UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Edge Computing for Drones

Diploma Thesis

Thodoris Kasidakis

Supervisor: Spyros Lalis

Volos 2021

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Edge Computing for Drones

Diploma Thesis

Thodoris Kasidakis

Supervisor: Spyros Lalis

Volos 2021

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Υπολογισμοί στην άκρη του δικτύου για Εναέρια Μη Επανδρωμένα Οχήματα

Διπλωματική Εργασία

Θοδωρής Κασιδάκης

Επιβλέπων: Σπύρος Λάλης

Βόλος 2021

Approved by the Examination Committee:

Supervisor	**Spyros Lalis** Professor, Department of Electrical and Computer Engineering, University of Thessaly
Member	**Dimitrios Katsaros** Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly
Member	**Christos Antonopoulos** Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Date of approval: 16-6-2021

Acknowledgements

As this long endeavor has reached its end, I would like to express my deep gratitude to a group of people who provided my with guidance, assistance, support, and encouragement.

First and foremost, I would like to deeply thank my supervisor, Prof. Spyros Lalis who have provided me with guidance, motivation and support throughout the years of my studies in University of Thessaly. His patience and excellent feedback were pivotal for the completion of this work. It was an honor working with him. What I learned from him will accompany me throughout my life. I would also like to show my warmest appreciation to Prof. Dimitrios Katsaros for his help and support through the last years of my studies. Our collaboration was something more than significant for me. I would like to express my greatest acknowledgment to Prof. Christos Antonopoulos for the valuable knowledge who provided me through his courses. I am also grateful to them for being members of the examination committee of my thesis.

I would also like to thank Manos Koutsoubelias who was by my side all this time, giving me valuable advices and support for the completion of my work. My sincere thanks to Nasos Grigoropoulos for his excellent help towards the creation of the simulation environment that was used in this work. I would not have accomplished it without him. I want to thank also Giorgos Polychronis for the help he gave me whenever I needed it.

It is impossible not to mention Akis, Christos and Apostolis. We have been together on this great journey since the beginning. We have common joys, sorrows, laughter and cries that I will never forget. Thank you for all these.

I am grateful to Marios for what we have achieved together. From the first day until now.

I want to thank Nikos, Fotis, Dimitris, Hraklis and Vasilis for the friendship of those years.

Last but not least, I want to thank my family, my sisters Mariarena and Dimitra, my parents Charilaos and Dafni. I have no words to express my gratitude for them, their support, patience and unconditional love for all my decisions. This thesis is dedicated to them.

This work has been co-financed 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 (contract T1EDK 02435)

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

The declarant

Thodoris Kasidakis
29-6-2021

Abstract

In data-driven applications, which go beyond simple data collection, drones may need to process sensor measurements at certain locations, during the mission. However, the onboard computing platforms typically have strong resource limitations, which may lead to significant delays and extended mission times. To address this problem, we explore the potential of offloading heavyweight computations from the drone to a nearby computing infrastructure. We discuss a concrete implementation for a service-oriented application software stack, which takes offloading decisions based on the expected service invocation times and the locations of the servers expected to be available in the mission area. We evaluate our implementation using an experimental setup that combines a hardware-in-the-loop and software-in-the-loop configuration, as well as via a suitable simulation environment. Our results show that the proposed approach can reduce the total mission time significantly, by up to 48% vs local-only processing, and by 10% vs more naive opportunistic offloading, depending on the mission scenario.

Περίληψη

Σε εφαρμογές που βασίζονται σε δεδομένα, οι οποίες υπερβαίνουν την απλή συλλογή, τα drones μπορεί να χρειαστεί να επεξεργαστούν μετρήσεις αισθητήρων σε συγκεκριμένες τοποθεσίες, κατά τη διάρκεια της αποστολής. Ωστόσο, οι ενσωματωμένες πλατφόρμες έχουν συνήθως ισχυρούς περιορισμούς πόρων, οι οποίοι μπορεί να οδηγήσουν σε σημαντικές καθυστερήσεις και παρατεταμένους χρόνους αποστολής. Για την αντιμετώπιση αυτού του προβλήματος, διερευνούμε τη δυνατότητα εκφόρτωσης υπολογισμών από το drone σε μια κοντινή υποδομή υπολογιστών. Συζητάμε την υλοποίηση μιας συγκεκριμένης στοίβας λογισμικού, η οποία λαμβάνει αποφάσεις εκφόρτωσης βάσει των αναμενόμενων χρόνων επίκλησης μιας συγκεκριμένης υπηρεσίας και της τοποθεσίας των servers που αναμένεται να είναι διαθέσιμοι στην περιοχή αποστολής. Αξιολογούμε την εφαρμογή μας χρησιμοποιώντας τόσο μια πειραματική εγκατάσταση που συνδυάζει μια διαμόρφωση υλικού-σε-βρόγχο και λογισμικού-σε-βρόγχο όσο και ένα περιβάλλον προσομοίωσης. Τα αποτελέσματά μας δείχνουν ότι η προτεινόμενη προσέγγιση μπορεί να μειώσει σημαντικά το χρόνο αποστολής, έως και 48%, σε σύγκριση με την τοπική επεξεργασία και 10% σε σύγκριση με την απλή ευκαιριακή εκφόρτωση, ανάλογα με το σενάριο αποστολής.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Motivation

Thanks to the rapid developments in control systems and embedded systems, aerial unmanned vehicles (UAVs), also referred to as drones, are becoming a key component of the cyber-physical computing landscape. As costs have dropped in the last years, drones have become affordable for a large number of organizations or even individuals with a small budget. As a result, drones are now being used in an increasing number of civilian applications, such as agriculture [1] and environmental monitoring [2] and rescue operations [3]. A very popular type of drone for a wide range of applications are polycopters, such as quadcopters and hexacopters. They are easy to fly, can hover above a specific position, can perform vertical maneuvers and can take-off/land virtually anywhere.

However, such drones have limited autonomy, the usual flight time being 20-30 minutes. Also, to keep the cost, weight and power consumption low, the onboard hardware platforms are typically embedded systems with limited computing resources. As a consequence, heavyweight computations can introduce significant delays, especially if these cannot be overlapped with the navigation tasks of the drone. In turn, this increases the total mission time and reduces the area that can be targeted without changing/switching batteries.

In this thesis, we tackle this problem by letting time- consuming computations be offloaded to servers that are located in the mission area, in a flexible and educated manner. This is achieved by adopting a service-oriented architecture in conjunction with a mechanism for offloading service invocations in a transparent way for the mission program running on the drone. The decision to use a server for offloading is taken at runtime, based on performance-related estimations and the location of the servers in the mission area. We have implemented the proposed approach on a real drone platform. To avoid the limitations of field experiments, we study the behavior of our implementation using an experimental setup that combines a hardware-in-the-loop (HITL) and software-in-the-loop (SITL) configuration. We also perform experiments using a simulation environment. The results show that our approach can significantly reduce the mission time. These time gains can, in turn, be exploited to target larger areas or scan them at a finer grain without having to switch batteries.

1.2 Contribution

The main contributions of this thesis are:

- We present a mechanism for offloading service invocations to remote servers in the context of a real application software stack for drones.
- We propose a policy for taking offloading decisions at runtime in an educated manner.
- We provide an evaluation based on a HITL/SITL setup and a simulation environment that are used to get realistic measurements.
- We show that the proposed approach can lead to significant gains.

1.3 Thesis Structure

The rest of the thesis is structured as follows. Chapter 2 introduces the system model. Chapter 3 describes a concrete implementation of the proposed system on top of a popular application software stack for drones. Chapter 4 presents the evaluation of the implementation. Chapter 5 gives an overview of related work. Finally, Chapter 6 concludes the thesis and points to directions for future work.

Chapter 2

System Model

The system we consider in our work consists of a drone that is used to visit a set of waypoints where it performs some sensing, processes the sensor measurements and based on the results notifies the user, and possibly also performs some actuation. For instance, a drone can scan a crop field in order to detect areas that have been attacked by some pest, in which case it sprays some pesticide. Another example is a drone tha monitors a certain industrial plant, such as a solar park or wind park, in order to detect faulty/damaged solar panels and blades, respectively. As one more application scenario, a drone can fly over certain main roads in a city to monitor traffic. There are many more applications that follow such sense-process-decide-notify/actuate loop. Of course, in the general case, an application may use multiple drones at the same time. This is largely orthogonal to the problem we address here, so we focus on a single drone.

Besides its sensors and actuators, the drone also features some general-purpose computing platform that can be used to perform computations locally. However, due to cost, weight and power restrictions, the local platform is limited compared to a typical ground infrastructure. As a consequence, heavyweight processing may still lead to significant delays during the mission.

As an add-on to this system, we assume one or more servers that may be located near the mission area. These servers can be part of a larger fog infrastructure, or in the more extreme case standalone computing-resource boxes (possibly powered using renewable energy, such as solar panels, wind turbines) that can be used in an ad-hoc fashion. In the spirit of edge computing, the drone can exploit such servers to offload some processing tasks so that these are executed faster to compared to a local execution. This way, the mission can be reduced and it becomes possible to target larger areas.

Without loss of generality, we assume that the existence and locations of such servers is known at mission design time. Thus, it is possible to ship mission-specific code to them before the mission starts. There are already very mature software packaging and deployment technologies for this, such as virtual machines or containers. Such a pro-active code shipment is important so that the potentially large image transfer delay does not have an impact on the mission. However, even if the code is already available on the server, some time is still required in order to load and initialize it. This has to be done during the mission, depending on the path that will be followed by the drone.

Of course, the fact that the servers are known when the mission is designed and software can be transferred to them before the mission starts, does not provide any guarantees regarding their availability during the mission. For instance, a server might be loaded and thus not be able to process a request coming from the drone. The server can also be down due to maintenance or a system/power failure. Thus, to be autonomous, the drone cannot rely exclusively on these servers being available during the entire mission. In the worst case, it should be able to perform the required processing locally.

Finally, we assume that each server is accessible through a dedicated wirelles network, which is used exclusively for the purpose of offloading. The network ID and any credentials that are needed to connect to the server's network are pre-loaded on the drone before the mission starts.

Chapter 3

Implementation

We start by describing the mission execution environment and how offloading is supported in this context. Then, we discuss in more detail the server management aspect and how offloading decisions are taken.

3.1 Software architecture

We have designed our mission execution environment following a service-oriented approach [4]. More specifically, the drone's sensing, actuation, navigation and data processing capabilities are exported to the application developer as first-class services with well-defined interfaces. The mission logic is a proper Python program, which invokes the drone's services to retrieve sensor data and to process it so that it can then take decisions and actions according to the mission objectives.

The calls to the autopilot are always executed locally via the Dronekit software [5]. The rest of the service invocations are captured by an intermediate layer, which calls the corresponding services and returns back the result/reply to the mission program. The fact that a service call can be performed remotely is transparent to the mission program. Internally, service calls are done through Pyro [6].

The main components and information objects of our implementation are shown in Figure 3.1. The server selection component decides whether to use one of the servers that are available in the mission area for offloading. In this case, the service check component proceeds to confirm the server's availability. A connection is made to the network of the server and a request is sent to check for the services the drone wishes to use. If the server has the

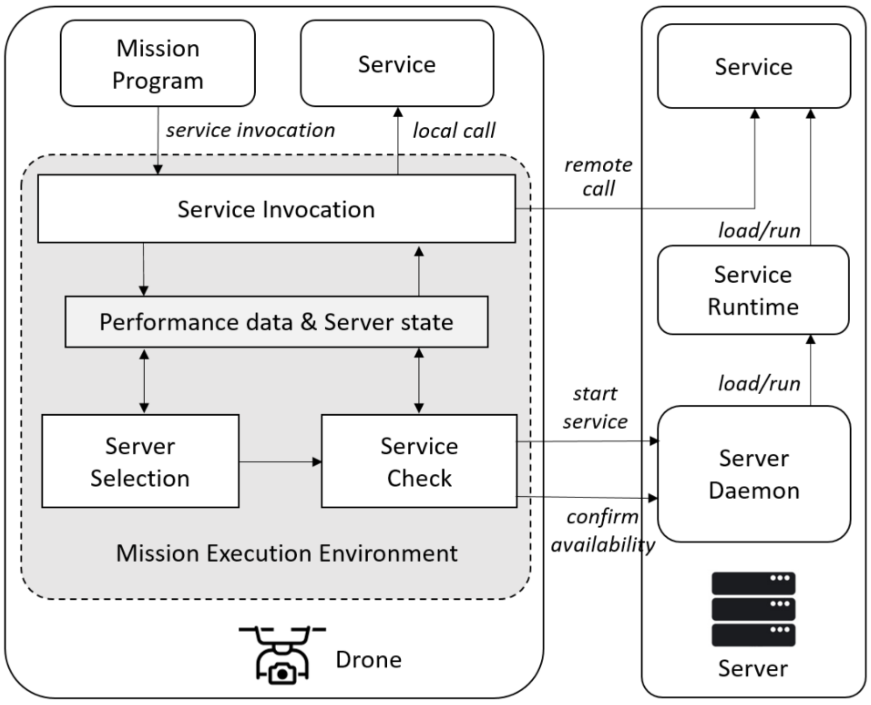

Figure 3.1: Software Architecture.

required software images and sufficient free capacity to serve the drone, it sends a positive reply, which includes the endpoints of the services that are already running (if any). Else, the server sends a negative reply.

If the desired service is not running on the server, a start request is sent. The server then loads/initializes the service and replies with the respective endpoint information, which is then passed on to the service invocation component. In case the service was already running on the server, the endpoint that was received in the first phase is used instead. If at a later point it is decided not to use the server, e.g., because a better option is found, the remote endpoint is updated accordingly.

The service invocation component intercepts the invocations of the mission program and calls the corresponding service. A remote call is made if a valid remote endpoint is available for the service in question. Else, the local service is called as usual. The delay of each call is recorded and is taken into account to re-evaluate the server selection / offloading decisions in the future.

On the server side, a daemon handles the interaction with the drone. When an availability confirmation request is received, it replies with the names of the services for which the software image is locally available and the endpoints for the ones that are already running. When a start request is received, the daemon asks the service runtime environment to load and start

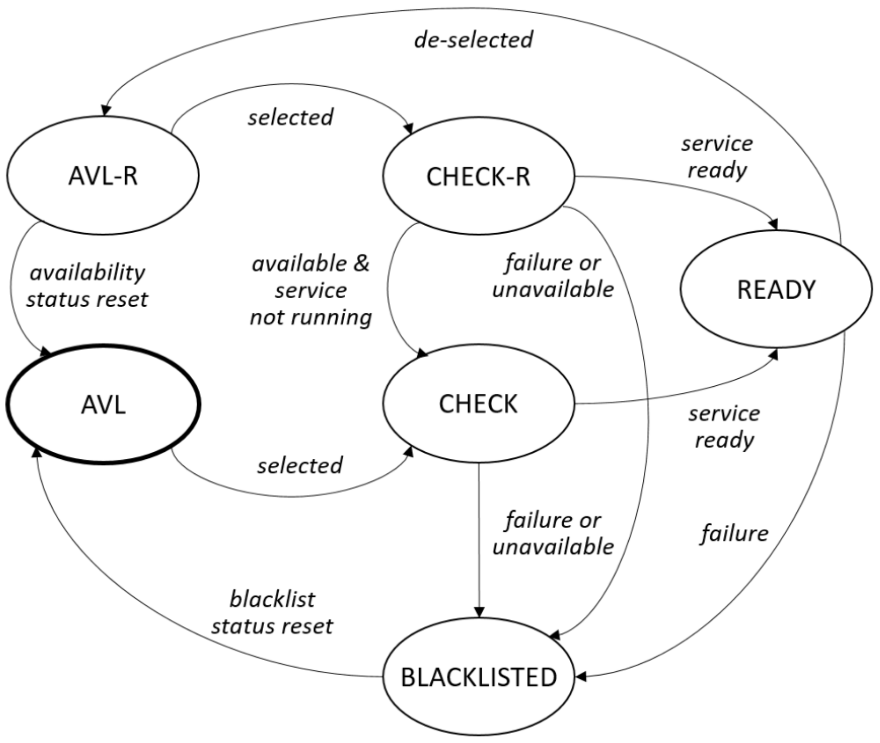

Figure 3.2: Server state diagram.

running the service in question. We use the LXD environment [7] for this, with each service being properly packaged as a Linux container. We assume that the services are stateless. Thus, one can switch between local and remote invocations without needing to checkpoint and transfer state information between the drone and the server. Many compute-intensive data processing functions fall under this category.

To focus on the essence of the mechanism, in the following, we assume a single service being considered for offloading. However, our implementation supports offloading for multiple services each having different performance characteristics.

3.2 State management

For each server in the mission area, an entry is kept with information about its expected performance, location, network ID, access credentials and the communication range of the wireless technology. Based on this information, function $inrange(s)$ indicates whether the current position of the drone is sufficiently close to servers to be within its range.

Also, each entry has a state that is consulted during server selection and usage. The states and transitions are shown in Figure 3.2. The -R flag in the AVL and CHECK states captures the assumption that the service is already running on the server. This determines the estimate for the service availability check delay (see Section 3.4), since the extra time for starting-up a service can be non-negligible. Initially, all servers are AVL as it is assumed that they are available for offloading but the service still needs to be loaded/activated on them.

When an AVL server is selected for offloading (see Section 3.3), its state is set to CHECK. Similarly, if the selected server is AVL-R, its state changes to CHECK-R. In both cases, the time of server selection is recorded in t_{check}. At this point, if another server was previously selected for offloading and is in the READY state, its state is set to AVL-R and the service check component disconnects from the server's network.

Subsequently, a connection is made to the network of the selected server (in CHECK or CHECK-R state) to confirm its availability and receive the service endpoint. If the server is available and the service is already running, its state is set to READY and can be used for offloading. If the server confirms its availability but the service is not already running there, its state is set to CHECK and the service is started via a follow-up request to the server. When the server replies with the corresponding endpoint, its state is set to READY.

If it is not possible to connect to the server's network, or the server does not respond, or it is unavailable, its state is set to BLACKLISTED so that it is not considered for offloading. The procedure is repeated until the next best selected server confirms its availability and the endpoint for the service is successfully received, or all candidates are blacklisted.

At most one server can be selected at any point in time (be in the CHECK, CHECK-R or READY state). The current selection is re-evaluated periodically (the period is configurable). For instance, the drone may come in range of new servers that may be better candidates for offloading, or go out of range of the currently selected server. Also, the remote invocation delay may turn out worse than expected, rendering the server unattractive for offloading. In these cases, the server is de-selected and its state reverts to AVL-R. Note that the server selection and the (potentially time-consuming) service check procedure are performed in the background, concurrently to the execution of the mission program.

Finally, BLACKLISTED servers are reset to AVL after a configurable period of time. A similar transition occurs from AVL-R to AVL for servers that have not been selected over along period of time, assuming that the service running there has most likely been de-activated/unloaded in the meantime.

Table 3.1: Key parameters of the server selection policy.

Symbol	Description
$Call$	Local end-to-end service call delay.
$Data$	Size of service request and reply data that need to be sent over the wireless networks.
$Proc_s$	Remote service processing time on server s.
$Call_s$	Remote end-to-end service call delay on server s, including data transfer over the wireless network.
$Check_s$	Connection & availability confirmation delay when the service needs to be started on s.
$CheckR_s$	Connection & availability confirmation delay when the service is already running on s.

3.3 Server selection policy

The server selection and offloading decision is based on a few basic parameters, listed in Table 3.1. Initially, $Call$ is obtained through offline profiling on the drone platform. The amount of data $Data$ that has to be sent over the network if the service is called remotely is obtained in a similar way. An initial value for $Proc_s$ is obtained by conducting offline tests on each server platforms. Then, $Call_s$ can be initially estimated as follows:

$$Call_s \approx Proc_s + Data/Bandwidth_s \tag{3.1}$$

where $Bandwidth_s$ is the nominal bandwidth of the server's wireless network. Finally, $Checks$ and $CheckRs$ are determined via suitable offline tests as well. Note that these include the time for connecting to the network of the server.

Once the mission starts, $Call$ and $Calls$ are updated online, after each local and remote service call, respectively. In this case, $Call_s$ is measured directly and does not need to be approximated via Equation 3.1. Similarly, $Checks$ and $CheckRs$ are updated each time the server's availability is checked. The updates are performed using a moving average based on the last 3 interactions; in the beginning, the missing values are filled with the corresponding offline estimates.

A server s is a candidate for offloading if it is in range of the drone, the estimated remote service call delay is smaller than the local call delay, and it is not blacklisted:

$$candidate(s) = inrange(s) \land Call_s < Call \land s.state \neq \text{BLACKLISTED} \tag{3.2}$$

If several candidates exist, the best is the one with the smallest estimated service call delay:

$$best(s) = candidate(s) \land \nexists s' : candidate(s') \land Call_{s'} < Call_s \tag{3.3}$$

The server that is currently selected for offloading, let cur, is de-selected in favor of a better candidate s only if this is expected to lead to a non-negligible gain relative to the local call delay:

$$select(s) = best(s) \land \frac{Call_{cur} - Call_s}{Call} > Gain_{switch} \tag{3.4}$$

This is done to avoid switching between servers that have more or less the same performance. The $Gain_{switch}$ threshold is flexibly configurable. Note that, based on Equation 3.2, the currently used server is de-selected even in the absence of a better candidate, if the remote call delay grows larger than that of the local call or the drone moves out of the server's range.

3.4 Service invocation

The server selection and service check process runs in the background, asynchronously to the execution of the mission program. For this reason, additional checks are performed by the invocation component when the mission program invokes the service, before making a call to the selected server cur.

First, it is checked whether cur is READY. If so and $inrange(cur)$ holds, the remote service is called via the corresponding endpoint, else the server is de-selected, its state changes to AVL-R and the local service is called instead.

If the selected server cur is in the CHECK or CHECK-R state, the amount of time where the server should confirm service availability and become READY is estimated as $waitT = Check_{cur} - (getTime() - t_{check})$ and $waitT = CheckR_{cur} - (getTime() - t_{check})$, respectively. If $waiT > 0$, it is checked whether:

$$\frac{Call - (Call_{cur} + waitT)}{Call} > Gain_{wait} \tag{3.5}$$

If the condition does not hold, the local service is called immediately. Else, the invocation component waits for the server to become READY and then calls the remote service. The rationale is that this waiting time will be immediately amortized due to the much faster remote vs local service invocation. It is also possible for $waitT \leq 0$, if the service availability check has already exceeded the expected delay. In this case, it is decided to wait for up to X, where X is the maximum value of $waitT$ so that Equation 3.5 is satisfied.

Note that the background service check may fail and the currently selected server may be BLACKLISTED. Also, no suitable server may be selected for offloading in the first place. In both cases, the service invocation component will simply call the local service on the drone.

Chapter 4

Evaluation

While our software stack has been successfully tested on areal drone, for practical reasons, we use a lab-based setup that combines a hardware-in-the-loop (HITL) with a software-in-the-loop (SITL) approach. This allows us to explore scenarios that are hard or even impossible to test in the field, mainly due to the strict flight and safety restrictions for urban areas. Still, the measurements obtained using this setup are realistic. In order to experiment in the future with multiple servers and multiple drones that use our software stack in a controlled and flexible way, we create a simulation environment. To test the reliability of our simulation environment, we perform the same experiments as the HITL/SITL setup. The measurements showed that the simulation environment provides trustworthy results.

4.1 HITL/SITL setup

As a typical drone platform, we pick that of a custom polycopter we have in our lab. The drone is controlled by the popular open-source ArduPilot autopilot [8], which runs on a dedicated CUAV nano V5 board. In addition, the drone has a companion board for hosting application software (Figure 4.1). This is a Raspberry Pi v.4 (RPi) with a 4-core Cortex-A72 ARM processor running at 1.5GHz and 2GB of memory. The RPi runs our application software stack, including Dronekit and the mission program, on top of an Ubuntu 18.04 distribution. The communication with the autopilot subsystem is done via the MavLink protocol [9] over a serial connection. For the server, we use a HP Pavilion Gaming Laptop 15-cx0xxx with a 12-core i7-8750H X84 processor at 2.20 GHz and 12 GB of memory. The laptop also runs Ubuntu 18.04, which provides support for LXD containers.

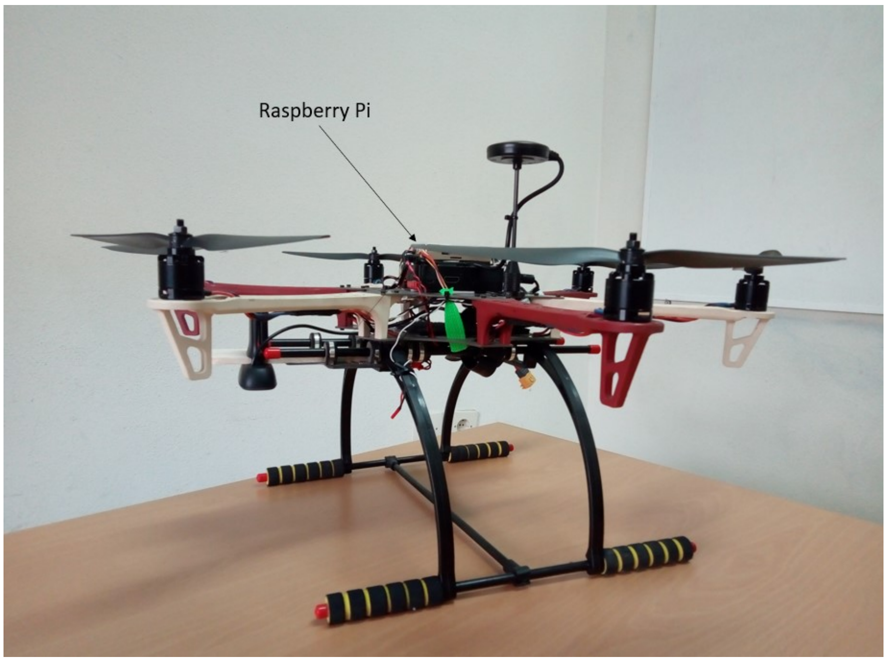

Figure 4.1: Custom polycopter of our lab with a Raspberry Pi as a companion board

To perform a wide range of experiments in a flexible and controlled way, we use a setup that faithfully simulates a real drone system through a combined HITL/SITL configuration, shown in Figure 4.2. The full application software stack (Dronekit and our mission execution environment with support for offloading) runs on a RPi that is identical to the companion board of the drone. For the autopilot, we use the official SITL configuration of ArduPilot, which runs on a standard Linux environment on a PC. The autopilot code is identical to the one running on the real drone, but is coupled to a physics model that simulates the dynamics and movement of the drone in the 3D space. In this setup, the MavLink-based interaction between the application software stack and the autopilot occurs over UDP/IP and an Ethernet link via a router.

The current position of the drone is communicated to the application software stack via MavLink, as usual. The positions, performance characteristics and communication range of the servers that are supposed to be available in the mission area are specified in a configuration file, which is pre-loaded on the RPi before the mission starts. The only difference compared to the native drone configuration is that the position of the drone is initialized and updated as part of the SITL operation (rather than being received from the GPS of the drone).

The communication between the RPi (drone) and the laptop (server) is done over WiFi, set to operate in ad-hoc mode with a nominal bandwidth of about 54Mbps. Server discovery/selection and remote service invocation runs as described in Chapter 3, as done in the

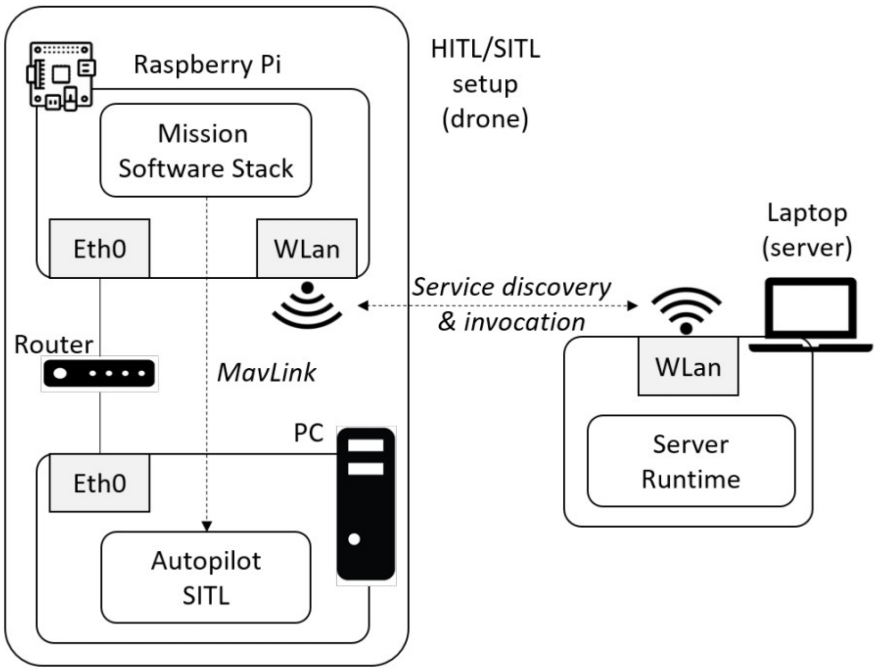

Figure 4.2: HITL/SITL setup.

configuration used for the real drone. The time needed for the RPi to perform a full discovery cycle, including network connection, initialization of the IP stack via the DHCP protocol and our service discovery protocol, is roughly 3.5 seconds.

To run experiments for scenarios with multiple (virtual) servers using the laptop, we modify the service discovery protocol so that the discovery request also contains the identifier of the target server (in addition to the identifier of the service in question). In turn, the laptop handles such requests independently for each server identifier, and can load the same service multiple times, once for each server. Although, in this particular setup, the RPi connects to, disconnects from and then re-connects each time to the same WiFi network (of the laptop), the availability check is performed exactly as if each server were accessible through its own private WiFi network, incurring each time the corresponding overhead.

4.2 Simulation setup

The design of our simulation environment is based on AeroLoop [10], a modular system for experimentation with virtual drones designed to run on off-the-shelf computing infrastructure. Each system entity, either it is a drone or it is a server, is packaged as a separate virtualized system (vDrone and vServer respectively). Instead of virtual machines (VMs)

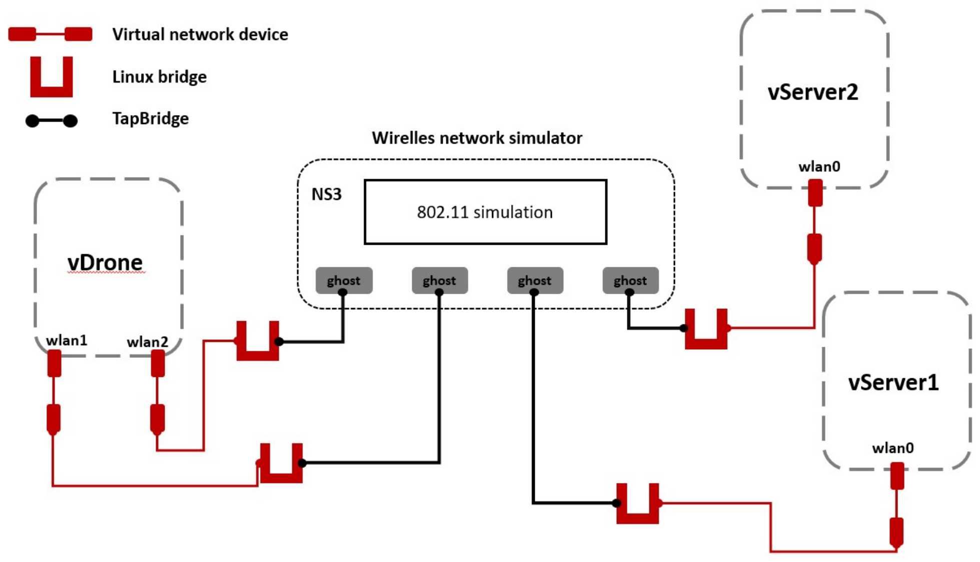

Figure 4.3: Simulation setup.

used in the AeroLoop system, in our case, we use Linux Containers (LXDs). LXDs offer a good compromise between isolation and resource efficiency, as they provide operating system level virtualization, which is more lightweight than full-fledged VMs, while offering a more complete virtual system (closer to a real drone or server) than Docker containers that share the same networking/storage stacks.

A vDrone consists of our full software stack and the ArduPilot, both run internally to a dedicated container. The interaction between the software stack and the autopilot is, once again, MavLink-based and occurs over UDP/IP and loopback. A vServer internally uses the LXD environment, with each service being packaged as a Linux container. To this end, for vServers running as LXDs we exploit the nested container functionality.

Wireless networking is implemented using ns-3 [11]. The setup is illustrated in Figure 4.3. For WiFi channels, each simulated ns-3 node, called ghost node, utilizes the ns-3 TapBridge device, which is connected to each vDrone/vServer through a combination of network bridges and virtual network devices (see [10]). Thus each vDrone communicates with each vServer through a dedicated, ad-hoc fashioned, Wi-Fi channel. The maximum available bandwidth is, roughly, 12Mbps. Each vDrone has as many virtual interfaces as the vServers in the mission area.

In order to simulate the connect/disconnect delays that would appear in real-world scenario, as part of the interaction of a drone with a server's Wi-Fi network, we introduce appropriate artificial delays, based to offline measurements. When a vDrone is going to connect

Algorithm 1 Mission program used in the experiments

```
Input: $WPList$                                        ▷ waypoints to visit
autopilot.arm()
autopilot.takeOff()
while $WPList \neq \emptyset$ do
    $wp \leftarrow$ getNxtWaypoint($WPList$)
    autopilot.goto($wp$)
    autopilot.waitToArrive($wp$)
    $pic \leftarrow$ camera.takePhoto()
    $objs \leftarrow$ detector.processPhoto($pic$)         ▷ may be offloaded
    if unexpected($objs$) then
        user.notify($objs$)
    end if
end while
autopilot.returnToHomeAndLand()
autopilot.disarm()
```

to a vServer, it uses vServer's identifier in order to obtain the IP from the appropriate virtual interface and starts the communication.

4.3 Mission template and services

On the drone, we run a mission program that visits a series of waypoints. At each way-point, a picture is taken that is processed to detect an unexpected or problematic situation, in which case an action needs to be taken, e.g., to notify the user who may wish a closer inspection. Note that the drone should not start moving towards the next waypoint before determining whether a problem exists at the current waypoint. Algorithm 1 shows the logic of the mission program.

Given that we run experiments using a static setup, the camera service on the RPi is configured to return static images from a directory in the local file system. As indicative input, we use pictures taken with a FLIR Duo R camera, which is a popular choice for drones (we also use this in our own drone). These pictures have a resolution of 1440 × 1080 pixels at a size of 400 to 450 KB. Each time the camera service is invoked from the mission program, it returns the next picture from the local directory. We add a delay of 1 second to account for the time that would be typically required to shoot a picture using such a camera.

The photos are processed by an object detection service, which internally employs an off-the-shelf version of YOLOv3 [12]. The service takes as parameters the image and the desired processing mode. The latter can take two values: light, in which case we run YOLOv3-320, and heavy where we run YOLOv3-608. The average service call delay on the RPi, measured offline, is about 3 and 10 seconds, respectively.

Since these delays are significant, the object detection service is a good candidate for offloading. To this end, the object detection service is packaged as a container that can run in the laptop (server) on top of LXD. The startup time, including the time needed to load the container and start running the service, is about 8 seconds. Note that this delay is not visible to the mission program as service startup is requested in the background before the selected server becomes ready for use. Once the service is running on the server, the average call delay, measured offline, is about 0.2 seconds for the light processing mode and 0.6 seconds for the heavy mode. Taking into account the time needed to send the call request and receive the reply over the WiFi network, the respective end-to-end remote call delays are initially estimated at roughly 0.3 and 0.7 seconds.

In the simulation setup, each vServer runs the object detection service as a (nested) container on top of LXD, with practically the same performance as in the HITL/SITL setup. Taking into account the maximum bandwidth supported by the network simulator, the end-to-end remote call delays are initially estimated to 0.6 seconds for the light processing mode and 1 second for the heavy mode. To properly simulate the call delay for local invocations, the object detection service running in the vDrone is configured with an extra artificial delay (where it simply sleeps) so that it exhibits the same response time as the service on the RPi.

Finally, in both setups, the $Gain_{switch}$ and $Gain_{wait}$ thresholds are set both to 25%.

4.4 HITL/SITL experiments

We have performed a wide range of experiments using the HITL/SITL setup. As an indicative example, here we discuss the experiments performed for an area of 200×200 square meters. The mission plan consists of 121 waypoints set every 20 meters in a grid-like pattern so that they cover the entire target area. Figure 4.4 illustrates the topology. The drone's plan

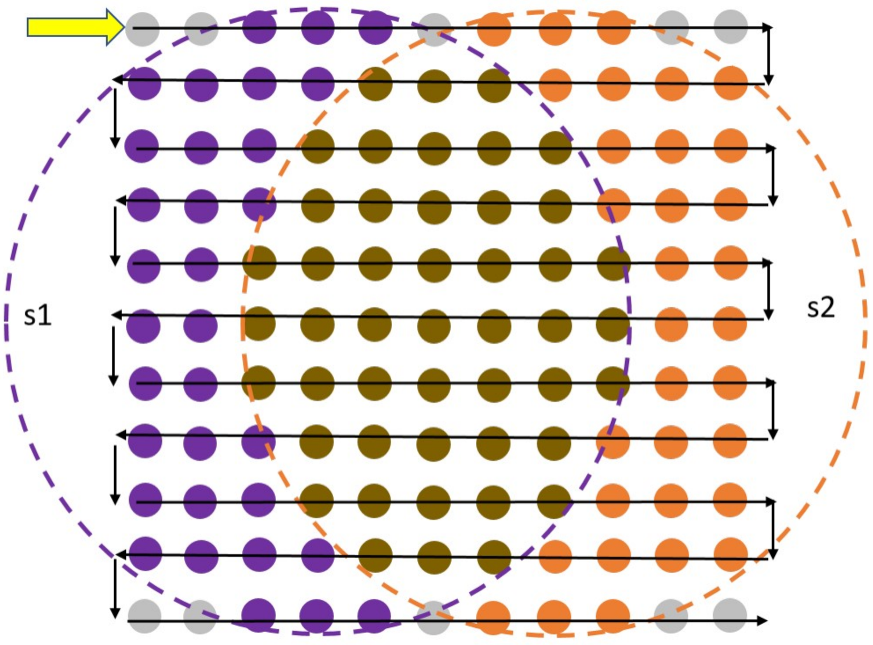

Figure 4.4: Mission waypoints and server coverage.

is to visit the waypoints starting from the top-left corner and moving row-wise from the top to the bottom of the area (as indicated by the black arrows). When moving between waypoints, the autopilot is instructed to fly at a speed of 4 m/s, which is the default for the field tests with our drone. Note, however, that as the drone moves away from the previous waypoint it needs some time to reach the target speed, and it also needs to slow down as it approaches the next waypoint.

We investigate a scenario with two servers in the mission area, s1 and s2, with a communication range of about 100 meters (based on tests we have performed in the field using our equipment). The dashed circles in Figure 4.4 indicate the respective coverage. The waypoints covered by both servers are brown. We run tests for both the light and heavy modes in the object detection service. As a reference, we use the default no-offload configuration where our mechanism is deactivated and all service calls are performed locally on the RPi.

4.4.1 Both servers at full capacity

In a first experiment, both servers provide the object detection service with the full performance. Figure 4.5 and Figure 4.6 shows the delay experienced by the mission program when invoking the object detection service in each waypoint in the light and heavy processing mode, when our offloading mechanism is enabled (blue) vs no-offloading (gray). The

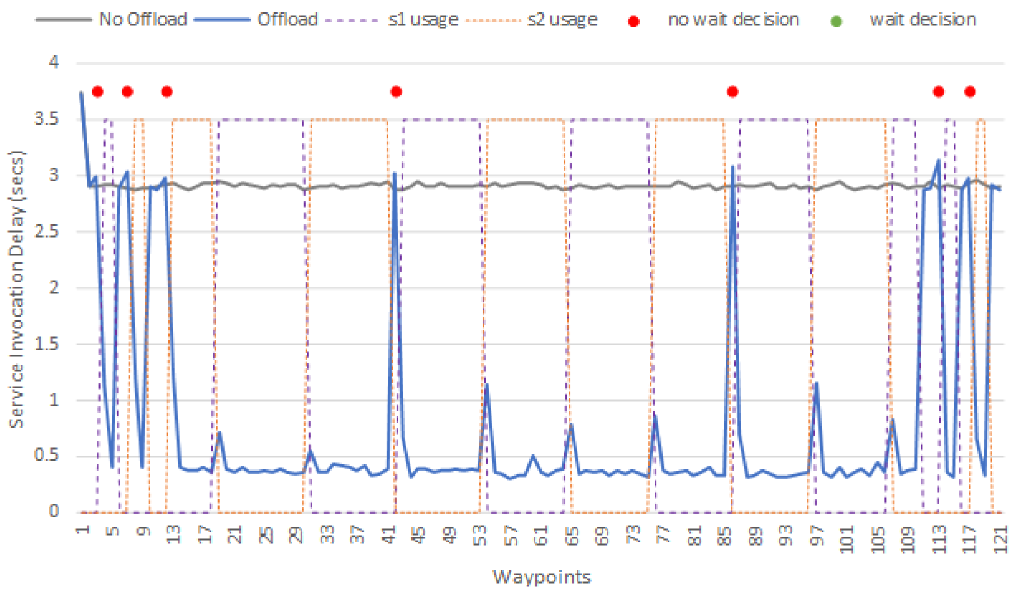

Figure 4.5: Call delay in the light processing mode, s2 is equally fast to s1 (HITL/SITL).

peaks correspond to waypoints where the local service is called, because no server has been selected or this is not yet ready. The areas under the dashed lines indicate when each server is being used for offloading, while the colored dots indicate the points where the mechanism decides to wait (green) or not (red) for the selected server to become ready.

Clearly, offloading achieves a very notable reduction of the service invocation delay vs. the no-offload configuration. In the light processing mode, improvement is about $3.7x$, yielding a mission time (including the travel time between waypoints) of 22.2 instead of 26.5 minutes, a reduction of 16%. The gains become even more significant for the heavy processing mode, where offloading reduces the total service invocation delay vs local-only invocation about $7x$, which shrinks the mission time by more than 48.5%, from 47.5 down to 24.4 minutes. This is less than the time needed for the no-offload configuration to perform the mission in the light processing mode. In fact, without offloading it would be impossible for a typical drone to complete the mission in the heavy processing mode without returning to base to change batteries.

In the light processing mode, the mechanism never decides to wait for the selected server to become ready, due to the relatively short call delay of the local service. In the heavy processing mode, this decision is taken several times during the mission, and, despite the extra waiting time, saves about 11.5% vs having called the local service at those waypoints.

Note that the first local service call, performed at the start of the mission, is slower than

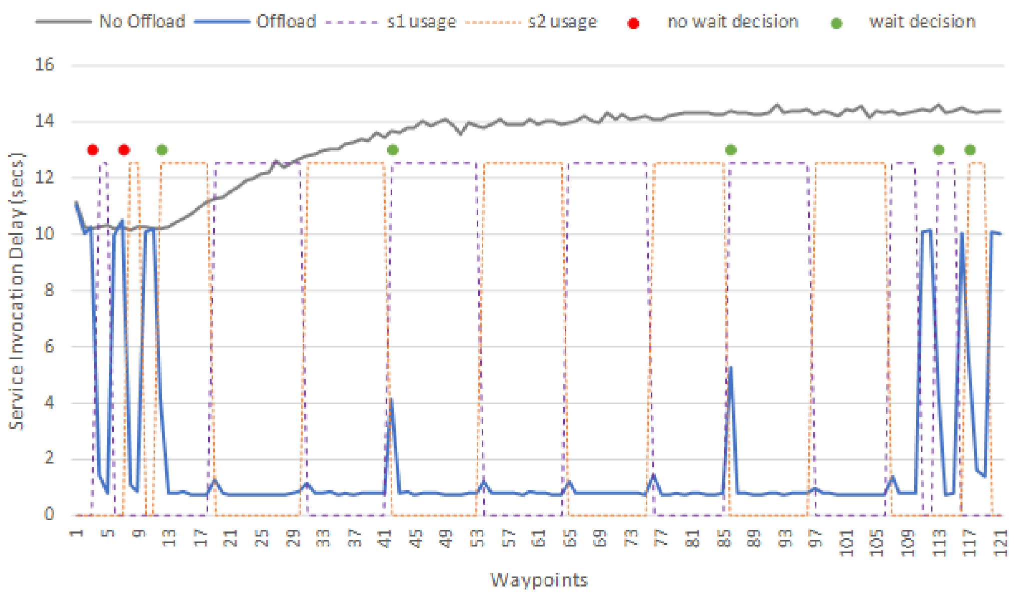

Figure 4.6: Call delay in the heavy processing mode, s2 is equally fast to s1 (HITL/SITL).

all subsequent invocations, due to a cold-start effect. There is a similar effect when calling a remote service that remained inactive for a while, due to time it takes to resume the inactive service container on the server. Also, in the light processing mode, the local calls that are performed concurrently to the attempt to connect to the network of the selected server, are slightly slower than in the no-offload configuration, due to the extra load. On the contrary, in the heavy processing mode, the no-offload configuration gradually exhibits a much larger service call delay, which stabilizes only after a large number of invocations. The reason is that processing in the heavy mode overloads the RPi and activates the frequency scaling mechanism to reduce CPU temperature. With offloading, local calls are performed at a much lower rate without any serious performance degradation.

4.4.2 One of the servers is loaded

In a second experiment, we slow down processing at s2 so that it performs worse than s1 (but still better than the local service), corresponding to a scenario where one of the two servers is loaded. Figure 4.7 and Figure 4.8 show the result for the light and heavy processing mode, where s2 processes incoming requests with an additional delay of 1 and 7 seconds, respectively.

The server usage pattern clearly shows that s1 is preferred over s2 in the waypoints covered by both servers. Also, due to the increased delay of s2, in several cases the mechanism

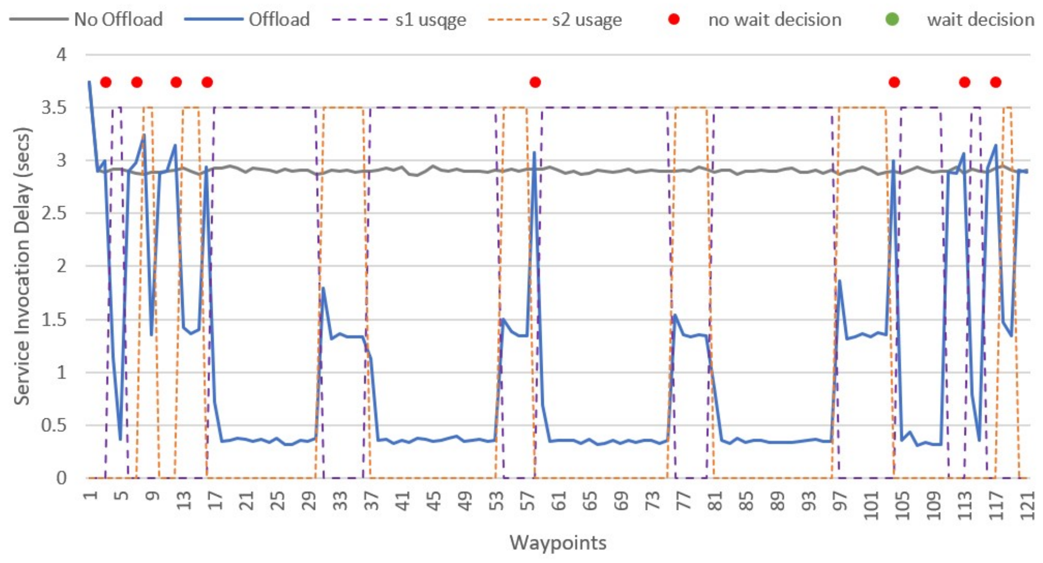

Figure 4.7: Call delay in the light processing mode, s2 is slower than s1 (HITL/SITL).

decides not to wait for s2 to become ready, and calls the local service, even in the heavy processing scenario. In the light processing scenarios, offloading reduces the total mission time to 22.7 instead of 26.5 minutes, by about 16%. In the heavy processing scenario, the reduction of the aggregated service invocation delay compared the no-offload configuration is about $3.6x$, leading to a total mission time of roughly 28 instead of 48.5 minutes.

To estimate the gain of this selection strategy vs a more naive approach that uses s2 with equal preference to s1 (as in Figure 4.6), we use the logs of the previous experiment and substitute for s2 the longer call delays recorded here. For the heavy processing mode, our calculations show that such a naive selection would increase the total service invocation delay by about 32.5%, which leads to an almost 10% longer mission time (extra 2.5 minutes).

Note that, for polycopter drones like the one in our lab, offloading cannot improve power consumption significantly. This is because the motors alone have a power consumption in the order of a few hundred watts, while the RPi consumes just a few watts even when loaded. However, offloading does lead to a significant reduction of the mission time thereby improving the effective operational capability of the drone.

4.4.3 Simulation experiments

We confirm the validity of the simulation setup, by performing the same experiments as in the HITL/SITL setup. To simulate the frequency scaling effects that occur on the RPi, we

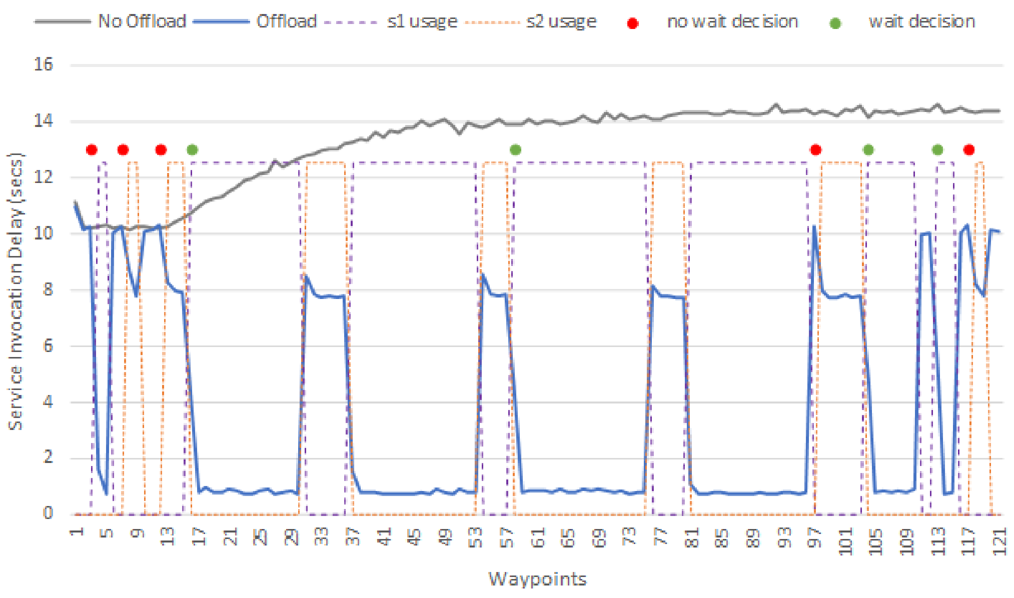

Figure 4.8: Call delay in the heavy processing mode, s2 is slower than s1 (HITL/SITL).

perform offline tests for the object detection service in heavy mode using different rates and apply polynomial curve fitting to the results. The approximate polynomials obtained this way are then used in the object detection service running in the vDrone to calculate the additional artificial processing delay to be applied as a function of the invocation rate.

The results for the case where both servers are unloaded and can provide the object detection service with full performance, are shown in Figure 4.9 and Figure 4.10 for the light and heavy processing mode, respectively. In the light processing mode the time gain is 13%, from 25.4 minutes for the no-offload configuration down to 22.3 minutes with offloading. In the heavy processing mode, offloading leads to a total mission time of 24.5 minutes instead of 46.8 minutes, with a gain of 47.5%.

In the light processing mode, the big peak early in the mission corresponds to a wait decision point where the selected server has not yet confirmed its availability as expected. Even though the decision is taken to wait for some time in hope of receiving the remote endpoint, in this case, the maximum waiting time is reached without success, and thus the service is invoked locally. Note that the decision to wait is taken one more time towards the end of the mission, with a successful outcome. In the heavy processing mode, the wait decision is taken successfully several times, like in the corresponding HITL/SITL experiment.

The results for the case where s2 is slower than s1are shown in Figure 4.11 and Figure 4.12 for the light and heavy processing mode, respectively. As in the HITL/SITL experiments, the

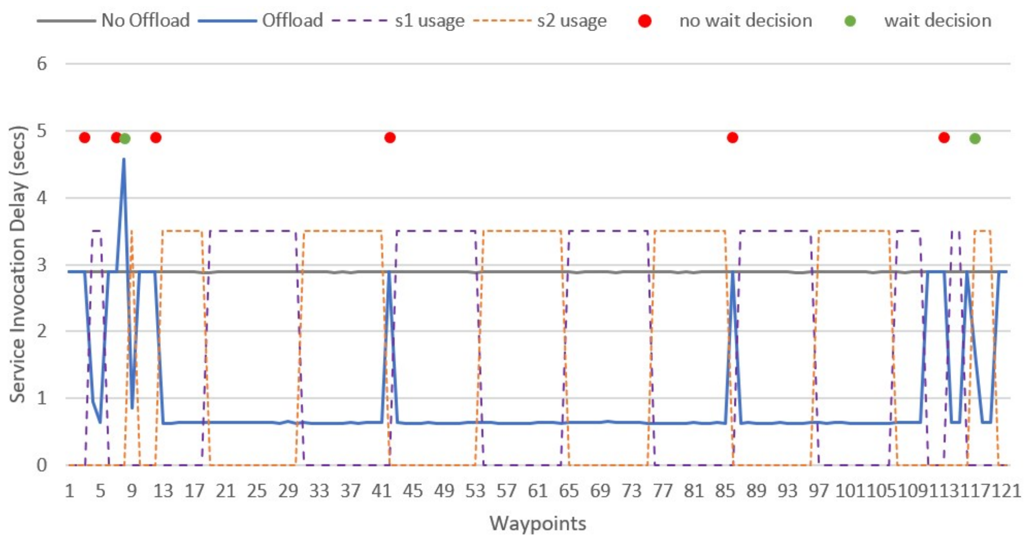

Figure 4.9: Call delay in the light processing mode, s2 is equally fast to s1 (simulation).

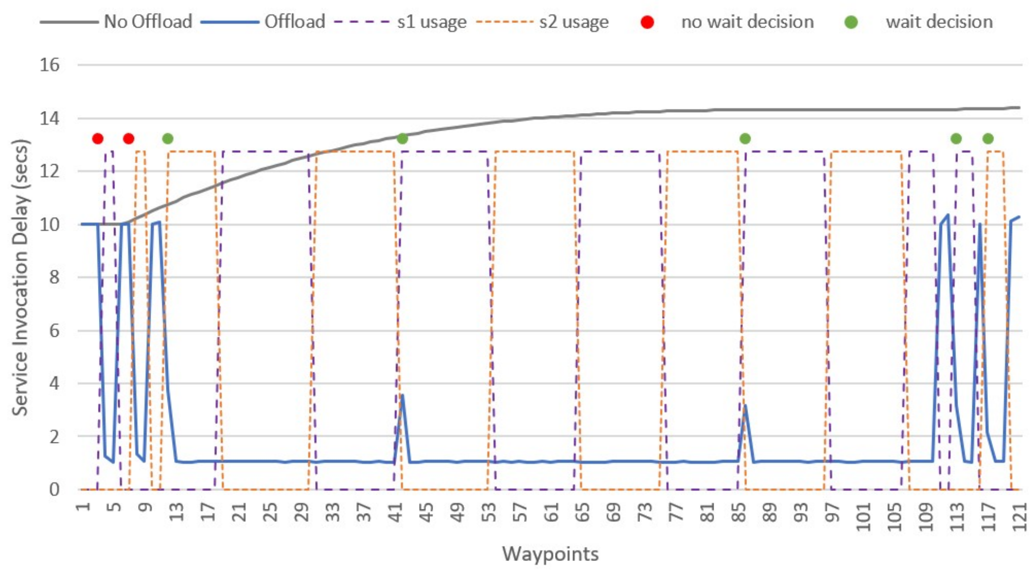

Figure 4.10: Call delay in the heavy processing mode, s2 is equally fast to s1 (simulation).

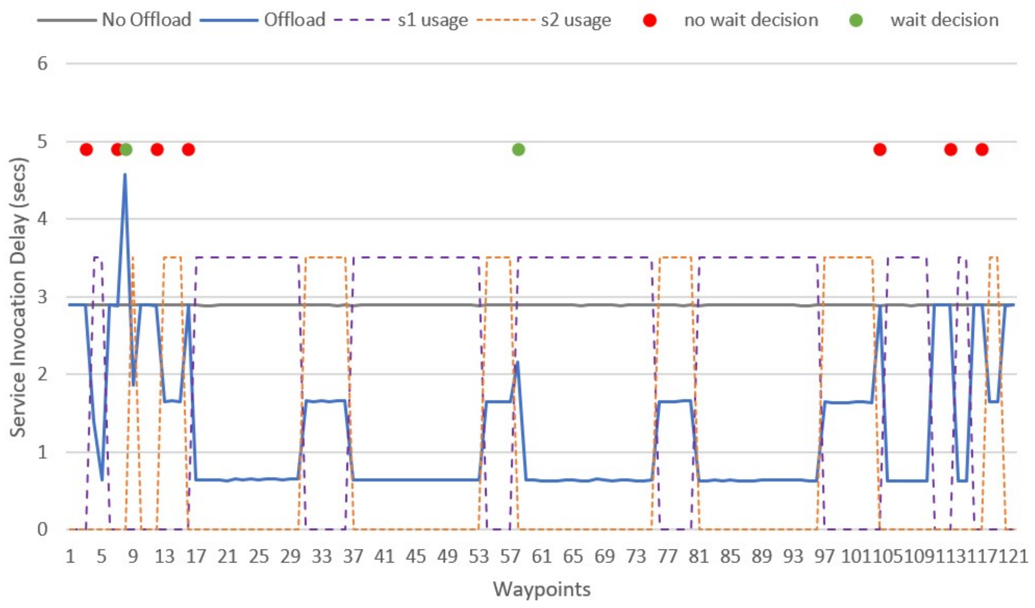

Figure 4.11: Call delay in the light processing mode, s2 is slower than s1 (simulation).

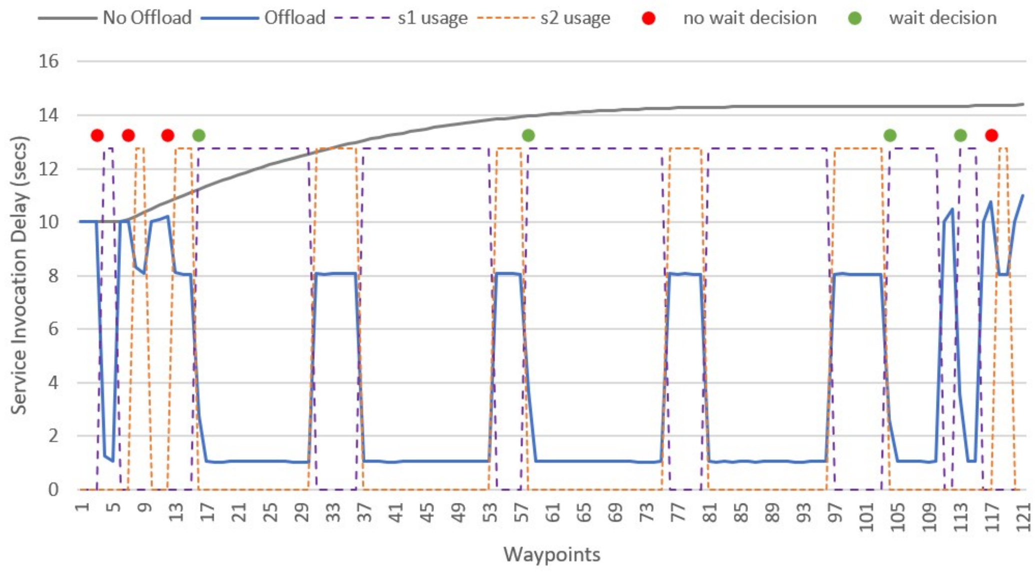

Figure 4.12: Call delay in the heavy processing mode, s2 is slower than s1 (simulation).

s1 is preferred at all waypoints that are covered by both servers.

In the light processing mode, offloading leads to a mission time of 22.3 minutes, a 13% gain vs the no-offload configuration. In the heavy processing mode, offloading achieves a reduction of 40.5%, leading to a mission time of 28.2 minutes. As above, in the light processing mode, we observe a big peak due a risky wait decision which does not pay-off and finally leads to a local call. Also the wait mechanism decides to wait one more time during the mission.

4.4.4 Results summary

Table 4.1 summarizes the results obtained from the HITL/SITL and simulation experiments. As can be seem, the simulation setup produces trustworthy results, very close to those obtained from the HITL/SITL experiments where we use the hardware and software of the real drone.

Table 4.1: Result overview for the mission times obtained in the HITL/SITL and simulation experiments (in min).

Experiment	**No-offloading SITL/HITL vs Simu**	**Offloading SITL/HITL vs Simu**	**Time gain SITL/HITL vs Simu**
Equal servers, light	26.5 vs 25.4	22.2 vs 22.3	16% vs 12%
Equal servers, heavy	47.5 vs 46.8	24.4 vs 24.5	48.5% vs 47.5%
Slow s2, light	26.5 vs 25.4	22.7 vs 22.3	16% vs 12%
Slow s2, heavy	47.5 vs 46.8	28.0 vs 28.2	42.2% vs 40.5%

Therefore, we believe that such simulations are perfectly valid to explore more complex scenarios. This is expected to be particularly useful for experiments that involve multiple drones, for which it is practically impossible to build and maintain a HITL/SITL setup.

Chapter 5

Related Work

Computation offloading has been investigated for general purpose mobile devices such as smartphones as well as in the context of robotic systems. Each application domain poses different requirements and comes with different challenges.

In mobile computing, the main objective of computation offloading is to conserve the energy of mobile devices. To this end, several platforms have been proposed that implement offloading at various levels. For instance, CloneCloud [13] migrates threads running in a smartphone to a VM in the cloud, which runs a clone/duplicate of the smartphone software environment. Instead of migrating entire threads, Jade [14] supports class-level offloading for Java applications, by providing an API that allows developers to control the application's partitioning and the interactions between remote and local code. Maui [15] allows finer-grade computation offloading, at the granularity of individual methods, for which the system automatically produces the required stubs at compile time. Cuckoo [16] offers similar functionality, but allows the programmer to provide two separate implementations for each remote method, one for the local and one for the remote execution. This makes it possible to exploit the differences and special capabilities of the respective hardware platforms.

Similar to our work, the above frameworks take offloading decisions at runtime, based on both offline and online profiling information. However, the key optimization objective is to minimize the energy consumption, rather than to reduce the execution time. From a software engineering perspective, our work is closer to Cuckoo in that it allows to use a different implementation for the service running on the server. Moreover, we adopt a service-oriented approach that further decouples the software technologies used in the drone and the server platform. In particular, it is straightforward to change our implementation to use a pure web-based invocation protocol [17], allowing the service on the server to be implemented using a different programming language and runtime system.

In [18], a unified model is presented for managing a computing infrastructure that consists of mobile, edge and cloud resources. To address the inherent heterogeneity of the computing environment and to enable a flexible placement and invocation of application-level functionality on the available nodes, the proposed approach is designed around the concept of stateless functions, which are offered to the mobile devices in the form micro-services. Our work also assumes stateless services that can be started/invoked in a flexible way on nearby servers that are discovered on the fly. An important difference is that our policy is aware of the servers that are located in the area where the drone operates, and exploits this information to take educated offloading decisions.

Computation offloading has also been studied in the context of mobile robots and UAVs. In [19], a mobile robot captures images from a camera and processes them in order to calculate the path for following a moving object. Part of the processing pipeline always executes on the robot, while the more heavyweight tasks can be executed either on the robot or on a remote server. The system offloads these computations via RPCs, based on an online estimation of the computational complexity for a particular image and the network conditions so as to meet a specified deadline. However, the estimation of the computing/communication time as well as the offloading decisions are deeply integrated into the application logic, rather than this being done transparently under the hood.

In [20], an aerial surveillance application running on the UAV captures a video stream from an onboard camera and detects the number of people in the frames using a video coprocessor (VPU). If the number of people exceeds a threshold, the next three seconds of the video stream are offloaded to a server for further analysis. A similar offloading strategy is followed in [21], where a UAV detects objects in real time. The application captures a low-resolution video stream and performs lightweight processing locally to estimate the existence of objects of interest. If the probability of object existence is high, the UAV captures a high-resolution image and offloads it to a server that executes the more heavyweight object detection process. In both cases, full server availability is needed as the application is statically designed to always offload certain computations without considering the possibility of local execution at all. In contrast, our mechanism preserves the application autonomy by taking offloading decisions at runtime and can flexibly switch to a local invocation if needed.

This is done in a transparent way for the application program.

The authors of [22] focus on a specific mission where a small swarm of drones collaborate to perform visual inertial odometry in an unknown area through image capture and IMU readings. Offloading to a nearby edge server or/and ground station is performed through LTE and WiFi, respectively. The task at hand is split into different independent subtasks, and for each one a separate offloading decision is taken based on the respective processing delay and data transfer latency over the available wireless links. Our approach naturally allows to implement a differentiated offloading policy, by modelling each subtask as a separate service. Also, more complex services can be implemented in a structured way, based on simpler ones for which separate/individual offloading decisions are taken at runtime. [23] studies how UAVs can be used as mobile servers to support a MEC system for other resource-constrained mobile devices. The objective is to minimize the average weighted energy consumption, by jointly considering stochastic computation offloading, resource allocation, and trajectory scheduling of the UAVs. Unlike our work, in this case, the UAVs play the role of servers / service providers that can be used to offload the computations of mobile devices.

A container-based edge offloading framework for autonomous driving is presented in [24]. To meet the efficiency, security and privacy requirements of autonomous driving, vehicles are allowed to offload parts of the applications to edge servers. In the general case, the vehicles have to consult a central entity that performs the server selection. Also, the respective containers are pre-run on the servers in order to reduce the respective load/boot times due to a cold start. In our work, offloading decisions are taken exclusively by the vehicle itself, without any intermediate coordinator. Our mechanism cold-starts a service, if needed, but this is done in the background without any negative interference with the execution of the mission program. The system transparently switches to remote invocation when the server becomes ready.

Chapter 6

Conclusion

We have presented a service-oriented approach for task offloading, implemented as part of a full software stack for autonomous drones. The decision whether to invoke a service locally or on a nearby server is taken at runtime, based on server availability and the estimated invocation delay. Our evaluation, using both a HITL/SITL and simulation setup, shows that the proposed approach can lead to a significant reduction of the mission time. Also, the results obtained using the simulation setup proved to be very consistent with those from the HITL/SITL setup.

As a direction for future work, one might explore more advanced offloading policies, which take into account the service call rate. Also, one could combine our offloading approach with higher-level server allocation and path-planning algorithms, to support smarter offloading in the presence of multiple drones that operate concurrently in overlapping mission areas.

Bibliography

[1] M. Kulbacki, J. Segen, W. Knieć, R. Klempous, K. Kluwak, J. Nikodem, J. Kulbacka, and A. Serester. Survey of drones for agriculture automation from planting to harvest. In *Proc. IEEE Intl Conf on Intelligent Engineering Systems (INES)*, pages 000353–000358, 2018.

[2] Muhammad Imran Majid, Yunfei Chen, Osama Mahfooz, and Wajahat Ahmed. Uav-based smart environmental monitoring. In *Employing Recent Technologies for Improved Digital Governance*, pages 317–335. IGI Global, 2020.

[3] Marzena Półka, Szymon Ptak, and Łukasz Kuziora. The use of uav's for search and rescue operations. *Procedia Engineering*, 192:748–752, 2017.

[4] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, 2005.

[5] Dronekit-python documentation. https://dronekit.netlify.app/.

[6] Python remote objects project. https://pypi.org/project/Pyro4/.

[7] Linux container management. https://linuxcontainers.org/LXD.

[8] Ardupilot web site. https://ardupilot.org/.

[9] Mavlink developer guide. https://mavlink.io/en/.

[10] M. Koutsoubelias N. Grigoropoulos and S. Lalis. A modular simulation environment for multiple uavs with virtual wifi and sensing capability. In *Proc. IEEE Sensors Applications Symposium*, 2018.

[11] ns-3 network simulator. https://www.nsnam.org/.

[12] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *arXiv*, (arXiv:1804.02767), 2018.

[13] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proc. ACM Conf on Computer Systems (EuroSys)*, pages 301–314, 2011.

[14] H. Qian and D. Andresen. Jade: Reducing energy consumption of android app. *Intl Journal of Networked and Distributed Computing*, 3(3):150–158, 2015.

[15] E. Cuervo, A. Balasubramanian, D.-K. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *Proc. ACM Int Conf on Mobile Systems, Applications and Services (MobiSys)*, pages 49–62, 2010.

[16] R. Kemp, N. Palmer, T. Kielmann, and H. Bal. Cuckoo: a computation offloading framework for smartphones. In *Proc. Intl ICST Conf on Mobile Computing, Applications and Services (MobiCASE)*, pages 59–79, 2010.

[17] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services*. Springer Science Business Media, 2004.

[18] L. Baresi, D. F. Mendonca, M. Garriga, S. Guinea, and G. Quattrocchi. A unified model for the mobile-edge-cloud continuum. *ACM Transactions on Internet Technology*, 19(2), Article Nr. 29, 2019.

[19] Y. Nimmagadda, K. Kumar, Y.-H. Lu, and C. G. Lee. Real-time moving object recognition and tracking using computation offloading. In *Proc. IEEE/RSJ Intl Conf on Intelligent Robots and Systems*, pages 2449–2455, 2010.

[20] M. S. Alam, B. Natesha, T. Ashwin, and R. M. R. Guddeti. Uav based cost-effective real-time abnormal event detection using edge computing. *Multimedia Tools and Applications*, 78(24):35119–35134, 2019.

[21] J. Lee, J. Wang, D. Crandall, S. Sabanovic, and G. Fox. Real-time, cloud-based object detection for unmanned aerial vehicles. In *Proc. IEEE Intl Conf on Robotic Computing (IRC)*, pages 36–43, 2017.

[22] M. A. Messous, H. Hellwagner, S.-M. Senouci, D. Emini, and D. Schnieders. Edge computing for visual navigation and mapping in a uav network. In *Proc. IEEE Intl Conf on Communications (ICC)*, pages 1–6, 2020.

[23] J. Zhang, L. Zhou, Q. Tang, E. C.-H. Ngai, X. Hu, H. Zhao, and J. Wei. Stochastic computation offloading and trajectory scheduling for uav-assisted mobile edge computing. *IEEE Internet of Things Journal*, 6(2):3688–3699, 2019.

[24] J. Tang, R. Yu, S. Liu, and J.-L. Gaudio. A container based edge offloading framework for autonomous driving. *IEEE Access*, 8:33713–33726, 2020.