded during the experiment in the system logs (as done during real flight tests). Once the drone takes off and starts the mission according to the first plan, its altitude remains constant at about 10 m, while its speed is roughly 4 m/s except when the drone slows down in order to turn at the specified waypoints. When the sensing task is aborted, the mission driver program activates the return-to-land behavior. As a result, the drone immediately rises to a default altitude of 15 m and then moves with a default speed of 5 m/s to the home position where it lands. The same basic pattern is repeated during the execution of the second plan. The difference is that, in this case, the entire mission is completed successfully, thus the drone follows the planned path back to home without changing the specified flight altitude or speed.

6 Related Work

There is a growing body of work on supporting the development of autonomous drone-based monitoring and tracking applications. Below, we briefly discuss indicative efforts. A modular system for object-tracking applications is presented in [28], consisting of a drone with an onboard computer, a camera and positioning sensors like GPS. The camera images and sensor data are processed on the onboard computer in order to detect the target object and infer the drone's position. The results fed in a mission planning module, also running on the onboard unit, which sends suitable control commands to the drone's flight controller so as to follow and approach the target object. [18] describes a drone-based system for structural health monitoring. The drone is equipped with a camera and an onboard computer that processes the camera images to detect special AR tags, which are used to mark the desired points (structural elements) of interest. It also features a separate stereo vision subsystem with two special (CCD point-grey) cameras. The drone flies autonomously to the points of interest where it takes stereoscopic images. After the flight, these images are transferred to a high-performance computer in order to perform the required data processing. The goal of the system presented in [29] is to pursuit intruder drones based on external alerts. When an intruder drone enters the area, another drone is used to detect, track and jam the intruder. The drone is equipped with a camera, an onboard unit that processes the camera images in order to detect the intruder drone and guide the drone by sending signals to the flight controller, and a software-defined

radio used to jam the intruder as well as for self localization. The authors in [11] propose a system for precision agriculture using UAVs and an edge server. The photos taken by the UAVs are processed on an edge server to extract relevant features for the crop in question, and to decide if more samples are needed, in which case the next flight plan for the UAVs is produced. The above efforts focus on the autonomous operation and flight control of the drone, in some cases with the help of edge computing infrastructure. Our work is complementary to such efforts, with the goal to automate the full operation cycle of drone-based sensing systems, including storage, battery-charging, weather checking, mission planning, mission execution and data processing, without any human intervention. The proposed design can be flexibly customized to support different specific application scenarios, including the ones outline above.

There are several efforts on supporting the coordinated usage of multiple robots or drones. TeCoLa [23] abstracts autonomous robots as nodes providing different services that can be accessed remotely via method calls with at-most-once semantics. Mobility is modelled as a special service. The application program can transparently discover and invoke the available nodes as needed, while there is also support for the dynamic formation and grouped invocation of smaller teams. Resh [12] is a domain-specific language and orchestration system for multi-robot systems. Each robot has an agent that communicates with a centralized runtime in order to advertise its capabilities/status and translate the Resh protocol to the local robot-specific API. Dolphin [26] also follows a centralized approach for task execution using a static team of autonomous vehicles, where the platform's runtime system is responsible for polling the vehicles, sending tasks to them and managing platform-specific operations. In [13], the ScaFi programming language is used to let a multi-agent system achieve a collective goal in a distributed fashion. An iterative protocol is followed, whereby each agent evaluates its local state, checks for any messages coming from its neighbors, runs an aggregation program based on the current context, performs corresponding actions, updates its state and notifies its neighbors. Maple-Swarm [21] allows the developer to submit partial high-level plans. These plans are used as input to generate specific tasks for one or more robots, based on their capabilities and the current state of the system. Our work is largely orthogonal to the problem of coordinated task execution. In terms of design, the proposed system architecture can be extended to support sensing tasks with more than one drones. In a nutshell, the Manager would need to handle multiple hangar and battery charging subsystems in order to keep track of the drones that are ready to be used for sensing tasks, the Planner would need to produce individual mission plans while ensuring the necessary safety boundaries between the drones that will be used at the same time, and the driver program would have to be written accordingly in order to execute these mission plans in a coordinated way. Our current implementation uses TeCoLa for mission execution, which already provides support for team-level operations that could be used to support sensing tasks with multiple drones. However, in principle, any other platform could be used to write the driver program for executing such sensing tasks.

Some researchers investigate the remote usage of UAVs following cloud-based approaches. In [32], UAVs appear as cloud services managed by a central coordinator, which takes care of the underlying communication and maintains the sessions between UAVs and their users. [14] proposes a cloud service through which users can interactively request missions using an Android application. The service plans the path, picks the best available UAV and sends the mission to the UAV's autopilot, while the user can monitor the mission by receiving live status data and video streaming. Dronemap Planner [22] virtualizes UAVs by offering suitable control and data retrieval web services, which run in the cloud and can be invoked by third-party applications via standard web-service protocols. All these approaches are definitely interesting if ones wishes for UAVs to become a shared resource that can be used by different users possibly even at the same time. Our work goes in the opposite direction, focusing on the automation of remote sensing systems that employ their own (possibly highly customized) drones and ground infrastructure in an exclusive way. Also, both control and data processing take place at the edge, making it possible for the system to operate with full autonomy even when having poor connectivity with the cloud. If needed, the proposed system can also work in the cloud, using a 5G link for the communication with the drone, provided the latency and bandwidth are sufficient for the application in question.

Our work has some common goals with the concept of autonomic computing [19]. The benefits of an abstract architectural approach for autonomic computing systems are stressed in [25], where the authors motivate their model inspired by the domain of robotics. Most work in this area of research focuses on resource/service monitoring and provisioning in cloud environments so as to maintain QoS and satisfy SLA requirements [15,17,30], while several self-* properties have also been studied in the context of flying ad-hoc networks [20,31]. In contrast, the approach proposed in this paper tackles the self-management of drone-based remote sensing systems, by monitoring key parameters of system operation and taking corrective or preventive actions as needed.

7 Conclusion

We have presented a complete architecture for edge-based remote sensing systems that rely on drones. The system is designed to support the full cycle of operation in an automated way, without requiring good connectivity to the cloud or any human intervention (except when serious, non-recoverable failures occur). We have implemented all software components of the proposed architecture and have tested their functionality using both a real drone and ground station as well as a suitable simulation environment. Thanks to its modular design, the system can be flexibly customized according to application requirements, in terms of both the drone and the mission types/driver programs that will be used to support the desired sensing tasks. Also, data processing can be performed during the execution or after the completion of a sensing task, as needed.

As a next step, we intend to investigate the support of sensing tasks via multiple drones in more depth. Apart from writing suitable driver programs

that perform the necessary coordination, which is already feasible in our current prototype, an additional challenge is to address the management of multiple hangars and battery chargers as well as to perform a dynamic replanning when a drone, hangar or battery charger fails so that the task is performed by the drones that remain fully operational.

References

1. ArduPilot autopilot. http://ardupilot.org
2. ArduPilot SITL. http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html
3. DroneKit. http://dronekit.io/
4. Gazebo simulator. http://gazebosim.org/
5. IRLock tracking system. https://irlock.com
6. MAVProxy software. http://ardupilot.github.io/MAVProxy/html/index.html
7. Picamera interface. https://github.com/waveform80/picamera
8. Raspberry Pi 3. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/
9. Raspberry Pi camera. https://www.raspberrypi.org/products/camera-module-v2/
10. V5 Nano. http://doc.cuav.net/flight-controller/v5-autopilot/en/v5-nano.html
11. Boubin, J., Chumley, J., Stewart, C., Khanal, S.: Autonomic computing challenges in fully autonomous precision agriculture. In: IEEE International Conference on Autonomic Computing (ICAC), pp. 11–17 (2019)
12. Carroll, M., Namjoshi, K.S., Segall, I.: The Resh programming language for multirobot orchestration. arXiv preprint arXiv:2103.13921 (2021)
13. Casadei, R., Aguzzi, G., Viroli, M.: A programming approach to collective autonomy. J. Sens. Actuator Netw. **10**(2), 27 (2021)
14. Ermacora, G., Rosa, S., Toma, A.: Fly4smartcity: a cloud robotics service for smart city applications. J. Ambient Intell. Smart Environ. **8**(3), 347–358 (2016)
15. Ferrer, A.J., Becker, S., Schmidt, F., Thamsen, L., Kao, O.: Towards a cognitive compute continuum: an architecture for ad-hoc self-managed swarms. arXiv preprint arXiv:2103.06026 (2021)
16. Gerkey, B., Mataric, M.: Pusher-watcher: an approach to fault-tolerant tightly-coupled robot coordination. In: IEEE International Conference on Robotics and Automation, vol. 1, pp. 464–469 (2002)
17. Gill, S.S., Chana, I., Singh, M., Buyya, R.: Radar: self-configuring and self-healing in resource management for enhancing quality of cloud services. Concurrency Comput. Pract. Experience **31**(1), e4834 (2019)
18. Kalaitzakis, M., Kattil, S.R., Vitzilaios, N., Rizos, D., Sutton, M.: Dynamic structural health monitoring using a DIC-enabled drone. In: International Conference on Unmanned Aircraft Systems (ICUAS), pp. 321–327 (2019)
19. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)
20. Kim, G.H., Nam, J.C., Mahmud, I., Cho, Y.Z.: Multi-drone control and network self-recovery for flying ad hoc networks. In: International Conference on Ubiquitous and Future Networks (ICUFN), pp. 148–150 (2016)
21. Kosak, O., Huhn, L., Bohn, F., Wanninger, C., Hoffmann, A., Reif, W.: Maple-swarm: programming collective behavior for ensembles by extending HTN-planning. In: International Symposium on Leveraging Applications of Formal Methods, pp. 507–524 (2020)

22. Koubâa, A., et al.: Dronemap planner: a service-oriented cloud-based management system for the internet-of-drones. Ad Hoc Netw. **86**(1), 46–62 (2019)
23. Koutsoubelias, M., Lalis, S.: TeCoLa: a programming framework for dynamic and heterogeneous robotic teams. In: International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous), pp. 115–124 (2016)
24. Koutsoubelias, M., Lalis, S.: Fault-tolerance support for mobile robotic applications. In: IEEE International Symposium on Industrial Embedded Systems (SIES), pp. 1–10 (2018)
25. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: Future of Software Engineering (FOSE), pp. 259–268 (2007)
26. Lima, K., Marques, E.R., Pinto, J., Sousa, J.B.: Dolphin: a task orchestration language for autonomous vehicle networks. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 603–610 (2018)
27. Riley, G.F., Henderson, T.R.: The ns-3 network simulator. In: Wehrle, K., Güneş, M., Gross, J. (eds.) Modeling and Tools for Network Simulation, pp. 15–34. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12331-3_2
28. Smyczyński, P., Starzec, Ł., Granosik, G.: Autonomous drone control system for object tracking: flexible system design with implementation example. In: IEEE International Conference on Methods and Models in Automation and Robotics (MMAR), pp. 734–738 (2017)
29. Souli, N., et al.: Horizonblock: implementation of an autonomous counter-drone system. In: International Conference on Unmanned Aircraft Systems (ICUAS), pp. 398–404 (2020)
30. Toffetti, G., Brunner, S., Blöchlinger, M., Dudouet, F., Edmonds, A.: An architecture for self-managing microservices. In: International Workshop on Automated Incident Management in Cloud, pp. 19–24 (2015)
31. Yang, T., Foh, C.H., Heliot, F., Leow, C.Y., Chatzimisios, P.: Self-organization drone-based unmanned aerial vehicles (UAV) networks. In: IEEE International Conference on Communications (ICC), pp. 1–6 (2019)
32. Yapp, J., Seker, R., Babiceanu, R.: UAV as a service: enabling on-demand access and on-the-fly re-tasking of multi-tenant UAVs using cloud services. In: IEEE/AIAA Digital Avionics Systems Conference (DASC), pp. 1–8 (2016)