# Dynamic offloading of application services to edge servers using docker swarm and microservices

UNIVERSITY OF THESSALY

George Georgiou

**Supervisors**
Spyros Lalis, Associate Professor
Christos D. Antonopoulos, Assistant Professor

2018, Volos Greece

*To those who withstood*

# Acknowledgements

First I would like to thank my supervisor Spyros Lalis for his help and guidance throughout this work. His high standards and tireless work vastly improved the quality of the presented work. I would also like to thank Christos D. Antonopoulos for his help in the evaluation of this work.

My gratitude also extends to Dimitris Syrivelis and Manos Koutsoubelias for their crucial contributions in the beginning of this thesis and its general direction. I would also like to thank all of my departments faculty and staff for our cooperation throughout the years

My special thanks to my family and friends for their unwavering and continuous support, both moral and material. Finally to Katerina Charisi; Thank you for being the light when all was dark.

# Περίληψη

Η επικράτηση του Internet of Things και των κινητών συσκευών έχει εισάγει ευρεία διαθεσιμότητα υπολογιστικών πόρων σε κοντινή απόσταση από τελικό χρήστη. Αυτό, με την σειρά του, έχει οδηγήσει στην εμφάνιση της δομής του edge computing, που προσπαθεί να αξιοποιήσει αποτελεσματικά τους πόρους αυτούς και έχει επιπλέον εισάγει υπολογιστικές συσκευές με την μορφή των micro-servers. Ως αποτέλεσμα υπάρχουν πρόσθετες προκλήσεις που αφορούν στην ετερογένεια των διαθέσιμων συσκευών καθώς και την βέλτιστη αξιοποίησή τους.

Σε αυτή την έρευνα προσπαθούμε να αντιμετωπίσουμε μερικές από αυτές τις προκλήσεις χρησιμοποιώντας την γνωστή αρχιτεκτονική των microservices και ελαφριές εικονικές μηχανές όπως τα Docker containers. Παρουσιάζουμε το πρότυπο ενός συστήματος που έχει αναπτυχθεί πάνω στο Docker Swarm. Λαμβάνει, ως είσοδο, μια microservices εφαρμογή και εκμεταλλεύεται ευκαιριακά, με βάση μια απλή πολιτική μετανάστευσης, κοντινούς υπολογιστικούς πόρους ώστε να αυξήσει την επίδοσή της. Αξιολογούμε το σύστημά μας χρησιμοποιώντας μια ρεαλιστική εφαρμογή και διαπιστώνουμε ότι σε ένα μετρίως δυναμικό περιβάλλον μπορεί να προσφέρει σημαντική αύξηση της επίδοσης στην προαναφερθείσα εφαρμογή.

# Abstract

The prevalence of the Internet of Things and mobile devices has introduced widespread availability of computational resources in close proximity to the end user. This has, in turn, led to the emergence of the edge computing paradigm, attempting to efficiently leverage these resources and further introducing computing devices in the form of micro-servers. As a result added challenges exist, pertaining to the heterogeneity of available devices as well as their optimal utilisation.

In this research we attempt to address some of these challenges by utilizing the well established architecture of microservices and lightweight virtualization in the form of Docker containers. We present the prototype of a system that is built on top of Docker Swarm. It receives, as input, a microservices application and opportunistically exploits, based on a simple migration policy, nearby computing resources in order to increase its performance. We evaluate our system using a realistic test application and find that in a moderately dynamic environment it can provide a significant performance gain to the target application.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

In recent years we have witnessed the evolution and prevalence of the Internet of Things (IoT) [1] alongside the widespread adoption of mobile devices (mainly smartphones) as the de-facto ubiquitous computational platform. This has created new opportunities for the development of novel applications, which are typically location-aware, time-critical and have access to a multitude of data produced by the numerous sensing devices. But this also introduces increased requirements in terms of processing power, storage, and low latency. Moreover, it brings additional challenges in terms of device/-platform heterogeneity, stable and fast connectivity, and security.

Traditionally, support for mobile and IoT computing has been provided using the Cloud Computing paradigm [2]. Cloud computing provides readily available computing power that leads to increased performance as opposed to a strictly local execution on much slower embedded platforms, as well as achieves better data reliability in case a device fails or is lost. Computation at large data centres also enables the management and processing of very large data sets, while taking care of software update and maintenance problems.

However, the fact that computation may take place at a server almost anywhere in the world, presents security concerns and removes control from the end user. In addition, the physical distance between the location where data is produced and where it is processed can introduce significant latency due to the uncontrolled nature of the Internet. These constraints, added to the requirement for constant Internet connectivity and availability of cloud resources, render cloud computing inappropriate for certain types of applications, for example time-critical applications that require fast service response.

The IoT as an inherently distributed paradigm is in contrast with the centralized approach of classic cloud computing. This has pushed the industry towards a more distributed architecture, leveraging more powerful computing infrastructure (e.g., based on microservers) near or inside the mobile access network, with a very fast communication to the mobile and IoT devices in the vicinity. This architectural approach is often referred to in literature as fog computing [3], mobile-edge computing [4], or simply edge computing. It makes computing resources readily available in close proximity to where the data is produced, can significantly reduce the amount of data that is sent to the cloud, decrease Internet congestion and improve the application's response time.

The challenges though still stand on how to leverage these resources, in a practical and easy way. The heterogeneity of the numerous IoT devices presents a number of technical problems, burdening developers with accommodating a large number of different execution environments. Moreover, since the computing power of edge-based infrastructures (micro-servers) is typically smaller than in the cloud, there exists a more apparent and urgent need for provision and optimal exploitation of such resources. Finally, in some cases, the security concerns can become even greater at the edge, if potentially untrusted devices need to collaborate and share their own provide resources, calling for increased isolation of hosted executions.

In this thesis we present a prototype of a system aimed towards enabling flexible resource sharing and offloading of computations at the network edge. Given a component-based application, our goal is to: (1) support the dynamic migration or offloading of selected application components by opportunistically exploiting heterogeneous nearby computing resources, be they microservers or other devices that offer their resources for this purpose; (2) propose and evaluate a simple policy for triggering/controlling this migration/offloading so as to improve application performance in terms of response time; (3) allow this migration/offloading to happen transparently in order to achieve an uninterrupted execution of the application, also in case the application host loses connectivity with part of the computing infrastructure.

To accomplish our goals we utilize the well established architecture and technology of microservices [5] and containerization with Docker [6] and Docker Swarm [7]. Microservices are a relatively new software architecture whereby an application is composed of a set of small independent services. This allows for an intuitive mapping to the distributed domain, where the computationally intensive components/microservices can be identified and migrated/offloaded to remote computing resources. Containers are a lightweight alternative to traditional virtualization techniques, having the advantage of smaller image sizes and easier deployment while retaining the advantages of security and isolation of service execution on a remote host. Docker is the most popular platform that provides substantive tooling for containers. In our approach, each microservice is realized inside a Docker container. To achieve flexible distributed execution, we build our system on top of Docker Swarm, an established solution for the orchestration and management of Docker containers.

## 1.2 Thesis Structure

The thesis is structured as follows. Chapter 2 briefly gives some background information about the utilized technologies. Chapter 3 outlines the related work. In Chapter 4 the system architecture and implementation is presented. Chapter 5 discusses the evaluation of our system prototype. Finally the thesis is concluded in Chapter 6 alongside goals for future work.

# Chapter 2

# Background

This chapter briefly introduces the main architectural concepts and technologies that were used to implement our system prototype.

## 2.1 Microservices

The term microservices refers to the architectural style of developing a software application as a suite of loosely coupled services [5]. It can be seen as an evolution or detailing of the more traditional Service-Oriented Architecture (SOA) model. In the microservice architecture, each service implements specific functions or business logic. Services aim to be fine-grained and communicate through lightweight mechanisms, typically HTTP/REST [8]. As in traditional SOA, microservice interfaces are typically language- and platform-neutral, so that they ca be easily invoked from a wide range of client programs that are written in many different programming languages and run on widely different platforms.

It is useful to examine microservices as opposed to traditional monolithic applications, as illustrated in Figure 2.1. In a typical client-server monolithic application, the server-side application is expected to process requests, execute all the processing and business logic, handle potential calls to a database, and produce a response for the client. Such an application scales by replicating the server-side as a whole, and is typically maintained and updated at this rather coarse of granularity. On the contrary, a microservice-based application requires minimal central governing as it consists of several relatively small and autonomous services. These can be deployed and scale

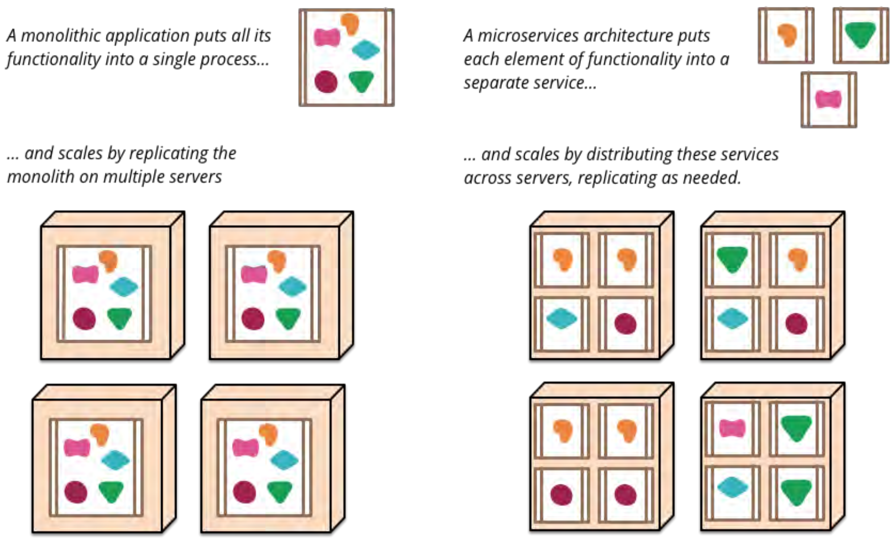
A monolithic application puts all its functionality into a single process...
... and scales by replicating the monolith on multiple servers
A microservices architecture puts each element of functionality into a separate service...
... and scales by distributing these services across servers, replicating as needed.

Figure 2.1: Monolithic and Microservice-based approach. Image taken from Martin Fowler[1]

individually as well as accommodate code or technology updates in a more fine-grained manner. Moreover, this approach provides better fault isolation and allows for a more granular/focused monitoring of individual application components.

Microservices also have drawbacks. For instance, they introduce larger architectural complexity in both deployment and testing. It is often not trivial to decompose a software application into meaningful components. At times, microservices also impose a significant communication overhead cost, unlike for components that interact and synchronize using faster inter-process communication that relies on shared memory. Lastly, if designed poorly, overly small services can lead to increased memory and resource overhead.

The concept lends itself particularly well to web-applications; although traditional desktop application could also be designed to use both locally resident and remote microservices. In our work, we utilize microservices to implement a component-based application since it naturally allows for per-service monitoring and profiling. Based on this monitoring, our system decides which service/component to migrate/offload on an available host, in order to utilize the available resources more efficiently.

[1]`https://martinfowler.com/articles/microservices.html`

## 2.2 Containers

Containerization is a method of virtualization at the Operating System (OS) level, achieved through isolation into multiple user-space instances, typically called containers. Containerization is a lightweight alternative to the more traditional Hypervisor-based virtualization. The main difference is that, instead of running an entire and completely separate system software stack on top of emulated hardware, containers share system calls and kernel features (see Figure 2.2). Containers typically have separate filesystem, processes, network stack etc, and are allotted resources such as CPU, memory and disk by the Host-OS. It is common for containers to be used in order to accommodate the execution of a single application in an environment that is isolated as well as restricted in terms of resources.

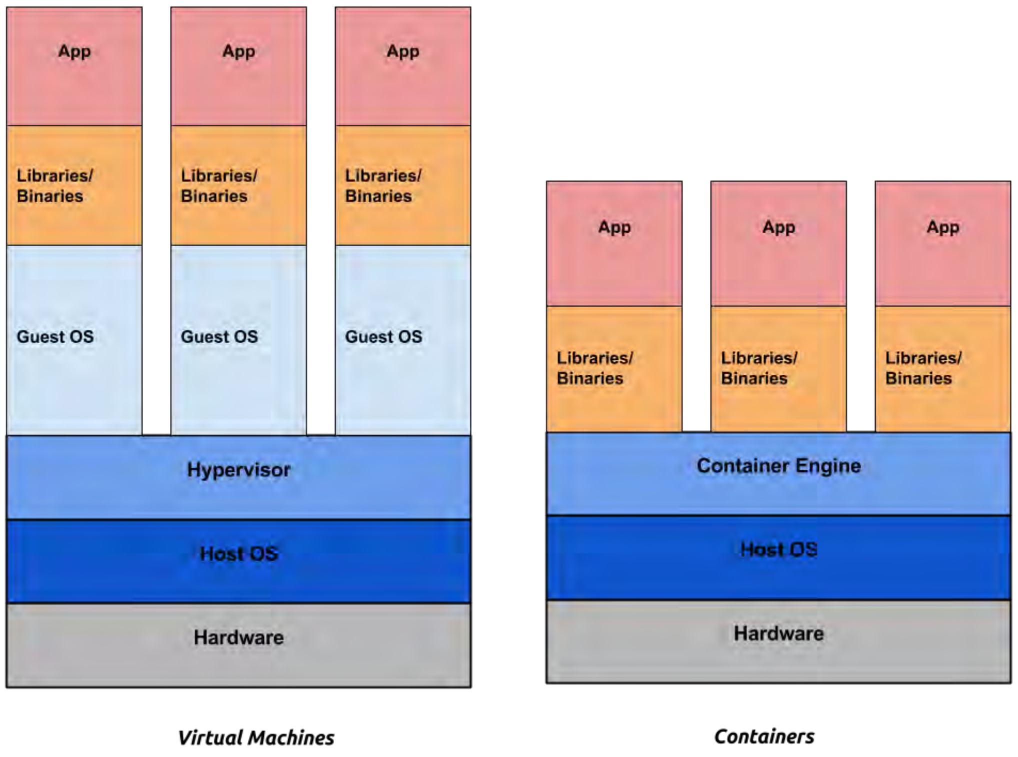

Figure 2.2: Virtual machines and containers.

Containers are an attractive method for application virtualization because they introduce minimal overhead compared to other methods. The fact that system calls are executed natively on the Host-OS and the kernel features are shared amongst different containers on the same machine, allow for increased creation and execution speed. In addition, the image size for containers is significantly reduced compared to full Virtual Machines (VM). These facts translate to highly increased portability and agility. Thanks to the more lightweight nature of containerization compared to hardware-level virtualization, a host can usually accommodate a larger number of containers but only a limited number of traditional VMs.

One of the disadvantages of containerization is that it raises security concerns due to the decreased level of separation between different containers. For this reason, to have a stronger separation among different container groups, it is not uncommon for a bundle of containers to be created and run inside a separate full VM. In addition, because of the sharing of kernel, containers are less flexible. For instance, it is not possible for containers to be accommodated inside a host that runs a different native OS. There are workarounds to this problem, but they give away a number of the traditional advantages of containerization.

In Linux systems, the assortment of tools, templates, libraries and language bindings that provide support for containerization is referred to as Linux Containers (LXC) [9]. LXC acts as a driver for the creation and execution of containers. Container engines, such as Docker or LXD, are built on top of it. The key kernel features that make containerization possible are *namespaces* and *control groups*. These are discussed below.

Namespaces is the underlying mechanisms that provides containers with an isolated environment in terms of system view. Currently there are 6 namespaces that are utilized by container technology, achieving isolation for different aspects/characteristics of each container. These are: (i) *mnt*: control of mount points as well as filesystems; (ii) *pid*: provision of separate process IDs; (iii) *net*: virtualization of the network stack; (iv) *ipc*: isolated System V interprocess communication; (v) *uts*: separate hostnames; (vi) *user*: root and privilege isolation, as well as separate user IDs

Control groups (cgroups) are responsible for limiting resource and monitoring resource usage, like memory, CPU, block I/O and network. They allow for the setting of soft and hard limits in terms of available memory, CPU shares and/or number of cores. Additionally, they provide network prioritization as well as limits for read and write I/O. Cgroups also monitor and report corresponding resource usage metrics. Notably, cgrouns function in a hierarchical manner where each subsystem (i.e. CPU, memory etc.) has an independent hierarchy tree consisting of nodes. Each node consists of a group of processes sharing a resource and each process belongs to exactly one node [10].

## 2.3 Docker

Docker [6] is an open platform and tool chain that provides an additional abstraction layer for managing and running an application program in a wrapped and isolated manner, as a separate container. It is built on top of the features of the Linux kernel that allow for OS-level virtualization like namespaces, cgroups etc. Docker used to employ LXC as its default execution driver, but currently employs its own *libcontainer* facility in order to communicate directly with the kernel (however, it still supports LXC, *libvirt* and *systemd-nspawn* as alternative execution drivers). Apart from the aforementioned kernel features, Docker uses a union-mount capable file system in order to build the container images. This means that images are built by creating multiple separate layers on top of each other, allowing multiple containers to re-use basic layers, thus making the process more lightweight. By images we refer to the read-only template that includes the necessary information to create a Docker container. This information is given by the user in the form of a simple syntax text-file called Dockerfile.

The Docker engine is responsible for creating, running and managing the containers. It is built on a client-server architecture that consists of 3 main parts (as illustrated in Figure 2.3): the Docker daemon/server, a Command Line Interface (CLI) client, and the REST API through which calls to the server are made. Users typically use Docker through the CLI client, which in turn communicates with the Docker daemon through the REST API over UNIX sockets or a network interface. The Docker daemon is responsible for building and running containers as well as for managing Docker objects like images, containers, networks and volumes. It can communicate with the clients or other daemons in order to manage Docker services. The Docker server and client are not required to run on the same machine, even though this is possible.

Aside from the basic engine, Docker offers additional tools for the management and extend use of containers. The most prominent ones are Docker Registry, Docker Compose and Docker Swarm. The Docker Registry stores images, which are retrieved from the Docker daemon in order to create new

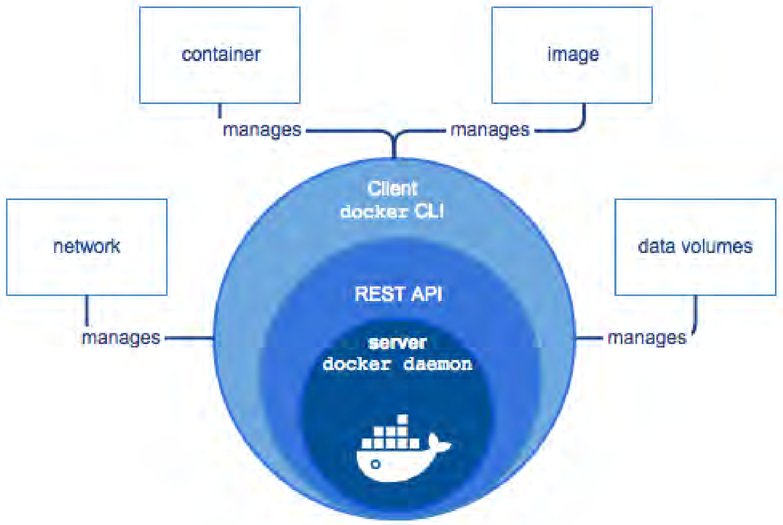

Figure 2.3: Docker engine components. Image taken from docker.com[2]

containers if a base image does not exist in the local machine. Docker Compose is a tool for defining and running applications that consist of multiple containers. The requirements and specifics are described in a YAML text-file by the user, which is then used by Docker in order to create the desired environment. Docker Swarm is a formerly stand-alone tool, now integrated in the Docker Engine. It provides native cluster management and orchestration functionality for Docker containers. More information about Docker Swarm is given in the next section.

## 2.4 Docker Swarm

As mentioned above, the Docker engine can be used to create a cluster or swarm of containers where application services can be deployed. The corresponding cluster management and service orchestration support is referred to as Docker Swarm [7]. This is based on a declarative service model whereby an application stack comprises a set of services that have a specific desired state. Configuration happens at runtime, meaning that a swarm can be

[2]`https://docs.docker.com/engine/article-img/engine-components-flow.png`

reconfigured and scaled dynamically. Docker Swarm offers multi-host networking by the creation of an overlay network to which it automatically assigns IP addresses for each node. It offers service discovery, as nodes are assigned a unique DNS name. Containers are load-balanced through ingress load balancing, and fault tolerance is achieved by utilizing the Raft Consensus Algorithm [11]. Security is handled through mutual TLS authentication and encryption between Docker Swarm nodes.

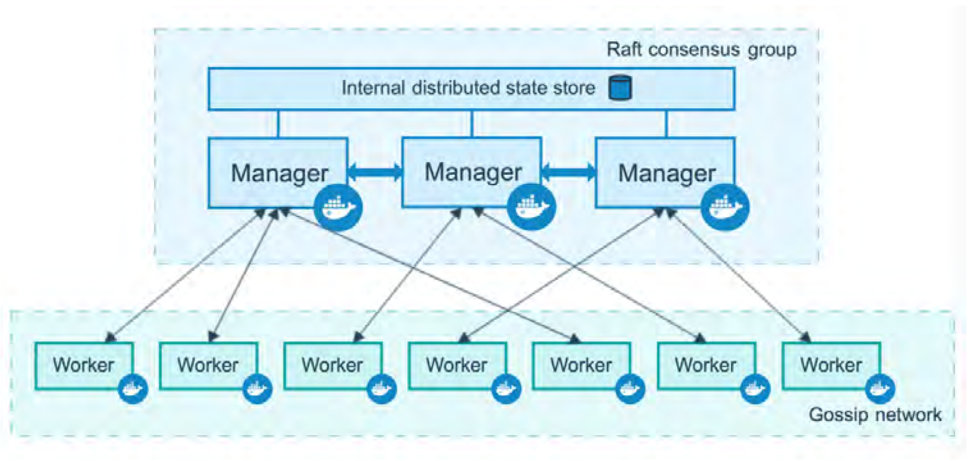

Figure 2.4: Docker swarm nodes. Image taken from docker.com [3]

A Docker Swarm consists of manager and worker nodes (see Figure 2.4). A node is a distinct instance of the Docker engine that participates in the swarm. Manager nodes are responsible for cluster management and orchestration. They maintain a consistent cluster state, schedule services and serve the swarm API endpoints. They elect a single leader that is responsible for handling orchestration issues. Worker nodes execute the work dispatched to them by the managers. They do not participate in any cluster decision. By default every manager node also serves as a worker.

An application that is deployed to a Docker Swarm consists of services. Each service that is created follows a desired state that is pre-defined by the developer. This includes the number of replicas, the network configuration, given resources etc. Docker Swarm tries to maintain this state, so as to always have the desired number of service replicas running. Services can be defined as global, meaning that each node in the swarm will execute a replica

[3] `https://docs.docker.com/engine/swarm/images/swarm-diagram.png`

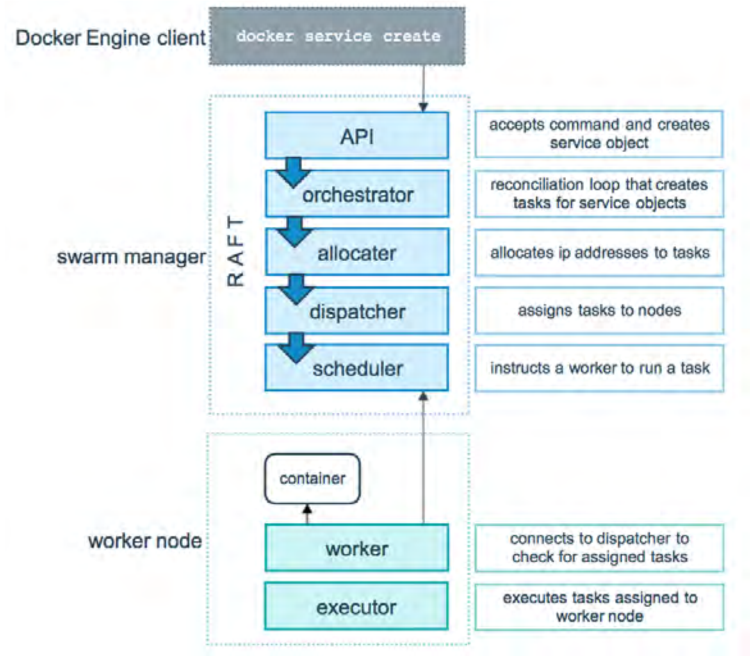
Docker Engine client
docker service create
swarm manager
RAFT
API
accepts command and creates service object
orchestrator
reconciliation loop that creates tasks for service objects
allocater
allocates ip addresses to tasks
dispatcher
assigns tasks to nodes
scheduler
instructs a worker to run a task
worker node
container
worker
connects to dispatcher to check for assigned tasks
executor
executes tasks assigned to worker node

Figure 2.5: Docker service management and execution functions. Image taken from docker.com[4]

of that service.

Tasks are the basic scheduling unit in a swarm. They are assigned to worker nodes by managers, and represent a running instance or replica of a service. Tasks do not migrate; they either successfully execute on the assigned node, or fail, in which case the manager may create a new task to counter-balance the failure. Figure 2.5 illustrates the main functions/responsibilities of the swarm manager and worker nodes with respect to service/task management and execution.

[4]`https://docs.docker.com/engine/swarm/images/service-lifecycle.png`

# Chapter 3

# Related Work

The research efforts related to the work presented in this thesis, outlined in this chapter, can be roughly divided into 3 thematics. First is literature focusing on the more traditional concept of mobile cloud computing, that aims to address the objective of offloading from mobile devices onto cloud infrastructure. Next are the works that expand this concept at the environment of the network edge and the Radio Access Network(RAN), with the presence of edge servers and other devices. Last recounted are the works that discuss the utilization of container technology, most notably Docker, alongside microservices for use at the network edge.

## 3.1 Mobile Cloud Computing

MAUI [12] is a system that focuses on energy saving by offloading code at the method level onto cloud infrastructure. It is built on top of the .NET CLR utilizing its *reflection* feature and partitions code based on manual annotations (*@Offloadable*) set by the application programmer. A profiler is continuously evaluating the mobile device, the application and the network conditions and then passes this data to the solver. In turn, the solver computes a global optimization problem to make the decision of whether to offload. The application state is transfered by leveraging the type-safenature of the .NET runtime. Only data which is potentially referenced by the method to be offloaded are transfered over the network, and is transfered back at the end of the remote execution. MAUI currently supports only single thread execution.

CloneCloud [13] relieves the programmer from having to indicate the offloadable code in a manual way. The system achieves this goal by performing static offline analysis to determine all the legal partitions, based on a set of constraints, e.g., not to include code that accesses device-specific features. Dynamic profiling is conducted for multiple executions, on both device and cloud, with random inputs to create behaviour profiles. Last in the offline analysis is an optimization solver that given the behaviour profiles picks the legal partition that minimizes costs referring to execution time or energy consumption. This process produces a database of legal partitions optimized for different conditions. During execution, a suitable partition is chosen based on this database and the specific running conditions.

Another effort closely related to MAUI is ThinkAir [14]. It also requires developers to annotate their methods with *@Remote* annotations. The system comprises 3 key components. The execution controller runs locally on the device, and is responsible for taking the offloading decision, based on previous method executions, the network environment and the policies for energy, execution time, energy and cost optimization. In the cloud runs a client handler that is responsible not only for remote code execution but also for automatically scaling the virtual machines (VM), that actually execute the offloaded code, to fit the required needs. In order to take the offloading decision ThinkAir uses a set of three different profilers for hardware (or device) features, software (or application) and network environment. They respectively collect data pertaining to the hardware state like CPU and WiFi utiliaztion, software execution like number of instructions and method calls and network state in the form of estimated network bandwidth. These data are then fed to an energy consumption model, based on which the actual offloading decision is made.

Similar work is presented in COSMOS [15], where computation offloading is offered as a service. The main component of the system is the so-called master service. Based on computation tasks given by the COSMOS clients and workloads given by the COSMOS servers, it decides the spawning and destruction of VM instances, in the authors implementation Amazon EC2 instances were used, and performs the allocation of the tasks that were offloaded. The COSMOS client is responsible for the task selection, in a manner similar to MAUI and CloneCloud (according to the authors), and takes the offloading decision. Finally the COSMOS server executes tasks on a first come first serve basis, and provides the COSMOS master with expected workload information, based on predictors such as Mantis [16].

An effort that focuses on *how* to perform offloading is COMET [17]. It follows a distributed shared memory approach, trying to keep in-sync memory states across local and offloaded execution threads. The work is aimed at the specifics of VM synchronization between a pusher and a puller endpoint. A "happens-before" relationship is established before lock acquisition, to keep memory states in sync. The system offloads every possible method, excluding native functions that utilize device hardware (these are manually pre-determined). Finally, a basic scheduler moves threads between endpoints if they have not executed a native function for more than a given time window. This time window is chosen to be twice the Round Trip Time, as to minimize the chances that a thread has to immediately migrate back to the device to execute a native function.

One of the earliest attempts is Cuckoo [18], that aims to enable offloading integrated with the Android system. One of its key contributions, according to the authors, is a programming model offered through the Eclipse IDE[1]. It is based on the pre-existing Android model of services (compute intensive parts) and activities (interactive parts). The application programmer writes the local service implementation, and the Cuckoo framework generates a dummy remote service implementation that must be completed/refined by the developer. On the smartphone runs a resource manager with which available remote resources, that are willing to execute the remote services, register. For the execution of remote methods the Ibis [19] communication middleware is used. The offloading decision made by the system is based on simple heuristics, always giving preference to remote resources if these are reachable.

Serendipity [20] is an approach that shifts focus on multiple devices (called nodes) rather than a single cloud server. The basic job component of the system is called a PNP-block, and consists of a pre-process program that processes input data, N parallel worker tasks, and a post-process program that merges the output of the worker tasks. Each Serendipity node runs three basic system components. The *job engine* is responsible for building job profiles, based on methods similar to the offline analysis in MAUI and CloneCloud, and for disseminating tasks using a greedy task allocation water-filling algorithm. The *master process* monitors task execution on the several *worker processes* that run on each node. The job engine launches a job initiator that initiates PNP-blocks and propagates them to Serendipity nodes.

[1]`https://www.eclipse.org/`

Once results are available they are gathered from the worker processes by the master and are sent back to the job initiator, which in turn returns them to the user.

## 3.2 Mobile Edge Computing

FemtoCloud [21] is a system that offers computation as a service by creating an on the spot compute cluster with participating devices in the vicinity. A client service that runs on the mobile devices is responsible for the reporting of resource sharing capabilities and limits set by the devices, as well as metadata about user behaviour. A server service that runs on a special "controller" device utilizes the data reported by the client devices to calculate the execution load introduced by devices and their expected presence time. The server service is also responsible for task scheduling that is performed based on greedy heuristics. There also exist modules for service discovery and network connectivity estimation.

In CloudAware [22] an abstract programming model that is based on top of the Jadex [23] middleware is presented. The model is meant to be used for elastic mobile applications and the design goal of the authors is to support ad-hoc and short-time interaction with centralized and nearby resources. The application developer is required to create his application with the Jadex components in mind, that are in turn utilised by the framework. The CloudAware theoretical framework consists of 5 key components. A discovery service, a partitioner and solver for determining which components and when to offload them, a context manager for the acquiring of device and user metrics and a coordinator that is tasked with error management.

REPLISOM [24] attempts to reduce traffic to the cloud, in offloading scenarios, via an LTE-optimized memory replication protocol that utilizes the LTE nodes together with neighbouring devices. ME-VoLTE [25] is a work that aims at reducing energy consumption in the event of Video Calls by utilising mobile edge servers located on LTE nodes.

There are also a number of research efforts that are focused on the optimization problems that arise in mobile edge offloading and/or migrating scenarios. Wang et al [26] utilize Markov Decision Processes for the prediction of mobility based on distance and random walk. In [27] a device offloads only if it doesn't break the Nash equilibrium and in [28] an effort is made for the joint optimization of radio and computational resources.

## 3.3 Containers & Microservices in Edge Computing

Stankovski et al in [29] design an Autonomous Self Adaptation Platform (ASAP) that focuses on satisfying specific Quality of Service (QoS) constraints for time-critical functionalities by deploying application specific containers to meet these constraints. The authors focus on the particular use case of File Upload. A monitoring system is responsible for the gathering of metrics metrics. This systems consists of monitoring "probes" that interact with the application, a "monitoring agent" that aggregates the data from the probes and a server that stores relevant information in a time-series database. A performance diagnoser estimates the desired QoS of the application and finally a decision maker specifies the optimal placing where a temporary File Upload Server is deployed inside a container.

Rufino et al in [30] propose a layered architecture for the orchestration of dockerized microservices for Industrialized IoT. The proposed architecture consists of 3 layers. In each of these layers a set of containerized services run on top of the Docker platform. First is the sensing layer that hosts cyber-physical system (CPS) end devices such as sensors and actuators that interact with the environment. Next is a mediation layer where exist microservices for Software-Defined Network (SDN) controllers, databases and Machine Learning Units that perform service monitoring and behaviour analysis. Last is an enterprise layer that is responsible for the execution of the more computationally intensive services and high level management and control of the system.

In [31] a fog-oriented middleware is presented. The middleware is realized as an extension of the Kura framework [32]. On each node a message broker queue is included in order to collect sensed information and possibly perform data filtering and the second extension relates to enabling cluster topologies, instead of strictly layered, for Kura Gateways. The proposed fog nodes act as IoT Gateways that provide a base skeleton to install and run a number of dockerized services. All container management and orchestration is proposed to be performed in the cloud computing level.

Morabito and Beijar in [33] present the design of an edge computation platform that functions through intermediate edge devices. These devices run fuctional *blocks* that in reality are dockerized services. These blocks can offer diverse functionality; the authors focus on data compression and processing to reduce load to cloud infrastructure. Orchestration is achieved through a special *IoT Application Orchestrator* and networking is handled through the creation of separate virtual network slices for each user.

In Poster [34] a very simple framework for the migration of applications is presented and evaluation of the performance of containers and traditional VM is conducted. The layered framework consists of a base layer consisting of the Operating System that is pre-installed on edge servers, the application layer that includes the application logic and is also pre-distributed and the instance layer that is the running state of an application. In terms of VM and container evaluation, the authors find that containers in conjunction with their model can offer reduced RAM usage as well as reduced use of bandwidth. Similarly Ramalho et al in [35], focus on the performance evaluation of Virtualization techniques at the Network Edge. By performing a number of synthetic benchmarks on hypervisor-based virtualization and the Docker platform, the authors arrive to the conclusion that while the first method introduces significant overhead, container-based virtualization looks promising

In [36] a web-based collaborative document editing, with face recognition, application is built utilizing microservices. The authors have developed a number of microservices that run inside Docker containers. They are deployed and orchestrated through the use of Docker Tools like Docker Compose and Docker Registry. The authors have created separate microservices for the collaboration, chat and face recognition servers as well as microservices for the corresponding document, chat history and face template databases. Similar work is presented in [37] where authors present their research about the characteristics of the Docker platform and arrive to the conclusion that it is well paired with microservices. To support that claim a code versioning system is enhanced by the authors with automated distributed deployment utilizing dockerized services.

# Chapter 4

# System Architecture & Implementation

This chapter discusses the system architecture. The main components of they system will be presented and analysed, alongside the most important implementation details.

## 4.1 Approach

In this work, we target component-based applications and focus on the offloading of one or several components in order to improve the performance / response time of the application. The key idea is to opportunistically exploit computing resources offered by nearby devices, such as edge-based servers, in order to offload computationally intensive services or components of an application that would otherwise run locally on a slower (potentially mobile) personal device, like a smartphone.

The developer creates his application with the microservices architecture in-mind and implements the application's components as independent microservices, packed inside Docker containers. The Docker images for these containers, are expected to be uploaded in an accessible registry such as the Docker Hub.

The service definitions, their relations, desired state and image location are all described in a "docker-compose" YAML [38] file. As an example, Listing 4.1 shows the docker-compose file for a sample microservices application, which receives speech as input, recognizes it, turns it to text and finally translates it to text on the chosen language. It consists of 3 services, that respectively handle the speech recognition, translation and service communication. The respective component diagram is shown in Figure 4.1.

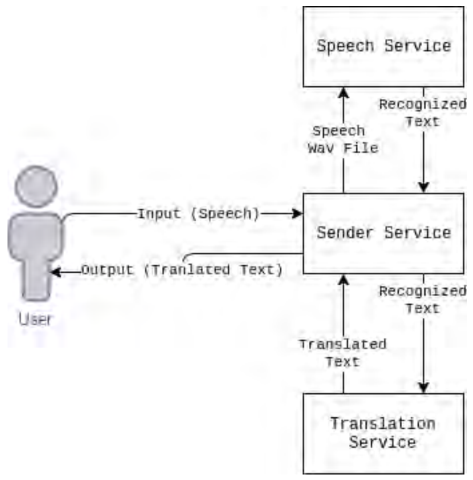

Figure 4.1: Structure of indicative application.

```
version: '3'

services:
  speech_service:
    image: georgeorg/speech_to_spanish:speech_service
    ports:
      - 6001:80

  translation_service:
    image: georgeorg/speech_to_spanish:translation_service
    ports:
      - 6002:80

  sender_service:
    image: georgeorg/speech_to_spanish:sender_service
    ports:
      - 6000:80
    depends_on:
      - speech_serviceh
  - translation_service
```
Listing 4.1: The docker-compose.yml for the indicative application.

This file is the only input that needs to be provided to our system. Based on this application description, the system deploys the application components (micro-services) and automatically tries to identify the most compute-intensive ones , and then tries to offload them to faster machines instead of running them as usual on the local device.

Our system utilises the Docker platform by creating an ad-hoc Docker Swarm where available resources are invited to join as workers. The offloading decision is made based on service profiling and device characteristics (such as CPU and memory) following a simple migration policy.

## 4.2 System Architecture

The high level architecture of the system as well as the main system components can be seen in Figure 4.2.

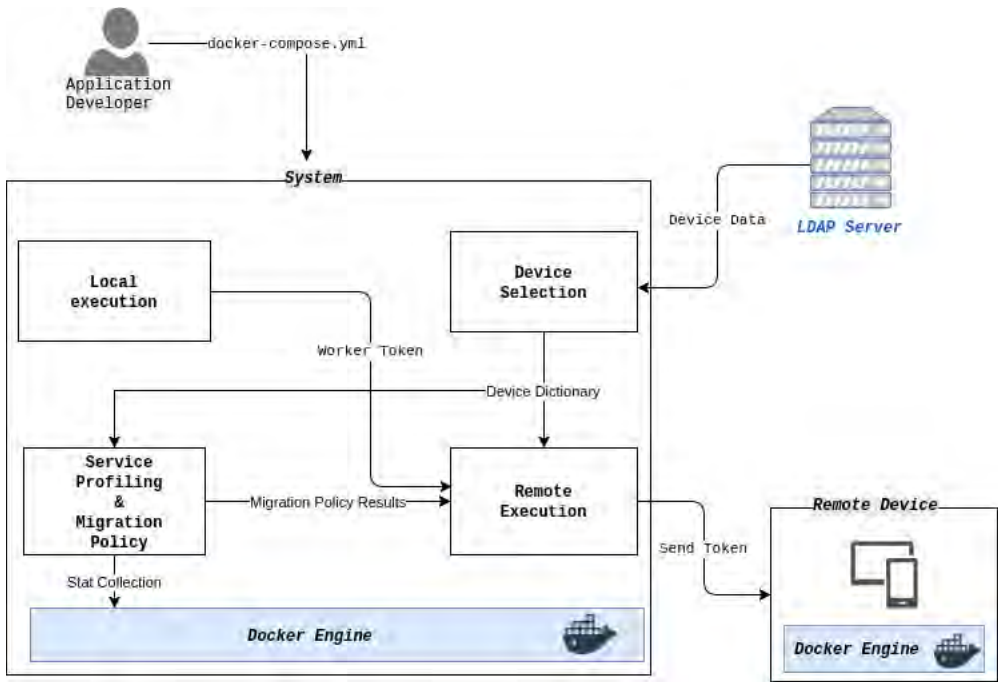

Figure 4.2: High Level System Architecture

Initially all services of the application are expected to run locally. On system start a Docker Swarm is deployed consisting of a single node, the local application device, that acts as leader and manager to the swarm. All the microservices of the application are deployed and run locally, on the application device. Remote devices, that want to share their resources, register with an LDAP [39] server.

Device selection made by the system starts with periodically querying the LDAP server and retrieving available device together with metadata describing their computing resources. After pruning ineligible devices, the average latencies to the remaining ones are calculated. In parallel our system's Service Profiling constantly monitors the running services, in order to calculate their CPU, memory and network usage.

Next, based on service statistics and device characteristics, the Migration Policy is applied and computes the optimal candidate device for each service. The devices that emerge from this procedure are then invited to join the docker swarm by the Remote Execution component.

Finally the Docker Swarm is updated to reflect the results of the migration policy.

## 4.3 Local Execution

The first part of the system is responsible for the local execution of the application. As mentioned, a "docker-compose" file describing the application services, their relations as well as their image locations, is to be provided by the application developer. The services are deployed in a single node Docker Swarm that consists solely of the local application device, acting as manager and sole worker to the swarm, adopting all administrative swarm duties as well as execution of required tasks. At this point a worker join-token is generated. This token is to be shared with potential devices that wish to participate in the swarm.

Alongside the application services, a special *cAdvisor* [40] service is deployed globally, meaning that it will be replicated in every node that joins the swarm. The cAdvisor service is responsible for the monitoring of statistics pertaining to the application services deployed; this process will be discussed thoroughly in the service profiling section of the chapter.

After all services are deployed we exploit the fact that Docker Swarm can be dynamically reconfigured at runtime. The local device/node is labelled and placement directives are placed on all of the services to remain on the device. This last step prevents Docker Swarm from auto-scheduling services on joining nodes and keeps all services running on the local node, unless explicitly defined otherwise.

At a more technical level, note that these directives can expressed either as a "soft rule" in the form of placement-preferences, or as a "hard rule" in the form of a placement constraint. The former option instructs Docker to make a best-effort attempt to spread services among nodes that satisfy the placement requirements. The latter explicitly schedules services on nodes fulfilling the requirements or leaves the service pending until a suitable node is found. We have found that while placement-preferences provide increased system stability, constraints allow for a more direct control over the swarm services. The current implementation of our system utilizes the latter option.

## 4.4 Device Selection

This part of our system is responsible for discovering all available devices that wish to share their computing resources. This is a 3-step process (as illustrated in Figure 4.3) that results in a dictionary listing devices by IP together with metadata describing their computational potency and network latency. It is periodically run, with an infrequent period.

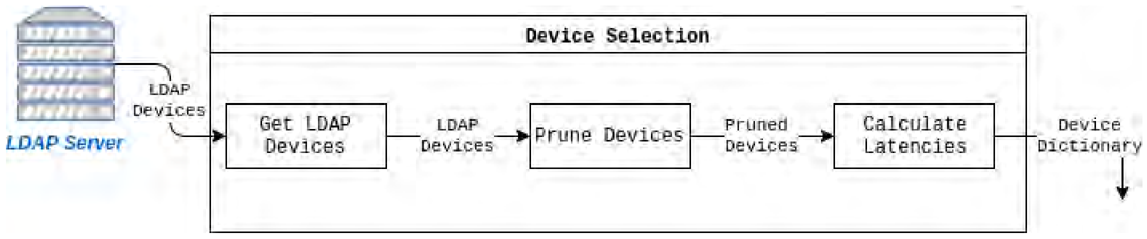

Figure 4.3: Device Selection

### 4.4.1 Device Discovery

Remote devices register with an LDAP server from where they are retrieved by the local application device. One of the reasons we have chosen LDAP as the directory service for device discovery is because of its tree-based hierarchical structure. In a network edge scenario, it is useful to attribute location awareness our directory service. This can be done by exploiting the LDAP hierarchy and retrieving a list of devices under an organizational unit describing a specific location (i.e the city of Volos). We assume that the devices that register with the server have reduced mobility and do not intent to very quickly move out of reach.

Each remote device that registers with the LDAP server also reports its computing resources in the form of metadata describing the number of cores the device has available, the amount of memory and a score describing its CPU. For the CPU score, we have not conducted separate benchmarks but rather use the Passmark CPU mark. Passmark [41] provides scores for a very wide number of CPUs based on a number of benchmarks conducted. The scores are relative, meaning that a CPU with score of $2x$ is roughly twice as fast as a CPU with a score of $x$. Scores are provided for overall and single thread performance. In our case we utilize the single thread mark and take in consideration the number of cores separately, since a remote server is not guaranteed to allow execution on all the cores of its CPU.

The application device periodically, with an infrequent period, queries the LDAP server to retrieve available devices. Our system taking the results as input, constructs a device dictionary that identifies each device by its IP and stores the corresponding metadata. This dictionary is used as input for the next step of the Device Selection process.

### 4.4.2 Device Pruning

After retrieving the data from the server, devices are pruned to exclude ineligible devices. The Prune Devices function receives the device dictionary constructed from querying the LDAP server as input and returns a trimmed version of it. For this purpose the combination of number of cores and the CPU score of available remote devices is considered and compared to those of the local device. The system currently follows a simple greedy approach. It prunes all devices that are computationally inferior to the local application device as they would never be selected for offloading purposes. The result of this function is a new "pruned" device dictionary that is then used for the last part of the Device Selection.

### 4.4.3 Latency Calculation

The final step of the Device Selection process is the calculation of latency between the local application device and the remote devices present in the pruned device dictionary. The system determines the ping latency by calculating the average between 5 pings for every remote device. Packet loss is accounted for by adding a large value for any packets lost; these are considered regularly in the calculations. After these operations, the device dictionary is updated by adding the information about latency in every device entry.

## 4.5 Service Profiling and Migration Policy

This next part of our system is making the offloading decision. In order to accomplish this, the system constantly monitors the running containers (corresponding to application services) and return metrics describing their CPU, memory and network usage. These statistics combined with device characteristics taken from the Device Selection procedure are used as input to a simple migration policy that in turn calculates the optimal device to schedule each service on. An overview of this procedure can be seen in Figure 4.4

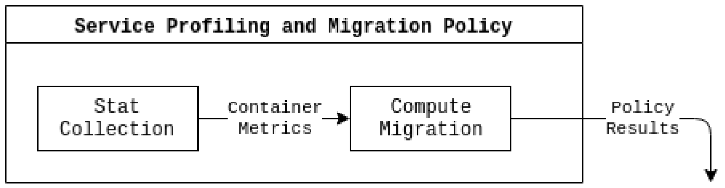

Figure 4.4: Service Profiling and Migration Policy

### 4.5.1 Collecting Service Statistics

As stated above, in order to monitor the performance of the running containers, at system startup a global cAdvisor(container Advisor) service is deployed. cAdvisor is an open-source monitoring tool for containers developed by Google. It collects a large number of metrics including CPU and memory use as well as network usage. These statistics derive mainly from the kernel's corresponding *cgroups*. cAdvisor runs as a daemon that polls for new statistics at roughly every second. Those metrics can be obtained for use through a variety of means. We draw them through calls to the REST API that cAdvisor exposes, in timestamps for each second in the last minute.

Our system gathers statistics for all the containers running on each host. The *stat collection* function is called periodically and reports the average CPU usage in the last minute, the average memory usage in the last minute and the average network usage as it accumulates based on the most recent data available. These metrics are calculated as percentages and are stored in a dictionary for every container corresponding to a an application service.

### 4.5.2 Compute Migration

After the collection of metrics, the system uses said metrics combined with the device characteristics stored in the device dictionary as input to determine the optimal candidate devices for offloading the application services. This decision is made by the system based on a given migration policy, making use of the specific gathered service statistics (CPU, Memory and Network usage) and optimizing for the requested parameters. The proposed migration policy and current implementation is based on simple heuristics and a greedy approach, aiming for minimal total response time. The proposed policy will be discussed more thoroughly on the corresponding section.

The migration policy calculation takes place for every container and device pair, every time new service statistics are collected. We determine the score for each service for every available device, including the local application device. After we have determined the device that provides the maximum score for each service, the policy results are produced in the form of a dictionary with a key-value pair of container-device IP address, describing the target device each service /emphshould be offloaded to. These policy results are then propagated to the remote execution part of our system, to actually perform the offloading.

## 4.6 Remote Execution

The final part of the system developed is expanding the Docker Swarm with the selected devices and offloading the corresponding services. An overview is presented in Figure 4.5.

This step includes communicating with chosen remote devices in order for

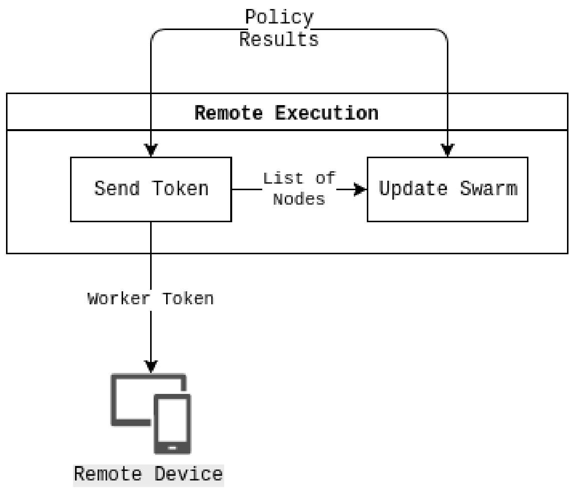

Figure 4.5: Remote Execution

them to join the swarm and then updating the swarm configuration accordingly. It follows the calculation of the migration policy, as it uses its results as input.

### 4.6.1 Swarm Scaling

Receiving as inputs the policy results together with the worker token generated at Swarm creation, the system invites remote devices to join the Swarm. Communication is done via sockets, following a simple protocol illustrated in Figure 4.6.

The application device sends an invitation message to the remote device. The remote device checks whether it is still available, by checking if it has joined another swarm or of it is overloaded in terms of processes. If available it replies with a request for the worker token. The token is exchanged and the remote device joins the swarm as a worker node, sending an acknowledgement message. If the exchange is successful and the device joins the swarm, it is then labelled as a node according to its hostname so that it can be identified in placement directives.

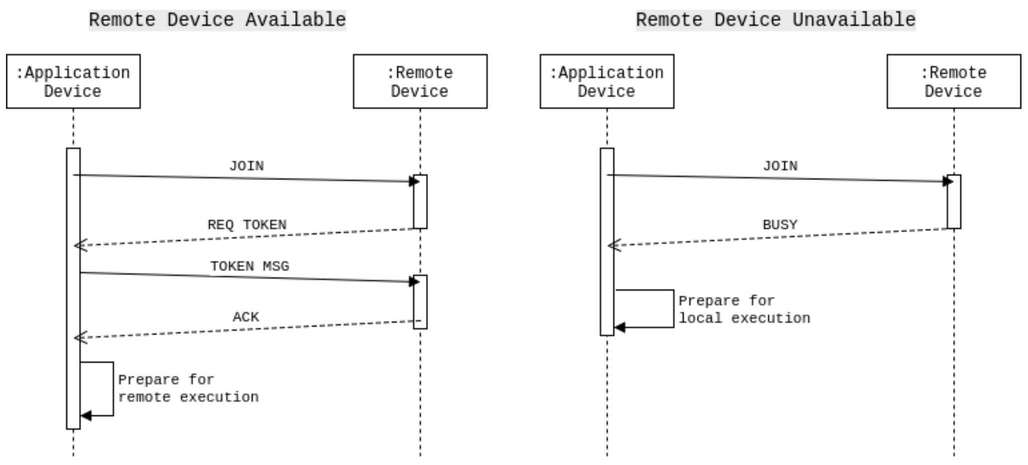

Figure 4.6: Token Exchange Protocol

### 4.6.2 Service Scheduling

After the chosen devices have joined the swarm as worker nodes, it is time to update the swarm configuration based on the migration policy results, so that services execute on the optimal nodes. First, all previous placement directives(placement-preferences or constraints) are removed. Next, all Swarm nodes are retrieved from the Docker daemon and the system checks if the optimal device for every service, is in fact a node of the Docker Swarm. If this is true, placement directives are placed on the service and the service is updated, so that it will execute on the desired node. This step essentially "offloads" the computational load of the service unto the new node.

Services are updated with a "start-first" policy, meaning that when a service migrates the Docker Swarm will wait first for the new service container to go up and then remove the existing service. This feature offers a seamless and transparent transition from local to remote execution. The user does not experience wait-time in order for the services to go up on the remote device. Instead services continue to run locally until they are readied. If the optimal device has failed to join the swarm, the service executes on the local application device.

## 4.7 Migration Policy

The proposed migration formula follows a greedy approach and attempts to determine the optimal device from those available(including the local application device) that minimizes total execution time. For every service a device score is calculated based on the service and device characteristics. The target device that accumulates the maximum positive score is the one that gets chosen to join the swarm and offload the service to. The algorithm behind the process can be seen in Algorithm 1

**Algorithm 1** Migration Algorithm

```
for Every Service do
    Max Score = 0
    Target Device = Local
    for Every Device do
        if $MemoryUsage < DestinationMemory$ then
            Compute Device Score
            if Device Score > Max Score then
                Max Score = Device Score
                Target Device = Device
            end if
        end if
    end for
end for
```

The available memory of the remote device is used as a first filter on whether it can accommodate the service to be offloaded, based on the service memory usage. We then calculate the device score that is broken down to 2 parameters. The CPU Score and the Communication Overhead. The Formula is seen in Equation 4.1

$$DeviceScore = CPUScore - CommunicationOverhead \qquad (4.1)$$

The CPU Score consists of first calculating what increase in performance we expect to see by migrating based on the Passmark single thread score and the number of cores, comparing the local application device and the remote device. We then factor in the service's CPU usage gathered from profiling to determine how much will the service take advantage of this increase. The exact formula for the CPU Score can be seen in Equation 4.2

$$\text{CPU Score} = \text{CPU Increase} * \text{CPU usage} \tag{4.2}$$

$$CPUScore = \frac{CPUDestination*CoresDestination-CPUScoreHost*CoresHost}{CPUScoreHost*CoresHost}*100*CPUusage$$

Next we calculate the communication overhead based on the average network usage of the service and the expected throughput of the connection with the remote device. Throughput is defined as the fraction of the TCP Window Size versus the Round Trip Time (RTT) or as is commonly known, the ping latency. The formula can be seen in Equation 4.3. Through this overhead that is calculated as a percentage we attempt to calculate the expected performance hit due to network latency.

$$CommunicationOverhead = \frac{NetworkUsage}{Throughput}*100 \tag{4.3}$$

$$CommunicationOverhead = \frac{NetworkUsage}{WindowSize/RTT}*100$$

$$WindowSize = Typically \quad 65535\ B$$

$$RTT = Ping \quad Latency$$

It must be specified that in order for the proposed policy to be effective, a "star" service topology is assumed. This means that all services receive input and send output to a master thread that runs the application logic. Else, if the application is an arbitrary graph, and services communicate with each other directly, this policy will not yield the expected results.

# Chapter 5

# Experimental Evaluation

In order to evaluate the performance of our system prototype, we have conducted a number of experiments using an indicative microservice-based application. This chapter presents the application we have developed and the experimentation setup, and discusses the experimental results.

## 5.1 Microservice-based Application

For the evaluation of our system we have developed an application, that is intended for use in tourist scenarios. It takes as input speech from the microphone of the personal device, converts it to text, and translates it to a target language. This application makes particular sense in an edge computing scenario. Speech recognition and translation naturally benefits from location awareness as natural language models can be divided and stored in servers for a specific language use, i.e. directions or a guided tour near tourist attractions. It can also be reasonably expected that such a service will be particularly popular in touristic areas, thus one can intelligently place additional computational resources, at those specific edge locations. This way the application would benefit from the low latency to a server located nearby, as opposed to using a service running on a cloud server anywhere in the world.

Our application consists of 3 independent microservices that are accessed through a REST interface over HTTP. The Speech service receives *.wav* files containing speech as input, and converts them to text files. The Translation service receives text files as input, and returns the text translated to the requested language. Finally, the Sender service accepts the application input, communicates with the Speech and Translation services as needed, and returns the end result. We call the application services through a simple call script. All services, as mentioned, communicate through a REST API. Consequently POST calls create the resource requested and GET calls retrieve it. A typical exchange that converts English speech to Spanish text can be seen in Figure 5.1.

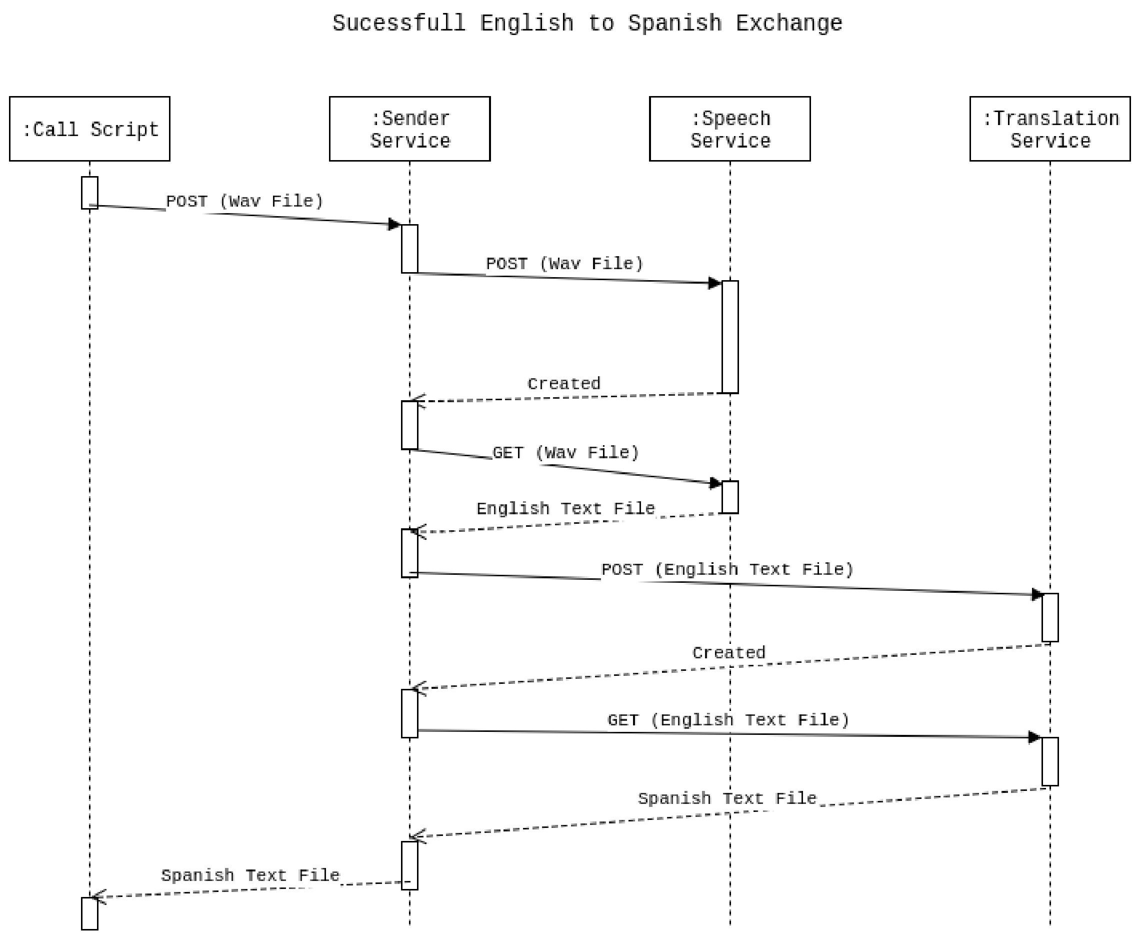

Figure 5.1: Typical message exchange for translating an oral (speech) message to text in a different language (here, English to Spanish).

The speech recognition service implementation utilizes the open source toolkit CMU Sphinx [42] developed at the Carnegie Mellon University [43]. In our test application, speech recognition currently accommodates only English, and is based on the acoustic model and test data offered by the Sphinx toolkit. We specifically use *pocketsphinx* in order to minimize execution time at the cost of recognition accuracy. The translation service utilizes the Apertium [44] open source platform. It is a platform for rule-based machine translation. All services are developed in Python.

Each service runs inside a Docker container that packs all its software dependencies and execution environment. Containerization is pivotal to the ability of the services to be migrated, as they require a large number of dependencies in order to run successfully in a remote execution environment.

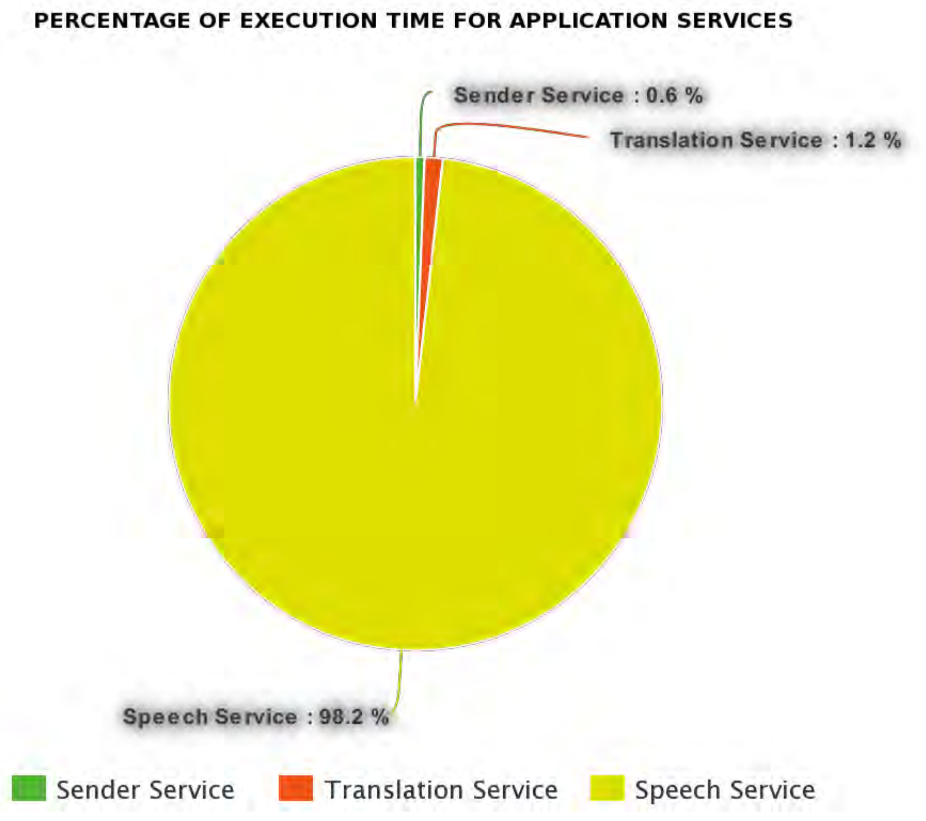

Figure 5.2: Evaluation Application Execution Time Percentages

The size for the corresponding Docker images can be seen in Table 5.1. Also, Figure 5.2 gives a breakdown of execution time, per service. It can be clearly seen that the speech recognition service takes most of the execution time and thus is pivotal for the performance of the application.

Table 5.1: Service Docker Image Size

| | Docker Image Size |
|---|---|
| **Speech Service** | 325 MB |
| **Translation Service** | 215 MB |
| **Sender Service** | 30 MB |
| **cAdvisor** | 24 MB |

## 5.2 Experimental Setup

For practical development and testing purposes the application device is a Laptop and the edge machines are personal computers on the cluster infrastructure of the department. The machine specifications can be seen on Table 5.2. Connection is over a WiFi local area network. The device setup can also be seen in Figure 5.3

Table 5.2: Testing Device Specifications

| | Application Device | Edge Device | Edge Server |
|---|---|---|---|
| **CPU** | Intel i5-4200U CPU | Intel Xeon E5-2620 v2 @ 2.10GHz | Intel Xeon E5-2630 @ 2.30GHz |
| **Cores** | 1 Active (4 Originally) | 4 Active (8 Originally) | 8 Active |
| **Memory** | 4 GB | 8 GB | 8 GB |

An effort was made to minimize any external network and CPU load by deactivating all external software, save for basic exceptions like the terminal emulator and the graphical interface (XFCE) in the application device. The system load was monitored through the *uptime* Linux utility and remained stable throughout different experiments. All experiments were conducted multiple times and the ones best representing the median values are the ones presented here.

For a rough evaluation of our system we have run a number of experiments. We have conducted the same long running experiment in 3 different scenarios (i) Monolithic version (ii) Local Execution (iii) Remote Execution . The experiment in each of these scenarios consists of the testing application receiving as input a list of 9 wav files described in Table 5.3. The wav files are recordings of Aesop's fables taken from LibriVox recordings [45] and differ in size and running time. In the sequence portrayed they are given as serial

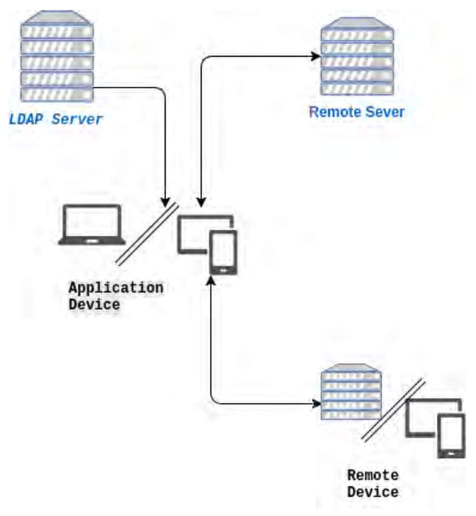

Figure 5.3: Experimental Device Setup

Table 5.3: Wav File Size and Running Time

| File Name | File Size | Running Time |
|---|---|---|
| Fox and Grapes | 1.5 MB | 36s |
| Dog and Sow | 1.4 MB | 44s |
| Golden Goose | 2.1 MB | 1m 6s |
| Fox and Crow | 3.1 MB | 1m 36s |
| Mice in council | 2.9 MB | 1m 30s |
| Lion and Mouse | 3.4 MB | 1m 47s |
| Cat and Mice | 3.2 MB | 1m 41s |
| Goods and Ills | 3.3 MB | 1m 43s |
| Mercury and Woodman | 5 MB | 2m 35s |

input to the application and are then recognized, turned to English text and translated to Spanish text that is then provided as output.

## 5.3 Results and Analysis

### 5.3.1 Monolithic version

The first experiment is conducted with no presence of Docker or our system. The test application is re-written to remove the microservices architecture and the consequent communication overhead. All functions run locally on the application device as it is described.

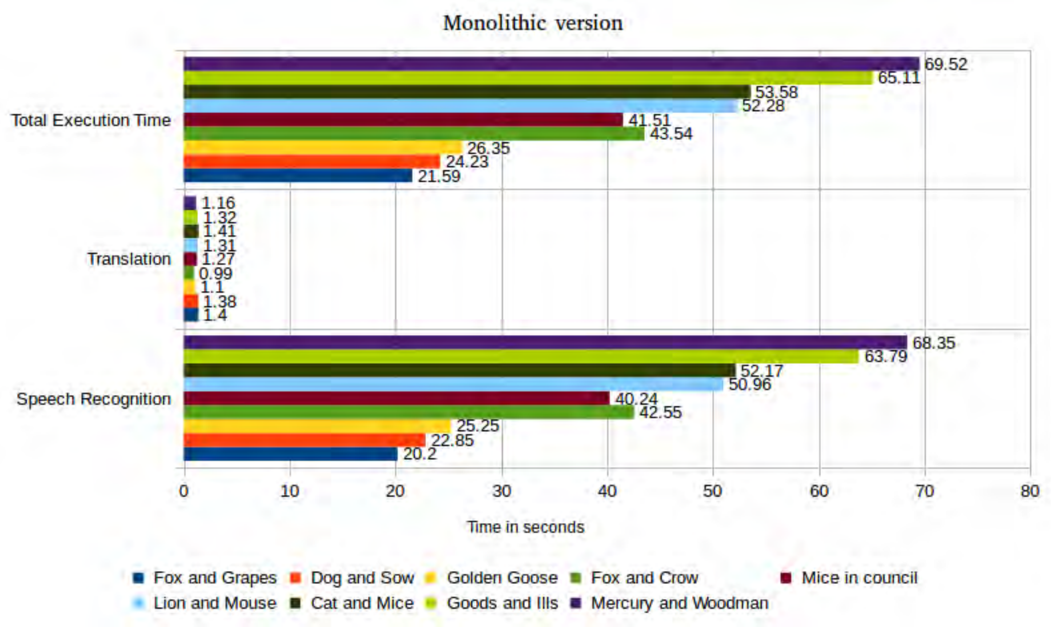

Figure 5.4: Monolithic version (per file performance)

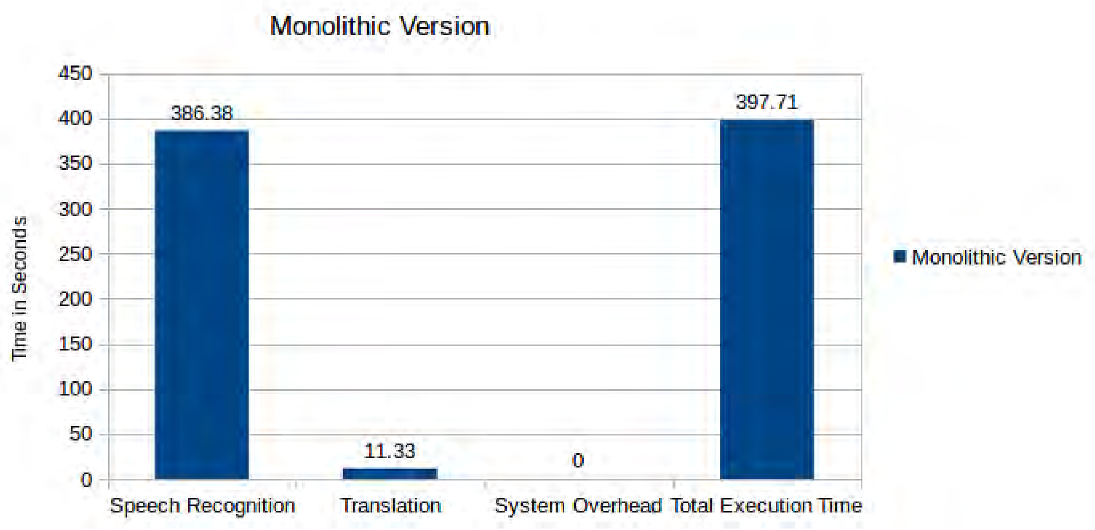

Figure 5.5: Monolithic version (overall, full experiment)

The results can be seen per file and for the full experiment in Figure 5.4 and Figure 5.5 respectively. As mentioned, it is easily observable that the speech recognition part of the application is central in determining the overall execution time.

### 5.3.2 Local Execution

For this experiment we deploy our system and run the test application in its microservices form, inside Docker containers. In this experiment no remote devices are available and all execution occurs locally on the application device. This scenario is meant to replicate conditions where either no devices are willing to join the swarm or there is no connectivity. The respective results can be seen in Figure 5.6 per file and for the full experiment compared to the Monolithic version execution in Figure 5.7.

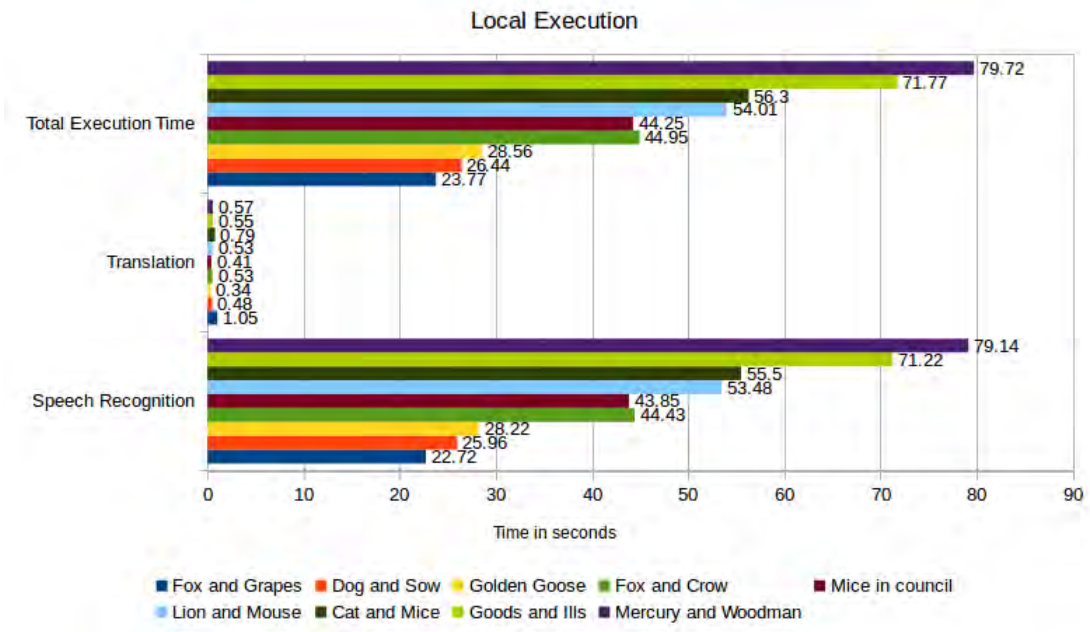

Figure 5.6: Local execution (per file performance)

In the per file results we can observer an increase of around 7% in execution time as expected. This is due to the introduced joint overhead of running Docker, the developed system's functions(like service profiling, migration policy calculations etc) and HTTP communication of microservices.

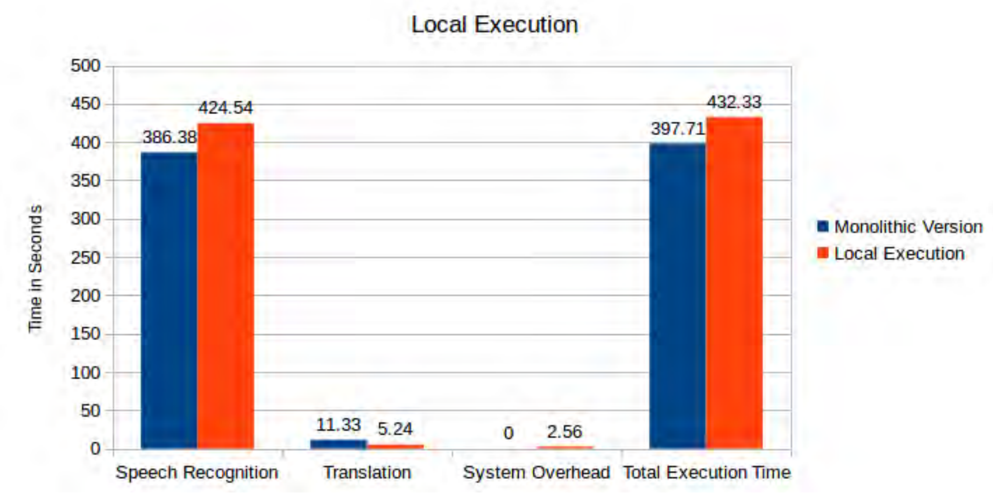

Figure 5.7: Local execution (overall, full experiment)

In a local execution environment latency is minimal to non-existent. As a result the introduced overhead by HTTP communication and microservices in general is similarly minimal. Thus the main bulk of the introduced overhead has to be attributed to Docker and system functions as in a local execution. Our system periodically (with different periods) collects service statistics, calculates the migration policy, attempts to contact the LDAP server and checks if available devices exist. Additionally the Docker platform introduces overhead of its own, for the management and running of Docker containers.

The delay is mostly present in the speech recognition service because the service takes up the most execution time; consequently due to the periodic nature of system functions a lot more function calls are likely to occur during the recognition process increasing its execution time. On the contrary no overhead is observed in the translation service. In fact the translation service seems to run faster on this experiment. As the translation service execution time is small it is less likely to collide with system functions. It should however experience some small overhead due to Docker and communication delay, we believe this isn't observed due to minor external factors.

On the overall execution time of the experiment we can also see an added overhead of 2,56 seconds. This added execution time can be attributed to the same factors as discussed but takes place in between service calls and thus was not apparent in the per-file results. This added delay brings the increase to execution time to around 9%.

It needs to be noted that the measurement do not take into account the initial setup time of the system that will be discussed separately along with service transition time.

### 5.3.3 Remote Execution

In the final experiment we test the performance of our system on the event of Remote execution. The application device, a remote device and an edge server, as described in Table 5.2, are present from the start. This scenario aims to gauge how the developed system performs in conditions where remote devices of different specifications are available to offload services to. Latency to the remote devices throughout the scenario is low and ranges from 3 to 10 ms.

The system's migration policy, as expected, chooses to offload services to the edge server as it is considerably faster to both the application and the remote device. It needs to be noted that in the this experiment our system consistently chose to offload all 3 application services. The per file results of this experiment can be seen in Figure 5.8 and in Figure 5.9 we can see a comparison of performance among all conducted experiments.

As seen the full system execution offers a decrease in execution time of around 15% compared to the monolithic version execution and of around 21% compared to the local-only execution. The overhead that occurs in-between service calls, can be seen as rather stable with a small decrease at 2.16 seconds compared to the 2.56 seconds of the local-execution experiment. This is due to the decreased Docker load, since the services execute on a faster machine.

The increase in performance can also be seen in the per-file results. Both speech recognition and translation services benefit considerably from decreased execution time, in an analogous manner. Another interesting observation is that the first 3 files of the experiment see no particular performance boost compared to the local execution. This is due to the service transition overhead that will be explained in the following section.

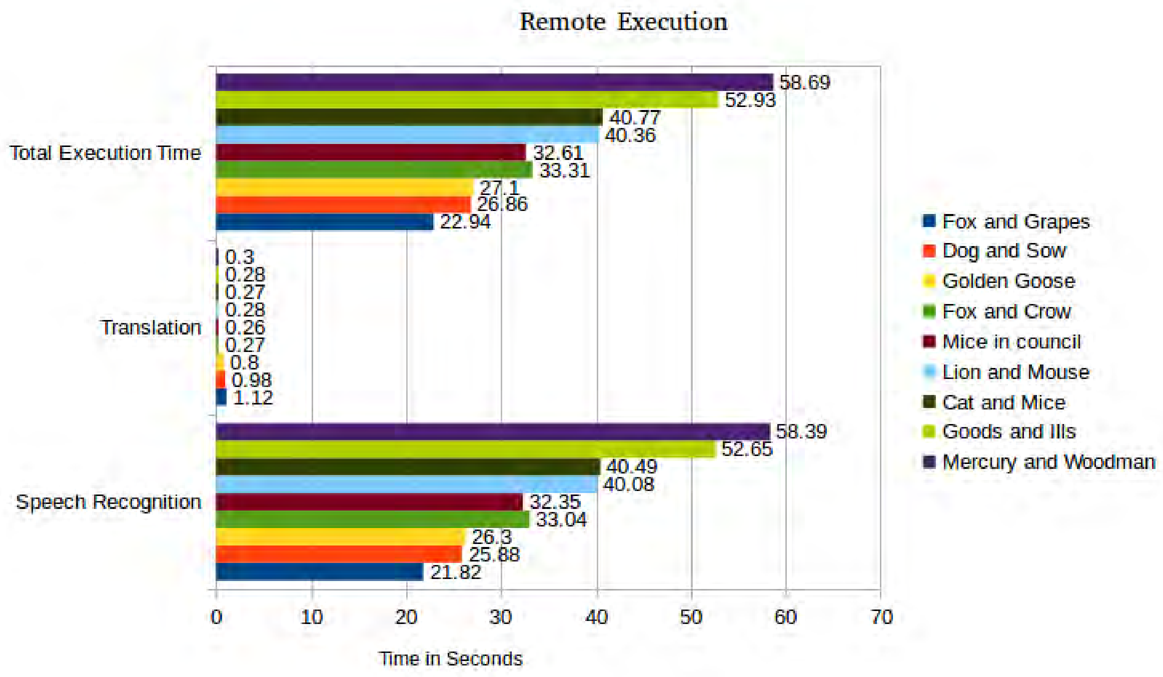

Figure 5.8: Remote execution. Per file performance

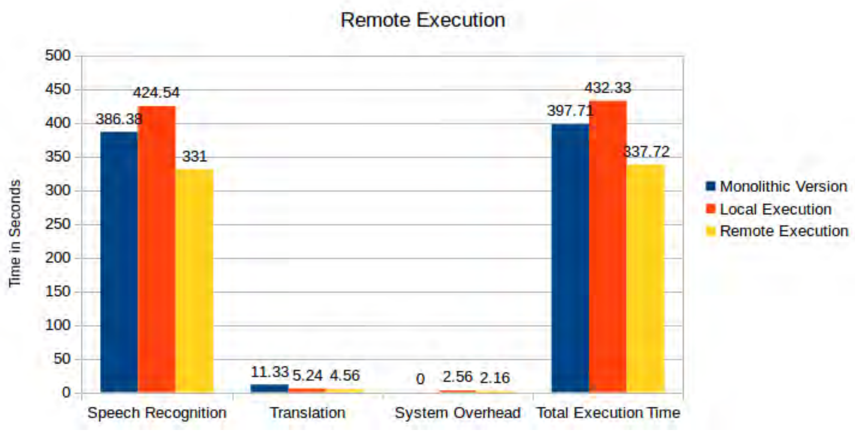

Figure 5.9: Remote execution (Overall, Full Experiment)

### 5.3.4 Setup and Service Transition Overhead

A number of experiments has also been conducted in order to measure the time needed by the system to setup and initially deploy the application services, as well as the time required for the application services to be installed/migrated and start running on a remote device once the decision to offload them has been made by the system we refer to this as the service transition time.

For the case of service installation/migration we investigate 2 scenarios. In the so-called "warm" scenario, the remote device has pre-downloaded or cached the Docker image for the service in question. In the "cold" scenario, before offloading can take place, the remote host also has to download the required image.

As seen in Table 5.4 on average the system requires 142 seconds to setup. This time can be further broken down; a considerable amount of time (about 60 seconds) are need specifically for the monitoring container of cAdvisor to be fully functional. The rest of the elapsed time is accounted to Docker creating the overlay network and launching the service tasks and the corresponding containers, and in order for the system to update the services based on the initial placement directives.

Table 5.4: Setup and Service Transition Overheads.

| | Average | Std Dev |
|---|---|---|
| Setup Time | 142.36 | 8.7 |
| Transition Time (Warm) | 66.02 | 0.24 |
| Transition Time (Cold) | 143.09 | 6.8 |
| Offloading Decision | 5.22 | 1.6 |

The service transition time averages on 66 seconds for a warm remote device, and 143 seconds for the cold scenario, the added time being due to the downloading of Docker images. This overhead directly relates to the number of services needed to migrate. In our case all 3 services of the application were migrated by the system. In Table 5.5 we can see a typical breakdown of transition time for each service. It is easily apparent that the transition time is roughly equal for all services in the "warm" scenario. This of course differs on the "cold" scenario, where individual service transition time is dependent on the image size and the time it needs to be download on the remote device.

Assuming that a valid remote device is available the developed system

Table 5.5: Individual Service Transition Overhead (Warm)

| | **Average** | **Std Dev** |
|---|---|---|
| **Speech Service** | 22.12 | 1.58 |
| **Translation Service** | 21.94 | 1.24 |
| **Sender Service** | 21.96 | 0.27 |

takes on average 5.22 seconds to make the offloading decision. This translates to the time need to begin migrating the services and is considerably smaller compared to the time needed for service transition. This value added to the average service transition time, yields on average 71.24 seconds until the user experiences benefit from the system in the "warm" scenario and 148.31 seconds in the "cold" scenario. These values also represent a minimum running time after which utilizing the developed system begins to make sense for a user.

While these overheads are considerable, in practice, they are almost unobservable to the user. This is because our system and Docker updates service on a "start-first" policy, meaning that services are readied on the background. As long as a service is not ready, all related calls are routed to the active service instance that is running on the current host device. Consequently, from the application's and user perspective, the benefits of offloading can be enjoyed without any visible performance degradation.

## 5.4 Summary

Overall the developed system seems to be able to, under specific circumstances, offer considerable performance gain for compute intensive application services. The introduced overheads constitute a small percentage, even though not insignificant, of the overall execution time in the experiments conducted even compared to the total absence of microservices architecture and its communication costs. This holds true even in worst-case scenarios where no available remote computing resources exist. It is also deemed important that most time-consuming operations are performed in the background, thus leaving the end user relatively unaffected.

It needs to be noted however that the experiments were conducted in a favourable network environment and their duration was considerable, at over 5 minutes. Applications that do not have computationally intensive components or have really fast execution times, can be unfit to take advantage of the developed system. In addition, even though the system is design to accommodate dynamic changes of present devices, these changes need to happen in a slow manner for the system to have time to converge. Otherwise the overhead costs of the system can be paid with no significant performance gain.

# Chapter 6

# Conclusions & Future Work

We have presented our system for the dynamic offloading of application services to devices at the network edge. To achieve our goal of opportunistically exploiting nearby devices we have used microservices architecture and built our system on top of Docker swarm. A simple migration policy has also been proposed. In order to evaluate our system, we conducted a series of experiment using a speech recognition and translation microservices application that we build. We have found that our system, under specific conditions concerning execution time and presence of remote devices, can provide a considerable performance boost to application services.

In the future our system can be extended or improved upon in various ways. The migration policy proposed can be more fine grained to include metrics like energy consumption and the offloading decision could be made to accommodate more complex information like expected device presence, expected service usage etc. The system could also be extended to become less greedy and aim to improve the overall network and system load of all devices present. Finally the system's response to the dynamic arrival and departure of devices and other application services could be improved.

# References

[1] Friedemann Mattern and Christian Floerkemeier. “From Active Data Management to Event-based Systems and More”. In: ed. by Kai Sachs, Ilia Petrov, and Pablo Guerrero. Berlin, Heidelberg: Springer-Verlag, 2010. Chap. From the Internet of Computers to the Internet of Things, pp. 242–259. URL: http://dl.acm.org/citation.cfm?id=1985625.1985645.

[2] *The NIST definition of cloud computing.* URL: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf.

[3] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, et al. “Fog Computing and Its Role in the Internet of Things”. In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing.* MCC ’12. Helsinki, Finland: ACM, 2012, pp. 13–16. ISBN: 978-1-4503-1519-7. DOI: 10.1145/2342509.2342513. URL: http://doi.acm.org/10.1145/2342509.2342513.

[4] *Mobile Edge Computing: Introductory Technical Whitepaper - ETSI.* URL: https://portal.etsi.org/Portals/0/TBpages/MEC/Docs/Mobile-edge_Computing_-_Introductory_Technical_White_Paper_V1%2018-09-14.pdf.

[5] James Lewis Martin Fowler. *Microservices, a definition of this new architectural term.* URL: https://martinfowler.com/articles/microservices.html.

[6] *Docker: About.* URL: https://www.docker.com/what-docker.

[7] *Docker Swarm.* URL: https://docs.docker.com/engine/swarm/.

[8] *Representational State Transfer (REST).* URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[9] *LXC: LinuX Containers.* URL: https://linuxcontainers.org/.

[10] *Anatomy of a Container: Namespaces, cgroups and Some Filesystem Magic - LinuxCon.* URL: `https://www.slideshare.net/jpetazzo/anatomy-of-a-container-namespaces-cgroups-some-filesystem-magic-linuxcon`.

[11] *The Raft Consensus Algorithm.* URL: `https://raft.github.io/`.

[12] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, et al. "Maui". In: *Proceedings of the 8th international conference on Mobile systems, applications, and services - MobiSys '10* (2010), p. 49. ISSN: 10058885. DOI: `10.1145/1814433.1814441`. URL: `http://portal.acm.org/citation.cfm?doid=1814433.1814441`.

[13] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, et al. "CloneCloud". In: *Proceedings of the sixth conference on Computer systems - EuroSys '11* (2011), p. 301. ISSN: 10058885. DOI: `10.1145/1966445.1966473`. URL: `http://portal.acm.org/citation.cfm?doid=1966445.1966473`.

[14] Sokol Kosta, Andrius Aucinas, Pan Hui, et al. "ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading". In: *Proceedings - IEEE INFOCOM* (2012), pp. 945–953. ISSN: 0743166X. DOI: `10.1109/INFCOM.2012.6195845`. arXiv: `1105.3232`.

[15] Cong Shi, Karim Habak, Pranesh Pandurangan, et al. "COSMOS : Computation Offloading as a Service for Mobile Devices". In: *Proceedings of MobiHoc* (2014), pp. 287–296. DOI: `10.1145/2632951.2632958`.

[16] Yongin Kwon, Sangmin Lee, Hayoon Yi, et al. "Mantis: automatic performance prediction for smartphone applications". In: (June 2013), pp. 297–308.

[17] Ms Gordon, Da Jamshidi, and Scott Mahlke. "COMET: code offload by migrating execution transparently". In: *Proceedings of the 10th ...* (2012), pp. 93–106. URL: `https://www.usenix.org/system/files/conference/osdi12/osdi12-final-11.pdf`.

[18] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, et al. "Cuckoo: a computation offloading framework for smartphones". In: *Mobile Computing, Applications, ...* (2012), pp. 59–79. ISSN: 18678211. DOI: `10.1007/978-3-642-29336-8_4`. URL: `http://www.springerlink.com/index/U6777405301NP063.pdf`.

[19] Rob Van Nieuwpoort, Jason Maassen, Gosia Wrzesińska, et al. "Ibis: A Flexible and Efficient Java-based Grid Programming Environment". In: 17 (Feb. 2005), pp. 1079–1107.

[20] Cong Shi and Vasileios Lakafosis. "Serendipity: Enabling remote computing among intermittently connected mobile devices". In: *... Networking and Computing* (2012), pp. 145–154. ISSN: 10848045. DOI: 10.1145/2248371.2248394. URL: http://dl.acm.org/citation.cfm?id=2248394.

[21] Karim Habak, Mostafa Ammar, Khaled A. Harras, et al. "Femto Clouds: Leveraging Mobile Devices to Provide Cloud Service at the Edge". In: *Proceedings - 2015 IEEE 8th International Conference on Cloud Computing, CLOUD 2015* (2015), pp. 9–16. ISSN: 2159-6182. DOI: 10.1109/CLOUD.2015.12.

[22] Gabriel Orsini, Dirk Bade, and Winfried Lamersdorf. "Computing at the Mobile Edge: Designing Elastic Android Applications for Computation Offloading". In: *Proceedings - 2015 8th IFIP Wireless and Mobile Networking Conference, WMNC 2015* (2016), pp. 112–119. ISSN: 19328184. DOI: 10.1109/WMNC.2015.10. arXiv: 1510.00888.

[23] Alexander Pokahr and Lars Braubach. "The Active Components Approach for Distributed Systems Development". In: *Int. J. Parallel Emerg. Distrib. Syst.* 28.4 (Aug. 2013), pp. 321–369. ISSN: 1744-5760. DOI: 10.1080/17445760.2013.785546. URL: http://dx.doi.org/10.1080/17445760.2013.785546.

[24] Sherif Abdelwahab, Bechir Hamdaoui, Mohsen Guizani, et al. "Replisom: Disciplined Tiny Memory Replication for Massive IoT Devices in LTE Edge Cloud". In: *IEEE Internet of Things Journal* 3.3 (2016), pp. 327–338. ISSN: 23274662. DOI: 10.1109/JIOT.2015.2497263.

[25] Michael Till Beck, Sebastian Feld, Andreas Fichtner, et al. "ME-VoLTE: Network functions for energy-efficient video transcoding at the mobile edge". In: (Mar. 2015), pp. 38–44.

[26] Shiqiang Wang, Rahul Urgaonkar, Murtaza Zafer, et al. "Mobility-Driven Service Migration in Mobile". In: (). DOI: 10.1109/MILCOM.2014.145. arXiv: 1503.05141.

[27] Xu Chen, Lei Jiao, Wenzhong Li, et al. “Efficient Multi-User Computation Offloading for Mobile-Edge Cloud Computing”. In: *IEEE/ACM Trans. Netw.* 24.5 (Oct. 2016), pp. 2795–2808. ISSN: 1063-6692. DOI: `10.1109/TNET.2015.2487344`. URL: `https://doi.org/10.1109/TNET.2015.2487344`.

[28] S. Sardellitti, G. Scutari, and S. Barbarossa. “Joint optimization of radio and computational resources for multicell mobile cloud computing”. In: *IEEE Workshop on Signal Processing Advances in Wireless Communications, SPAWC* 2014-Octob.October (2014), pp. 354–358. ISSN: 2373-776X. DOI: `10.1109/SPAWC.2014.6941749`. arXiv: `1412.8416`.

[29] Vlado Stankovski, Jernej Trnkoczy, Salman Taherizadeh, et al. “Implementing time-critical functionalities with a distributed adaptive container architecture”. In: *Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services - iiWAS '16* (2016), pp. 453–457. DOI: `10.1145/3011141.3011202`. URL: `http://dl.acm.org/citation.cfm?doid=3011141.3011202`.

[30] Joao Rufino, Muhammad Alam, Joaquim Ferreira, et al. “Orchestration of containerized microservices for IIoT using Docker”. In: *Proceedings of the IEEE International Conference on Industrial Technology* (2017), pp. 1532–1536. DOI: `10.1109/ICIT.2017.7915594`.

[31] Paolo Bellavista and Alessandro Zanni. “Feasibility of Fog Computing Deployment based on Docker Containerization over RaspberryPi”. In: *Proceedings of the 18th International Conference on Distributed Computing and Networking - ICDCN '17* (2017), pp. 1–10. DOI: `10.1145/3007748.3007777`. URL: `http://dl.acm.org/citation.cfm?doid=3007748.3007777`.

[32] *Eclipse Kura: Open Source Framework for IoT*. URL: `https://www.eclipse.org/kura/`.

[33] Roberto Morabito and Nicklas Beijar. “Enabling Data Processing at the Network Edge through Lightweight Virtualization Technologies”. In: *2016 IEEE International Conference on Sensing, Communication and Networking, SECON Workshops 2016* 607728 (2016). DOI: `10.1109/SECONW.2016.7746807`.

[34] Andrew Machen, Shiqiang Wang, Kin K. Leung, et al. "Migrating running applications across mobile edge clouds". In: *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking - MobiCom '16* (2016), pp. 435–436. DOI: 10.1145/2973750.2985265. URL: http://dl.acm.org/citation.cfm?doid=2973750.2985265.

[35] Flavio Ramalho and Augusto Neto. "Virtualization at the network edge: A performance comparison". In: *WoWMoM 2016 - 17th International Symposium on a World of Wireless, Mobile and Multimedia Networks* (2016). DOI: 10.1109/WoWMoM.2016.7523584.

[36] Cristian Gadea, Mircea Trifan, Dan Ionescu, et al. "A microservices architecture for collaborative document editing enhanced with face recognition". In: *SACI 2016 - 11th IEEE International Symposium on Applied Computational Intelligence and Informatics, Proceedings* (2016), pp. 441–446. DOI: 10.1109/SACI.2016.7507409.

[37] David Jaramillo, Duy V. Nguyen, and Robert Smart. "Leveraging microservices architecture by using Docker technology". In: *Conference Proceedings - IEEE SOUTHEASTCON* 2016-July (2016), pp. 1–4. ISSN: 07347502. DOI: 10.1109/SECON.2016.7506647.

[38] *Docker Compose File Reference*. URL: https://docs.docker.com/compose/compose-file/.

[39] *Open LDAP*. URL: https://www.openldap.org/.

[40] *cAdvisor*. URL: https://github.com/google/cadvisor.

[41] *Passmark CPU Benchmarks*. URL: https://www.cpubenchmark.net/.

[42] *CMU Sphinx*. URL: https://cmusphinx.github.io/.

[43] *Carnegie Mellon University*. URL: https://www.cmu.edu/.

[44] *Aperitum, a free/open source machine translation platform*. URL: http://wiki.apertium.org/wiki/Documentation.

[45] *LibriVox Aesop's Fables*. URL: https://librivox.org/aesops-fables-volume-1-fables-1-25/.