UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ADAPTIVE DEPLOYMENT AND MOBILITY OF CONTAINERIZED SERVICES ON THE EDGE

Diploma Thesis

Foivos Pournaropoulos

Supervisor: Christos D. Antonopoulos

Volos, September 2022

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ADAPTIVE DEPLOYMENT AND MOBILITY OF CONTAINERIZED SERVICES ON THE EDGE

Diploma Thesis

Foivos Pournaropoulos

Supervisor: Christos D. Antonopoulos

Volos, September 2022

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΠΡΟΣΑΡΜΟΣΤΙΚΉ ΑΝΆΠΤΥΞΗ ΚΑΙ ΜΕΤΑΚΊΝΗΣΗ CONTAINERIZED ΥΠΗΡΕΣΙΏΝ ΣΤΟ ΆΚΡΟ ΤΟΥ ΔΙΚΤΎΟΥ

Διπλωματική Εργασία

Φοίβος Πουρναρόπουλος

Επιβλέπων/πουσα: Χρήστος Δ. Αντωνόπουλος

Βόλος, Σεπτέμβριος 2022

Approved by the Examination Committee:

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

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

Member **Nikolaos Bellas**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Acknowledgements

I would like to express my gratitude to Prof. Christos D. Antonopoulos for his priceless guidance throughout my undergraduate studies and Prof. Spyros Lalis for his invaluable help for the completion of my Thesis. Finally, a special thanks to my family for the continuous support in order to achieve my goals.

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 also declare that the results of the work have not been used to obtain another degree. 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

Foivos Pournaropoulos

Diploma Thesis

ADAPTIVE DEPLOYMENT AND MOBILITY OF CONTAINERIZED SERVICES ON THE EDGE

Foivos Pournaropoulos

Abstract

Modern applications tend to embrace the microservices approach, which, in combination with the proliferation of the edge computing paradigm over the past years, create the need for dynamic resource allocation and energy efficiency. An additional main concern is to provide applications with high availability and low response time. Moreover, the mobility of the user or vehicle that utilizes edge computing intensifies the uncertainty and provokes a plethora of challenges to guarantee to the application high quality of service and uninterrupted operation.

In this work, we propose an extension to an existing orchestration framework for the Drone-Edge-Cloud continuum, to provide flexible and dynamic deployment of application components at the edge. We present the design and implementation of the proposed extension in detail, and discuss and differentiate previous research in related fields from our extended framework. In the experimental evaluation, we use a realistic lab-based distributed cluster setup to present the main benefits of edge computing as well as the potential and limitations of the proposed mechanism. We examine the different aspects of this study through system-level and application-level metrics.

Keywords:

Adaptive Deployment, Service Migration, Edge Computing

Διπλωματική Εργασία

ΠΡΟΣΑΡΜΟΣΤΙΚΉ ΑΝΆΠΤΥΞΗ ΚΑΙ ΜΕΤΑΚΊΝΗΣΗ CONTAINERIZED ΥΠΗΡΕΣΙΏΝ ΣΤΟ ΆΚΡΟ ΤΟΥ ΔΙΚΤΎΟΥ

Φοίβος Πουρναρόπουλος

Περίληψη

Οι σύγχρονες εφαρμογές τείνουν να υιοθετούν την προσέγγιση των microservices, η οποία, σε συνδυασμό με την ταχεία ανάπτυξη του προτύπου υπολογισμού στα άκρα του δικτύου (edge computing), δημιουργούν την ανάγκη για δυναμική δέσμευση των υπολογιστικών πόρων και ενεργειακή αποδοτικότητα. Ένας επιπρόσθετος κύριος προβληματισμός είναι να παρέχουμε στις εφαρμογές υψηλή διαθεσιμότητα και χαμηλό χρόνο απόκρισης. Επιπλέον, η κίνηση του χρήστη ή του οχήματος που χρησιμοποιεί το edge computing εντείνει την αβεβαιότητα και προκαλεί πληθώρα προκλήσεων ώστε να εγγυηθεί στην εφαρμογή υψηλή ποιότητα υπηρεσιών και αδιάκοπη λειτουργία.

Σε αυτή την εργασία, προτείνουμε μία επέκταση σε ένα υπάρχον πλαίσιο λογισμικού (framework) ενορχήστρωσης, στο Drone-Edge-Cloud συνεχές, για να παρέχουμε ευέλικτη και δυναμική ανάπτυξη των μερών της εφαρμογής στο άκρο του δικτύου. Παρουσιάζουμε το σχεδιασμό και την υλοποίηση της προτεινόμενης επέκτασης σε λεπτομέρεια, καθώς και συζητούμε και διαφοροποιούμε την προϋπάρχουσα έρευνα σε σχετικούς τομείς από το δικό μας εκτεταμένο framework. Κατα την πειραματική αξιολόγηση, χρησιμοποιούμε μία ρεαλιστική, βασισμένη στο εργαστήριο, κατανεμημένη ομάδα από υπολογιστικά συστήματα (cluster) για να παρουσιάσουμε τα κύρια ωφέλη του edge computing καθώς και την προοπτική και τους περιορισμούς που θέτει ο μηχανισμός. Εξετάζουμε τις διαφορετικές πτυχές αυτής της μελέτης μέσω μετρικών σε επίπεδο συστήματος και εφαρμογής.

Λέξεις-κλειδιά:

Προσαρμοστική Ανάπτυξη, Μετακίνηση Υπηρεσιών, Υπολογισμός στο ’Ακρο του Δικτύου

Table of contents

List of figures

List of tables

Abbreviations

API	Application Programming Interface
CRD	Custom Resource Definition
IoT	Internet of Things
P2P	Peer-to-Peer
RPi	Raspberry Pi
VIP	Virtual IP
VM	Virtual Machine
VPN	Virtual Private Network
YAML	YAML Ain’t Markup Language

Chapter 1

Introduction

The edge computing paradigm enhances application's quality of service (QoS) by moving computations to the edge of the network, closer to the point where the data are produced. This provides applications with the opportunity of lower communication latency compared with communicating with a remote cloud server. Edge computing also favors the overall system performance and stability, as it reduces the data sent to the cloud resulting in bandwidth savings. Furthermore, new missions and challenges arise in this field, especially with the proliferation of self-driving cars or unmanned aerial vehicles, which are characterized by time-sensitive demands and also amplify cluster heterogeneity, rendering the overall system's flexibility and adaptation capability crucial.

With the evolution of edge computing, a plethora of edge-cloud applications arise, which consist of components distributed across a set of nodes in the Edge-Cloud continuum. Thus, the monolithic approach is not a proper paradigm for many modern applications. Quite often, such applications adopt the microservices paradigm, in which an application is organized as several independent entities which cooperate with each other to produce the final result or functionality. Microservices-based applications solve real world problems with flexibility and efficiency. The code implementing microservices is usually packaged and deployed in the form of containers. Containers provide efficiency as they execute on top of the host's Operating System and are, therefore, a lightweight solution for hosting microservices.

To exploit the benefits of edge computing for applications organized as microservice chains, there is an emerging need to enable orchestration and adaptivity at runtime.

1.1 Subject

In this Thesis, we design and implement a flexible application deployment mechanism in the Drone-Edge-Cloud continuum based on the Fractus orchestration framework [1], [2]. Fractus handles automated application deployment and management, to realize a system that adapts to the user's requirements as well as node mobility at runtime. The aforementioned framework manages applications following the microservices paradigm as independent containers that coexist and communicate to perform real-world missions. Also, it leverages direct communication opportunities with edge nodes to transparently exploit the aforementioned advantages of edge computing, while at the same time enabling high availability for the application.

However, Fractus deploys all instances of a given component at once, thus having a subset of idle components during the operation. In this Thesis, we extend Fractus to support just-in-time provisioning, as well as deployment adaptivity at runtime. In order to improve the overall system's resource utilization and energy efficiency, we design a solution in which the Fractus-related entities are communicating and cooperating, while having a single point of management information which facilitates keeping track of potential cluster evolution and enables hosting only the necessary application components at any given time. In addition, we enable Fractus to readjust due to node mobility and to modify the deployment plan based on various triggering scenarios.

1.2 Contributions

The contributions of this Thesis can be summarized as follows:

1. Extended the design and implementation of Fractus, to support different scenarios of adaptive deployment at runtime.

2. Design and execution of a comprehensive experimental evaluation, using a realistic lab-based cluster for a mobile application and dynamic component migration scenario, to quantify the overheads and identify weaknesses of our mechanism.

3. Experimental quantification of the advantages of Edge Computing in terms of reduced latency and increased bandwidth compared with the mobile-to-cloud communication.

4. Validation of the benefits of deployment flexibility via application-related metrics.

1.3 Thesis structure

The rest of this Thesis is organized as follows: Chapter 2 introduces the Fractus framework, which serves as the basis of this work. Chapter 3 describes the architecture and the design aspects of our extended framework. Chapter 4 presents detailed insights on the implementation. Chapter 5 focuses on the experimental validation and evaluation of the system on a distributed cluster. Chapter 6 provides a review of previous work. Finally, Chapter 7 summarizes the most valuable results and the contributions of this Thesis, while proposing various directions for possible improvements.

Chapter 2

Background

In this chapter, we introduce the most significant aspects of Docker and Kubernetes, technologies required for this work. Afterwards, we present the Fractus framework, which we use as a starting point of this Thesis to introduce the desired deployment adaptivity at runtime.

2.1 Docker

Docker [3] is a platform used to create, deploy and run applications in an isolated and secure environment, in the form of containers [4]. Containers enable faster development and deployment of modern applications following the microservices approach and provide portability, in terms of packaging, as they can run on a large variety of systems regardless of their unique peculiarities. Moreover, containers provide a virtualization technology which renders them more lightweight comparing with Virtual Machines [5], as the former use the host's Operating System, thus improving the system's overall performance and lowering the footprint of the enclave.

2.2 Kubernetes

Kubernetes [6] is a container orchestration platform useful for application deployment and monitoring in an automated fashion. Referring to the architecture, every Kubernetes cluster has at least one control plane to supervise the cluster state, so that it can be as "close" as possible to the user's demanded state. Also, the cluster consists of several worker nodes that communicate with the control plane via a well-defined API. To this end, the API Server is an

essential entity of the control plane, responsible for managing all of the API-related tasks.

A Kubernetes node needs to host an agent called *kubelet* in order to maintain the desired Pods in a running state. A Kubernetes Pod is the system's smallest deployable computing entity. In addition, nodes host the Kubernetes network proxy, an entity that manages Pod communication by using packet redirection.

To achieve the desired result, every application description is deployed to the API server as one or more YAML files, consisting of all the needed parts for a successful operation. Each independent part of the application is deployed as a Kubernetes Pod which hosts the necessary containers.

2.3 Fractus

2.3.1 Overview

Fractus [1] [2] is a framework "built" on top of Kubernetes, responsible for the deployment and coordination of distributed applications in the form of containerized microservices in the Drone-Edge-Cloud continuum.

The user provides an application specification to Fractus, describing the application components, their desired placement on the different types of nodes, the possible interactions between them and their resource requirements for a successful deployment. Subsequently, Fractus decides the deployment plan based on the aforementioned description and deploys the components in alignment with the user's demands, if possible at the moment of the request. Each application component is deployed as a Kubernetes Pod, running the respective container with the desired functionality. In our case, every Pod hosts one and only container. The user can also specify how many instances are needed for each component.

In addition, applications registered to Fractus have the flexibility to let their components use of two types of system services, which can be hosted by each node with respective capabilities: the *mobility* service to handle the node's movement and the *camera* service used for image capturing and processing. As a result, Fractus can be used for experimentation with a variety of real-world tasks which require mobile sensors.

Fractus also provides handling of drone-specific operation. This is accomplished by using information such as the regions of interest as well as the forbidden zones in which the drone lacks the permission to use the *camera* or the *mobility* services.

2.3.2 Design

The key operational units for the deployment process are Fractus *Controller*, *Monitor*, *Agent* and *net-proxy* components.

The *Controller* inspects the application's deployment requests and decides the nodes that host each component. In order for the deployment to be accomplished, the *Controller* considers information such as the cluster nodes' status, the resource requests by each Kubernetes Pod and the communication relations between components. Consequently, the *Controller* starts the *Monitor* which periodically performs health-checks for the running components.

Furthermore, every node of the cluster hosts a Fractus *Agent* instance in order to periodically update state-related information to the Kubernetes registry, such as the node type, the location, and the available resources. The Fractus *net-proxy* components are used to perform data redirection and exploit opportunities for direct peer-to-peer (P2P) communications between the application components.

As for the application components, there are four basic categories based on the type of the nodes which host them:

1. *Drone* component, which is hosted only by a drone.
2. *Hybrid* component, meaning that it can run both on cloud and edge nodes.
3. *Edge* component, hosted only by static nodes at the edge of the network.
4. *Cloud* component, which resides in nodes in the cloud.

As mentioned before, the application model follows the microservice architecture paradigm, in the form of containers. As a result, a subset of the registered application components need to use several services, for the purposes of any given application. For example, a component placed on a drone could send the captured pictures to a component at the network edge in order to process them and make a decision with the results. Each one of these services is provided with a Virtual IP (VIP) from Kubernetes.

In our study, we primarily focus on *hybrid* components, which are used for the evaluation of the deployment adaptivity in one of the possible cases, namely the component migration between cloud and edge nodes. For the *hybrid* components, the Fractus *Controller* chooses the most suitable host based on a metric provided by the user at the time of application registration to Fractus. For example, some indicative metrics would be the maximum communication

bandwidth, the minimum communication latency, or the preference of communication with the edge nodes over WiFi rather than with the cloud nodes using a 4G internet connection. However, Fractus always deploys instances of such components on all selected candidate nodes, even if some of those instances may be required only for a short amount of time during the application's lifetime.

2.3.3 Implementation

Every necessary part of the cluster, such as the different types of nodes, drones, applications and their components are represented via Custom Resource Definitions (CRDs). The lifecycle of the above resources, through specific operations, is controlled by the Kubernetes API server.

The Fractus *Controller* watches for new application descriptions to be registered to Kubernetes, creates the respective Kubernetes custom resources and services and deploys the distributed components as independent Kubernetes Pods. When the deployment phase is finished, the *Controller* spawns the Fractus *Monitor* thread to check the execution status of the application's components.

If multiple instances of the same component are deployed, the service-invoking components are able to communicate with a single service-providing component by applying the respective ingress and egress rules to allow traffic to or from the appropriate instance. Every distinct application service is granted a unique VIP from Kubernetes and all the instances of the same service provider have the same VIP.

Chapter 3

Design-Architecture

In this chapter, we discuss the objective of the Thesis, the assumptions made during this work and our extended design and architecture compared with the Fractus prototype. We will refer to our extension as Fractus++.

3.1 Objective

Figure 3.1 illustrates the architecture of a model cluster used as our running example. The cluster consists of two edge nodes, one cloud node, which also hosts the control plane, and one mobile node. All the nodes are connected over a VPN, each one of them using an Ethernet interface, except from the mobile node which employs a 4G internet connection. In order to reduce the application-level response time we exploit direct WiFi channels for mobile-to-edge communication.

Our primary goal is to bring the computations closer to the mobile node, thus exploiting the communication but also computing advantages that edge computing offers, such as the lower latency, the higher bandwidth as well as potentially faster or underutilized computing resources. Another main concern is to ensure the application's high availability with minimal downtime. In addition, we aspire to improve the overall resource utilization by adding adaptivity to the application deployment at runtime, as we deploy, migrate, and remove application components based on user-provided metrics combined with the dynamic topology of the member nodes. We mainly focus on distributed applications running on a set of edge, cloud and mobile nodes.

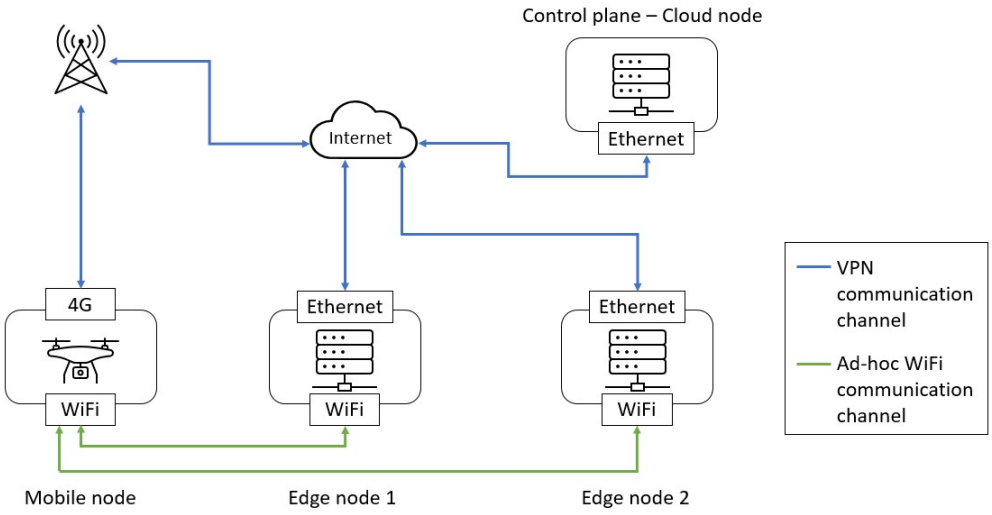


Figure 3.1: Cluster connectivity

3.2 Assumptions

The basic assumptions made in this work are the following:

- The communication between the cluster components is reliable and secure.
- Application components do not fail.
- Any communication-related errors between the application components are handled as needed at the application-level to provide the desired functionality.

3.3 Deployment adaptivity

Modern applications have increased demands in terms of executing and monitoring their desired state to guarantee successful operation. Furthermore, real-world tasks in the form of distributed applications need to reconfigure their component placement based on various events which are not known *a priori*.

In this study, we render the Fractus++ framework more versatile by (i) providing the user with the flexibility to perform application-specific modifications during runtime, (ii) automatically adapting the application deployment due dynamics in resource availability or mobility

of application components, as well as (iii) letting the application's components themselves lead to a revision of the application's deployment plan.

More specifically, the initial application description can be replaced with another modified version by the user to reconfigure its characteristics. If the user needs the application to be self-operating, this can be achieved by having the application modify its description to Kubernetes programmatically. The above choices refer to the user's own demands to rearrange the deployment.

Fractus++ also supports adaptation based on cluster status. If a new node joins the cluster and directly affects the application, meaning it can satisfy the application's needs, the deployment will automatically change to include the new node to the plan at any given moment during operation. Additionally, an application consisting of a mobile node can exploit the benefits of edge computing by offloading several application components from the cloud to edge devices, if needed.

To accomplish adaptivity, the deployment phase occurs in a repetitive way as it can be triggered by the aforementioned events.

3.4 Fractus++ custom resources

We introduce a new mobile node custom resource, which refers to a node type capable of moving and changing its position in the physical space. The motivation for the above addition is to test and evaluate our mechanism in several component migration scenarios, where the mobile node enters or leaves the proximity range of one or more edge nodes.

3.5 Fractus++ entities

The responsibilities of the main Fractus++ entities are the following:

1. Fractus++ *Controller*: In the initial deployment case, it is responsible for the candidate nodes filtering, the host selection and the component-to-node mapping, as it was on the prototype version as well. The new capabilities of the *Controller* are the following:

    - The *Controller* is continuously informed about cluster status or application modifications and performs the necessary adaptation to satisfy user's demands. During

the adaptive deployment stage, the unused Pods are removed from their hosts and the new ones are deployed to the selected hosts.

- When the user requests the deletion of an application, the *Controller* removes and cleans all running Pods from the cluster.

2. Fractus++ *Monitor*: The *Monitor*'s main responsibility is to periodically check for any updated or new Fractus++ custom resources. Whenever this occurs, the *Monitor* inspects the application description, in order to decide if the corresponding update leads to a change in the current state of the cluster and, as a consequence, the *Controller* must be notified. Also, the *Monitor* periodically checks the status of the running Pods.

3. Fractus++ *Agent*: Same as in the prototype, the *Agent* placed on cloud, edge or mobile nodes, keeps track of the node's state and resource availability and sends updates to the Kubernetes registry. When the *Agent* is hosted by a mobile node, apart from the aforementioned, it also retrieves and sends to the Kubernetes registry its coordinates periodically.

4. Fractus++ *net-proxy*: It is responsible for the redirection of the application's traffic through a direct WiFi channel instead of the conventional one, if the mobile node is within the proximity range of an edge node and the edge node has direct communication capabilities. The Fractus++ *net-proxy* running on the edge nodes, behaves as an access point and waits for new connections, while the same entity running on the mobile node receives notifications on the availability of Ad-Hoc connection opportunities from the *Controller*.

3.6 Node-level requirements

Figure 3.2 depicts the software stack of different types of cluster nodes. The cloud node runs the control plane for both Kubernetes and Fractus++, thus hosting the Kubernetes API Server and the Fractus++ *Controller* and *Monitor* entities. The rest of the nodes host the *kubelet* agent. Also, each node hosts the Fractus++ *Agent* and the essential container runtime software to be able to execute the desired application components in the form of containers. The edge as well as the mobile nodes host a *net-proxy* component to exploit direct communication opportunities that may occur at runtime.

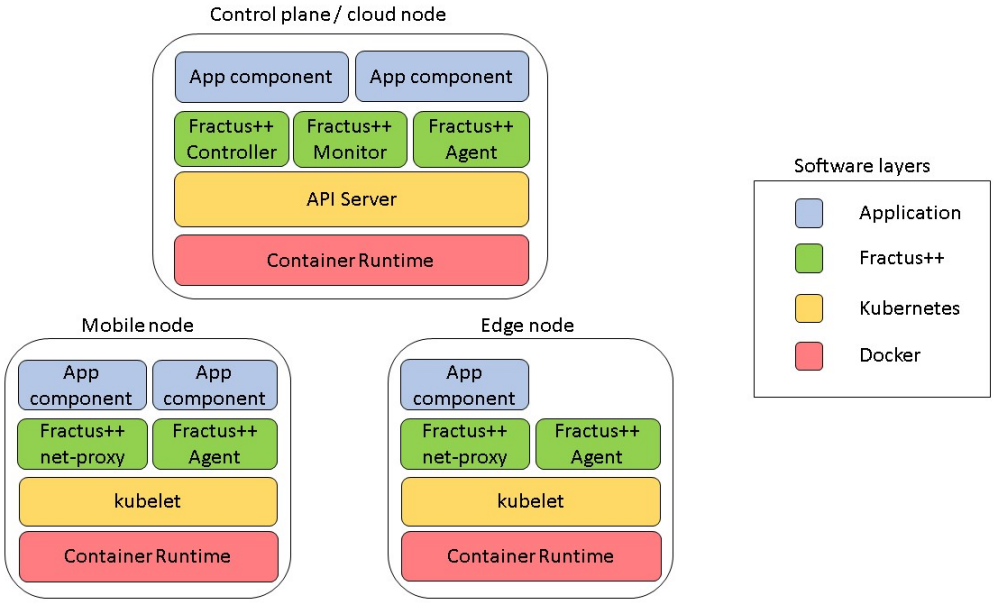


Figure 3.2: Software stack

3.7 Application deployment

Figure 3.3 presents a typical application deployment request to Fractus++. The *Controller* is notified by Kubernetes when a new application has been registered. The application description is stored to internal data structures and translated to various dependencies between the involved components. The *Controller* updates the cluster's resources and it then builds an initial deployment plan. The latter also spawns the *Monitor* to start monitoring for resource updates and the status of the already deployed Pods.

3.8 Adaptation triggering scenarios

In Fractus++, we consider various modifications in the application's deployment plan which can be initiated from the user, the application itself, or by other external factors. More specifically, the possible rearrangement-triggering scenarios supported are the following:

- Movement of an application component
- Introduction of a new node in the cluster
- Application description modification by the user or by the application itself

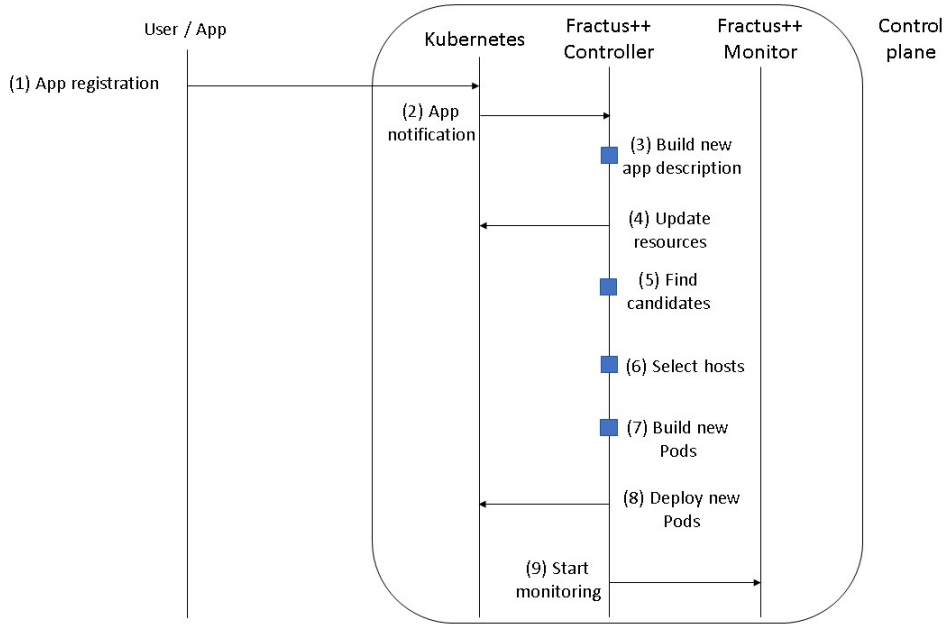


Figure 3.3: Application registration by the user - control flow

To visualize the control-flow process, we present an explanatory figure for each of the scenarios.

Figure 3.4 depicts the main scenario we evaluate, thus providing more details compared with the other adaptation scenarios. The mobile node updates its coordinates periodically to Kubernetes. When the node enters the proximity range of an edge node, the *Monitor* detects the possible need for migration and notifies the *Controller* to adjust the current deployment, if it may lead to a more efficient setup for the application. In contrast with the first registration case, the *Controller* also removes the unused Pods. In addition, an alternative scenario for this case would be a new node introduction to the cluster.

In Figure 3.5, we present another scenario for readjustment, when the user or even a component of the application decides to adjust the current application description. For example, the user or the application component may modify the region of interest for the application, meaning that new nodes would be candidates to host a subset of the application components, or several hosts would no longer be needed. The *Controller* spots the differences between the outdated and the updated description and reconfigures the plan by repeating the deployment procedure.

A key observation for the extended functionality is the fact that we keep a single point of information storage, which is the Kubernetes registry. Every time a Fractus++ entity needs to update a custom resource, the invocation results to the API Server.

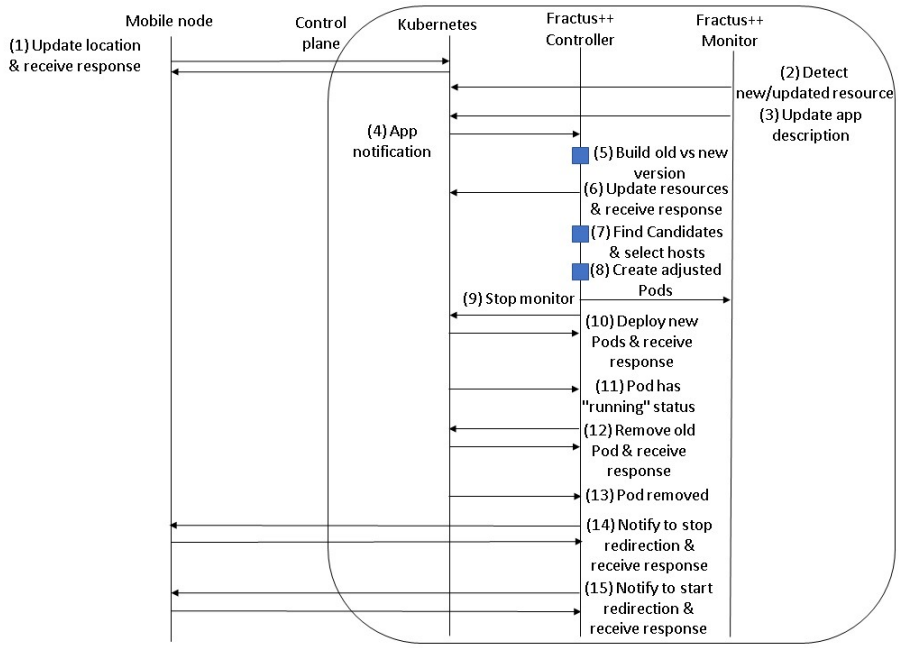


Figure 3.4: Adaptive deployment due to node movement - control flow

3.9 Application removal

In addition to the Fractus prototype, our extended framework supports application removal by the user, as shown below. In Figure 3.6 the *Controller* is notified for the application removal, it retrieves the needed stored information about the current deployment plan and it deletes all application modules.

3.10 Data-traffic redirection

When a *hybrid* component is moved to one of the edge nodes equipped with a WiFi interface, the possibility of direct communication with the application components hosted on mobile nodes is emerging to optimize the system's overall performance. The Fractus++ *net-proxy* entity is responsible for redirecting the data traffic to pass through the direct WiFi channel instead of using the default internet connection between the interacting components. To leverage this potential, a *net-proxy* instance is running on all of the edge and mobile nodes.

Figure 3.7 illustrates the process of mobile node connection to the direct channel that the edge node provides. The mobile node is notified from the *Controller* to establish the direct connection with the edge and it then sends to the latter the service component's name and the port number. When the edge node has modified its routing rules it responds to the mobile

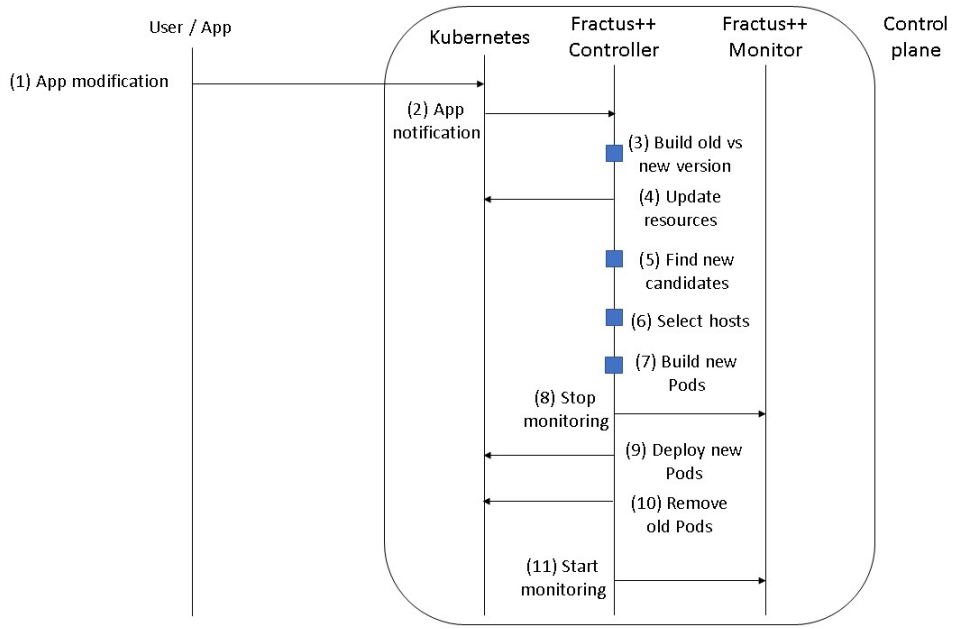


Figure 3.5: Application modification by itself or by the user - control flow

node to let it update its own rules.

Figure 3.8 outlines the reverse procedure, when a direct communication link must be destroyed. Same as in the connection case, the *Controller* initiates the disconnection process by informing the mobile node. The latter restores the routing rules previously modified to support direct connection and notifies the edge node to follow the same course of action.

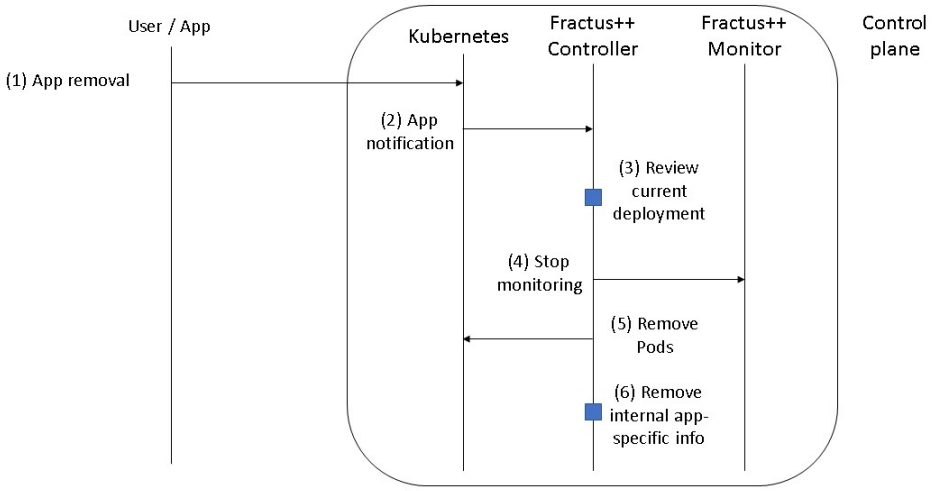


Figure 3.6: Application removal by the user - control flow

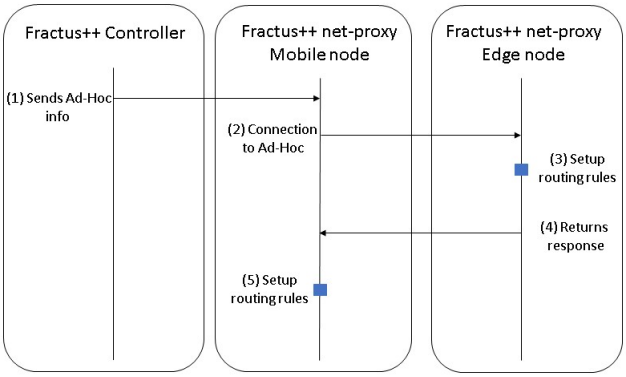


Figure 3.7: Mobile connection to Ad-Hoc

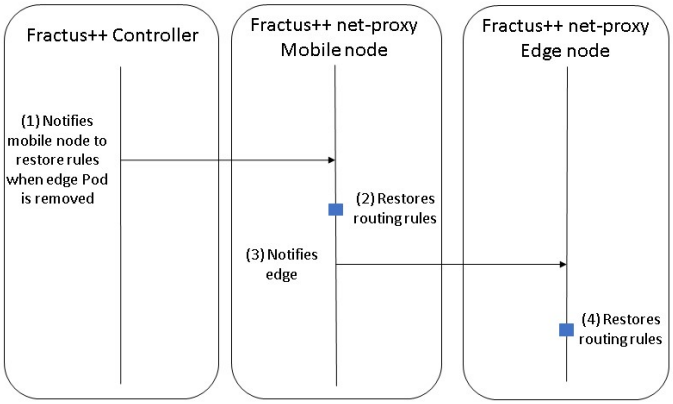


Figure 3.8: Mobile disconnection to Ad-Hoc

Chapter 4

Implementation

In this chapter, we discuss the main implementation details of the proposed work.

4.1 Underlying frameworks

The Fractus++ framework uses k3s [7], a lightweight Kubernetes distribution appropriate for edge computing. In addition, Docker [3] is leveraged as the container runtime software and Flannel [8] provides a layer 3 overlay network among the cluster members for the internet-based communication.

4.2 Fractus++ *Controller*

The *Controller* is aware of application resource events to Kubernetes and readjusts the deployment plan. It also cooperates with the *Monitor* to have global knowledge about the dynamic behavior of the cluster.

The *Controller* initializes its internal structures and starts the *Scheduler* thread responsible for serving deployment-related actions. The *Controller's* main thread watches for new application registration, modification, or removal events posted to Kubernetes. The main thread inserts the type of each event as well as the application's specification and credentials to a queue, which the *Scheduler* thread accesses and consumes the plan configuration requests.

The *Scheduler* thread recognizes two types of application description modifications: 1) specification changes initiated by the user or an application component, and 2) one or more new resource type entries in the *UpdatedResources* list.

To further explain the latter case, list entries indicate the resource types that have been updated since the last readjustment and may lead to a possible deployment plan rearrangement. The aforementioned entries are produced by the *Monitor* at runtime. Whenever a reconfiguration request is finished, the *Scheduler* clears the *UpdatedResources* list, if it is not already empty. The list is introduced via a new property in the application specification and can include the following entries:

(i) *Edge Node Entry*, (ii) *Cloud Node Entry*, (iii) *Mobile Node Entry* and (iv) *Drone Entry*.

For example, when the mobile node is in the proximity range of an edge component, the *Mobile Node Entry* will be inserted to the *UpdatedResources* property of the application's specification. Alternatively, when a new edge node joins the cluster, the *Edge Node Entry* will be added.

In terms of deployment adaptivity, every application component has a dedicated list containing the host nodes for it. The *Scheduler* marks every host in the above list with the custom status values shown in Table 4.1:

Table 4.1: Host-specific custom status values per application component

Value	**Description**
Pending	The Pod needs to be deployed to the host node
Active	The Pod is running on the host node
Inactive	The Pod needs to be removed from the host node

Whenever a new node is selected to host a specific component, the node is marked as *Pending* in order for the component's Pod to be deployed to it immediately. From the moment the given Pod is up and running, the host's status is marked as *Active*. During the deployment plan reconfiguration, if one of the component's hosts is no longer a candidate, the host is marked as *Inactive*.

4.3 Fractus++ *Monitor*

The *Monitor* detects cluster resource updates, informs the *Controller* to apply plan reconfigurations and performs Pod status monitoring.

The *Monitor* entity consists of two threads. The *HealthChecker* thread is responsible for the application Pods' status inspection. The *ResourceChecker* thread periodically checks the

cluster's resources in order to detect any deployment rearrangements that may be required due to such changes, and notifies the *Controller* accordingly.

In particular, the *ResourceChecker* thread examines mobile node-related updates lead to check if such nodes enter or leave the proximity range of an edge node. If a readjustment is needed, it inserts the respective resource type to the the *UpdatedResources* list of the application's specification. Note that the *ResourceChecker* only checks the resources which are directly affecting the specific application in question. For instance, an application which does not use a mobile node, does not require any management over mobile node resources.

The *Controller* needs to share application-related data structures with the *Monitor*. To ensure proper synchronization, whenever the *Scheduler* readjusts the deployment, it uses two binary semaphores to block the *HealthChecker* and the *ResourceChecker* threads, respectively. The described functionality is shown in Figure 4.1.

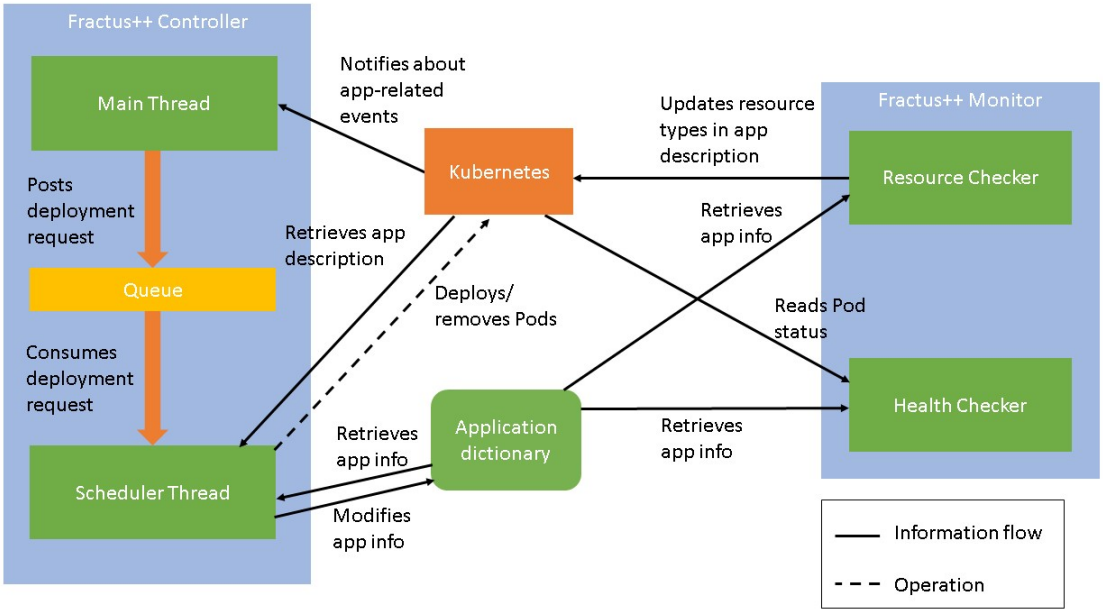


Figure 4.1: *Controller-Monitor* interaction

4.4 Fractus++ *Agent*

The *Agent* sends the node status and resource availability to Kubernetes. In case the *Agent* is running on an mobile node, it periodically updates the coordinates to Kubernetes.

4.5 Fractus++ *net-proxy*

The *net-proxy* cooperates with the *Controller* and performs data-traffic redirection to let the application prefer direct communication with the edge transparently.

In general, distributed applications use several components which serve other application components. Thus, it is crucial for a client component to be able to communicate with the service-providing component without having to discover it after a possible service migration or restart. As a consequence, the server component is represented via a Kubernetes *Service*, provided with a Virtual IP (VIP). The aforementioned VIP and the service's port number are assigned to the service-invoking Pod as environment variables. This address will remain constant during the application's lifetime, as long as the Kubernetes *Service* is not modified.

During a traffic redirection scenario, the *Scheduler* thread sends to the mobile node's *net-proxy* the credentials in order to connect to the respective edge node's *net-proxy* instance which operates as an access point. The mobile node sends to the edge node the desired component's unique name which the redirection will affect, as well as the service's port number. The *net-proxy* running on the edge node firstly retrieves the requested component's IP address in the overlay network which is used for Pod communication, and it assigns a new IP address to the specific component, dedicated to application-level communication through WiFi. In addition, it modifies the routing rules to redirect to the specific Pod traffic sent to the component's assigned address of the direct channel. Subsequently, it returns a response with the component's assigned address. Finally, the mobile node modifies its own routing rules to redirect all the data sent to the service's VIP and port to the dedicated address through the WiFi channel. Figure 4.2 illustrates the explained procedure.

Whenever the redirection needs to be destroyed, the *Scheduler* thread notifies the mobile node. The latter restores the *iptables* [9] rules to enable communication using the default connection and forwards the same notification to the edge node to delete the undesired routing rules.

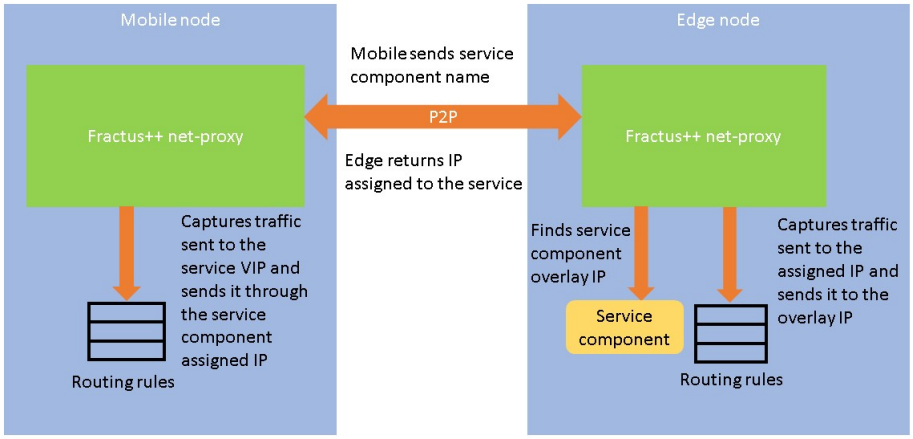


Figure 4.2: *Net-proxy* component interaction

Chapter 5

Evaluation

In this chapter, we will present the experimental evaluation undertaken to validate the operation and quantify the performance of the proposed deployment adaption mechanism. The first section describes the experimental setup. The second part presents the experimental results.

5.1 Preliminaries

5.1.1 Cluster nodes

Within our cluster, we consider Mobile, Edge and Cloud nodes. Each node runs the respective Fractus++ *Agent* based on its type. The *Agents* register the node type to Kubernetes. The *Agent* for the Mobile node updates the coordinates periodically. The *Agents* at the edge also submit their location to Kubernetes.

We evaluated our work on a realistic lab-based cluster which consists of the following members as outlined in Table 5.1. It is important to clarify that the Cloud node hosts both the application-level components as well as the Kubernetes and the Fractus++ control planes.

As for the connectivity, all the cluster nodes are connected via a VPN. Additionally, the Mobile node [10], the Edge 1 and the Edge 2 are employing their WiFi interface, using the 2.4GHz frequency band, for the ephemeral P2P connections. The Mobile node uses a 4G/LTE USB modem [11] for the standard and permanent communication to the Internet. The 4G network coverage is provided by COSMOTE [12].

Table 5.1: Node specs

Node	**Raspberry Pi 3 Model B (Mobile node)**	**Desktop (Edge node 1)**	**Laptop - VM (Edge node 2)**	**Cloud - VM (Cloud node)**
Processor	ARM Cortex-A53 @1.20 GHz	Intel(R) Core(TM) i5-4670 @ 3.40GHz	Intel(R) Core(TM) i5-7200U @ 2.50GHz	Intel(R) Xeon(R) Silver 4110 @ 2.10GHz
Cores	4	4	2	4
RAM	1GB	8GB	4GB	16GB
OS	Raspbian GNU/Linux 11 (bullseye)	Ubuntu 20.04.4	Ubuntu 20.04.1	Ubuntu 20.04.4
Kernel	5.15.32-v7+	5.15.0-46-generic	5.15.0-46-generic	5.4.0-122-generic

5.1.2 Test application

The distributed application used in the experiments, as shown in Figure 5.1, contains three different components. The *MobileViewer* component is running on the mobile node. It accesses the local camera service, to capture images periodically. In turn, these images are sent to the *ImageChecker* component. The latter implements the image processing service of the application and forwards the subset of the images considered as “interesting” for the application to the *DataStore* component, which saves them for further usage / inspection. The *ImageChecker* component is a *hybrid* component, as it can reside in both Edge and Cloud nodes. In this experiment, we focus on the migration of this *hybrid* component, namely the adaptive deployment of the *ImageChecker* as a function of the mobile node’s current position in order to exploit nearby Edge nodes. The *DataStore* component is a pure *cloud* component. The component used to test the adaptation mechanism has an image footprint of 331MB and implements the *hybrid ImageChecker* component.

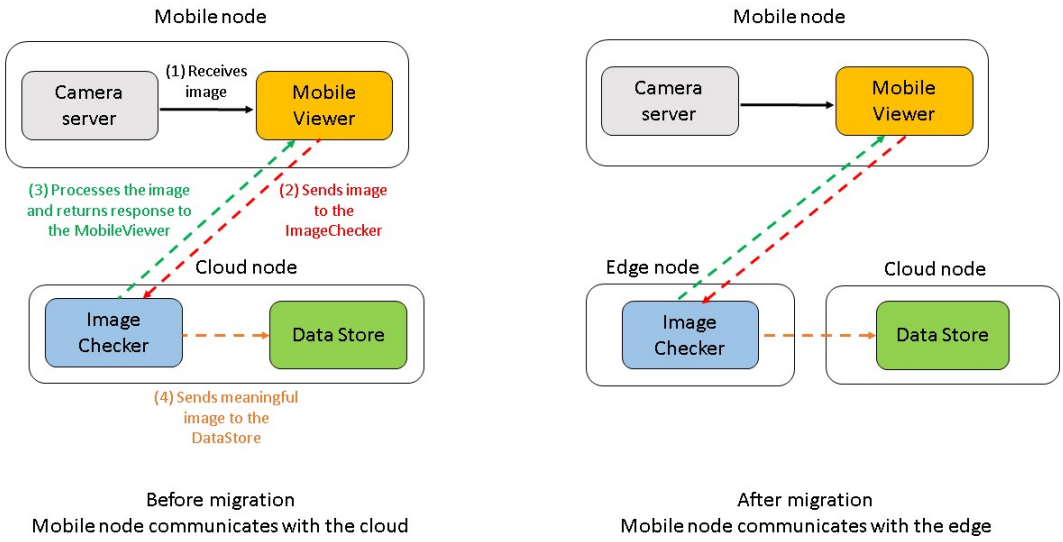


Figure 5.1: Interaction between application components

5.1.3 Testing scenario

For the purpose of our lab-based experiments, we extend the default *Agent* functionality in order to support virtual node mobility. This is done by letting the *Agent* read a file, specified via a configuration parameter, which encodes the waypoints and the (constant) velocity of the node. The distance traveled between two consecutive locations is periodically calculated assuming the node moves in a straight line, and the node's (virtual) current position is updated in the respective Kubernetes resource.

In order to exploit the advantages of the edge computing at runtime, the *hybrid* component is migrated to an Edge node, when the Mobile node approaches its proximity range, thus the Edge node is preferred over a Cloud node. If the Edge node is equipped with a WiFi interface, the Mobile node adopts a temporary P2P interaction with the Edge node over WiFi, as long as it remains within the WiFi range.

We test the deployment adjustment mechanism in the following component migration scenario: During the experiments, the proximity range of Edge nodes is equal to 25 meters and the Mobile node's velocity equals 2.5 m/s. As shown in Figure 5.2 the Edge nodes are being placed in equal distances, forming three regions without Edge node coverage and two regions with edge coverage.

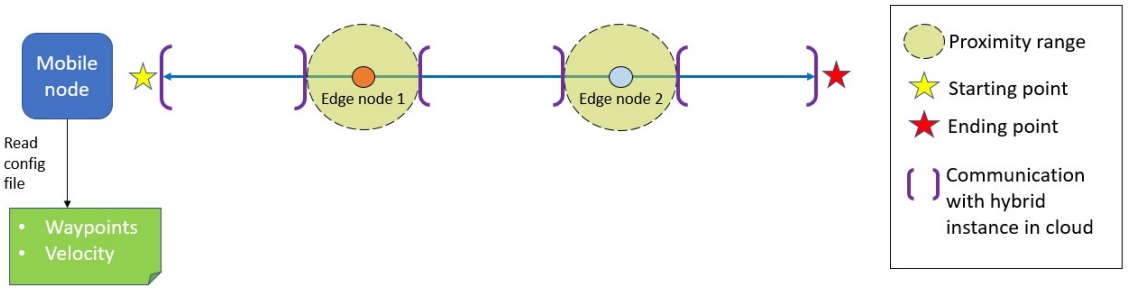


Figure 5.2: Migration scenario

5.2 Performance Evaluation

5.2.1 Communication latency & bandwidth

To begin with, we quantify the benefits in the case when the Mobile node is communicating with the Edge nodes instead of the Cloud node. The main favorable characteristics of edge computing are the lower communication latency and the higher communication bandwidth utilization compared with the communication with the cloud. Thus, modern applications typically opt for interaction with the edge of the network whenever possible. This is particularly true for time-sensitive applications. In the following evaluation, the Mobile node uses the WiFi Ad-Hoc channel when sending data traffic to the Edge nodes and the 4G internet connection for the Cloud node.

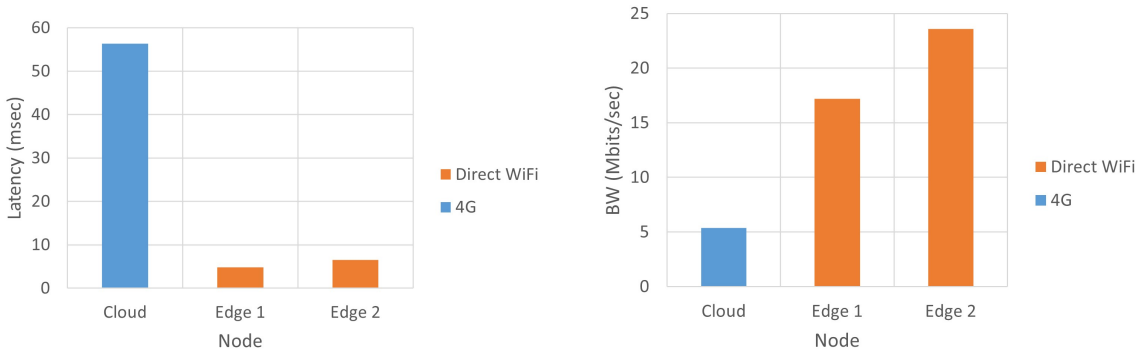
The first set of experiments leverages the *ping* command to quantify the communication latency and the *iperf* tool [13] for the communication bandwidth. In the system-level case, the nodes involved are not using the Fractus++ software.

From Figure 5.3 it is clear that the interaction with the Edge nodes significantly improves the communication latency and bandwidth. The latency with the Edge 1 and the Edge 2 is 91% and 88% respectively lower than with the Cloud node. As for the measured bandwidth, it is 68.8% higher with the Edge 1 and 77% higher with the Edge 2. The above results are

expected, as the P2P interaction is more efficient compared to the 4G internet connection when the RPi communicates with the cloud. The latter also involves passing the data through a base station, which can be located anywhere, to consequently have access to the internet.

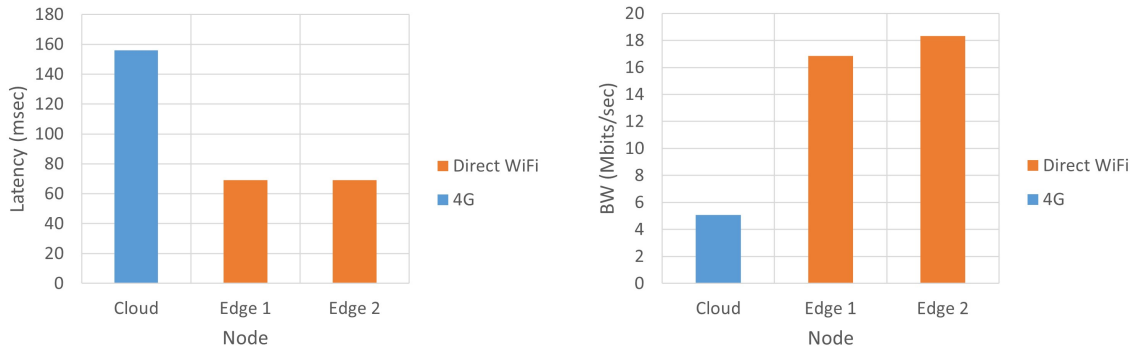
The next step is to involve the test application we discussed, as shown in Figure 5.5, to measure the communication latency and bandwidth when the *MobileViewer* is sending: 1) a simple string (latency) and 2) a 6MB image (bandwidth) to the *ImageChecker* component.

In Figure 5.4 we observe that the communication latency with both of the Edge nodes is 55% lower on average, compared with that to the Cloud. Similarly, the communication bandwidth is approximately 70% higher to the edge rather than to the cloud.



Latency - *ping* command Bandwidth - *iperf* tool

Figure 5.3: System-level latency & bandwidth



Latency Bandwidth

Figure 5.4: Application-level latency & bandwidth

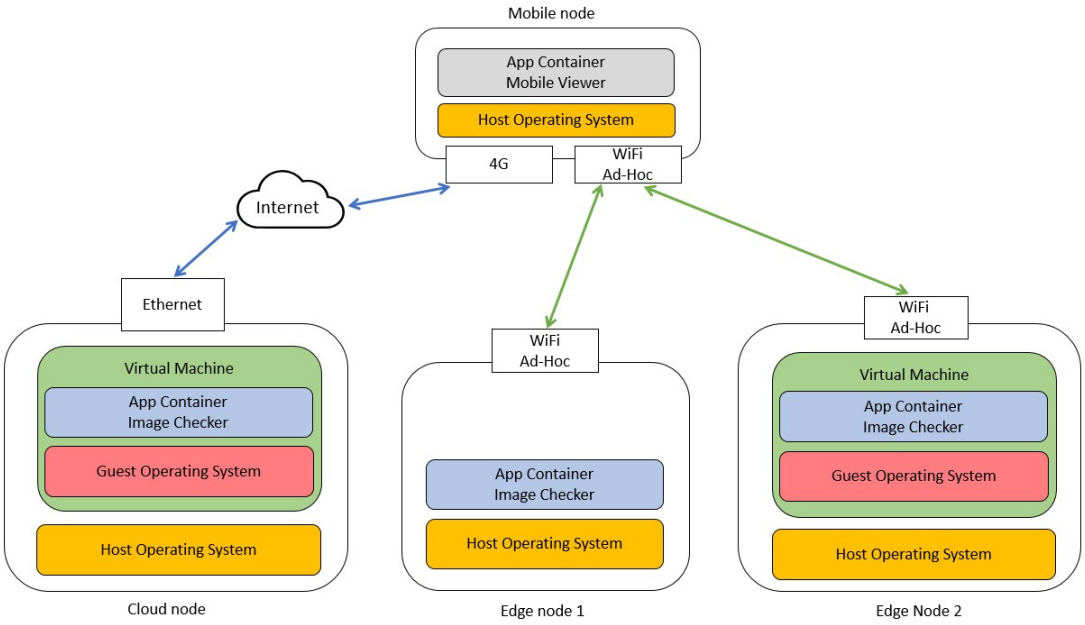


Figure 5.5: Application-level evaluation setup

5.2.2 *ImageChecker* component - Image processing evaluation

At this point, the *hybrid ImageChecker* component activates its true functionality, meaning that it will process the images sent from the *MobileViewer* to perform face detection and blurring, if one or more faces have been detected.

Figure 5.6 asserts that the application shown is favored in its entirety, in terms of response time to the *MobileViewer*, whenever the *ImageChecker* component is hosted by an Edge node. Communication with the edge reduces the overall response time by 71% on average. Additionally, the time spent for image processing is also decreased by 74% on average. This result is explained considering the following factors:

- The Cloud VM handles a larger workload as it hosts the Kubernetes and Fractus++ control planes as well as the *ImageChecker*, *DataStore* and Fractus++ *Agent* Pods. The Edge nodes only host the Fractus++ *Agent*, the Fractus++ *net-proxy* and the *ImageChecker* entities. As a consequence, the Cloud node has constantly very high CPU utilization compared with the Edge nodes.

- The Edge nodes have computing capabilities which are comparable with the Cloud VM's.

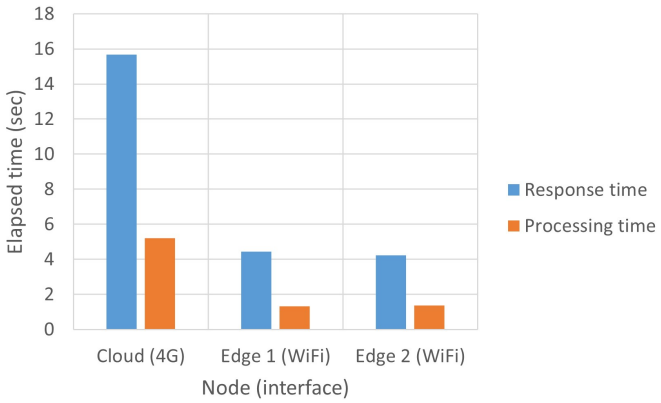


Figure 5.6: Image Processing

5.2.3 Adaptive deployment mechanism - Application-level metrics

To have a better understanding of the potential and the possible limitations of the implemented mechanism for deployment adaptivity, we examine various internal delays and application-level metrics.

The following table 5.2 summarizes application-related metrics:

Table 5.2: Application-related metrics

Metric	**Description**	**Evaluation**
Availability	The total number of successful invocations (from the *MobileViewer* to the *ImageChecker*) to the total number of invocations	Higher is better
Accuracy	The number of the successful invocations at the desired *hybrid* instance based on the specified application rule for migration to the total number of successful invocations	Higher is better

Concerning availability, in the migration scenario 193 of the total invocations of the *MobileViewer* to the *ImageChecker* have been completed successfully and 1 failed, resulting in an availability of 99.48%. When a migration is decided, the old *hybrid* component is being deleted with zero grace period in order to use the new application Pod immediately. Thus, communication errors can occur when the *MobileViewer* is sending an image to the *ImageChecker*.

As mentioned before, we assume that these kind of errors are handled at the application-

level. For example, an image of great significance for the application can be selectively retransmitted. When the *ImageChecker* is migrated and the image transmission fails, there is a high possibility for the application to achieve sending the image faster, if it retransmits the image through the P2P channel, than it would have if the *MobileViewer* was sending the image without errors using the default channel. The above statement is valid in the case when the component migrates from a node which uses the internet-based channel to an Edge node that has direct communication capabilities.

Concerning *accuracy*, the rule we use to select the new host for the *hybrid* instance is the physical proximity from the Mobile node. In case of a migration, we remove the old Pods immediately, with zero grace period. Thus, we observe that the *accuracy* is 100%, as when the Mobile node is inside the proximity range it sends any upcoming successful invocations to the new *hybrid* instance of the *ImageChecker*. If a communication error occurs when the *MobileViewer* starts sending an image during the removal of the old Pod or the teardown of the data-traffic redirection, the invocation is considered as failed.

5.2.4 Adaptive deployment mechanism - Internal delays

In order to quantify the performance overhead for the migration of the *hybrid ImageChecker* component, Table 5.3 summarizes the main Fractus++-oriented delay components, along with their descriptions, while the Figure 5.7 depicts the whole migration process annotated with the aforementioned components.

The last part of the experimental evaluation focuses on the latency related to the Ad-Hoc connection of the Mobile node leveraging the Edge node's direct WiFi channel. Table 5.4 presents and describes the respective delay components. Figure 5.8 depicts the Fractus++ functionality along with the respective delay component for the Ad-Hoc connection. Similarly, Figure 5.9 shows the Ad-Hoc disconnection case.

The Mobile node's *location update* to Kubernetes occurs every 2 seconds and lasts for 342 msec on average. Also, the Fractus++ *Monitor* is periodically checking for new or updated resources every 1 second. Thus, the average delay starting from the end of step (1) until the beginning of step (2) is approximately 544 msec. However, the bottleneck during this time window is the delay for the *Monitor* to update its resources from the Kubernetes registry, added to the delay for the *Monitor* to update the application description. The delays last 42 msec and 33 msec on average respectively. Thus, the average delay for this time window in

the worst case scenario, assuming that the *Monitor* is updating the resources with zero idle time and is not blocked by the Fractus++ *Controller*, is approximately 75 msec.

Table 5.3: Major components of adaptation overhead

Delay component	**Description**
Location update	The delay for the Mobile node to update its location to Kubernetes and receive the response
Controller notification	The delay from the moment when the Fractus++ *Monitor* detects the need for migration until the Fractus++ *Controller* receives the deployment adjustment request
Decision	The delay from the moment that the Fractus++ *Controller* is notified to adjust the deployment until it decides the new component-to-node mapping
Pod-file creation	The delay for the Fractus++ *Controller* to build the new Pod-related files
Deployment	The delay for the Fractus++ *Controller* to stop the Fractus++ *Monitor* and to complete the deployment request to Kubernetes
Instantiation	The delay for the deployed component instance to become operational
Removal	The delay for the Fractus++ *Controller* to check for Pods to delete and complete the removal request to Kubernetes
Termination	The delay for the undesired Pods to be completely removed
Redirection teardown notification	The delay for the Fractus++ *Controller* to notify the Fractus++ *net-proxy* of the Mobile node to stop data-traffic redirection
Redirection setup notification	The delay for the Fractus++ *Controller* to notify the Fractus++ *net-proxy* of the Mobile node to start data-traffic redirection

Table 5.4: Ad-Hoc delay components

Delay component	**Description**
Setup connection	The delay for the Fractus++ *net-proxy* to connect to the Ad-Hoc channel of the respective Edge node
Teardown connection	The delay for the Fractus++ *net-proxy* to disconnect from the Ad-Hoc channel of the respective Edge node

The *Controller notification* delay has an average value of 57 msec. As for the *Pod-file creation* procedure, it has negligible overhead equal to 1 msec.

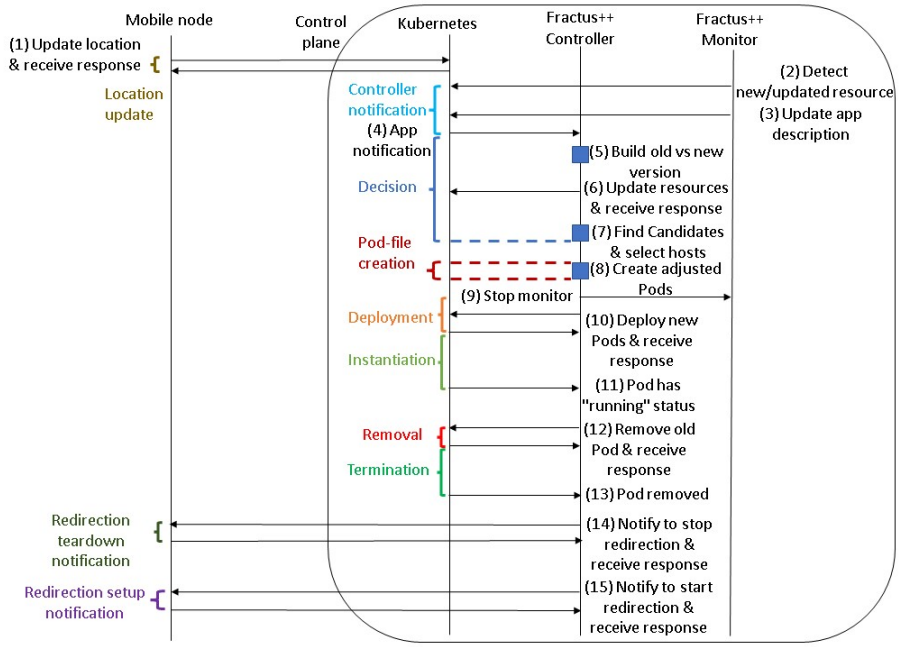


Figure 5.7: Migration process - Functionality & delay components

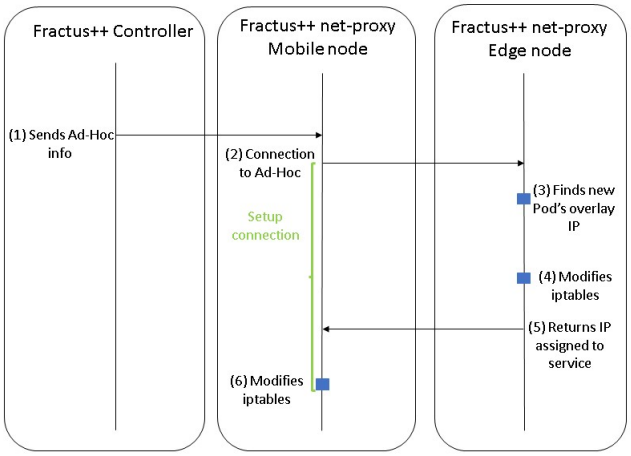


Figure 5.8: Ad-Hoc connection - Functionality & delay components

Figure 5.10 illustrates the average *decision* delay for all the possible migration cases. We observe increased average *decision* overhead by approximately 27.5% when the migration target is the Cloud node. This result is explained because the mechanism performs additional calls to Kubernetes, when none of the Edge nodes can host the *hybrid* component, by asking

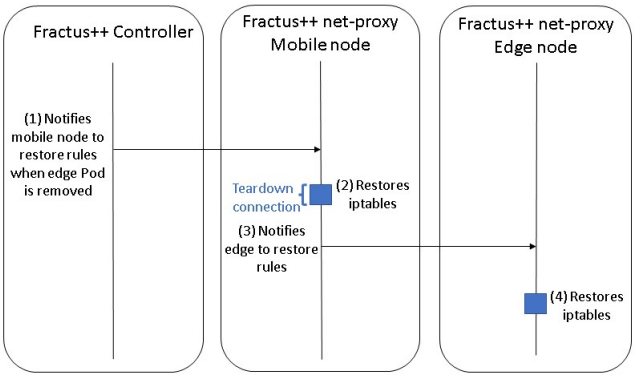


Figure 5.9: Ad-Hoc disconnection - Functionality & delay components

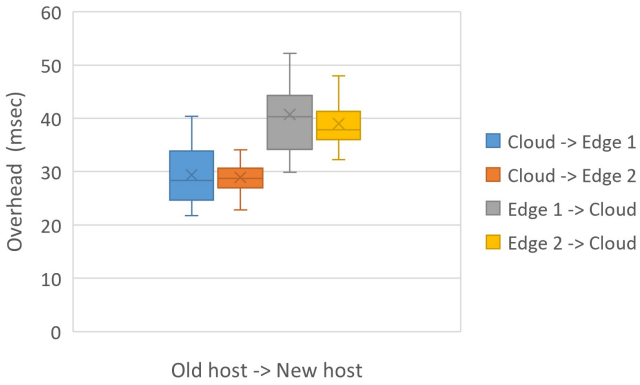


Figure 5.10: Average *decision* delay

the status of the available Cloud node.

From the presented results, we can observe that, the mechanism is limited by an acceptable overhead from the moment when the mobile node sends its location until the new migration decision is made and the deployment plan is readjusted.

In Figure 5.11 we compare the *instantiation* overhead for the *hybrid* component across the different cluster members. As expected, the Edge 1 has the lowest *instantiation* overhead as it is the only node which is running on top of the machine's host Operating System and not within a VM, thus it is spared from virtualization overheads. The Edge 1 has approximately 20% and 30% lower *instantiation* delay than the Edge 2 and the Cloud respectively. The Cloud node has 10% higher overhead due to the fact that it hosts a larger workload than the Edge 2, which leads to a significantly higher CPU utilization percentage compared with the

Edge nodes. As a consequence, the mechanism's performance is favored by migrating the component to the Edge nodes instead of the Cloud node.

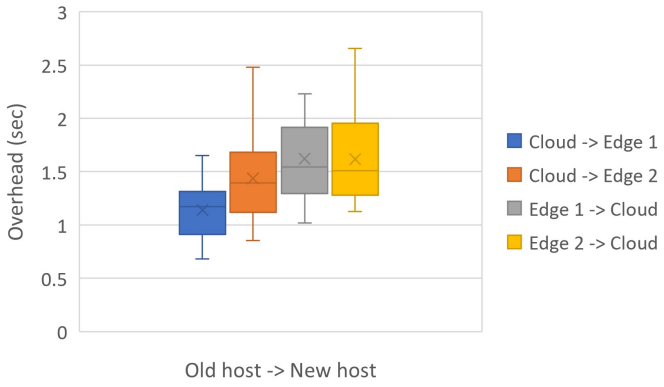


Figure 5.11: Average *instantiation* delay

Next, we evaluate the performance of the stages of the *deployment*, the *removal* and the *termination* in the same graph as shown in Figure 5.12. As observed from the graph, the average overhead for the *removal* and the *termination* is practically equal on all nodes with a difference lower than 2 msec for both cases. Referring to the *deployment* delay, the maximum difference is approximately 6 msec. The *deployment* delay varies due to the fact that it includes the additional overhead of the acquisition cost for two semaphores used for synchronization between the Fractus++ *Controller* and the Fractus++ *Monitor*. The results show that these components cause acceptable delays to the overall migration process.

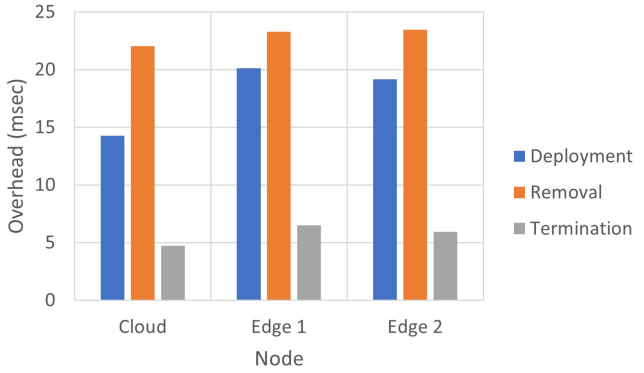


Figure 5.12: Average *deployment*, *removal* & *termination* delay

When the old Pod is removed from the system, the Fractus++ *Controller* notifies the Mobile node if there is an opportunity for connection or disconnection to or from an Ad-Hoc network. From Figure 5.13, comparing the *redirection setup notification* and the *redirection teardown notification* delays we notice that the former is 52% higher, as the message to

transmit is longer. This relates to the need of the Mobile node for the essential credentials to connect to the Ad-Hoc channel in which the Edge node operates as an access point. On the contrary, whenever a connection needs to stop, the Mobile node does not require rich information, rather than a simple message with the suitable status code.

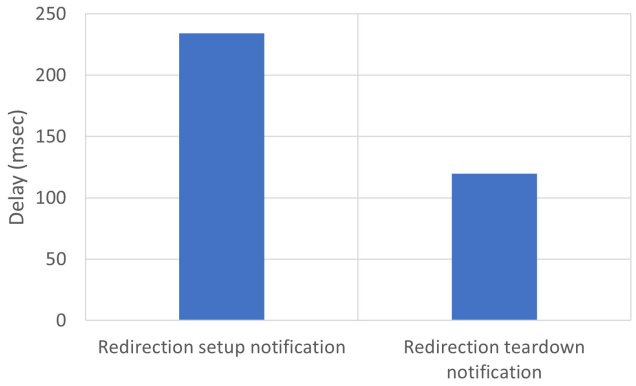


Figure 5.13: Average *Controller*-to-mobile notification delay

From the Edge node's point of view, we evaluated the average delay to create and delete *iptables* rules for each migration case. It is worth noticing that in the case of the *redirection setup* delay, the Fractus++ *net-proxy* on the Edge node initially finds the Pod's IP address in the overlay network at runtime and subsequently inserts the appropriate routing rules to start the redirection. On the other hand, the *redirection teardown* only includes the delay of restoring the *iptables* rules.

Figure 5.14 illustrates that the *redirection setup* delay on the Edge 1 is 43% lower than on the Edge 2, while the *redirection teardown* overhead of Edge 1 is 65.9% lower than the respective overhead of Edge 2. That is explained as the Edge 1 runs directly on top of the host's Operating System and the Edge 2 runs as a VM. Thus, the Edge 2 suffers virtualization overhead on top of the overhead of network management within the Operating System.

Figure 5.15 shows that the Mobile node has 10% lower *connection* delay with Edge 1 compared with Edge 2. That is expected due to the fact that Edge 1 has lower *redirection setup* than Edge 2, as derived from Figure 5.14. The *disconnection* average delay is similar for both Edge nodes. A percentage equal to 92% of the total *disconnection* delay occurs due to the concurrent *iptables* rules modification by Kubernetes, thus preventing the Fractus++ *net-proxy* to restore the routing rules immediately. The *connection* and *disconnection* delays are bottlenecks for the performance of the mechanism as they induce the highest delays among the presented overhead components.

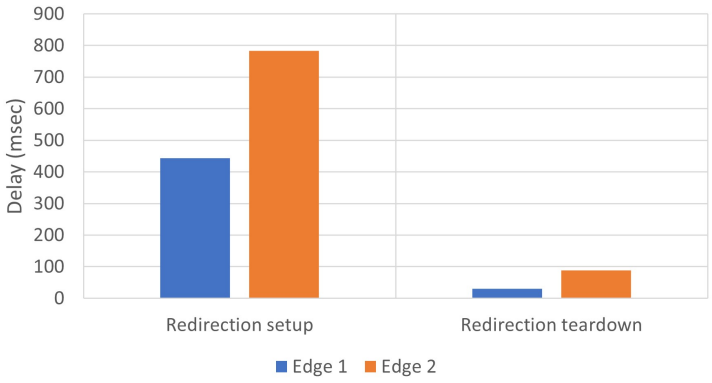


Figure 5.14: Average edge redirection *setup* & *teardown* delay

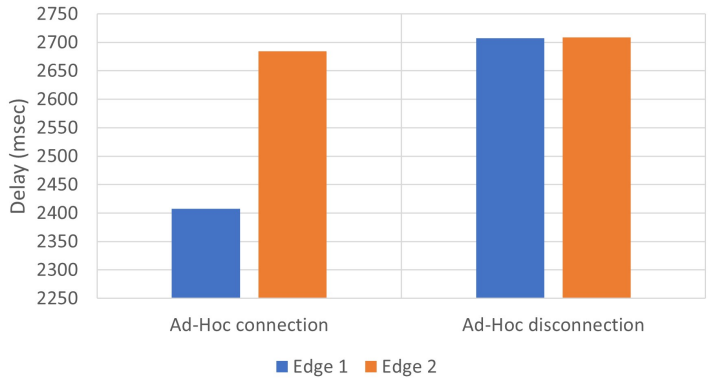


Figure 5.15: Mobile node - Average Ad-Hoc *connection* & *disconnection* delay

5.2.5 General limitations

Having examined the above experiments in order to evaluate our proposed mechanism in terms of performance, it is important to extract general conclusions for the limitations of this work.

The successful operation and communication with the edge is restricted by two factors:

1. The Edge node's wireless proximity range

2. The Mobile node's velocity

We assume a wireless proximity range equal to 50 meters and a constant drone velocity equal to 5 m/s. The overall migration process, from the Cloud to an Edge node, lasts 4.69 seconds on average. Based on relation 5.1 and having an Edge node with the assumed proximity range, the mobile node's velocity must not exceed 21 m/s, as the mobile node travels across a straight line which equals two times the physical proximity range of the Edge node. Exper-

imentally, the maximum supported velocity equals 26 m/s. This behavior can be explained because:

- The mobile node remains in the proximity range for a longer time interval than the theoretical one, due to the delays of the mobile node's location updates to Kubernetes.
- The mobile node is moving virtually, as the Fractus++ *Agent* simulates the location updates of an actual node with mobility. In practice, it constantly remains in the physical proximity of the edge node and its WiFi interface remains active during the Ad-Hoc *disconnection* process.

$$Velocity < \frac{2 * ProximityRange}{MeanMigrationDelay} \tag{5.1}$$

An additional observation is related with the expected invocations of the *MobileViewer* to the edge. In this experiment, the period of accessing the camera service is zero, meaning that the *MobileViewer* captures a new image immediately after the previous one is sent to the *ImageChecker*. The formula 5.2 follows an approximate model to calculate the *MobileViewer's* average invocations to the edge for a single traversal.

$$CallsToEdge = \left\lfloor \frac{2 * ProximityRange}{Velocity * MeanResponseTime} \right\rfloor \tag{5.2}$$

The mobile node's velocity significantly affects the accuracy of the proposed model, as increased velocity produces faster migration triggers and less invocations. Thus, the model becomes vulnerable as the velocity increases. This deduction can be validated from experiments shown in Table 5.5. As observed, in the first case of having the velocity equal to 5m/s, the model has a high deviation between the theoretical calls and the experimental ones. While decreasing the speed to 2.5m/s, the accuracy is increased by 10% on average. In the third step, we observe that the accuracy is increased with the a lower rate, equal to 1.5%. If we keep reducing the speed, the same behavior will be continued.

The above limits that our mechanism sets can be mitigated using a proactive approach, which would start the migration process earlier to have the Mobile node start communicating with the edge as soon as possible when it enters the respective proximity range.

In our experimental evaluation, we mainly focused on the component migration scenario due to the mobility of a node. However, the implemented mechanism not only provides the

Table 5.5: Model accuracy

Velocity	**Model accuracy (%)**
5 m/s	78%
2.5 m/s	88%
1.25 m/s	89.5%

component migration option, but also a more general deployment adaptivity mechanism that improves the overall system's resource utilization and optimizes performance-related metrics specified by the user.

Chapter 6

Related Work

In this chapter, we outline previous research on the fields of static and flexible application deployment, service and computation migration, as well as the main differences compared to our proposed work.

6.1 Application deployment & Edge, Cloud computing platforms

Beyond static deployment, flexible deployment is emerging as a requirement by modern, complex applications.

Bahl et al. [14] introduce the concept of service offloading from the mobile devices to a powerful set of computers which are operating at the edge of the network, called as ”cloudlets”. The cluster uses a VM virtualization technology to host the respective services. The offloading process starts whenever the mobile node approaches the proximity range of the ”cloudlet” to exploit the communication benefits and terminates when it leaves the range, thus having to continue the communication with a remote cloud.

Liu et al. [15] mainly focus on IoT applications in the form of containerized services, in which computations can be offloaded to the gateways, as the latter usually remain underutilized. The authors leverage a centralized cloud-centric approach for the resource orchestration and management and evaluate the variety of technologies used for virtualization before the final selection of containers was made. [16] follows the same pattern as the previous work by offloading services to the Edge Cloud and also considers security as a major concern with the introduction of trusted environments for the container operation. In terms of evaluation,

they present the security-related overheads induced by the platform.

Comparing with the first study, we focus on a container migration approach instead of VM synthesis. Referring to the last two works, apart from the initial deployment phase, our major consideration is the plan readjustment behaviour at runtime and the different scenarios which trigger the readjustments.

In [17], the authors justify the emerging need of running applications in which both common and serverless components are used concurrently. The common components are used in the form of containers or VMs. As a consequence, the primary focus is on supporting the adaptive deployment of applications following the above model, by expanding the capabilities of an existing platform. Moreover, the platform can handle the deployment of cross-cloud applications. The deployment procedure is repeated whenever the captured metrics for the monitored application suggest it. The design is significantly influenced by applications that combine both AI and Big-Data fields, which is also the type of application used for the evaluation. More specifically, the same application is examined in two scenarios, comparing the case when only VMs are used with the case when serverless components are also part of the application. The authors prove that the latter case has not only decreased cost but also higher scalability.

In [18], the authors discuss several advantages of using multi-cloud applications nowadays. This extended framework serves applications with such characteristics and also consisting of components with various service levels. In this case, the deployment readjustment is supported by complex rearrangement rules, activated by events. The rules are correlated and converted to workflows, which can be modified in the course of time to optimize the used plan. Moreover, supervision of the application's precedent recorded activity is also provisioned for possible reuse, or deployment reshaping. The platform evaluation tests a paradigm of an application using the microservices approach and shows the successful operation and handling of adaption scenarios.

Work presented in [19] focuses on the adaptive application deployment across multiple cloud providers, but without solving any plan reconfiguration scenario. A provider-independent approach is embraced by combining two existing application specification technologies to exploit both the application components' topology as well as a uniform method to distribute the components to multiple cloud vendors. To achieve that, the authors use an intervening graph which combines essential information about the application's components and the in-

teractions between them. For the functionality validation, a multi-cloud application sample is presented.

The key difference between the last three studies and our work is that, apart from the functionality testing with an application sample, we also quantify the internal overhead that our mechanism introduces during the deployment adaptation. In addition, we provide application-level metrics to ensure the successful operation using a realistic cluster.

6.2 Service migration

Service migration is a technique commonly used in edge computing in order to achieve better resource utilization and to improve the quality of service for the user.

The authors of [20] introduced a service migration mechanism which exploits cognitive computing at the edge of the network. The migration process occurs proactively, to send a subset of the needed services or tasks to the next expected edge node where the user is more likely to travel, if the node provides the essential required resources. In addition, each service has three different types, depending on the user's demands, in terms of the tradeoff between the quality of the results and the response delay of the running application. To decide for a possible migration, the mechanism realizes a reinforcement learning approach. The mechanism is also compared with the just-in-time migration case and the case without migration, using a mobile user in an emotion detection scenario.

[21] focuses on IoT environments, while operating in the Cloud-Fog-Edge continuum. Each layer is responsible for handling the respective workload based on its computing capabilities. The Cloud layer manages the most demanding tasks in terms of resource utilization and decides the deployment plan based on global system analysis. The Fog layer accelerates the interaction with the edge devices. Similarly to our work, the nodes are annotated with placement-specific labels according to their layer. To provide the overall system with deployment flexibility, the Cloud layer exploits data mining solutions and the results are consequently shared with the Fog layer. To validate the correct operation, a smart home scenario is used.

Our work is capable of handling more complex applications and adaptivity scenarios, such as having an application component to trigger a component migration at runtime.

6.3 Target mobility & computation migration

An additional related field is the computation migration in order to provide flexible support due to the target mobility.

Bramberger et al. [22] apply migrations in a distributed system consisting of multiple smart cameras for an object tracking application. Smart cameras are used as embedded systems that cooperate to accomplish several real-world missions. When the target inserts a specific zone of interest or there is a resource modification, in terms of availability, the migration process is triggered. The system uses mobile agents, meaning that the agent that performs the required task is moved between cameras.

Although there are similarities with our proposed work, we handle a larger variety of applications which can also use systems with sensors.

Chapter 7

Conclusions and future work

In summary, we mainly focused on the need of introducing adaptivity to the deployment at the edge of the network.

7.1 Conclusions

During this work, we have introduced and extended Fractus, an orchestration framework for distributed applications in the form of containerized services, which is capable of exploiting possibilities of temporary P2P communication with the edge of the network, based on user's requirements. We introduced the functionality to evolve the deployment adaptation and readjust the plan at runtime, by detecting application description and cluster modifications, while also considering node mobility. We conducted an experimental evaluation to verify the advantages of edge computing and to examine the limitations of the mechanism by the capturing component migration-related overheads. In addition, we validated the successful operation with several application-level metrics.

7.2 Possible improvements

To optimize the overall system performance and mitigate the limitations presented in the evaluation, we can perform proactive migration to the new target-host based on the measured overheads, in order to let the mobile node start communicating with the edge just-in-time with minimal delay. To achieve that, we also need to predict the node's mobility to recognize the next edge node that the mobile node will approach.

To further reduce the migration process overhead of the adaptation mechanism, one could initiate the direct connection between the mobile and an edge node before the *Controller*-to-mobile notification is sent, to start the data traffic redirection with lower delay.

Bibliography

[1] Athanasios Grigoropoulos. *System Support for the Fault Tolerance, Testing and Orchestration of Drone Applications*. PhD thesis, University of Thessaly, Dept. of Electrical and Computer Engineering, Apr. 2022.

[2] Nasos Grigoropoulos and Spyros Lalis. Fractus: Orchestration of Distributed Applications in the Drone-Edge-Cloud Continuum. *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 838–848, 2022.

[3] Docker. `https://www.docker.com/`. Access date: 2022-03-09.

[4] Container. `https://www.docker.com/resources/what-container/`. Access date: 2022-20-09.

[5] Virtual Machine. `https://www.vmware.com/topics/glossary/content/virtual-machine.html`. Access date: 2022-21-09.

[6] Kubernetes. `https://kubernetes.io/`. Access date: 2022-21-09.

[7] k3s. `https://k3s.io/`. Access date: 2022-03-09.

[8] Flannel. `https://github.com/flannel-io/flannel`. Access date: 2022-03-09.

[9] iptables. `https://www.netfilter.org/projects/iptables/index.html`. Access date: 2022-27-09.

[10] Raspberry Pi. `https://www.raspberrypi.com/products/raspberry-pi-3-model-b/`. Access date: 2022-03-09.

[11] Huawei 4G USB Dongle. `https://consumer.huawei.com/en/routers/e3372/`. Access date: 2022-03-09.

[12] COSMOTE GR. `https://www.cosmote.gr/hub/`. Access date: 2022-03-09.

[13] iperf. `https://iperf.fr/`. Access date: 2022-27-09.

[14] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.

[15] Peng Liu, Dale Willis, and Suman Banerjee. Paradrop: Enabling lightweight multi-tenancy at the network's extreme edge. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 1–13. IEEE, 2016.

[16] Ketan Bhardwaj, Ming-Wei Shih, Pragya Agarwal, Ada Gavrilovska, Taesoo Kim, and Karsten Schwan. Fast, scalable and secure onloading of edge functions using airbox. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 14–27. IEEE, 2016.

[17] Kyriakos Kritikos and Paweł Skrzypek. Towards an optimized, cloud-agnostic deployment of hybrid applications. In *International Conference on Business Information Systems*, pages 435–449. Springer, 2019.

[18] Kyriakos Kritikos, Chrysostomos Zeginis, Eleni Politaki, and Dimitris Plexousakis. Towards the modelling of adaptation rules and histories for multi-cloud applications. In *CLOSER*, pages 300–307, 2019.

[19] Jose Carrasco, Javier Cubo, and Ernesto Pimentel. Towards a flexible deployment of multi-cloud applications based on tosca and camp. In *European Conference on Service-Oriented and Cloud Computing*, pages 278–286. Springer, 2014.

[20] Min Chen, Wei Li, Giancarlo Fortino, Yixue Hao, Long Hu, and Iztok Humar. A dynamic service migration mechanism in edge cognitive computing. *ACM Transactions on Internet Technology (TOIT)*, 19(2):1–15, 2019.

[21] Muhammad Alam, Joao Rufino, Joaquim Ferreira, Syed Hassan Ahmed, Nadir Shah, and Yuanfang Chen. Orchestration of microservices for iot using docker and edge computing. *IEEE Communications Magazine*, 56(9):118–123, 2018.

[22] Michael Bramberger, Andreas Doblander, Arnold Maier, Bernhard Rinner, and Helmut Schwabach. Distributed embedded smart cameras for surveillance applications. *Computer*, 39(2):68–75, 2006.