UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Support for Parallel Drone-based Task Execution at Multiple Edge Points

Diploma Thesis

Joana Tirana

Supervisor: Spyros Lalis

Volos 2021

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Support for Parallel Drone-based Task Execution at Multiple Edge Points

Diploma Thesis

Joana Tirana

Supervisor: Spyros Lalis

Volos 2021

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Υποστήριξη για Παράλληλη Εκτέλεση Εργασιών από Drones σε Πολλαπλά Άκρα του Δικτύου

Διπλωματική Εργασία

Ιωάννα Τιράνα

Επιβλέπων: Σπύρος Λάλης

Βόλος 2021

Approved by the Examination Committee:

Supervisor **Spyros Lalis**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Christos D. Antonopoulos**
Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Dimitrios Katsaros**
Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Date of approval: 15-7-2021

Acknowledgements

First and foremost, I would like to thank my supervisor Prof. Spyros Lalis for his valuable guidance and support, not only during the work of this thesis, but also throughout my studies. Without his contribution, this work would not be the same. Also, I want to thank him for always being available for me, and providing me with the best advice and corrections which definitely helped me improve. His devotion, knowledge and way of thinking are an inspiration to me.

Additionally, I want to thank my family for their unconditional love, support and their sacrifices. Without their presence, I would have never imagined being the person I am today. Last but not least I would like to thank my friends for their encouragement and understanding through these years.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Joana Tirana
15-6-2021

Abstract

Drones are used in a significant variety of applications. They can offer access to places where human intervention can be extremely difficult or dangerous. However, the building of such systems can be extremely challenging, since they generate an enormous amount of data that usually are transferred to a remote server in the cloud. This work tries to tackle this problem by creating a distributed system consisting of a server in the cloud and multiple servers on edge nodes. Each edge node is located nearby a group of drones, with direct access to them. Edge nodes can process the generated data in parallel and independently of each other. The process running at the edge points uses the framework of TeCoLa to orchestrate the associated group of nodes. The system offers to users a shell interface through which one can initiate tasks to specific edge nodes and afterwards combine the results. The communication of server and edges is done without any user's intervention. We assess the potential improvements of the proposed approach for different system configurations, using an estimation model that is created using metrics that are extracted from experimental testing.

Περίληψη

Η χρήση μη επανδρωμένων οχημάτων (drones) συναντάται σε ένα μεγάλο φάσμα εφαρμογών καθώς προσφέρουν τη δυνατότητα πρόσβασης σε μέρη που η ανθρώπινη παρέμβαση είναι δύσκολη έως και αδύνατη. Ωστόσο, ο χειρισμος τέτοιων οχημάτων ελλοχεύει πολλές προκλήσεις, όπως είναι το θέμα της διαχείρισης του μεγάλου όγκου πληροφορίας που παράγουν. Συνήθως τα δεδομένα αυτά μεταφέρονται σε κάποιον απομακρυσμένο κόμβο που έχει τη λειτουργία του διαχειριστή (server) και βρίσκεται σε μια Cloud υποδομή.

Η εργασία αυτή προσπαθεί να αντιμετωπίσει το πρόβλημα αυτό δημιουργώντας ένα κατανεμημένο λογισμικό εργαλείο στο οποίο εκτός από τον κεντρικό διαχειριστή χρησιμοποιεί ακριανούς κόμβους (edge nodes). Κάθε τέτοιος κόμβος βρίσκεται τοπολογικά κοντά σε κάποια διαθέσιμα drones, οπότε επεξεργάζεται άμεσα, παράλληλα και ανεξάρτητα από τους ομότυπούς του κόμβους τα παραγόμενα δεδομένα. Οι ακριανοί κόμβοι χρησιμοποιούν το λογισμικό περιβάλλον TeCoLa για να διαχειριστούν την ομάδα των drone για τα οποία είναι υπεύθυνα. Ο χρήστης μπορεί να εκκινεί παράλληλες διαδικασίες στους διαθέσιμους ακριανούς κόμβους μέσω μιας διεπαφής τερματικού που προσφέρει το σύστημα και ύστερα να συνδυάσει τα αποτελέσματα χωρίς να εμπλέκεται στη διαδικασία μεταφοράς/συλλογής των δεδομένων από τους edges. Τέλος, αξιολογούμε το σύστημά μας χρησιμοποιώντας ένα μοντέλο αξιολόγησης που δημιουργήσαμε βασιζόμενοι σε πειράματα που διεξήγαμε.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Context

Unmanned aerial vehicles (UAVs), also known as drones, will be an integral part of the next-generation technologies that will shape future developments in many industrial sectors and human activities. The list of industries that have begun employing drones for various applications is expanding continuously. For instance, drones are already being used in agriculture [1, 2], surveillance [3], rescue operations [4], the monitoring of critical infrastructure [5] as well as for environmental/marine monitoring [6, 7].

The reason why drones are becoming so popular is the fact that they can be equipped with several different cameras, a wide variety of other sensors but also actuators, and can reach locations in a safe and low-cost manner, which are practically impossible, costly or dangerous be approached by humans. Moreover, multi-rotor drones (polycopters) can pilot themselves very efficiently in a highly autonomous manner. This is thanks to their embedded autopilot subsystems, which continuously gather data from various on-board sensors, process them and take all the necessary steering decisions. They can also perform precise maneuvers and even hold completely still, hovering in their current position. As a result, such drones can be flown via relatively 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 programming abstractions that simplify the development of computer-driven missions [8, 9, 10]. Moreover, using available precision landing sensor technologies, such as IRLock [11], the drone can accurately land on a target, such as a hangar

that can provide shelter or serve as a battery recharging/switching station. By combining these capabilities, it is actually possible to fully automate the entire cycle of drone operation, thereby opening the way to a new class of automated drone-based systems. This is particularly attractive if the drone routinely needs to perform the same tasks over a well-known area, as this is typically the case in several monitoring and surveillance application scenarios.

1.2 Problem

One of the problems that arises in such deployments is the traffic and latency due to the transfer of data collected by the drone to the cloud, where it is processed/analyzed to extract useful information. If, as this is typically the case, the drone is used to capture images / provide live video coverage of a target area, a very significant amount of data has to be sent over the public Internet. This, in turn, can lead to network overload and big delays. Also, the target location may be remote with poor or expensive Internet connectivity, making such heavy data transfers too slow or just too costly.

The above problem is further amplified in case there are several different locations of interest that have to be monitored at the same time. For a large scale of deployment, a conventional approach where all data is gathered for processing in cloud-based datacenters, can be practically infeasible.

1.3 Contribution

This work tackles the problem by adopting a hybrid could-edge computing approach, whereby a substantial part of the required data processing can be performed in parallel using edge-based computing infrastructure in the area of drone operation.

The desired functionality is provided via a script-like command language, which can be used to interactively start and monitor the execution of drone-based jobs at different edge points. The language has some similarities to that of a typical command-line shell in Unix systems. In particular, the user can specify a drone-task and the end points where this should be executed while providing suitable arguments. It it also possible to chain several such task executions in a pipelines fashion. Further, the task pipeline can include data processing tasks that combine the data produced at the edge.

From a system design perspective, the desired functionality is implemented using a distributed master-slave architecture. The task manager (master) entity can run in the cloud or directly on the user's computer. It is responsible for executing the user commands and driving all the necessary interaction and synchronization under the hood with the edge servers. Each edge server is installed at an edge point where one or more drones are stationed. It is responsible for reporting the special sensor/actuator resources and availability of the local drones, receiving drone-based tasks, executing them and informing the master about the status and respective data updates that take place during execution.

For the task execution, edge servers rely on the TeCoLa [10] framework. In particular, the drone-based tasks that are specified by the user have to be proper TeCoLa programs. A few minor syntactical extensions are introduced in order to support external data variables that are shared between the task manager and edge server environments. We note, however, that the proposed approach is not tightly-coupled to TeCoLa and could be adapted to support different drone programming languages.

1.4 Structure of thesis

The rest of the thesis is structured as follows. Chapter 2 briefly discusses indicative related work. Chapter 3 presents the system design, while Chapter 4 discusses the supported user commands and task description format. Chapter 5 provides more details about specific implementation aspects. Chapter 6 discusses the experimental setup used to test the implementation. Finally, Chapter 7 concludes the thesis.

Chapter 2

Related Work

This chapter briefly discusses indicative related work, pointing out the similarities as well as the differences with our work. We start with a brief description of TeCoLa which we also use in our work. Then we discuss the two most popular parallel distributed computing frameworks that adopt the MapReduce model which has some similarities with our work.

2.1 TeCoLa framework

The TeCoLa framework [10] aims to simplify the development of mission programs that involve one or more unmanned vehicles (UVs). TeCoLa assumes UVs that can move autonomously by performing all the necessary control loops locally, but still need to be coordinated at a higher level in order to complete a combined mission in an orchestrated manner.

The programming model of TeCoLa follows a service-oriented approach, whereby each UVs is modeled as a node providing one or more services that can be remotely accessed via remote calls. The mission logic is written as a regular program, which invokes the UVs in order to retrieve their status, location and data, processes this data to take decisions and then instructs the UVs to move possibly also to perform some actions according to the application goals. To simplify mission programming, TeCoLa introduces special support for the flexible construction and management of teams of UVs, along with team-level operations that can be invoked in the same way as when addressing a single UV. an efficient 1-N request/reply transport protocol to support such interactions, with good scalability even for a team with a large number of members.

The architecture of TeCoLa is illustrated in Figure 2.1. The mission programs runs on the

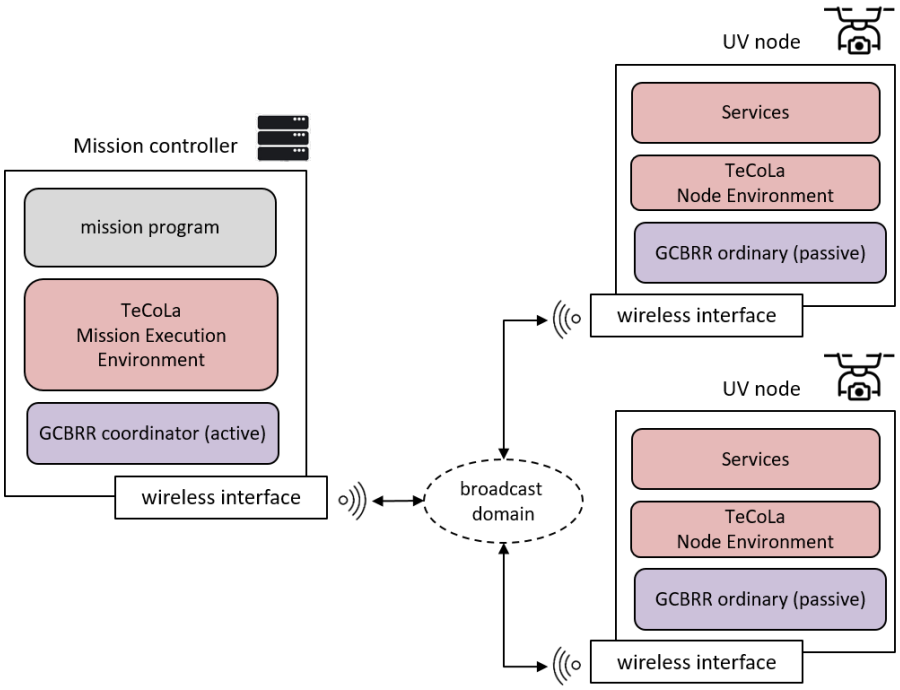


Figure 2.1: High-level architecture of the TeCoLa software stack.

so-called Mission Controller, on top of the TeCoLa Mission Execution Environment. This intercepts the service invocations of the mission program, and maps them to corresponding request-reply interactions that take place over the network, while handling all the related communication and synchronization issues. Each UV node runs the TeCoLa Node Environment, which receives the requests of the Mission Execution Environment, invokes the corresponding local service and sends back a reply.

Under the hood, the request-reply interactions between the Mission Execution Environment and the Node Environment are performed using the GCBRR transport protocol [12], which provides support for coordinated group management and efficient 1-N request/reply interaction on top of a wireless medium with support for a physical broadcast domain. The Mission Controller runs the active (coordinator) part of GCBRR, while each UV runs the passive (ordinary) protocol.

While TeCoLa offers specific support to simplify the programming of missions with multiple UVs, its design assumes that the Mission Controller can directly communicate with the UVs through a local area wireless network. Even if TeCoLa is configured to communicate with each drone in a point-to-point fashion (this option exists), e.g., over long-range RF links, a minimal physical proximity between the Mission Controller and the UV is still needed. As a consequence, the same instance of TeCoLa cannot be employed to run drone-based tasks at

widely different regions. Our work builds on the strong capabilities of TeCoLa and offers a meta-programming abstraction for task execution on multiple instances of TeCoLa deployed at different edge points.

2.2 MapReduce

MapReduce is a programming model used for the parallel and distributed processing of big data. It is inspired by the *map* and *reduce* primitives found in Lisp and other functional languages [13]. The main idea is that given task code written in a functional style and a potentially very large input in the form of key/value pairs, tasks are executed with the maximum possible parallelism given the available computing resources and data dependencies.

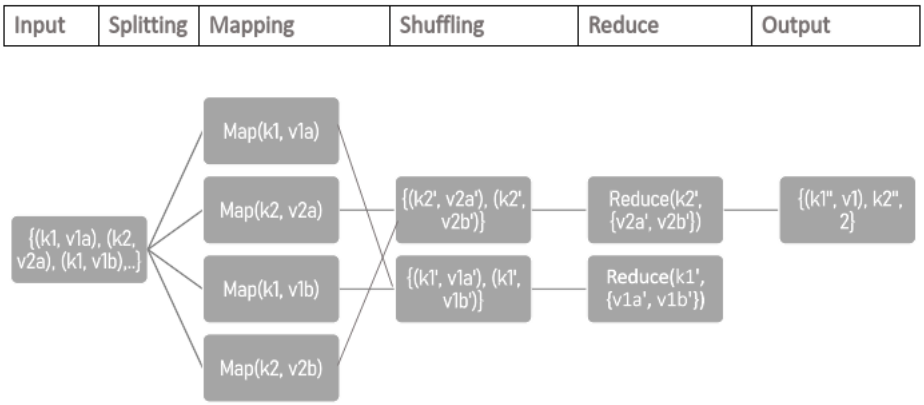


Figure 2.2: A simplified view of MapReduce phases

Figure 2.2 illustrates the typical high-level flow of the MapReduce, as described in detail in [13]. Initially, the input is split into M parts, each part corresponding to a separate *map* task. Each task reads the corresponding part of the input and executes the user-defined map function for each key/value pair. The intermediate key/values produced this way are buffered in memory. In addition, there are R *reduce* tasks.Each such task reads the buffered data, sorts it based on the key information (grouping occurrences with the same key) and for each unique key it applies to the corresponding values the user-defined reduce function in order to produce the output set of key/values pairs.

The map and reduce tasks are executed in a cluster of worker machines. When a worker becomes idle, it starts executing the next map or reduce task. The MapReduce runtime environment takes care of the distribution and the scheduling of the tasks, the interactions that

need to take place over the network with the workers as well the handling of failures. The programmer's only concern is to write the code of the map and reduce functions. The map function needs to conduct separately an operation to each record of the input, and produce a set of intermediate key/value pairs to which the reduce function will be subsequently applied to produce a merged output where all pairs with the same key are combined together.

Hadoop. A well-known open source framework that supports the MapReduce program model is Hadoop [14], which can be installed on a commodity Linux cluster. Besides some possible upgrades in order to fulfill minimum suggested RAM, disk space and other node requirements, no hardware modifications are required. Also, developers can write code in a choice of languages, including Java, C++ and Python. In Hadoop, each task is divided into two stages the *Map* and the *Reduce* stage. The *Map* stage can have one or more map tasks while *Reduce* can have zero or more reduce tasks. The two phases follow the standard of MapReduce workflow, as discussed above. Hadoop also allows the programmer to define an additional function, called the *combiner* function, which can be used to perform some filtering or aggregation operations on the intermediate values in order to reduce the number of keys passed on the reduce phase. For performance reasons, the Hadoop task engine overlaps the phase of map and reduce, passing over the intermediate values to the reduce function via an iterator. More specifically, when a certain percentage of the map task terminates, the produced values are shuffled and passed to the reduce task. The shuffle, merge and sort stages execute concurrently allowing partitions to be continuously merged while being fetched. Furthermore, this also makes it possible to handle large amounts of intermediate data that can not be all stored in memory at once. Last but not least, Hadoop provides its own distributed file system (HDFS), especially designed to handle the storage, distribution and processing of very large data sets (a typical size of a HDFS file can be from gigabytes to terabytes).

Spark. Another open source framework that (among other utilities) offers MapReduce-like support is Spark [15], known for having the quickest data processing tools. Spark's main goal is to provide iterative jobs and interactive analytics that cannot be efficiently supported using Hadoop [16]. Unlike Hadoop which basically relies on disk, Spark processes everything in memory. The main entity of the Spark engine is the Driver, which is a master program responsible for analyzing, distributing, scheduling and monitoring tasks. The system also consist of one or more Executors, which execute the code that is assigned to them by the Driver and report their status back to it. The cluster of machines used to run Spark tasks is

managed by an independent manager entity, such as Spark's standalone cluster manager [17], YARN [18], Mesos [19] or Kubernetes [20]. Data processing in Spark is based on so-called Resilient Distributed Datasets (RDDs), which are immutable, read-only collections of objects partitioned among a set of machines, which can be easily retrieved in case a portion is lost [16]. The user can create an RDD by loading data from a source or via the transformation of a existing RDD. Spark offers two types of operations on RRDs: *transformations*, which creates a new dataset from an existing one, and *actions*, which returns a value to the driver program after running a computation on the dataset. The former is characterized as a "lazy" operation while the latter as a strict one, "immediate", i.e., the results of a transformation operation are not computed unless an action requires a result to be returned to the Driver program. For example, a map transformation that applies a function to all elements of an RRD to produce a new RDD is performed in a lazy way, whereas a reduce operation that combines the elements of an RRD using an associative function to produce a result that will be exposed to the Driver program, is executed as an action.

Both Hadoop and Spark are powerful frameworks, with the former typically being more appropriate for dependable offline processing and the latter being a preferred option for more lightweight online processing. Like the MapReduce approach, our work also aims to support parallel distributed processing on a potentially very large scale. However, our work focuses on data that is collected on-the-fly through live drone-based missions, rather than data that is already stored in the cloud. Also, our design does not rely on the usage of a centralized cluster, but supports parallel execution of drone-based mission and data processing at different edge points while minimizing the amount of data that needs to be sent over the Internet. At the same time, the user can add arbitrary intermediate data processing steps, at both the local (edge) and global (cloud) scope, in the form of Python scripts.

Chapter 3

System Architecture

This chapter presents the system design. It introduces the main system components and describes the high-level interactions between them.

3.1 Concept

We envision a system that will allow the user to launch drone-based data acquisition tasks targeting different areas of interest. The goal is for these tasks to be automatically distributed to the respective edge points in a transparent manner, without the user having to take any explicit deployment and control actions. Further, it should be possible for the collected data to be processed in a flexible manner, and whenever possible in a parallel way by exploiting the available computing resources at the different edge points. Figure 3.1 gives a high-level overview of the system architecture. Next, we discuss each component in more detail.

3.2 Registry

The Registry is to be started before all other system components. It is used to keep up-to-date information regarding the edge points that are part of the system and can be used to execute drone-base tasks. For each edge point, this information includes (i) address information (IP and port) of the corresponding Edge Server; (ii) the drones that are locally stationed there and can be controlled through the edge server, (iii) the geographical area that can be covered using these drones, also referred to as ”zone”, and (iv) the identifier for that zone. Also, for each drone stationed at an edge point, additional information is provided, such as

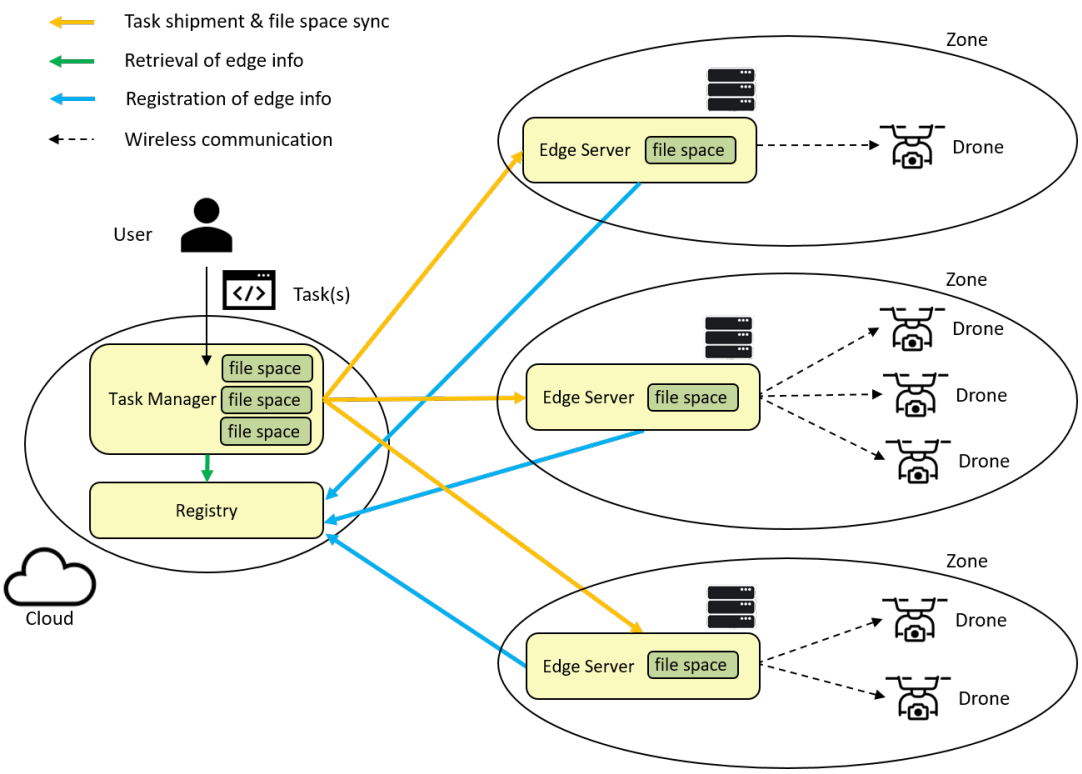


Figure 3.1: System architecture.

the identifier of the drone and the identifiers for the services provided by the drone[1].

The Registry receives this information via registration messages sent by the corresponding Edge Servers. Each entry has a limited lifetime, which is refreshed each time the Registry receives a new registration message from the Edge Server. If the Registry stops receiving such messages, the respective entry will be eventually removed from the Registry (the corresponding edge point is considered to be unavailable).

3.3 Edge Server

The Edge Server is responsible for receiving tasks from the Task Manager targeting a certain zone, and executing these tasks using the drones and computing resources that are locally available.

On the one hand, the Edge Server discovers and connects to the drones that are stationed in the respective end point. Based on this information, it sends a registration message to the Registry. A new registration message is sent when new drones become available or an existing

[1]The name of each service is unique and it is linked with a service ontology that is defined inside the framework of TeCoLa. This ontology specifies each service class, including its methods and semantics.

drone is (temporarily) decommissioned. Even if this information does not change, the Edge Server sends to the Registry periodic heartbeat in order to confirm its availability.

On the other hand, the Edge Server waits to receive tasks from the Task Manager. When a new task arrives, the Edge Server starts executing it while keeping the Task Manager informed about the execution status. Finally, when task execution completes, the Edge Server informs the Task Manager accordingly.

3.4 Task Manager

The Task Manager is responsible for managing the execution of the tasks submitted by the user, by interacting as needed under the hood with the relevant Edge Servers.

It periodically queries the Registry to receive up-to-date information about the running tasks and drones. For every zone, the Task Manager creates and maintains a runtime object that contains this information. In addition, each zone object is associated with an own pool of files, called ”zone file space” (discussed in the next section).

Further, the Task Manager supports a few basic user-level commands for reviewing the available zones and drones, submitting one or more tasks for execution on specific zones in a pipelined fashion, getting corresponding execution status information, and cleaning the file space. The execution of a given task on a zone and the required synchronization of zone file spaces, is achieved in a transparent way for the user, via a suitable interaction between the Task Manager and the corresponding Edge Server.

3.5 Zone file spaces

Zone file spaces are used for input/output between the user and tasks as well as between tasks. At the Task Manager side, a separate directory is created for each zone, which can be accessed by both programs and the user via the standard file system operations. Similarly, each Edge Server keeps the zone’s file space in a local directory. These file spaces (directories) are kept loosely synchronized via suitable view update and file transfer interactions that take place before and after task execution at the edge. As an exception, files created in the $tmp\backslash$ directory of a zone file space are not handled by the synchronization mechanism.

Chapter 4

User & Application Programming Model

This chapter introduces the user model. It discusses commands through which the user can interact with the Task Manager, and discusses the format and various conventions that apply to the tasks that are submitted for execution.

4.1 User Commands

The Task Manager offers a command-line interface through which the user can interactively inspect the available zones and submit tasks for execution. Short descriptions of each command are given in the following.

`list <zone>`

Prints the information available for the specified zone (identifier). If the zone does not exist (the Registry does not contain an entry for it), an error message is printed. If the special wildcard character `*` is given instead of a zone identifier, information will be printed for all available zones.

`reset <zone>`

Removes all the files inside the file space of the specified zone (identifier). It also will trigger the Edge Server to do so. In this case the Edge Server will also remove all files in the $tmp\backslash$ directory. If the specified zone does not exist, an error message is printed. If the special wildcard character the `*` is given instead of a zone identifier, the files of all available zones will be removed. Note that this command is allowed only if there is no task running or pending for execution, else it fails and an error message is printed.

```
run <edgeTask>@[zone1<arg1,arg2,..>;..] | ... |
    <managerTask>@[zone1;..]<arg1,arg2..>
```

Runs a pipeline consisting of one or more tasks on the specified zones. There are two types of tasks, depending on where these are executed: so-called edge tasks and manager tasks. For an edge task, the user specifies the zones where it should be executed and provides an argument list for each target zone. For manager tasks, the user specifies the target zones and a single list of arguments (independently of the zones specified). If the user attempts to start a new job while another job is already running, an error message is printed and the user command is ignored.

`stat`

Prints the status of the current job. More specifically, for each task that is part of the job, its execution status (pending, running, done, failed) is printed for each of the target zones.

`kill`

The job that is currently running is stopped, no new tasks are sent. However, tasks that are already running will not be terminated just will be ignored. If no job is currently running, this command has no effect.

Listing 4.1 below gives an indicative interaction sequence that employs these commands. The user starts by listing information for all available zones. Then, the user submits for execution on these zones a task pipeline, consisting of the *scannerTask*, the *detectorTask* and the *aggregatorTask*.

```
1  > list *
2  zone: zoneA
3  node: Drone1
4  position: (37.9278579,23.6500001)
5  services:
6      MobilitySvc
7      CameraSvc
8  zone: zoneB
9  node: Drone2
10 position: (37.9278579,23.65570508004129)
11 services:
12     MobilitySvc
```

```
13     CameraSvc
14     VideoSvc
15 >
16 > run scannerTask@[zoneA<'Drone1','tmp\raw\','mission1.txt'>;zoneB<'
     Drone2','tmp\raw\','mission2.txt'>] | detectorTask@[zoneA<'tmp\raw\','
     marked\'>;zoneB<'tmp\raw\','marked\'>] | aggregatorTask@[zoneA,zoneB]<
     'zoneA','zoneB','marked\','output'>
17 >
18 > stat
19 scannerTask: zoneA DONE, zoneB RUNNING
20 detectorTask: zoneA RUNNING, zoneB PENDING
21 aggregatorTask: PENDING
22 >
23 > stat
24 scannerTask: zoneA DONE, zoneB DONE
25 detectorTask: zoneA DONE, zoneB RUNNING
26 aggregatorTask: PENDING
27 >
28 > stat
29 scannerTask: zoneA DONE, zoneB DONE
30 detectorTask: zoneA DONE, zoneB DONE
31 aggregatorTask: RUNNING
32 >
33 > stat
34 scannerTask: zoneA DONE, zoneB DONE
35 detectorTask: zoneA DONE, zoneB DONE
36 aggregatorTask: DONE
37 >
38 > reset *
```

Listing 4.1: Indicative user interaction example based on the supported commands.

The *scannerTask* is an edge task for scanning a target area using a drone while taking photos via the drone's camera. The task takes as arguments (for each zone) the name of the drone to use, the name of a directory where to store the photos taken by the drone and the name of a file that contains the mission parameters (start waypoint of the mission, the length/side of the square area to be scanned, the flight altitude, and the photo shooting step / distance between the photos to be taken). The *detectorTask* task is also an edge task, which processes the photos in order to detect objects of interest. It takes as arguments (for each zone) the name of the

directory with the raw images to be processed and the name of a directory where to store the processed photos where objects of interest were detected. All directories and file names are relative to the file space of each zone. Finally, the *aggregatorTask* is a manager task, which takes as arguments the names of the zones file spaces to examine, the name of the directory (in each zone file space) where to find the processed photos where objects of interest were found, and the file path name where to store its output. For the tasks' source codes, refer to the Appendix section.

In the sequel, the user periodically checks the task execution status. When all tasks finish, the user resets the file space of all zones.

4.2 Task Scripts

A task is a Python script stored in a plain text file. However, the script contains certain special header elements, as follows. The first line of the script serves as a type specifier, which can be `!$EDGE` or `!$MANAGER`, depending on whether the task is designed for execution in an edge or the task manager environment, respectively. The second line of the script file declares the names for the script's arguments, via the $\#input$ directive. These names correspond to variables will can be accessed from within the script as usual. The user is expected to provide the correct number and type of arguments, in the order the corresponding names appear in the declaration.

Edge tasks may contain TeCoLa objects and method calls in order to implement drone-based missions. However, they may also be regular Python scripts without any TeCoLa-specific elements. Manager tasks have to be regular Python scripts and are not allowed to contain any TeCoLa primitives.

By default, the execution directory of an edge task is the local zone file space, and all relative file references are resolved in this context. In contrast, the execution directory of manager tasks is the top of the file space tree, wherein the directories of the zone file spaces reside. This way, a manager task can access whichever zone file space is required, e.g., based on its arguments.

Finally, edge tasks can record arbitrary status information during their execution. This can be done using the standard Python logging support, through invocations of the form `logger.levelname.(<message>)`, where `levelname` can be `debug`, `info`, `warning`,

`error` or `critical`. In the Edge Server's runtime, a corresponding logger object is created, which transparently sends this information to the Task Manager environment. There, it is automatically stored in a special log file in the file space of the corresponding zone, which can be polled/accessed by the user via the usual file operation calls and command line tools. However, this option should be used with care as it can lead to increased traffic between the Edge Server(s) and the Task Manager during task execution.

4.3 Task execution

Edge tasks are executed in parallel in the respective target zones. If a job includes a pipeline of several consecutive edges tasks, this will run independently on the respective zones / edge points.

Manager tasks are executed once for all target zones. They can be used to implement reduce-like functionality and act as synchronization barriers. More specifically, the execution of a manager task begins only once all preceding edge tasks that operate on the same target zones have run to completion.

Chapter 5

Implementation

This chapter discusses the most important implementation aspects of the system we have developed.

5.1 Service-oriented design

The main communication between the different software component follows a service-oriented approach. More specifically, each entity provides and/or invokes certain services, which are accessed via Remote Procedure Calls (RPCs).

The RPCs are implemented using Pyro4 [21], a library that supports the remote invocation of Python programs over the network. With Pyro4 one can create applications in a server-client logic. The server provides a service that is declared with a python class, that can contain methods and variables like a common class. The programmer can declare in an explicit way which parts of the class are exposed and can be called remotely by a client. Additionally, the server side to manage all the remote requests has to create a Pyro4 daemon thread associated with a specific service, which will communicate with the clients in the background.

The server must give the service a unique name and register it using the API of Pyro4. This makes it possible for the engine of Pyro4 to send each request to the proper server process that supports the corresponding service. Respectively, the client by knowing the IP/port address of the server program and the name of the service can create a proxy object of the specific service and run remotely the methods that are supported by the server.

Finally, we assume that the ip/port address of the Registry and Task Manager are known to all Edge servers (e.g., are supplied as arguments when booting the edge environment).

5.2 Main software components

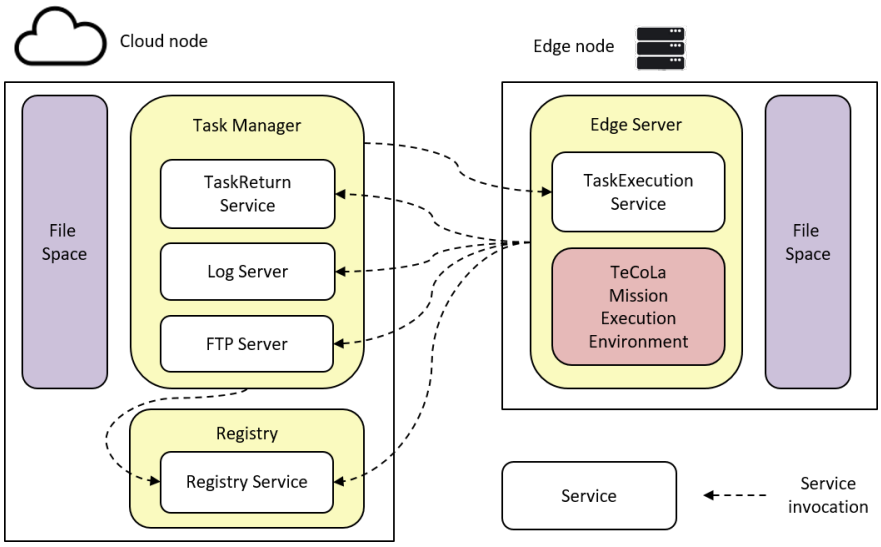


Figure 5.1: Main software components and services.

The main software components, services and service invocation relationships are shown in Figure 5.1. More details are provided in the following sections.

5.3 Registry

The Registry stores all the information about the available zones by creating for each of them an entry that contains in XML format a description of the zone's resources, as shown below.

```
<zone id = 'zoneA' ip ='172.19.0.5' port = '9559'>
    <node name ='uav1' lat ='37.9283026' lon ='23.651712'>
        <service name='CameraSvc'/>
        <service name='MobilitySvc'/>
    </node>
    <node name ='uav2' lat ='37.927858' lon ='23.661409901'>
        <service name='CameraSvc'/>
        <service name='MobilitySvc'/>
        <service name='VideoSvc'/>
    </node>
</zone>
```

Listing 5.1: Indicative zone registration in XML form.

A new zone entry is created when an Edge Server registers for the first time. Zone entries have a limited lifetime, which is refreshed each time a corresponding registration is received. If the lifetime of an entry expires, it is considered a no-available zone and it is removed from the Registry.

The Registry provides the *Registry* service. This is used by the Edge Servers and the Task Manager, to inform the Registry about the zone and its current drone resources, and to retrieve information about the available zones, respectively. This service provides the function call `update(string zoneID, string msg)`, where `zoneID` is the identifier of the zone that is supported by the Edge Server and `msg` is a registration in the expected format. The Edge Server calls this function at initialization time and each time the local drone resources are updated. Additionally, consists of the `getAll()` function, which is used by the Task Manager and returns all current zone entries of the Registry in the above XML format, and the `get(string zoneID)` function, which returns the entry only for the zone with the specified `zoneID` identifier (if available).

Internally, the Registry uses one Pyro4 server/daemon thread, for the aforementioned service, and another thread to garbage-collect entries when their lifetime expires.

5.4 FTP Server

The file transfer between the Task Manager and Edge Servers, is done via FTP (File Transport Protocol) on top of TCP/IP. More specifically, the Task Manager runs an FTP server process, which creates a FTP handler using the `ThreadedFTPServer` module of the `pyftpdlib.servers` Python library. In turn, the handler creates a separate thread for each client to be able to manage clients concurrently. This is important since individual FTP transactions can take a long time. Edge Servers invoke the FTP server as part of the file space synchronization procedure that takes place before and after task execution at the edge.

5.5 Log Server

The Task Manager runs a server that handles the log messages sent by the tasks that are being executed on Edge Servers. The Log Server accepts TCP connections, waits for new status/logging messages, and prints them in the user terminal. In turn, the logging object

created in the runtime of the Edge Server opens a TCP connection to the Log Server and writes the messages of the locally executing task in the socket. The Log Server processes requests asynchronously; for each request a new thread is generated, ensuring that a client will not be affected by other's client interaction with the Log Server.

5.6 Task Manager

The Task Manager is responsible for the user interaction and task execution, with communicating with the Registry and Edge Servers as needed under the hood. It keeps information for each available zone and creates a separate directory for the corresponding file space.

The Task Manager provides the *TaskReturn* service, which offers the `terminated(string zoneID, string taskID, boolean returnStatus)` function through which the Task Manager is informed about the termination of a task. If the execution of the task was completed successfully the `returnStatus` argument will be set True, otherwise False. This function is invoked remotely by Edge Servers as well as locally upon the termination of a manager task. In the latter case, the `zoneID` argument is a list that contains the identifiers of the zones for that task.

The Task Manager includes four threads: a Pyro4 deamon for the *TaskReturn* service, a thread that handles user interaction, a thread that periodically invokes the Registry to receive up-to-date zone information, and a thread that is responsible for the execution of manager tasks (which invokes the `terminated` function of the *TaskReturn* service when the task completes its execution).

5.7 Edge Server

The Edge Server includes TeCoLa environment so that locally running tasks can use the primitives of TeCoLa to control one or more drones as needed. After its initialization, the Edge Server is ready to accept and execute tasks upon the request of the Task Manager.

The Edge Server provides the *TaskExecution* service, which offers the `startTask(string taskID, string scriptCode, string zoneFSView)` function. This is used by the Task Manager to start the execution of a new task with the specified `taskID` by running the specified `scriptCode` in the local runtime environment. The `zoneFSView` contains the

view of the zone file system in the runtime of the Task Manager.

The Edge Server internally employs three threads: a thread that periodically polls TeCoLa for new drones and updates the Registry accordingly, a Pyro4 daemon for the *TaskExecution* service, and a thread that is responsible for the actual task execution (which invokes the `terminated` function of the *TaskReturn* service when the task completes its execution).

5.8 Job Execution

When the user submits a new job, the Task Manager creates a list of task objects, one for each task. A task object contains the following attributes: (i) task type, (ii) list of zones, (iii) for each zone the list of arguments that were supplied by the user, (iv) the code of the task script, and (v) the zones where task execution has completed. Then, job execution starts by calling the `scheduleTask()` function to start the first task in the pipeline. This function is also invoked from within the handler of the `terminated` function of the *TaskReturn* service, to proceed with the execution of the next task in the pipeline (if any and whenever this is possible) when a task completes.

The core logic of the `scheduleTask()` function is described in Algorithm 1. The function iterates through the task list of the job and checks which tasks are ready to be scheduled for execution, either on an Edge Server (zone) or locally on the Task Manager. Note that to schedule a manager task, all the previous tasks have to be completed. Finally, if during the iteration there is an unavailable zone the ”zone failure policy” is applied, to decide if there should be a violent termination of the job.

The main logic of the the `terminated()` function is shown in Algorithm 2. When this function is called the first thing it does is to mark all the zones that returned from the task as available (lines 2-19). However, if there are zones that crashed or the task failed, it checks if the job is safe to proceed with the execution of the following tasks. This decision is made following, again the ”zone failure policy” that will be explained below. Afterwards, the `scheduleTask()` function is called (line 20). If there are no other tasks to schedule and all tasks have returned, the entire job finishes and the user is informed accordingly (line 21).

Algorithm 1 Logic of `scheduledTask` function.

Input: $task$
Output: number of newly started tasks

1: $started \leftarrow 0$
2: **while** $task$ not $null$ **do**
3: $zonesPending \leftarrow task.getNonCompletedZones()$
4: **if** $task.type = EDGE$ **then**
5: **for** $zone$ in $zonesPending$ **do**
6: $z \leftarrow zoneObjs$.get($zone$)
7: **if** $z = null \wedge$ CheckIfBlocks($task, zone$) **then**
8: $return(-1)$
9: **end if**
10: **if** $z \neq NULL \wedge \neg\, z.isRunning$ **then**
11: $z.isRunning \leftarrow True$
12: $fileSpace \leftarrow z.getFileSpace()$
13: $scriptCode \leftarrow setArguments(task.code, task.getInput(zone))$
14: $z.TaskExecutionService.startTask(task.id, scriptCode, fileSpace)$
15: $started \leftarrow started + 1$
16: **end if**
17: **end for**
18: **else**
19: **if** $\neg task.prevTasksCompleted()$ **then**
20: $return(started + 1)$
21: **end if**
22: **for** $zone$ in $zonesPending$ **do**
23: $z \leftarrow zoneObjs.get(zone)$
24: **if** $z = NULL$ **then**
25: $return(-1)$
26: **end if**
27: $z.isRunning \leftarrow True$
28: **end for**
29: $pendingTasks.add(task)$
30: $return(started + 1)$
31: **end if**
32: $task \leftarrow task.nextTask()$
33: **end while**
34: $return(started)$

Algorithm 2 Logic of `terminated` function.

Input: $zonesCompleted, taskID, returnStatus$

1: $task \leftarrow jobChain.get(taskID)$
2: **if** $task = NULL$ **then** ▷ not waiting for this task
3: $return$
4: **end if**
5: **if** $\neg returnStatus \wedge task.type = MANAGER$ **then**
6: $TerminateJob(-1)$
7: **end if**
8: **for** $zone$ in $zonesCompleted$ **do**
9: $z \leftarrow zoneObjs.get(zone)$
10: **if** $z \neq NULL \wedge returnStatus$ **then**
11: $z.isRunning \leftarrow False$
12: $task.setDone(zone)$
13: **else**
14: $task.setFailed(zone)$
15: **if** $CheckIfBlocks(task, zone)$ **then**
16: $TerminateJob(-1)$
17: **end if**
18: **end if**
19: **end for**
20: $res \leftarrow scheduleTask(task)$
21: **if** res <= 0 **then**
22: $TerminateJob(res)$
23: **end if**

5.9 Task Execution and File Space Synchronization

The workflow of the task execution on an Edge Server is illustrated in Figure 5.2. Below, we discuss these steps in more detail.

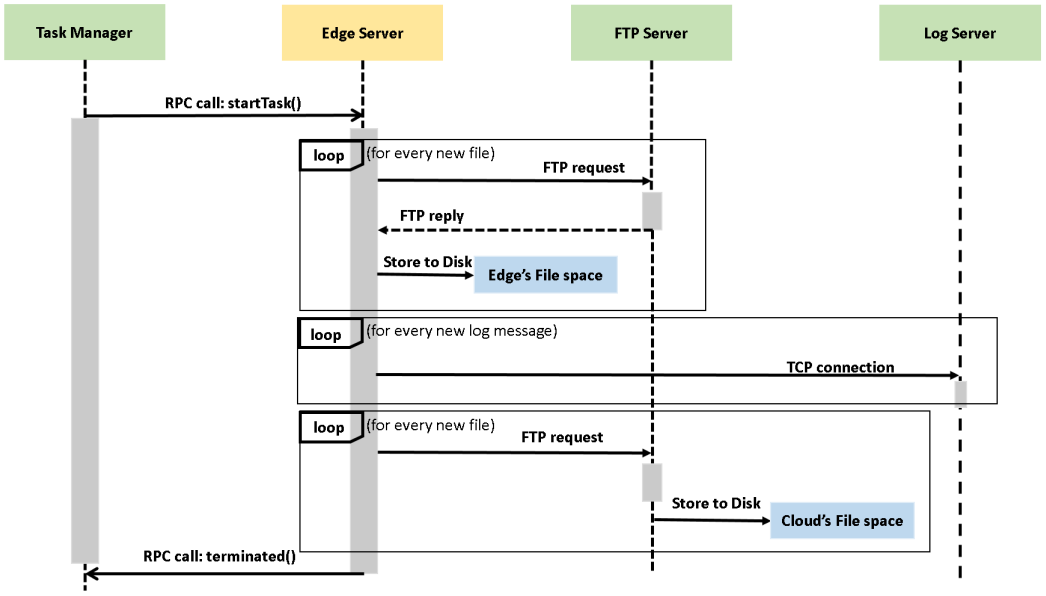


Figure 5.2: Main steps of task execution at the edge

To start task execution, the Task Manager invokes the `startTask()` function of the *TaskExecution* service on the Edge Server of the corresponding zone. Upon receiving such a request, the Edge Server performs a first file space synchronization step to download any new files from the zone file space in the Task Manager runtime to the local file space in its own runtime environment. Afterwards, the Edge Server starts executing the task. During task execution, any log messages written by the task in the logger object are forwarded to the Log Server. When task execution completes, another file space synchronization step takes place in which the Edge Server stores in the zone file space in the Task Manager runtime any new files that were created during the execution of the task. Finally, the Edge Server informs the Task Manager that the task was completed.

In the first step of file space synchronization, the Edge Server checks for new files by comparing the view of the local zone file space with that of the Task Manager's zone file space, which is given in the `zoneFSView` argument of the `startTask()` function call. For each new file, an FTP transaction is performed to in order to receive and store the file locally. In the second step of the zone file space synchronization, the Edge Server compares the

`zoneFSView` with the view of the local zone file space as was formed after the execution, and performs an FTP transaction to upload to the Task Manager any new files.

The execution of manager tasks is simpler as this is done from a local thread of the Task Manager, which checks periodically the list of pending tasks. If there is a new task, it executes it and then informs the Task Manager about its completion.

Note that an Edge Server may crash or get disconnected from the Internet while a job is running. If this happens during the execution of a job the ”zone failure policy” is applied. This policy ensures that is safe for the job to keep running regardless the fact that some zones may fail. If an edge task fails and the specified zone is not part of an upcoming manager task, then its failure can be ignored. That is safe since the execution of an edge task is independent. On the other hand, in a manager task the zones are dependent to each other since the task is executed as one for the whole group (to support a reduce functionality). So, a manager task can be executed only if the occupied zones have completed the previous tasks successfully. Finally, if a manager task fails the job is terminated unsuccessfully and does not proceed to the successive tasks.

Chapter 6

Functional Testing

This chapter explains how we have tested our implementation. We start by describing the setup that is used to perform our tests and experiments. We then demonstrate the functionality of the system by giving an indicative application example.

6.1 Experimental setup

In order to thoroughly test all system functions and several corner cases in a flexible, controlled and safe way, we use a suitable simulation setup, illustrated in Figure 6.1. For practical reasons, we use a setup with a single machine. The entities that belong to the cloud side (Registry, FTP Server, Log Server and Task Manager) run on the machine as normal processes. All other entities are packaged and deployed as separate Docker containers [22].

Each edge node container includes the Edge Server together with the TeCoLa Mission Execution Environment. Note that these containers are practically identical to those that would be used in a real deployment. Each edge point is associated with at least one but possibly more drones, assumed to be stationed nearby. The respective containers include the TeCoLa Node Environment along with the mobility and the camera services. The mobility service invokes via Dronekit [23] the off-the-shelf Ardupilot [24], a popular autopilot used in polycopter drones. Internally, Dronekit interacts with the autopilot via the MAVLink protocol [25]. In this setup, Ardupilot is configured to operate in the software-in-the-loop (SITL) mode [26], where it is coupled with a physics engine and copter dynamics simulator. Note that the autopilot is identical to the one used in a real drone, except for the fact that its sensor and actuator back-ends are coupled to the simulation engine and internal drone model, instead of accessing

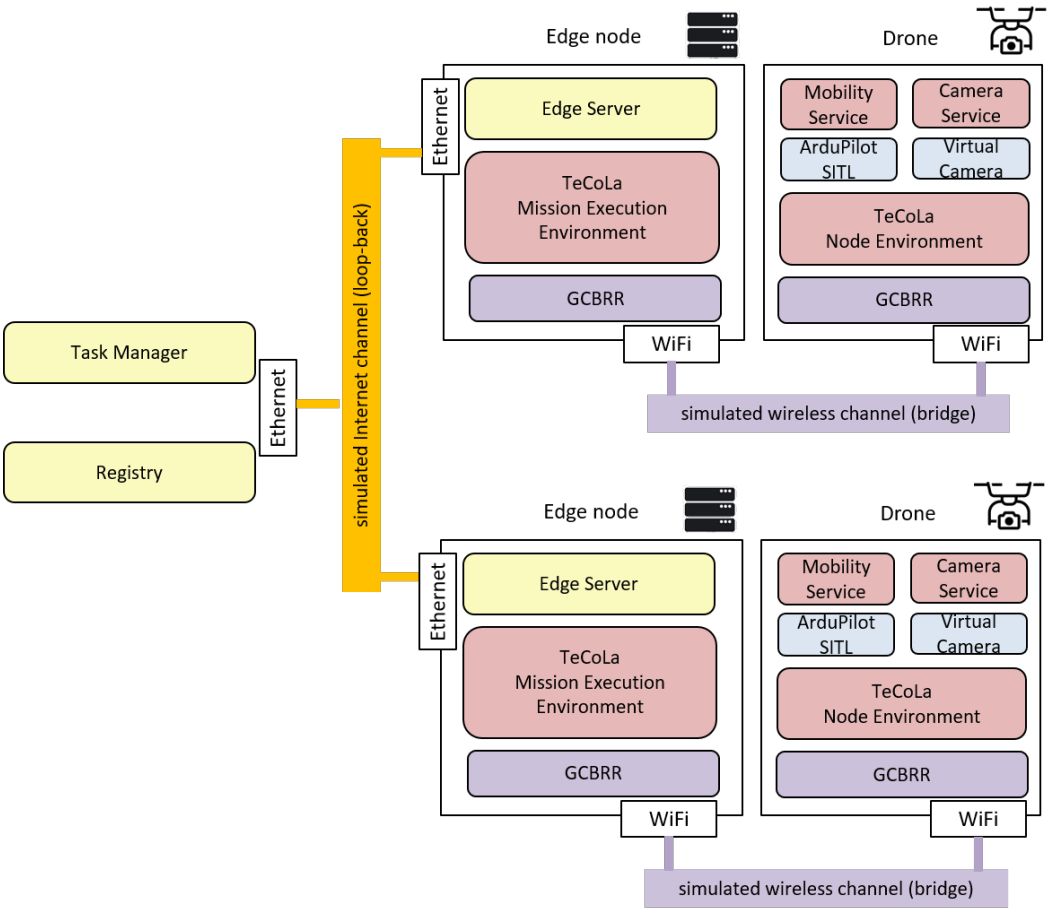


Figure 6.1: Experimental setup used to test the functionality of the system.

physical devices. The camera service is configured to access a virtual camera device, which simply returns pre-recorded images from the local file system.

The communication between the each edge node and the drones for which it is responsible, is performed over a separate/private simulated wireless network. Given that our work does not focus on the aspects of this wireless communication, each such network is implemented using a simple bridge between the corresponding containers. The Internet connectivity between the Task Manager, the Registry and the different Edge Servers is provided over IP-level via loop-back.

6.2 Test application

To test our system, we use an application in the spirit of the example discussed in Section 3, which consists of three tasks: the *scannerTask* that uses a drone to fly over a target area and take photos, the *detectorTask* that processes these photos to detect objects of interest

via a cv2 model using the weights and configure files from darknet-yolo [27], and the *aggregatorTask* that provides an aggregated report to the user. To illustrate the potential gains of edge-based processing, we develop two versions of these tasks, as follows.

Edge-based processing. The *scannerTask* and *detectorTask* run at the edge, while the *aggregatorTask* runs in the cloud. The *scannerTask* stores the photos taken in the $tmp\backslash$ directory of the zone file space, while the photos produced from the *detectorTask* (photos with at least one detected object) are stored in a synchronized directory in the file space. As a result, only the photos of interest are transferred back to the cloud and taken into account by the *aggregatorTask*.

Cloud-based processing. Only the *scannerTask* runs at the edge, while the *detectorTask* and *aggregatorTask* run in the cloud. In this case, the *scannerTask* stores the photos taken in a synchronized directory in the zones' file space. As a result, all captured photos are transferred to the cloud and are processed there by the *detectorTask*. Finally, the *aggregatorTask* processes the outputs of the *detectorTask* as usual.

We have performed several tests to verify that our implementation works as expected in terms of functionality.

6.3 Performance Model

As we do not have separate physical machines at our disposal to use as edge servers, we build a model in order to estimate the job execution time as a function of the number of edge points (servers). The model relies on some basic metrics obtained through a baseline execution as well as on additional input parameters for the scanning and detection processes performed by the respective tasks.

In the baseline execution, used to derive the basic performance metrics for the model, we use a single edge server with one drone. Figure 6.2 illustrates the different steps of job execution, for the edge-based processing case (upper part) and the cloud-based processing case. Table 6.1 lists the basic delays that were measured in these executions. These delays are, in turn, used in a model we build in order to estimate the job execution time for a wide range of scenarios, described next.

Table 6.2 lists the so-called input parameters of the model, which allow us to explore a wide range of scenarios. For instance, the speed of the drone, the length of the square target

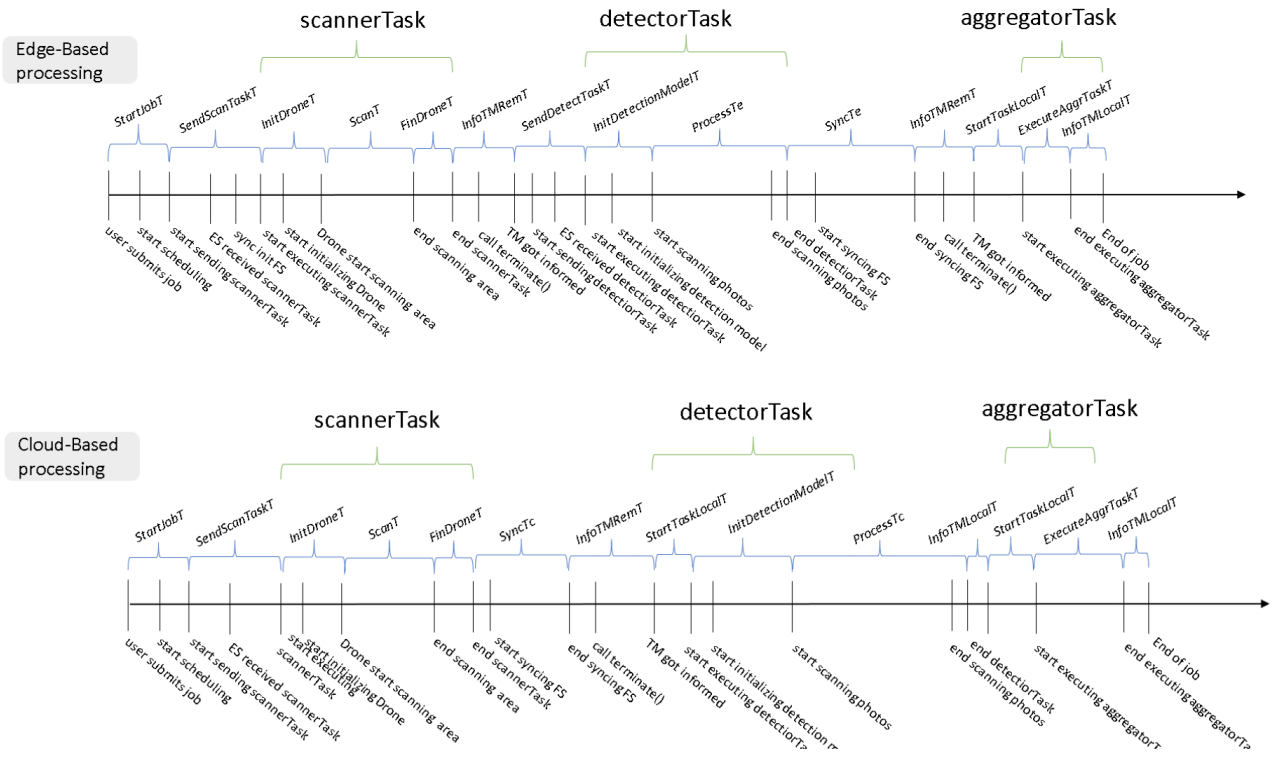


Figure 6.2: Breakdown of job execution time.

area and the distance between the points where photos are taken influence the time needed for the drone to scan a target area and the number of photos taken. Also, the detection factor affects the number of photos taken that actually contain objects of interest at each edge point, while the number of edge servers determines the total amount o processing and data transfer over the network. Finally, the image size and the network bandwidth directly affect the time it takes to complete file space synchonization.

Based on the basic parameters (measured via the base executions for the edge-based and cloud-based processing cases) and the input parameters (which can be freely given any desirable values), we calculate a set of so-called derived parameters, as follows:

The number of photos taken by a drone during scanning:

$$Photos = (Length/Step)^2 \tag{6.1}$$

The number of photos taken that actually contain one or more detected objects of interest:

$$PhotosD = Photos \times Detect \tag{6.2}$$

The time needed to scan the area of interest:

$$ScanT = Photos \times (Speed/Step) \tag{6.3}$$

Table 6.1: Basic measured delays of edge-based and cloud-based processing cases.

Symbol	Interval description
$StartJobT$	Start a job (once submitted by the user)
$StartTaskLocalT$	Start task execution on the local host
$InfoTMLocT$	Inform manager about local task completion
$InfoTMRemT$	Inform manager about remote task completion
$InitDroneT$	Arm drone, take off and fly to initial position
$FinDroneT$	Land drone and disarm
$InitDetectionModelT$	Load the cv2 detection model
$ProcessT$	Process a photo to detect objects of interest
$SendScanTaskT$	Send scannerTask to edge sever and sync files
$SendDetectTaskT$	Send detectosTask to edge sever and sync files
$ExecuteAggrTaskT$	Execute the aggregatorTask on the manager

Table 6.2: Input parameters.

Symbol	Description
$Speed$	The speed of the vehicle (m/s)
$Length$	The length of the side of the square area to be scanned
$Step$	The distance between two point of interest
$Detect$	Percentage of photos where objects of interest are detected
$Edges$	Number of target edge points (servers)
$NetBW$	The networks bandwidth (Mbits)
$ImgSize$	Image size (Mb)

The time needed to transfer a photo between the ES and TM over the network:

$$TransferT = ImgSize/NetBW \tag{6.4}$$

Then, for the edge-based processing case, the total amount of time required to process the photos taken by the drone and the total amount of time required to synchronize the file space between the edge server and the manager, is equal to:

$$ProcessT_e = Photos \times ProcessT \tag{6.5}$$

$$SyncT_e = PhotosD \times TransferT \tag{6.6}$$

Similarly, for the cloud-based processing case, these times are calculated as follows:

$$ProcessT_c = Edges \times Photos \times ProcessT \tag{6.7}$$

$$SyncT_c = Edges \times Photos \times TransferT \tag{6.8}$$

Based on the above, we can estimate the job execution time for edge-based and cloud-based processing cases for larger scales (more edge servers / zones).

Concretely, for the case of cloud-based processing, the scanning of the target areas will be executed in parallel, but all other phases will be executed sequentially. Thus, the total job execution time can be calculated as:

$$\begin{aligned} JobT_c = {} & StartJobT + \\ & (SendScanTaskT + InitDroneT + ScanT + FinDroneT + SyncT_c + InfoTMRemT) + \\ & (StartTaskLocalT + InitDetectModelT + ProcessT_c + InfoTMLocT) + \\ & (StartTaskLocalT + ExecuteAggrTaskT + InfoTMLocT) \end{aligned} \tag{6.9}$$

In the case of edge-based processing, the detection task will be executed in parallel at the edge servers and only the photos where objects of interest were detected will be transferred back to the task master, which then runs the aggregation task as usual. Thus, the job execution time is:

$$\begin{aligned} JobT_e = {} & StartJobT + \\ & (SendScanTaskT + InitDroneT + ScanT + FinDroneT + InfoTMRemT) + \\ & (SendDetectTaskT + InitDetectModelT + ProcessT_e + Edges \times SyncT_e + InfoTMRemT) + \\ & (StartTaskLocalT + ExecuteAggrTaskT + InfoTMLocT) \end{aligned} \tag{6.10}$$

Note that in both cases we make some simplifying assumptions regarding the communication that takes place between the task manager and the edge servers. More specifically, we assume that: (i) the task manager can send tasks to different edge servers in parallel, (ii) edge servers can notify the task manager about task completion in parallel, and (iii) the file transfers that take place as part of the file space synchronization are performed sequentially over the network.

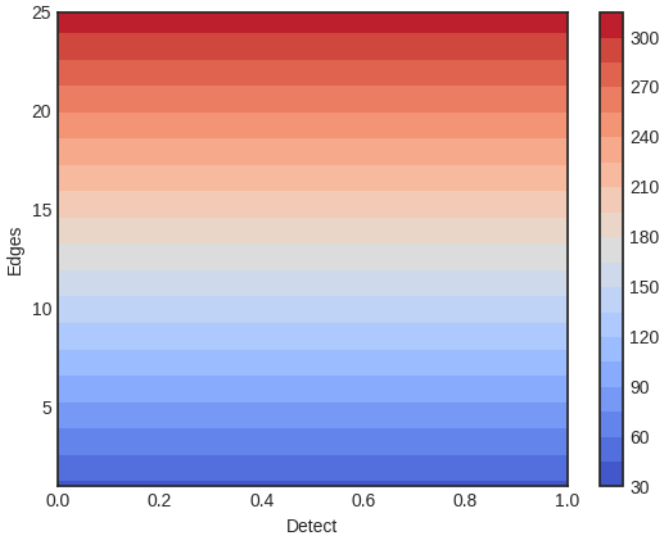
6.4 Results

We use the above formulas to compare the job execution time for the cloud-based vs edge-based processing scenario. To focus on the essence, we make some assumptions that eliminate some of the open parameters. Specifically, we assume that edge servers have a vDSL connection to the Internet with an upstream bandwidth $NetBW$ of 40 Mbits. Furthermore, we assume that the scanning of a zone is done using a single drone flying at a $Speed$ of 5 m/s taking a photo at a $Step$ of 20 meters. Finally, assuming that a drone can fly autonomously for about 30 minutes (covering a distance of about 9 Km), the total number of $Photos$ at each edge point (zone) is 450.

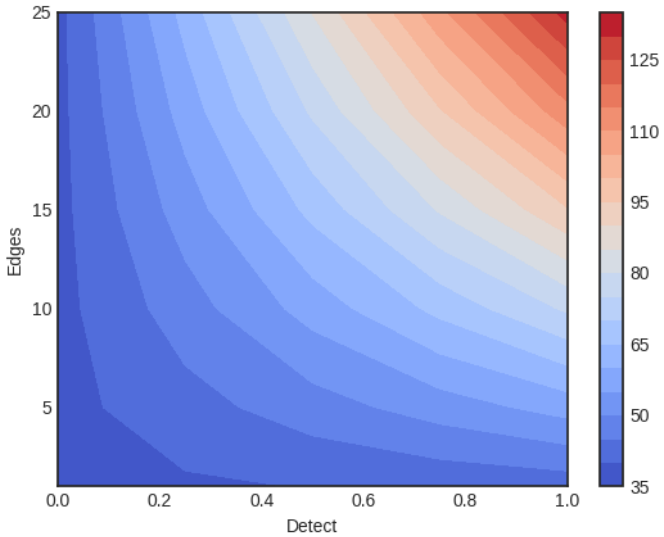
Figure 6.3a shows the job execution time for the cloud-based processing case as a function of the number of edge points ($Egdes$) and detection percentage ($Detect$), based on Equation 6.9. Naturally, the job time increases for a larger number of edge points as this leads to an larger number of photos that need to be transferred from the edge to the cloud and also need to be processed there. Note that the job execution time is not affected by the percentage of photos where objects of interest are detected, as this has no effect on the data transfers from edge to cloud and the amount of processing that needs to be performed there (the $Detect$ parameter does not appear in the job execution equation for the cloud-based processing case).

Figure 6.3b shows the job execution time for the edge-based processing case as a function of the number of edge points ($Egdes$) and detection percentage ($Detect$), based on Equation 6.10. In this case, both parameters play a key role in determining the time that is needed to complete the job, as they both affect the data traffic between the edge and cloud and the amount of processing that needs to be performed in the cloud.

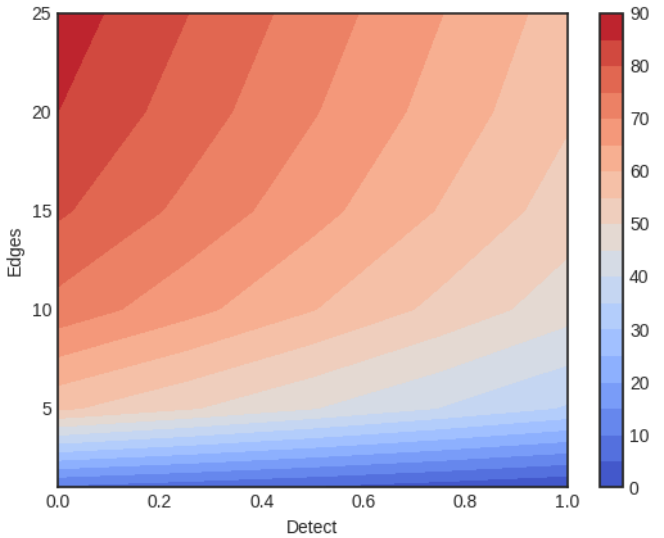
Finally, Figure 6.3c shows the relative improvement of the job execution time for the edge-based vs. cloud-based processing case. It can be clearly seen that edge-based processing can significantly reduce the job execution time. The difference is already notable even for a very small number of edge points, where edge-based processing can lead to a reduction of more than 20% and up to 50% for low detection percentages. The reduction becomes substantial for a larger number of edge points, from 50% to 90% depending on the percentage of photos with objects of interest.



(a) Job execution time for cloud-based processing (minutes).



(b) Job execution time for edge-based processing (minutes).



(c) Reduction of execution time achieved through edge-based processing (%).

Figure 6.3: Cloud-based vs edge-based job execution.

Chapter 7

Conclusion & Future Work

We have presented a hybrid cloud-edge computing system for a parallel-independent execution at edge nodes and data aggregation at the cloud. With a shell-like interface, the user can use issue commands to run drone-based data collection and processing jobs in a map-reduce manner. Edge nodes can process data that are collected on-the-fly from drones, while the user receives a live report of the execution.

To support this functionality, we created a distributed master-slave architecture in which entities share a file space. Also, nodes discover each other using a simple register policy implemented by a specific server-process. Finally, we evaluate our system using an estimation model that we created and found out that the scaling of a system via the expansion of the edge nodes can have a significant reduction to a job's time.

The present system can be extended in several ways. One direction is to extend our system to support fault-tolerance so that, in case the Task Manager crashes while a job is running, the Edge servers continue the task execution, buffer the respective log entries temporarily and postpone the file synchronization process, until the Task Manager recovers. Additionally, the synchronisation of the file spaces can be made asynchronous to task execution, so that the underlying file transfers start concurrently to task execution in order to further reduce the file synchronization delay. Last but not least, it would be important to add support for live streaming between drone-tasks and tasks running at the edge and cloud via suitable data forwarding abstractions and mechanisms.

Bibliography

[1] Qiang Ren, Rongde Zhang, Wanlin Cai, Xinfeng Sun, and Limeng Cao. Application and development of new drones in agriculture. *In IOP Conference Series: Earth and Environmental Science. IOP Publishing*, 440(5):052041, February 2020.

[2] van der Merwe D., Burchfield D. R, Witt T. D., Price K. P., and A Sharda. Drones in agriculture. *Advances in Agronomy*, 162:1–30, 2020.

[3] Vishal Sharma, Ilsun You, Giovanni Pau, Mario Collotta, Jae Deok Lim, and Jeong Nyeo Kim. Lorawan-based energy-efficient surveillance by drones for intelligent transportation systems. *Energies*, 11(3):573, 2018.

[4] Daniel Câmara. Cavalry to the rescue: Drones fleet to help rescuers operations over disasters scenarios. In *2014 IEEE Conference on Antenna Measurements Applications (CAMA)*, pages 1–4, 2014.

[5] Francesco Flammini, Concetta Pragliola, and Giovanni Smarra. Railway infrastructure monitoring by drones. In *2016 International Conference on Electrical Systems for Aircraft, Railway, Ship Propulsion and Road Vehicles & International Transportation Electrification Conference (ESARS-ITEC)*, pages 1–6. IEEE, 2016.

[6] Guobao Xu, Weiming Shen, and Xianbin Wang. Applications of wireless sensor networks in marine environment monitoring: A survey. *Sensors*, 14(9):16932–16954, 2014.

[7] David W. Johnston. Unoccupied aircraft systems in marine science and conservation. *Annual Review of Marine Science*, 11(1):1–4, 2019.

[8] Keila Lima, Eduardo RB Marques, José Pinto, and Joao B Sousa. Dolphin: a task orchestration language for autonomous vehicle networks. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 603–610. IEEE, 2018.

[9] Roberto Casadei, Gianluca Aguzzi, and Mirko Viroli. A programming approach to collective autonomy. *Journal of Sensor and Actuator Networks*, 10(2):27, 2021.

[10] Koutsoubelias M. and Lalis S. Tecola: A programming framework for dynamic and heterogeneous robotic teams. In *Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, pages 115–124, 2016.

[11] Irlock target tracking system. `https://irlock.com`.

[12] M. Koutsoubelias and S. Lalis. Coordinated broadcast-based request-reply and group management for tightly-coupled wireless systems. In *IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1163–1168, 2016.

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[14] Hdfs architecture guide. `http://hadoop.apache.org/docs/r1.0.4/hdfs_design.html`. Date of visiting site: 10-06-2021.

[15] Spark home page. `https://spark.apache.org/`. Date of visiting site: 10-06-2021.

[16] M. Zaharia, Chowdhury, M. J. M., Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[17] Spark standalone mode. `https://spark.apache.org/docs/latest/spark-standalone.html/`. Date of visiting site: 27-06-2021.

[18] Yarn home page. `https://yarnpkg.com/`. Date of visiting site: 27-06-2021.

[19] Apache mesos home page. `http://mesos.apache.org/`. Date of visiting site: 27-06-2021.

[20] Kubernetes. Production-Grade Container Orchestration. `https://kubernetes.io/`, 2021.

[21] Pyro - python remote objects - 4.80. `https://pyro4.readthedocs.io/en/stable/`. Date of visiting site: 19-06-2021.

[22] Docker home page. `https://docs.docker.com/`. Date of visiting site: 15-06-2021.

[23] DroneKit. Developer tools for drones. `http://dronekit.io/`, 2021.

[24] ArduPilot. Open source autopilot. `http://ardupilot.org`, 2021.

[25] MAVLink. Drone communication protocol. `https://mavlink.io`, 2021.

[26] ArduPilot. SITL Simulator. `http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html`, 2021.

[27] Yolo: Real-time object detection. `https://pjreddie.com/darknet/yolo/`. Date of visiting site: 05-09-2021.

Chapter

Task Scripts

In this appendix we are listing the source codes of the tasks we used for our experiments.

1 Script of the scannerTask

The scannerTask is listed in .1. This is an edge task, that runs a drone mission for scanning an area of interest by taking photos. The input arguments of this task are: (i) the name of the drone, (ii) the path of the directory where captured photos will be stored, and (iii) path to the file in which are stored the missions attributes (altitude, length of the square area to be scanned, distance between the photos to be taken, the initial waypoints)

```
1  !$EDGE
2  #input drone outputdir inputfile
3
4  # functions for calculating new waypoints
5  OFFSET = 111111
6
7  fun GetNewLat(lat, distance, direction):
8      nLat = lat + (distance / OFFSET)*direction
9      return nlat
10
11 fun GetNewLon(lon, distance, lat, direction):
12     convert_rad = (lat*PI)*180
13     nlon = lon + (distance/ (OFFSET*(cos(convert_rad))))*direction
14     return nlon
15
16 # mission code
```

```
17
18 with open(inputfile, 'w') as f:
19     alt = f.readline()
20     length = f.readline()
21     step = f.readline()
22     lat, lon = (f.readline(), f.readline())
23
24 os.makedirs(outputdir)
25 path_output_file = outputdir + 'waypoints_file.txt'
26 ofile = open(path_output_file, 'w')
27
28 logger.info('started')
29 n = mco.group.getNodeByNameStr(drone)
30
31 n.MobilitySvc.Arm()
32 logger.info('waiting to arm')
33 mco.wait([n.MobilitySvc.GetArmedStatus,'==','ARMED'],1,20)
34
35 n.MobilitySvc.TakeOff(alt)
36 logger.info('waiting to take off')
37 mco.wait([n.MobilitySvc.getDistanceFromTakeOffAlt,'<',1.0],1,20)
38
39 n.MobilitySvc.GotoWaypoint(lat,lon,10.000000)
40 logger.info('waiting to go to init position')
41 mco.wait([n.MobilitySvc.getAirSpeed,'>',0.8],1,300)
42 mco.wait([n.MobilitySvc.getDistanceFromTarget,"<",1.0],1,300)
43
44 direction = 1
45 possition = 0
46 steps = length/step
47
48 # start scanning
49 for x in range(0, steps):
50     for y in range(0, steps):
51         lat = getNewLat(lat, step, direction)
52
53         n.MobilitySvc.GotoWaypoint(lat,lon,10.000000)
54         logger.info(f'waiting to go to({lat},{lon})')
55         mco.wait([n.MobilitySvc.getAirSpeed,'>',0.8],1,300)
```

```
56          mco.wait([n.MobilitySvc.getDistanceFromTarget,'<',1.0],1,300)
57
58          logger.info("Taking photo")
59          n.MyCameraSvc.TakePicture('tmp/photo_'+str(possition))
60
61          ofile.write('%d = (%f, %f)'.format(possition, lat, lon))
62
63          possition++
64
65  lon = getNewLon(lat, lon, step, 1)
66  direction = -direction
67
68  # end of scanning
69  ofile.close()
70  n.MobilitySvc.Land()
71  logger.info('waiting to disarm')
72  mco.wait([n.MobilitySvc.GetArmedStatus,'==','DISARMED'],1,60)
```

Listing .1: script code of scannerTask

2 Script of the detectorTask

The detectorTask is listed in .2. This is an edge task, which processes the photos gathered from a drone mission in order to examine if there are objects of interest. The input arguments of this task are: (i) the path of the directory where input photos are stored, and (ii) the path of the directory where photos of interest will be stored

```
1  !$EDGE
2  #input inputdir outputdir
3
4  LABELS = open('names.txt').read().strip().split('\n')
5  logger.info('loading YOLO from disk...)
6  net = LoadNetwork('yolo.cfg', 'weights.cfg')
7
8  listOfImages = os.listdir(inputdir)
9  ofile = open('CountResults', 'w')
10
11 t1=time.time()
12 # start processing images
```

```
for im in listOfImages:
    image = cv2.imread(inputdir+'/'+im)

    start = time.time()
    detection_results = forwardPhoto(image)
    end = time.time()

    if len(detection_results.detectedObjects()) > 0:
        new_image = 'result-'+ im + '.jpg',image
        reateDetectedObjects(detection_results, new_image).saveImg()

    ofile.write(im+':'+str(len(idxs))+'\n')
# end processing images
ofile.write('Execution time:'+str(time.time()-t1))
ofile.close()
```

Listing .2: script code of detectorTask

3 Script of the aggregatorTask

The aggregatorTask is listed in .3. This is a manager task, that displays to the user the combination of the results. The input arguments of this task are, the names of the zones file space.

```
!$MANAGER
#input zone1 zone2

output_file1 = f'zone1/CountResults'
output_file2 = f'zone2/CountResults'

counter = 0
with open(output_file1, 'r') as f:
    lines = f.readlines()
    for line in lines:
        counter = counter + line.split(':')[1]

with open(output_file2, 'r') as f:
    lines = f.readlines()
    for line in lines:
        counter = counter + line.split(':')[1]
```

```
17
18 print('We detected %d objects' %counter)
```

Listing .3: script code of aggregatorTask