FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Tommi Järvenpää

Distributed Microservices Evaluation in Edge Computing

Bachelor's Thesis
Degree Programme in Computer Science and Engineering
May 2020

Järvenpää T. (2020) Distributed Microservices Evaluation in Edge Computing. University of Oulu, Degree Programme in Computer Science and Engineering, 38 p.

ABSTRACT

Current Internet of Things applications rely on centralized cloud computing when processing data. Future applications, such as smart cities, homes, and vehicles, however, generate so much data that cloud computing is unable to provide the required Quality of Service. Thus, edge computing, which pulls data and related computation from distant data centers to the network edge, is seen as the way forward in the evolution of the Internet of Things.

The traditional cloud applications, implemented as centralized server-side monoliths, may prove unfavorable for edge systems, due to the distributed nature of the network edge. On the other hand, the recent development practices of containerization and microservices seem like an attractive choice for edge application development. Containerization enables edge computing to use lightweight virtualized resources. Microservices modularize application on the functional level into small, independent packages.

This thesis studies the impact of containers and distributed microservices on edge computing, based on service execution latency and energy consumption. Evaluation is done by developing a monolithic and a distributed microservice version of a user mobility analysis service. Both services are containerized with Docker and deployed on resource-constrained edge devices to conduct measurements in real-world settings.

Collected results show that centralized monoliths provide lower latencies for small amounts of data, while distributed microservices are faster for large amounts of data. Partitioning services onto multiple edge devices is shown to increases energy consumption significantly.

Keywords: Internet of Things, Container, Latency, Energy consumption

Järvenpää T. (2020) Hajautettujen Mikropalveluiden Arviointi Reunalaskennassa. Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 38 s.

TIIVISTELMÄ

Nykyiset Esineiden Internet -järjestelmät hyödyntävät keskitettyä pilvilaskentaa datan prosessointiin. Tulevaisuuden sovellusalueet, kuten älykkäät kaupungit, kodit ja ajoneuvot tuottavat kuitenkin niin paljon dataa, ettei pilvilaskenta pysty täyttämään tarvittavia sovelluspalveluiden laatukriteerejä. Pilvipohjainen sovellusten toteutus on osoittautunut sopimattomaksi hajautetuissa tietoliikenneverkoissa tiedonsiirron viiveiden takia. Täten laskennan ja datan siirtämistä tietoliikenneverkkojen päätepisteisiin reunalaskentaa varten pidetään tärkeänä osana Esineiden Internetin kehitystä.

Pilvisovellusten perinteinen keskitetty monoliittinen toteutus saattaa osoittautua sopimattomiksi reunajärjestelmille tietoliikenneverkkojen hajautetun infrastruktuurin takia. Kontit ja mikropalvelut vaikuttavat houkuttelevilta vaihtoehdoilta reunasovellusten suunnitteluun ja toteutukseen. Kontit mahdollistavat reunalaskennalle kevyiden virtualisoitujen resurssien käytön ja mikropalvelut jakavat sovellukset toiminnallisella tasolla pienikokoisiin itsenäisiin osiin.

Tässä työssä selvitetään konttien ja hajautettujen mikropalveluiden toteutustavan vaikutusta viiveeseen ja energiankulutukseen reunalaskennassa. Arviointi tehdään todellisessa ympäristössä toteuttamalla mobiilikäyttäjien liikkumista kaupunkialueella analysoiva keskitetty monoliittinen palvelu sekä vastaava hajautettu mikropalvelupohjainen toteutus. Molemmat versiot kontitetaan ja otetaan käyttöön verkon reunalaitteilla, joiden laskentateho on alhainen.

Tuloksista nähdään, että keskitettyjen monoliittien viive on alhaisempi pienille datamäärille, kun taas hajautetut mikropalvelut ovat nopeampia suurille määrille dataa. Sovelluksen jakaminen usealle reunalaitteelle kasvatti energiankulutusta huomattavasti.

Avainsanat: Esineiden Internet, Kontti, Viive, Energiankulutus

TABLE OF CONTENTS

FOREWORD

I would like to thank M.Sc. Lauri Lovén for helping me with R, Ph.D. Leena Ruha, the second examiner, and especially D.Sc. (Tech.) Teemu Leppänen, the supervisor of this thesis. I would also like to thank Professor Jukka Riekki for the opportunity to work at Ubicomp. Working here has been an excellent experience.

This thesis was financially supported by the MEC-AI-project, funded by the Future Makers program of Jane and Aatos Erkko Foundation and Technology Industries of Finland Centennial Foundation, and by the Academy of Finland 6Genesis Flagship (grant 318927).

Oulu, 6th May, 2020

Tommi Järvenpää

ABBREVIATIONS

IoT	Internet of Things
QoS	Quality of Service
VM	Virtual Machine
OS	Operating System
UI	User Interface
AP	Access Point
ED	Edge Device
UE	User Equipment
CPU	Central Processing Unit
SBC	Single-Board Computer
RPi	Raspberry Pi
API	Application Programming Interface
SOA	Service-Oriented Architecture
DB	DataBase
MTA	Mobility Trace Analysis
IP	Internet Protocol
MMS	Mobility MicroService
SCP	Secure Copy Protocol
CSV	Comma-Separated Values

1. INTRODUCTION

The future applications of the Internet of Things (IoT), such as smart cities, vehicles, and homes will increase the volume of generated data massively [1]. Smart cars, for example, will generate approximately 4000 Gigabytes of data per hour [2] that should be processed in near real-time to avoid accidents. Today, IoT system architectures are based on centralized cloud computing [3]. However, with cloud computing, data must be transmitted to centralized data centers before processing, accumulating latency, consuming network infrastructure resources, and energy, which reduces the battery life of resource-constrained end devices. Current network bandwidths are also unable to handle the volume of data, creating bottlenecks for computation [4]. Cloud computing is thus ill-equipped to provide the necessary Quality of Service (QoS), i.e. latency. Because of this, a new computational paradigm, edge computing is seen as the way forward [5].

Edge computing pulls application resources, i.e. computation and data from the distant cloud to the network edge into close proximity of IoT devices. Moving processing as close as one hop away [6] reduces network infrastructure usage and application execution latencies, consequently improving the Quality of Experience for users.

In the edge approach, application-specific components are deployed to resource-rich edge servers, which have large amounts of memory and computational power [7] in a distributed fashion inside self-contained packages, i.e. Virtual Machines (VM) [3]. VMs typically have a large overhead in the form of a guest operating system (OS) and hardware emulation, making deployment, instantiation, scaling, and relocation very resource consuming. However, containers have recently emerged as a lightweight alternative for VMs [8]. Containers include a single process and its execution environment, but unlike VMs, they do not virtualize hardware. This makes the size of containers smaller and further enables the deployment of distributed applications to the edge.

As server-side IoT applications are typically developed as monoliths [9], i.e. applications with a single code base, edge computing and the usage of containers are not enough to solve problems with application development and scalability. To address these issue, a new approach to application development has emerged. Microservices [9, 10] divide applications on the individual process level into independent and lightweight packages, which can then be containerized and deployed to the edge. The benefits include independent development and deployment, isolation, and scalability. These benefits make microservices an attractive choice for edge application development.

In this thesis, an edge computing service is implemented as a monolith and in accordance with the microservices paradigm. Both versions are then containerized and deployed on low-resource edge devices in order to create two real-world prototypes of a user mobility analysis service. The effects of the two software architectures on two important edge computing workload allocation optimization metrics latency, and energy consumption [4] are then compared in real-world settings.

The rest of the thesis is organized as follows. Section 2 provides an overview of the IoT system architecture, edge computing, virtualization, and microservices.

Section 3 presents the structure and processing of the mobility data used in the services, the design and implementation of both services, the measurement testbed, and the results. In Section 4, the gathered results are analyzed and discussed. Section 5 concludes the thesis.

2. BACKGROUND

2.1. Internet of Things

Internet of Things, originally coined by Kevin Ashton in 1999 for the usage of radio frequency identification tags in supply chain management [11] has been a major topic of research. Rose et al. define IoT as a scenario where internet connectivity and computational capabilities are extended to everyday objects allowing them to generate, exchange, and consume data [12], but no universal definition exists. The goal, however, is to make devices operate without human interaction through massive distributed networks of heterogeneous devices connected across the internet.

IoT systems can be applied from personal to national level. In everyday life, IoT devices can e.g. be seen as home appliances and wearable devices. Different industries such as healthcare and transportation have their applications, and on the national level, IoT systems include smart water and energy grids. The combination of such technologies can be used to create smart cities, e.g.

IoT is able to provide seamless ubiquitous computing, but to achieve this, hardware, middleware, and data analysis tools are required[1]. Current IoT systems rely on a cloud-centric architecture that contains a cloud, edge, and perception layer [3, 13]. IoT applications and the data analysis tools reside in the cloud layer and use the data generated by the perception layer that they receive through the edge layer. These layers are depicted in Figure 1.

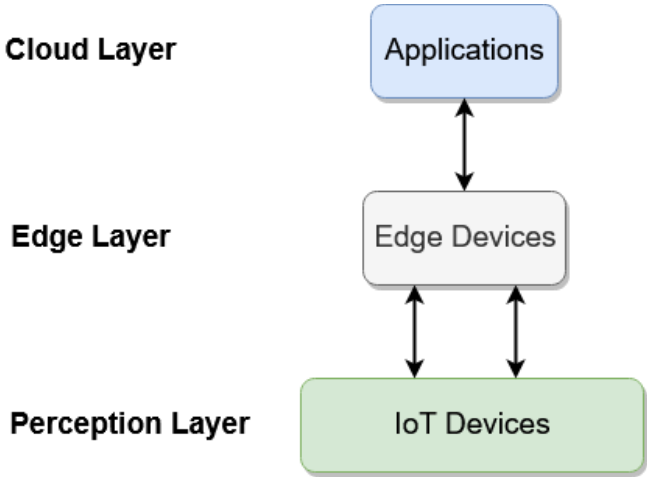


Figure 1. Cloud-centric IoT architecture.

2.1.1. Cloud Layer

The cloud layer is the topmost layer of the IoT system architecture. It consists of applications, hosted in data centers, i.e. the cloud, that provide the required visualization and interpretation tools. The resources of the cloud layer are

virtually unlimited, but distances from IoT devices are typically long and the generated data must be transmitted across multiple networks.

The cloud layer is centered around cloud computing, which has dominated many areas of the IT industry. Cloud computing, defined by NIST as a model for enabling ubiquitous, convenient, on-demand network access to shared and configurable resources, has five essential characteristics [14]:

- On-demand self-service
- Broad network access
- Resource pooling
- Rapid elasticity
- Measured services

This means that resources are shared between multiple users, automatically controlled and optimized, as well as, rapidly scaled up or down when needed. The cloud can also be accessed with any device with internet connectivity.

Applications in the cloud follow the physically two-tiered client-server model [15], which is an example of a centralized system. As the model name suggests, in centralized systems computational elements, i.e. nodes, are divided into clients and a centralized server which communicate over a network. Nodes refer to hardware devices or software processes. Often only the User Interface (UI) is implemented client-side while the rest of the application, including processing and data, is located server-side. The server provides a specific service and waits for incoming requests from clients. When a request is received, service execution starts, and after completion, the server sends a reply back to the client. In the cloud-centric IoT architecture, IoT devices are the nodes and the cloud is the server.

The cloud layer suffers from the problems of cloud computing [16, 17], and centralized systems [15]. Major cloud issues for IoT include long distances between devices and data centers, and privacy, as cloud providers have full control over data stored in the cloud. Problems in centralized systems include the server being a single point of failure and network reliability. If the server crashes or becomes otherwise unavailable, the whole system stops working. Network transmission failures also hinder system performance as requests and replies can be lost.

2.1.2. Edge Layer

The edge layer is located in the middle of the IoT system architecture. In the cloud-centric approach, the main objective of this layer is to transmit data between the two other layers [13]. Networking infrastructure in the form of hardware and middleware supports this bidirectional communication. The edge layer includes devices such as servers, routers, Access Points (AP), gateways, and

cellular base stations that use various communication technologies and protocols due to device heterogeneity.

In the cloud-centric IoT architecture, the computational resources at the network edge are left untapped. However, edge computing, a paradigm that has grown in popularity during recent years [5], makes the utilization of these resources possible. Edge computing is defined by Open Edge Computing as computation taking place at the network edge in small data centers located near the users [18]. However, in academic research, definitions for edge computing vary greatly as no universally accepted definition exists.

The edge computing paradigm pulls computation from centralized cloud data centers to the network edge by exploiting the computational resources of Edge Devices (ED) capable of processing, storing, and visualizing data. Utilizing edge computing and the close proximity of EDs provides the following benefits [4, 6]:

- Improved connectivity
- Reduced end-to-end latency
- Reduced energy and network infrastructure resource consumption
- Better privacy

Edge computing can be used to enhance cloud computing by masking outages and pre-processing device-generated data with EDs on the route to the cloud [4] or even completely remove the need for cloud computing. Overall edge computing provides better QoS for IoT than cloud computing. However, edge computing suffers from issues, such as:

- Programmability of heterogeneous IoT systems
- Finding the balance between latencies created and energy consumed by computation and communication.
- Handling crashes and ensuring that the whole system is not rendered useless in case of one part failing

Existing edge computing solutions provide application resources requested by devices inside VMs that are then executed in close proximity to devices [3]. An example of this is an edge-centric code offloading system for Android-based devices implemented in [19].

The edge layer is an example of a distributed system [15]. Such systems are characterized by two features:

- Consisting of autonomous nodes
- Appearing as a single coherent system

Distributed systems can contain a handful or even millions of physically distributed heterogeneous nodes. These nodes are programmed to achieve common goals by exchanging messages over a network [15]. Depending on the system overlay network, each node can have either a set of known neighbors or

a list of application-specific nodes to communicate with. These connections, as well as the difference between distributed and centralized systems, are illustrated in Figure 2.

To address the heterogeneity of nodes, middleware is utilized in distributed systems. Middleware adds a layer between the node's OS and applications, that provides a unified interface everywhere in the system and hides differences in node OSs and underlying device hardware. Common types of middleware services include communication, transaction, and service composition.

When designing distributed systems, certain goals must be met to make the process worthwhile and to have the system fulfill the characteristics of distributed systems. These goals include:

- Easy resource sharing between users
- All aspects of distribution being transparent
- Scaling being easy and possible in the size, geographical, and administrative dimension
- The system always acting accordingly to user expectations
- Components being small, adaptable, portable and easily integrated into or used by other systems

Striving for the single coherent system characteristic, however, creates problems as nodes can fail at any time, and the effects can be very unexpected. A failed node can, for example, have no noticeable effect or bring the whole system to a grinding halt. A high degree of transparency also reduces system performance, and achieving full transparency is impossible.

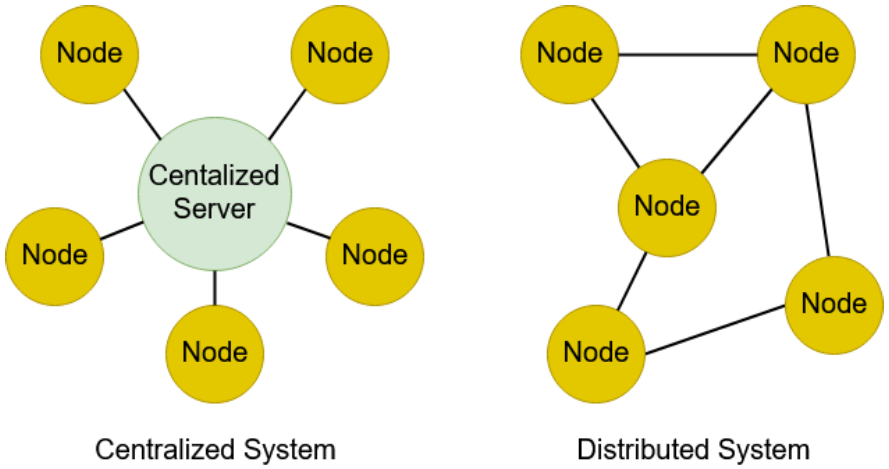


Figure 2. Node connections in a centralized and distributed system.

Since accommodating for increased traffic is extremely expensive or impossible for an unscalable system [20], achieving scalability with any centralized or distributed system is crucial for ensuring proper QoS. However, these architectures scale differently.

System scalability consists of three dimensions, size, geographical, and administrative [15], which all affect the system differently.

- Size: How easily users and resources can be added to a system, and how they affect its performance
- Geographical: How distances effect communication latencies
- Administrative: How easily a system spanning multiple independent organization can be managed

From the three dimensions of scalability, centralized systems mainly scale in the size dimension. When traffic to a centralized server increases, the server can only be scaled up, i.e. upgrading the CPU, adding memory and storage, etc. However, scaling up is effective up to a certain point, and as request volume increases, even modern machines will eventually run into problems. If clients are connected to the server over a network, geographical scaling comes into play in the form of transfer speed and physical distance [20]. Aside from well-optimized transfer protocols, centralized systems do not have good means to address increases in latencies and lost messages due to wide-area network reliability. For centralized systems, administrative scalability only determines the effect of the increased workload on the administrative workload [21], making administrative the least significant dimension.

Compared to centralized, distributed systems are a lot more scalable as they can scale in all three dimensions of scaling. In the size dimension, distributed systems can scale up as well as out. Scaling out means adding more nodes to the system. Adding nodes reduces computational bottlenecks and distances between nodes. Well executed geographical scalability negates the effects of latencies created by the distance and reduced reliability of wide-area networks. The importance of administrative scalability is increased as distributed systems can span multiple independent organizations making administration tasks more complex. Systems should thus be made easily operable to avoid a large increase in administrative workload.

When scaling systems, one should keep in mind that as a system gets larger, operating it becomes difficult for humans, and fine-grained control is lost [21].

2.1.3. Perception Layer

The perception layer is the bottom-most layer of the IoT system architecture. It controls and interacts with physical devices, as well as collects data [13]. Data collection happens through ubiquity enabling hardware including stationary sensor networks, actuators, and sensors embedded into User Equipment (UE), such as smartphones. The volume of data generated by devices is massive [1], and in many use cases requires near real-time processing and a high level of privacy.

Modern UEs are capable of locating themselves through different means, including the Global Positioning System or by using Wi-Fi APs with the Wireless

Positioning System. Monitoring user movements is crucial for providing advanced mobile services. The ability of UEs to locate themselves enables opportunistic mobile computing, i.e. opportunistically leveraging resources from other devices in the environment [22], in this case, resources are leveraged from the network edge, creating a distributed system. For example, location-aware crowdsensing solutions have been developed for the edge with microservices [23] and web-integrated smart objects [24].

Since processing data consumes energy, the most important resources for devices in the perception layer, it is offloaded to the higher layers through an interface. However, offloading computation to the cloud or the edge still presents a trade-off between latency and energy consumption [4, 25].

2.2. Development Technologies for Edge Computing

In edge computing, the focus of this thesis, virtualization is used to provide resources as virtual packages to devices requesting them [3], and the microservices paradigm [9, 10] is used to design suitable applications on the distributed edge layer. Due to these reasons, an overview of the two technologies is provided.

2.2.1. Virtualization

As described by Campbell and Jeronimo [26], virtualization is the process of decoupling an operating system from the underlying hardware of a physical machine. Virtualization can thus be thought of as a computer implemented in software running inside another computer. Virtualization enables us to run multiple isolated instances of heterogeneous OSs on the same host machine. An OS running inside a virtual environment is called a Virtual Machine. VMs emulate physical hardware such as Central Processing Units (CPU), memory and physical storage. Benefits of virtualization include more efficient use of available resources, easier system management, and the ability to deliver pre-configured environments, i.e. an application and its libraries, binaries, and data.

Virtualization is a crucial technology for edge computing for three reasons [27]

- An application running inside a VM works on any ED that supports virtualization regardless of the underlying hardware
- Abstraction of underlying hardware makes application development easier
- VMs isolate applications from each other, and thus the failure of one will not have an effect on others in a well-designed system

However, the inclusion of a guest OS and the virtualization of hardware makes VMs multiple gigabytes in size making deployment, instantiation, relocation, and maintenance resource consuming for edge computing. Containers have recently emerged as a lightweight alternative for VMs [8]. Containers provide by and large the same benefits as VMs, but their size is smaller as they only

emulate the guest OS and not the hardware. Unlike VMs, containers are typically designed to run a single process allowing applications to be partitioned into small independently developable and deployable packages. A container typically contains an application-specific functionality, and potentially the data it uses. Figure 3 presents the architectural differences between VMs and Docker containers. The performance of containers has been shown to be close to bare-metal for edge computing [28].

Real-world IoT services, e.g., can consist of multiple containers, making their management arduous. Such problems can be solved by utilizing container orchestration platforms, such as Kubernetes [29], which is a cloud-based, open-source platform for container management automation.

However, when deciding between VM and container-virtualization, some things should be noted. In [30], it is argued that when concerns for edge computing attributes such as platform integrity and multi-tenant isolation are dominant, VMs should be used instead of containers. In addition, in [27], it is stated that container compromisation or crashes can in some cases affect the whole system. As an example of container utilization in edge computing, a low-latency video analytics system based on containers was implemented in [31]. In [32] the usage of Docker containers was evaluated for IoT edge computing using Single-Board Computers (SBC). The results show that the activation times of containers on SBC are relatively short and utilizing container-virtualization has a nearly negligible effect on performance.

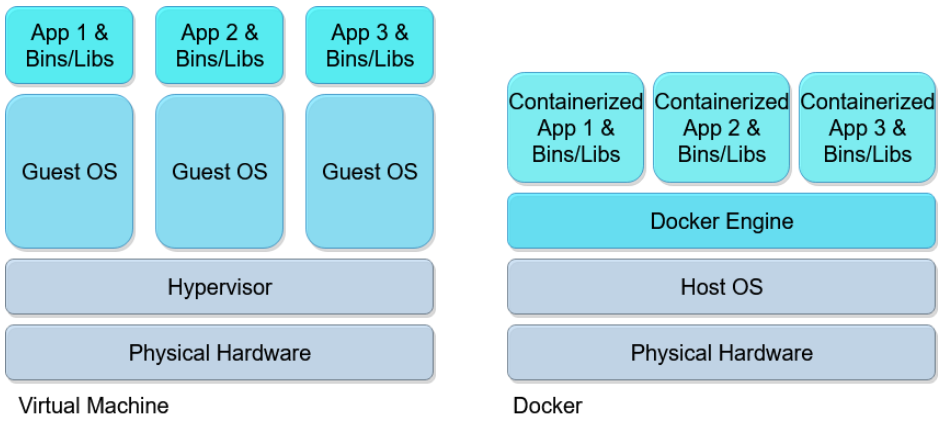


Figure 3. VM and Docker container architectures.

2.2.2. Microservices

Microservices are an auspicious paradigm for distributed IoT systems, as their design goals match each other well. Microservices modularize applications at the process level, transforming processes into individually developable and deployable packages, that communicate with each other through lightweight Application Programming Interfaces (API). Such partitioning turns applications into sets of

loosely-coupled small and autonomous services. In the real-world, applications decompose into hundreds or even thousands of microservices [9].

Microservices are implemented and deployed with containers. Compared to the traditionally monolithic server-side applications [9] and VM deployment, this combination of two lightweight technologies consumes fewer resources making it beneficial for edge computing. Other benefits of the microservices system architecture include [10]:

- Heterogeneity of technologies as each microservice can be created using different technologies
- Isolation of failure
- Scaling in all three dimensions of scalability
- Composability
- Reduced complexity of individual processes

While no established microservices development practice exists [33], microservices have adopted the Service-Oriented Architecture (SOA) for distributed systems [33]. Thus it is important to examine their similarities and differences [34]. Both software architectures are used to partition monoliths into co-operating processes integrable across heterogeneous platforms. In SOA, processes communicate through an enterprise service bus, and typically the data for all services is stored in the same DataBase (DB). Microservices, on the other hand, communicate through APIs, and all have their own DB. Overall, microservices are more autonomous and loosely-coupled. Figure 4 illustrates the differences between the two paradigms.

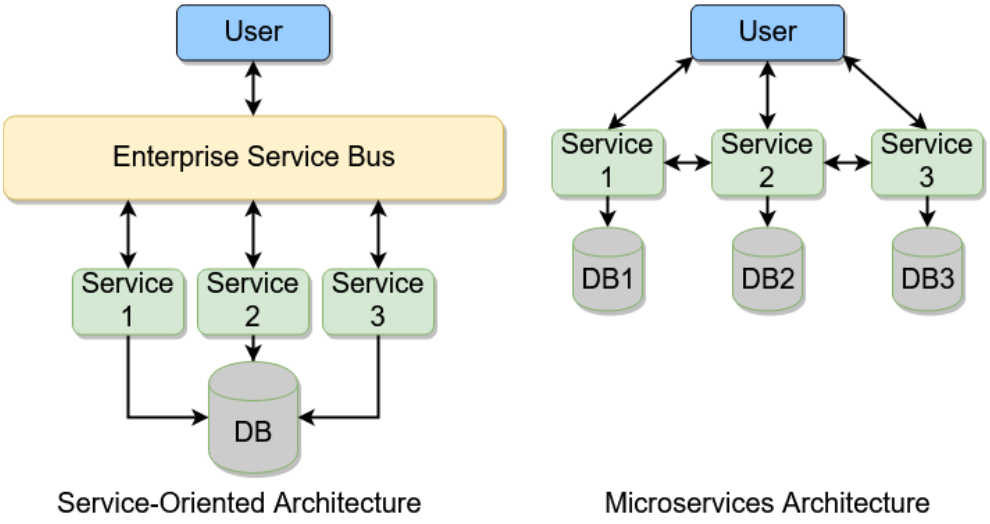


Figure 4. SOA and Microservices.

Challenges with microservices include increased complexity of systems overall, API design [35], orchestration of services, and error handling.

The performance of Docker with microservices was evaluated in [36] and was shown to be close to bare-metal. Microservices have, e.g., been applied to building a Smart City IoT platform in [10], and in [37] a vehicle-to-edge real-time emergency detection system deployed at the edge was presented.

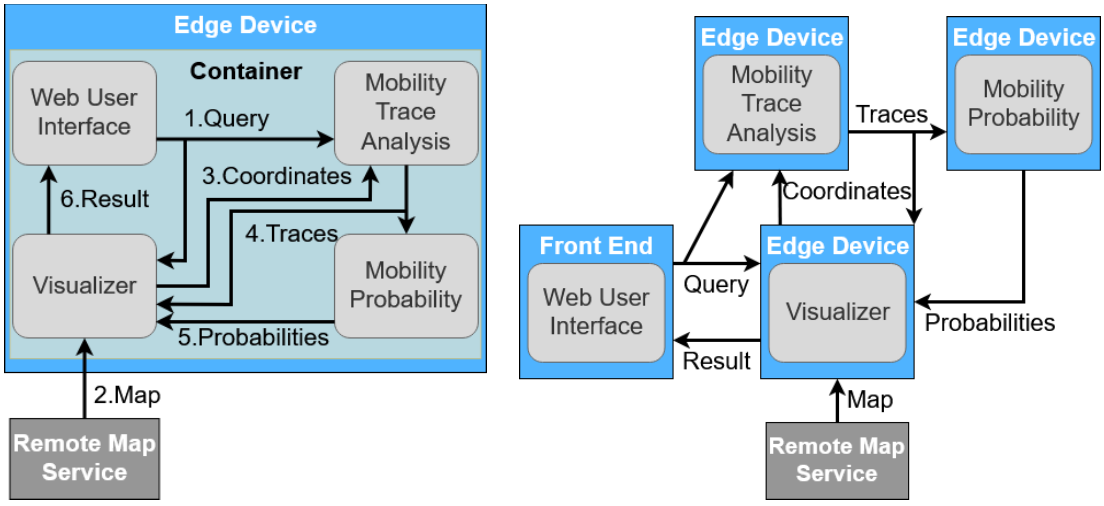
In this thesis, Docker containers are used to deploy a monolithic application and its microservices counterpart on Raspberry Pi (RPi) 3 Model Bs (SBC, Broadcom BCM2837 CPU, 1.2GHz, 1GB memory), which act as the EDs.

3. EXPERIMENTS

The purpose of this thesis is to evaluate the feasibility of distributed microservices versus centralized monolithic applications in edge computing, based on the workload optimization parameters latency and energy consumption. To achieve this, two versions of a user mobility analysis service were created. The goal of the services was to provide refined information on user trajectories and movement probabilities based on mobility data collected from UE connections to Wi-Fi APs, located in the city center of Oulu, Finland.

The first service version was a centralized monolith, i.e. all functionalities were contained in a single package. As seen in Figure 5a, the monolith composed of four distinctly different functionalities. The second service version modularized the monoliths functionalities into separate entities according to the microservices paradigm, creating a distributed Mobility MicroService (MMS), hosted on four EDs. The distributed MMS architecture is illustrated in Figure 5b. Developing the services in the two previously mentioned ways was done to enable the testing of microservices usage in a real-world prototype.

To make evaluation plausible, the main functionalities of both services are very similar. Both service versions were containerized with Docker and deployed on RPi 3 Model Bs in order to measure latency and energy consumption in real-world settings.



(a) Monolith execution flow.

(b) Mobility Microservice.

Figure 5. Monolithic and MMS architecture.

3.1. User Mobility Analysis

As user mobility analysis is only a means to an end in the monolithic and microservice application development in this thesis, a brief overview is given. User mobility analysis is a long researched topic in academia due to its importance in tracking and predicting human mobility patterns [38], as well as determining

points-of-interest people are likely to visit [39]. Results provided by user mobility analysis are necessary for example in location-based IoT and edge computing applications.

Human mobility has been shown to display simple reproducible patterns [38], for example, due to time-based habitual movements such as commuting. The relatively small variation in these movements makes prediction feasible. Traditionally, predictions are done based on analysis of large-scale data sets of social media data taken from Facebook user locations, Twitter's user-generated content, and location sharing services [38]. Analysis is typically done in the cloud.

3.2. Mobility Data Processing

To create any functionality or a UI, an understanding of the data is required. Both versions of the mobility analysis service use the data provided in [40]. It contains approximately 114 million entries of session data from UE connections to almost 1300 Wi-Fi APs of the open panOULU Wi-Fi network located in Oulu, Finland, and was gathered during the years 2007-2015. Table 1 depicts the variables included in the data. AP coordinates were provided as a separate data set.

Table 1. Example of the unprocessed mobility data

Wi-Fi trail ID	Timestamp	Origin AP	Destination AP	Device ID
78	2007-02-28 0:14:46	132	155	5
27179	2007-03-06 12:35:35		308	471
25727450	2010-03-06 13:09:36	137	138	4532
25727450	2010-03-06 13:09:39	281	138	4532
63641599	2015-02-03 12:53:12	964	965	45208
63664813	2015-02-03 22:11:24	825	824	90765

To simplify code and reduce computation time, the data was further processed using R [41], a free programming language and environment for statistical computing and graphics. Because of the data set's size, base R was not adequate for processing, thus the Data.Table package was used. It provides an improved version of R's standard data structure designed to reduce computation time.

To address data quality, entries containing missing origin APs were removed, as devices suddenly appearing do not tell much about user mobility. Next, the amount of consecutive UE movements with 0-60 seconds between them was calculated. The resulting time distribution depicted in Figure 6 contains multiple peaks. The first peak was on very fast movements whose mode was 0 seconds

and the corresponding local minimum on 4 seconds. Fast movements are typically unnecessary device handoffs between APs, i.e. a device continuously switching between APs as the user moves [42]. Redundant device handoffs are common in urban areas and happen due to objects interfering with received signal strength [42], other reasons include network congestion, temporarily being unable to find signal coverage [43] or temporarily connecting to an AP with a stronger signal. Because redundant UE handoffs do not explain user mobility, consecutive jumps with 0-4 seconds between them were removed from the data.

After the previous preprocessing, the data contained approximately 39 million entries. However, to align with the research previously done in the MEC-AI project, only the entries Tuesday, February 3rd, 2015, were used in the services. Since all the data was for the same day, timestamps were divided into hour, minute, and second. Wi-Fi trail and device IDs were also removed. Table 2 contains a version of Table 1 with the previously mentioned modifications and four of the rows deleted as they included data for the wrong date.

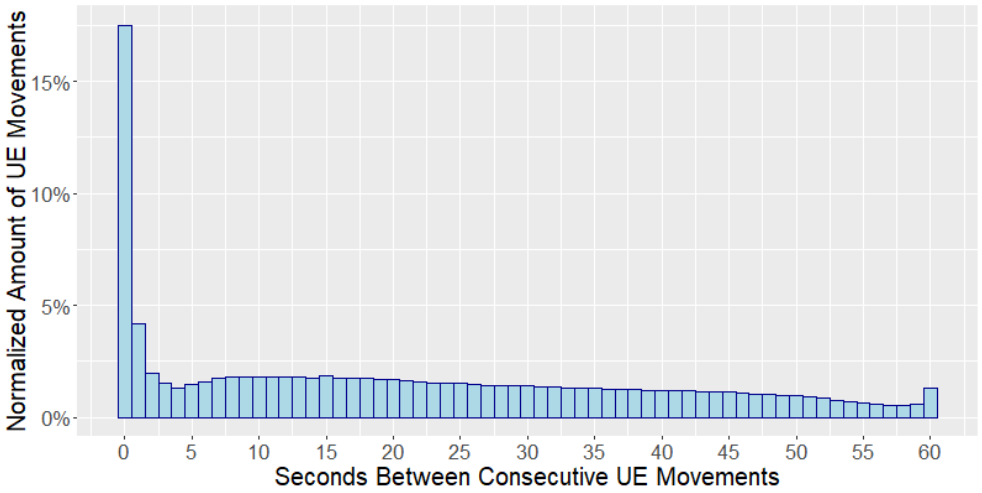


Figure 6. Consecutive UE movements for for each second.

Table 2. Processed version of data from Table 1

Origin AP	Destination AP	Hour	Minute	Second
964	965	12	53	12
825	824	22	11	24

3.3. Service Design and Implementation

The objective of the user mobility analysis services developed in this thesis is to provide refined information on user trajectories and movement probabilities in the form of a map image containing the visualized results. To achieve this, the

services must be able to take user input, subset the mobility data accordingly, extract movement trajectories, calculate movement amounts and probabilities, and visualize the results. Thus, as seen in Figure 5, the services were designed to have the following workflow.

1. User accesses the service's web UI and specifies the area, time frame and included APs for the analysis
2. The back-end fetches the selected area's map from Google Maps
3. A subset of the mobility data is created based on user input and map coordinates
4. User trajectories and movement amounts are extracted from the subset
5. Probabilities are calculated based on movement amounts
6. APs, movement trajectories, and probabilities are added on to the map
7. The resulting map image is visualized to the user in the web UI

Figure 7 depicts the UI designed for the services, as well as, an example result of a user mobility analysis request. The resulting image shows movements leaving from AP 864 from 8-9 am. The circles represent APs, the arrows represent UE movements. Probabilities are shown as text on top of the destination AP. The settings used in the analysis can be modified using the options panel.

According to edge computing conventions, the services were implemented as virtualized packages using Docker containers. The r-base-dev package was installed on top of the containers' Debian base, allowing the deployment of R software. However, R with no additional packages is not enough to create the services. Thus, depending on functionalities implemented inside a container, one or more of the R packages Shiny, Ggmap [44], Data.Table and Ggplot2 were also installed. Data required by each service was stored in a directory on the host ED and accessed through Docker Volume. The following presents the goal of each service component and its implementation.

The front-end web UI allows users to send mobility analysis requests and view the results. The UI is created with the Shiny package, which enables web applications to be created using only R code. Shiny also functions as the service's server, exposing it to connections on the port 8000. The service front-end can be accessed by entering the host machine's Internet Protocol (IP) address and the port 8000 into a browser. From the UI, users can specify the area to be analyzed, select which movement amounts to not include in the image, define specific origin and destination APs, and select a time frame. The front-end passes queries on to the Mobility Trace Analysis (MTA) and visualizer service components and visualizes the result to users. The web UI was also used to measure service execution latencies, i.e. the time it takes for the resulting image to appear on the UI after a request is sent, with a logging functionality implemented with the R function *Sys.time()* and by saving the results in the Comma-Separated Values (CSV) file format.

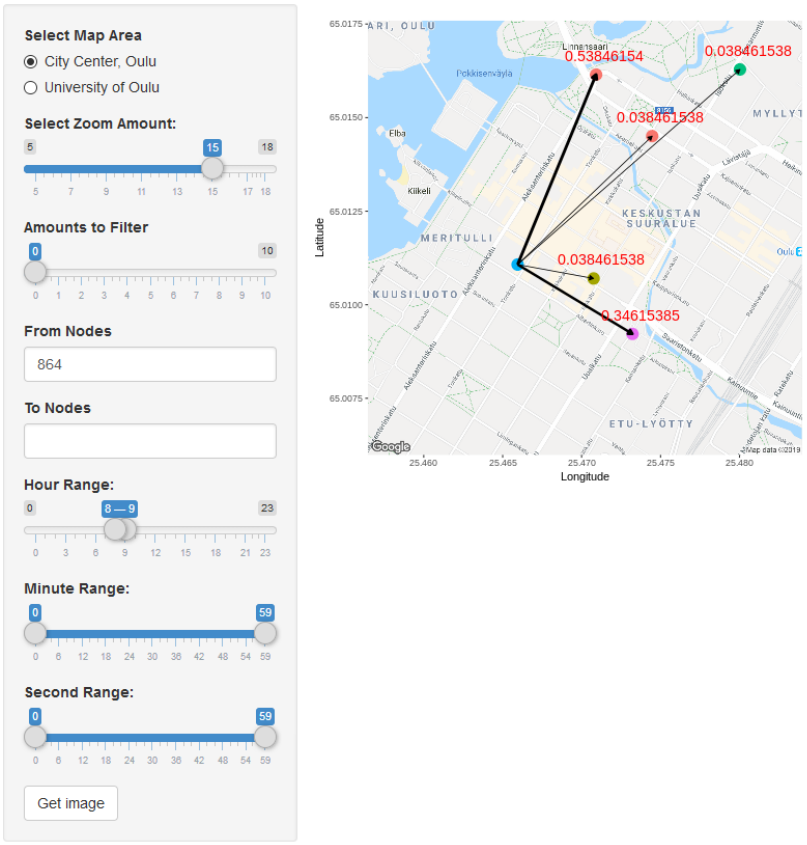


Figure 7. Service UI and an example result.

Mobility Trace Analysis identifies movement trajectories from the data and calculates the number of movements corresponding to each trajectory. This is done based on coordinates received from the visualizer service and user input. Suitable mobility observations are selected by

1. Identifying observations inside the given time frame
2. Identifying APs inside the selected area based on coordinates
3. Identifying observations with origin and destination APs inside the area

This data subsetting is done with the Data.Table package. Unique movements are then extracted from the subset of observations and the total amount of such actions is calculated. Origin and destination AP identifiers are replaced with their coordinates. R's ability to vectorize functions is utilized to reduce the time used in MTA. A vectorized function is able to modify all values in a vector at the same time. The results of the analysis are passed on to the mobility probability and visualizer components.

The movement probability service calculates the probability of each mobility trajectory found by the MTA. However, as movement prediction is not the focus of this thesis, movement probabilities are calculated for movements between APs with one-hop distance by summing up the number of movements leaving an origin

AP and dividing the number of movements to each of the destination APs with the sum. Results are passed on to the visualizer. Table 3 contains an example result of MTA and mobility probability calculation.

Table 3. Example MTA and probability calculation result

Orig Lat	Orig Long	Dest Lat	Dest Long	Amount	Probability
65.0108271	25.4812945	65.0238644	25.5200225	1/5	0.20
65.0108271	25.4812945	65.0431872	25.4877352	4/5	0.80
65.0860519	25.4812949	65.0587324	25.5604638	3/3	1.00

The visualizer service fetches the map requested by a user and layers the MTA and probability calculation results on top of it. Requested maps are fetched from Google Maps using the Ggmap package. Coordinate information is extracted from the map and sent to MTA. Unique APs found in the MTA result are drawn on top of the map as circles in their real location. Movements between APs are illustrated as arrows. The width of an arrow illustrates the number of movements. Movement probabilities are added near the destination APs. Drawing on top of the map is done with Ggplot2 graphics objects. The final image is sent to the web UI.

3.3.1. Centralized Monolith

The monolithic service version was realized as a single container. Its architecture can be seen in Figure 5a. As all service components are running on the same container, no additional functionalities had to be added to enable inter-component communication. All parts of the analysis are done successively to the whole data set being analyzed, making service execution linear, as depicted in Figure 8.

3.3.2. Distributed Mobility Microservice

The distributed MMS consists of the four different functionalities modularized into separate components deployed on different EDs inside containers. The MMS architecture is depicted in Figure 5b. The microservices communicate with each other by sending the user's query and results in R's native rds file format using Secure Copy Protocol (SCP). An exception to this is the resulting image, which is saved by the visualizer in the Portable Network Graphics file format. Implementing inter-service communication required error checking in the form of making sure that services do not attempt to read files as they are still being received, to be added, as this would crash the service. The data required by each service is stored on their host EDs and accessed through Docker Volume. Unlike in the monolith, the MMS's service components are able to run in parallel, as the MTA divides unique movements into chunks and processes them one at a time. The size of created chunks is defined in the MTA code. Results for each chunk

are sent to the visualizer and mobility probability microservices so that they are able to fulfill their task while new MTA results are calculated. As the visualizer receives data from the other services, it layers them on top of the map. The effect of MMS architecture on the execution timeline is illustrated in Figure 8.

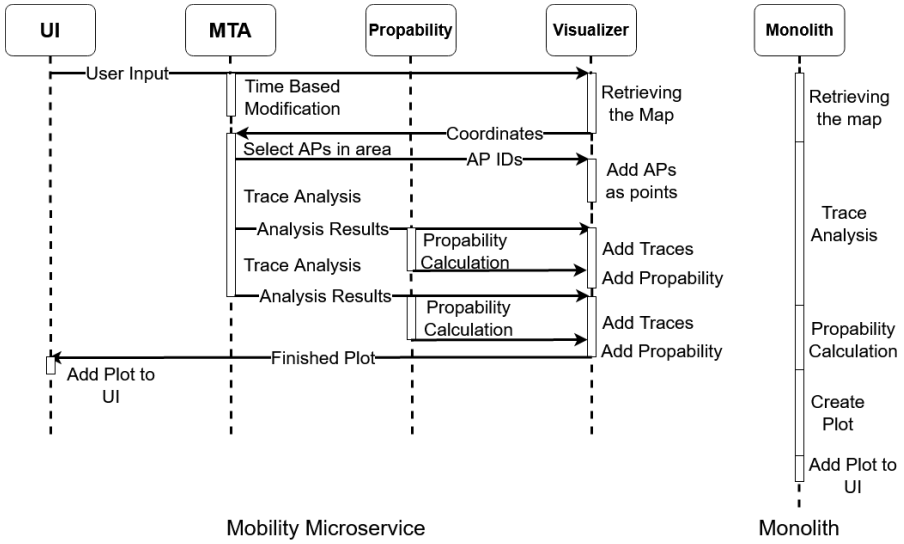


Figure 8. Execution timeline for the MMS and the Monolith.

3.4. Testbed and Experiments

To test the effects of the distributed MMS, in comparison to the monolith, a real-world prototype was created for both service implementations. The prototypes were then used to collect energy consumption and latency data for service requests. The prototypes were created by deploying the services, containerized with Docker, on RPis and by orchestrating them with Kubernetes. This is illustrated in Figure 9. The monolith was deployed on one RPi, and the MMS was deployed on four RPis connected to the same wireless network so each microservice was running on its own host machine. Each microservice knew the IP address of the RPis to whom they had to send messages to. The required AP location and mobility data were included in the RPis and accessed through Docker Volume. Service requests were sent through the web UI.

During all measurements, the time frame was set to include the whole day, and the size of the area being analyzed ranged from approximately 150m x 150m to 5km x 5km. Latencies gathered in [45] for a laptop and desktop computer are a part of the testbed, as results gathered in this thesis are also compared to them. However, it should be noted that the implementation evaluated in [45] did not use R's vectorization and was not distributed to multiple machines.

Energy consumption for each service was measured with the Monsoon Power Monitor [46]. One RPi at a time was connected from its 5V power input and ground pin to the Power Monitor's main channel. The voltage output provided by the Power Monitor ranges between 2.1V-4.55V, but RPi 3 typically uses 5.1V

[47]. Setting the Power Monitor output to 4.55V was, however, sufficient in this case. Figure 10 depicts the testbed during energy consumption measurements.

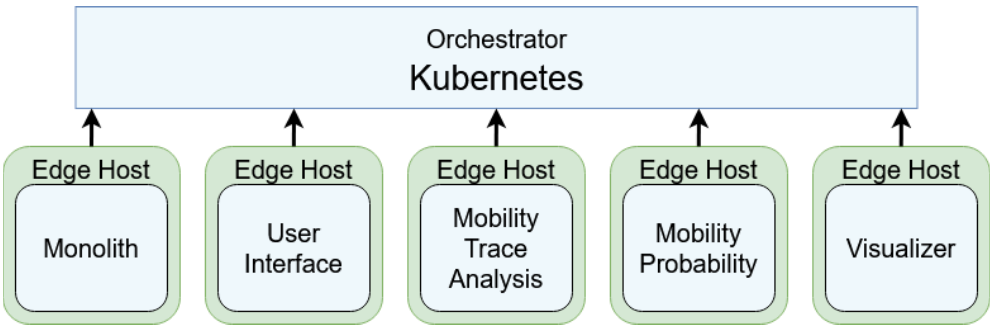


Figure 9. Service orchestration with Kubernetes.

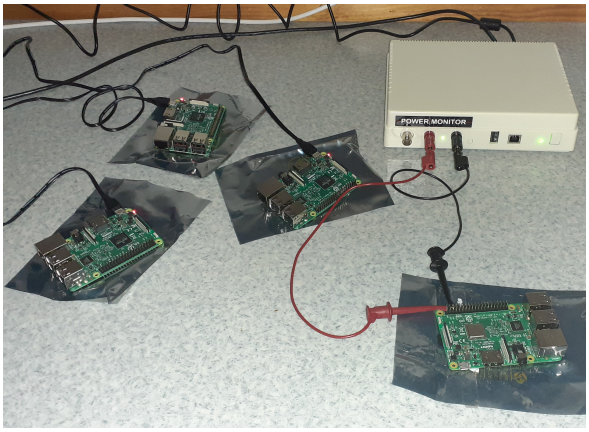


Figure 10. Energy consumption testbed.

3.5. Results

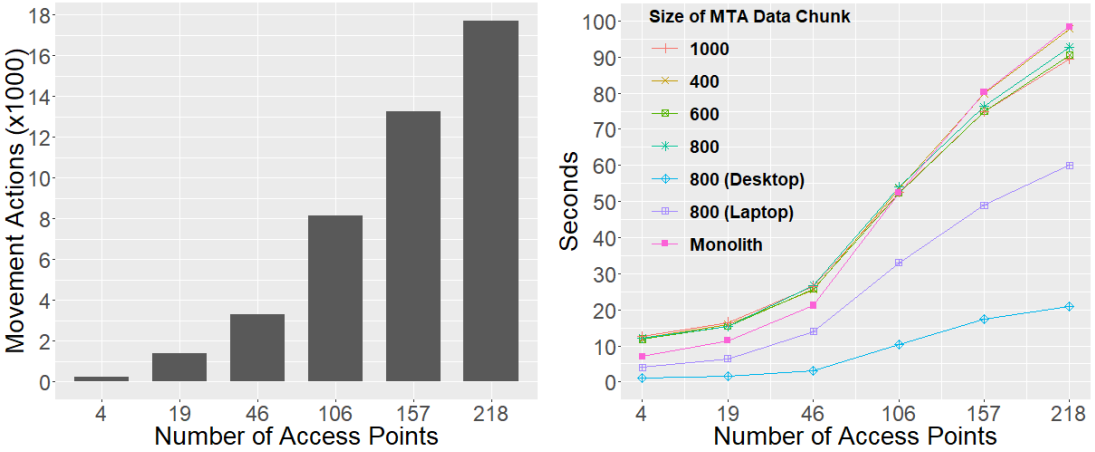
During the early stages of result collection, it was observed that connecting a RPi to the power monitor increased service execution latencies significantly. Because of this, the measurements were done separately.

Figure 11a depicts the number of APs and movement actions for each measurement point in Figure 11b. Service execution latencies and the results used as comparison [45] are illustrated in Figure 11b. The effects of increasing communication were measured by using 400, 600, 800, and 1000 observations as the MTA chunk size. Energy consumption was measured for

- RPI idling, i.e. the baseline energy consumption
- Service idling
- Services processing data for 4 APs

- Services processing data for 218 APs

Results collected with the Power Monitor were exported in CSV format and later visualized and analyzed using R. Tables 4 and 5 show the mean energy consumption of each service for the different MTA chunk sizes. Figure 12a depicts a RPi idling and Figures 12b-12f depicts service energy consumption transients when analyzing data for 218 APs with MTA chunk size set to 800.



(a) APs and corresponding movements.

(b) Service execution latencies.

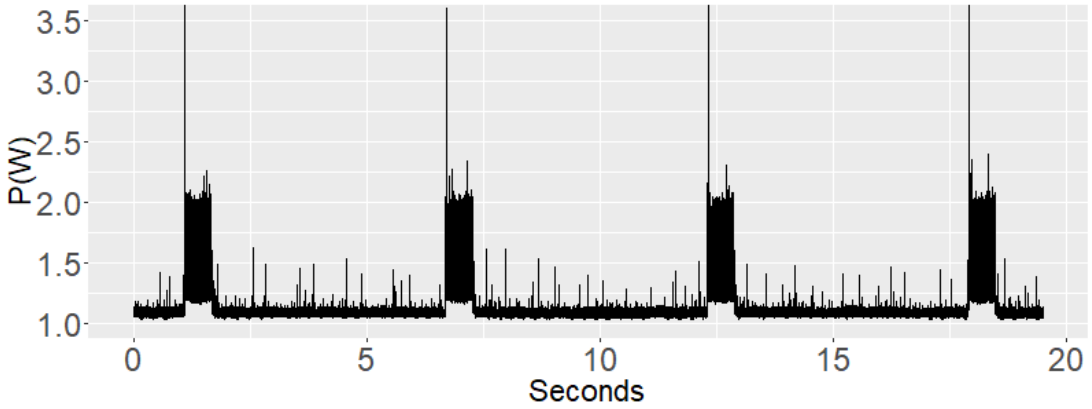
Figure 11. Latency evaluation results.

Table 4. Mean service energy consumption for 4 APs

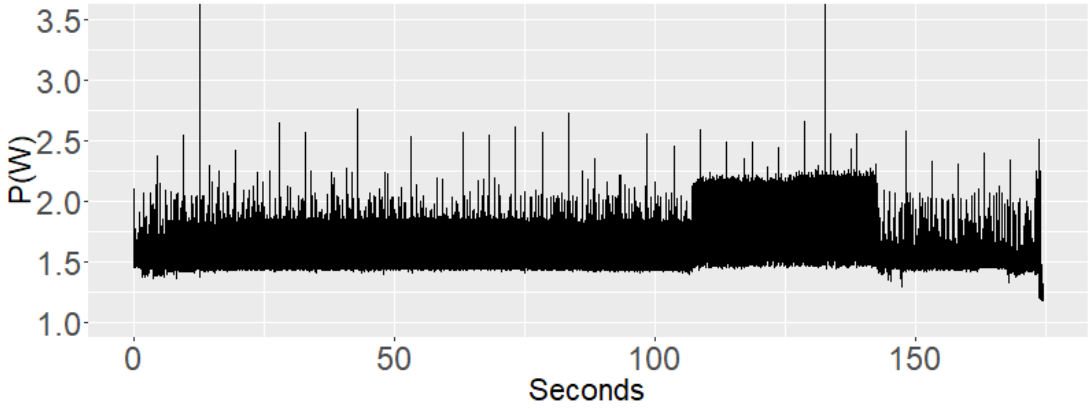
		MMS			
	Monolith	UI	MTA	Probability	Visualizer
Idle	1.21W	1.21W	1.22W	1.21W	1.20W
400	1.49W	1.26W	1.36W	1.35W	1.48W
600	1.49W	1.27W	1.36W	1.37W	1.49W
800	1.49W	1.27W	1.37W	1.34W	1.49W
1000	1.49W	1.27W	1.37W	1.38W	1.49W

Table 5. Mean service energy consumption for 218 APs

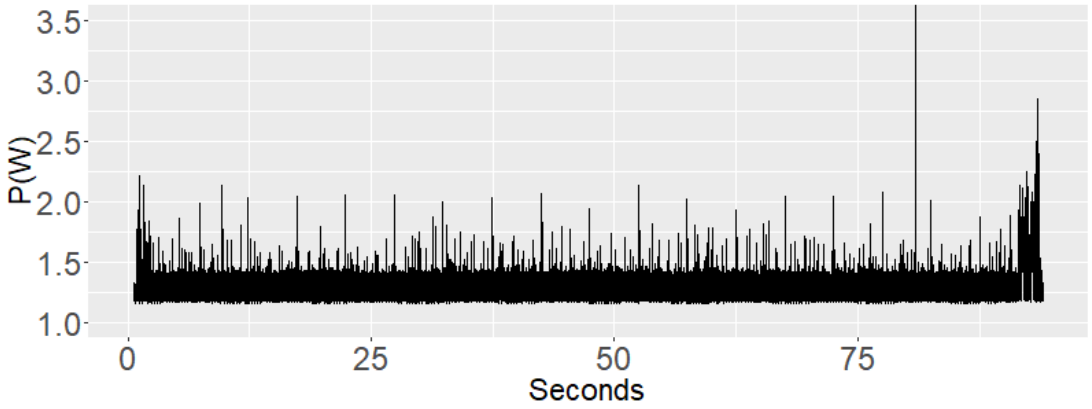
		MMS			
	Monolith	UI	MTA	Probability	Visualizer
Idle	1.21W	1.21W	1.22W	1.21W	1.20W
400	1.62W	1.22W	1.58W	1.59W	1.49W
600	1.62W	1.22W	1.62W	1.61W	1.49W
800	1.62W	1.22W	1.62W	1.63W	1.49W
1000	1.62W	1.22W	1.63W	1.63W	1.48W



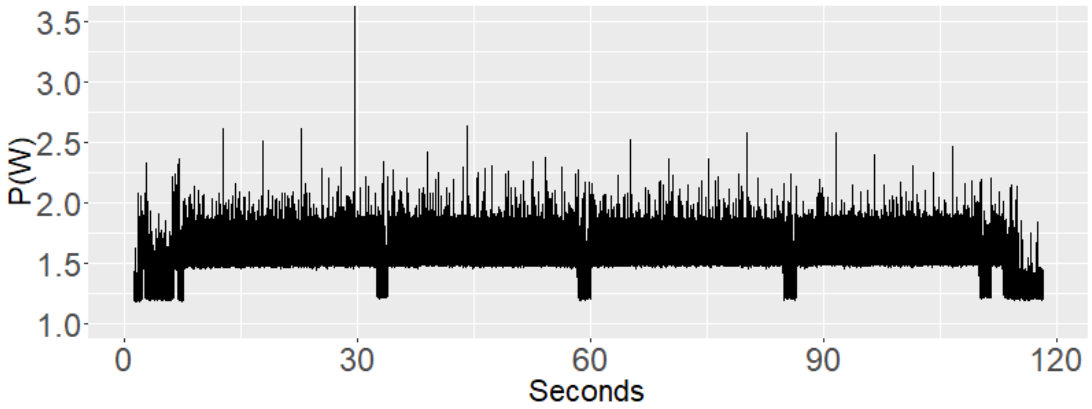
(a) Raspberry Pi 3 Model B idling.



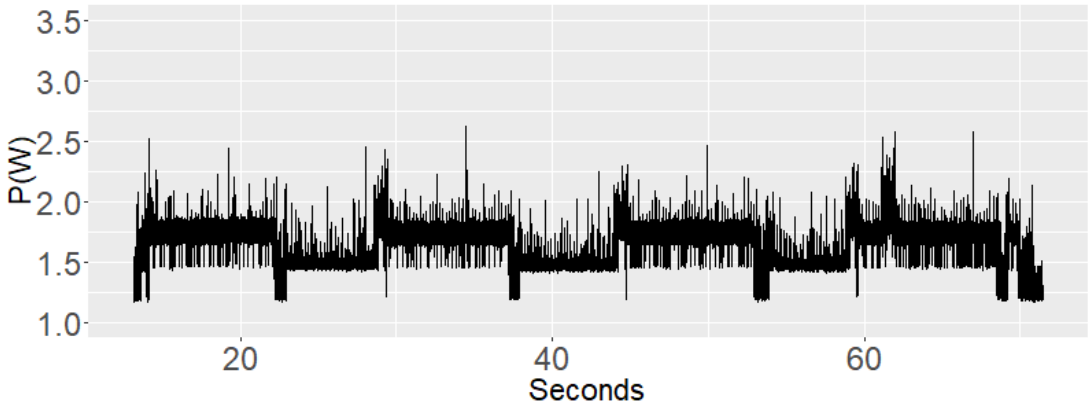
(b) Monolith, 218 APs.



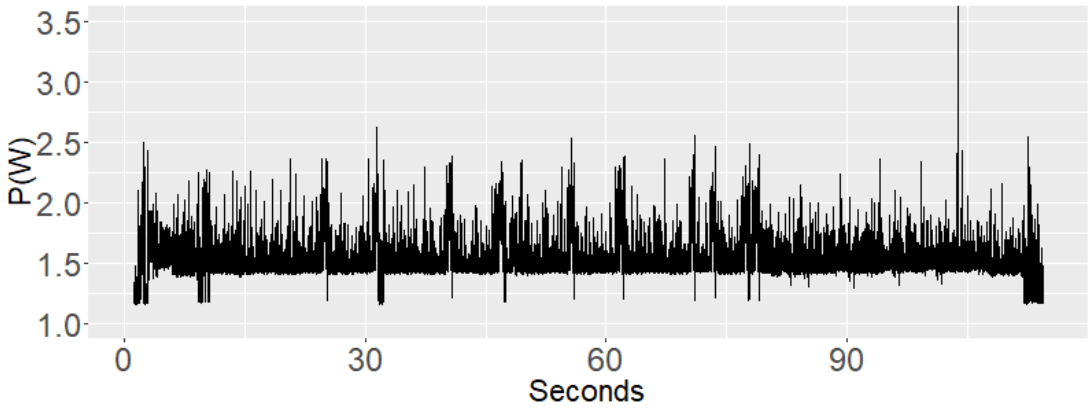
(c) UI, 218 APs with 800 chunk size.



(d) MTA, 218 APs with 800 chunk size.



(e) Mobility probability, 218 APs with 800 chunk size.



(f) Visualizer, 218 APs with 800 chunk size.

Figure 12. Energy consumption transients.

4. ANALYSIS OF RESULTS

In this section, the collected latency and energy consumption data is analyzed. The gathered results and the developed services are also discussed.

4.1. Latency

From the measured latencies shown in Figure 11b, it can be observed that resource-constrained EDs such as the RPi are able to handle service requests for geographically small areas within a reasonable time. Processing the data for 4 APs took approximately 7 seconds for the centralized monolith, and 14 seconds for the distributed MMS. The difference between the MMS's and the monolith's service execution latency is caused by inter-service communication, which makes the MMS lag behind for 4, 19, and 46 APs. After reaching 106 APs, the distributed MMS provides lower latencies than the monolith, and at 218 APs, the MMS with 1000 MTA chunk size is 9 seconds faster than the monolith. Decreasing MTA chunk size reduces this latency difference, nonetheless, the chunk size of 600 is interestingly shown to provide lower latency than 800. Using 400 as the MTA chunk size was nearly as slow as the monolith for 106-218 APs.

4.2. Energy Consumption

Figures 13a-13f illustrate the means and standard deviations calculated from the energy consumption measurement results presented in Tables 4 and 5. For service executions, results were calculated based on energy consumed when processing data for the smallest and largest number of APs.

From Figure 13a, it can be observed that while idling, the RPi 3 Model B consumes approximately 1.1W on average, and after starting a service, consumption is increased by approximately 0.1W. As seen in Table 6, the centralized monolith consumed 1.49W for 4 APs and 1.62W for 218 APs. Depending on the MTA chunk size, the distributed MMS consumed in total 5.45W-5.51W for the smallest number of APs and 5.88W-5.96W for the largest number of APs. From the individual service components, the UI consumed 1.265W for 4 APs and 1.22W for 218 APs. This is due to the UI doing nothing while waiting for the resulting image, the mean for the smallest amount of data is thus higher, as the time spent being idle is shorter. The visualization service consumes approximately 1.49W regardless of data volume or MTA chunk size. MTA and mobility probability calculation both used a very similar amount of energy, 1.34W-1.38W for 4 APs and 1.58W-1.63W for 218 APs.

The data analysis done in chunks by the MMS is especially visible in the MTA's and mobility probability's energy consumption transients seen in Figures 12d and 12e. After a chunk of data is analyzed, the energy consumption decreases while MTA and mobility probability send results forwards. The drop lasts longer for probability calculation, as it completes analysis faster than MTA and has to also wait for MTA results.

When focusing on the whole process, the total energy consumption in edge computing is the accumulation of each layer's energy cost [4]. Table 6 presents the total energy cost while idling and processing data for the monolith and the MMS with each MTA chunk size. The results show that, while idling, the mobility MMS consumes 4 times more energy than the monolith and during data processing, approximately 3.6 times more.

Table 6. Energy consumption accumulation for the monolith and the MMS

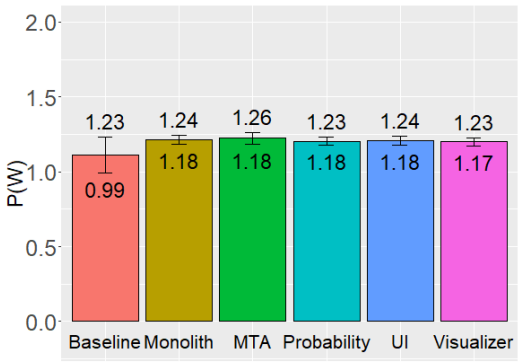
		MTA chunk size			
	Monolith	400	600	800	1000
Idle	1.2W	4.84W	4.84W	4.84W	4.84W
4 APs	1.49W	5.45W	5.49W	5.47W	5.51W
218 APs	1.62W	5.88W	5.94W	5.95W	5.96W

4.3. Discussion

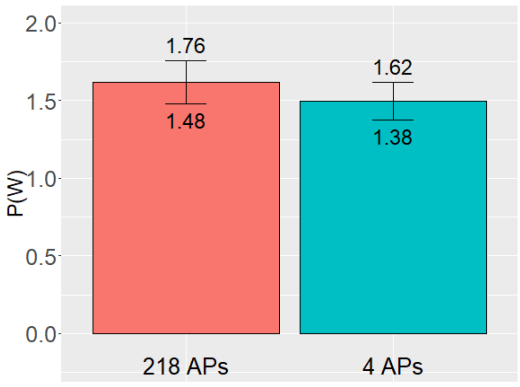
In this thesis, an edge computing user mobility analysis service was developed as a centralized monolith and further modularized into a distributed Mobility MicroService utilizing the microservices paradigm. The purpose of the services was to provide refined information on user trajectories and movement probabilities from the panOULU mobility data set [40]. The monolith was realized as a single Docker container and the MMS as a set of four containers. Both service implementations were deployed on low-resource EDs to create a real-world prototype with the goal of testing the feasibility of distributed microservices in edge computing, compared to traditional monoliths, based on service execution latency and energy consumption.

Before service development, the panOULU data set was processed with R to suit this use case. Unnecessary device handoffs were identified and removed from the data, as they provide no real information on user mobility. Working with the data set proved difficult due to its large size. However, utilizing the R package Data.Table made the process feasible. When observation with missing values and redundant device movements were removed, the data was significantly smaller. To align with previous research in the MEC-AI project, only data for the date February 3rd, 2015, was used. Timestamps were further divided into hour, minute, and second. Wi-Fi trail and device identifiers were removed, as the services were not meant for tracking the movements of specific users.

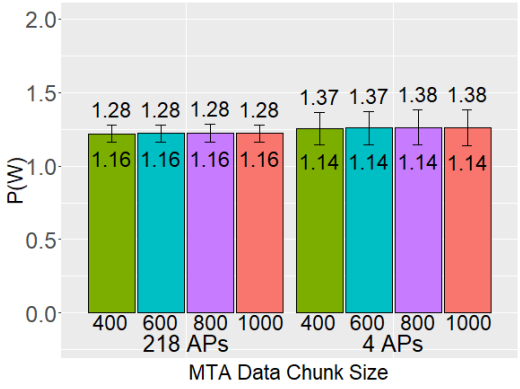
The service implementations relied on R based software to process varying amounts of mobility data. The usage of R in microservice development is atypical and likely effected gather results by increasing latencies, e.g. In this case, the usage of R can be attributed to the developer's skills. The service development was an evolutionary process, as many functionalities were redesigned multiple times. Compared to the monolith, the MMS was more difficult to create. The distribution of services increased complexity as a way for services to communicate,



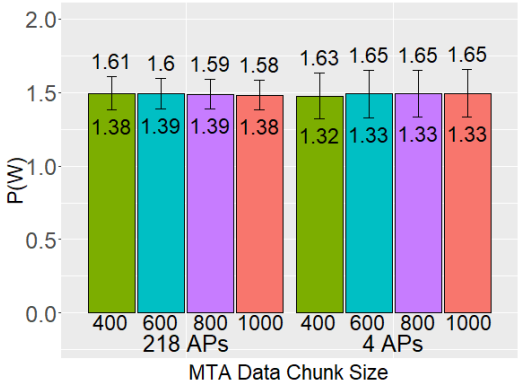
(a) Idling.



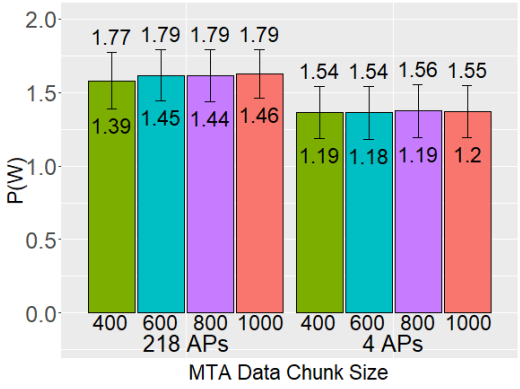
(b) Monolith.



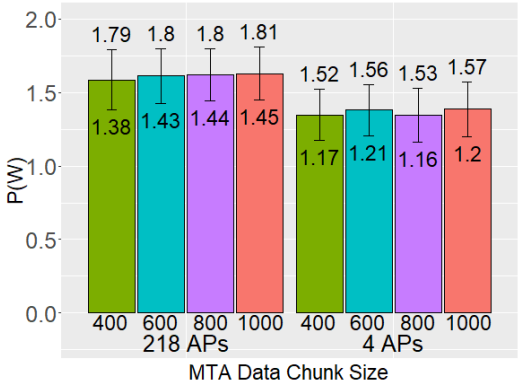
(c) User Interface.



(d) Visualization.



(e) MTA.



(f) Mobility Probability.

Figure 13. Energy consumption means and standard deviations.

as well as additional error checking had to be added. Deciding to use SCP for service communication was not the most microservice-esque solution. The additional packages installed on the containers to make SCP work also increased overhead. Although, creating a typically used API would have made the scale of this project too large.

The creation of the real-world prototypes required the services to be deployed on resource-constrained EDs, which were RPi 3 Model Bs. Kubernetes was used to orchestrate the services. Deploying the monolith took a significantly longer time than the MMS, demonstrating that deploying monoliths to the edge is not very preferable. As development was not done on the RPis, a problem was noticed when attempting to deploy the web UI. The Later package, a Shiny dependency, did not work properly on RPi. This issue was fixed according to [48]. Each microservice in the MMS had to know the IP address of other microservices that they communicated with. The target IP addresses were defined in the microservice code. Due to this, an IP changing could require multiple microservice containers to be rebuilt. This issue should be fixed in further development.

Looking back at the design goals of distributed systems, it can be noted that most of the goals were met in the distributed MMS. Each service can easily exchange data with each other using SCP. When a user interacts with the MMS, they only see the UI, thus the distribution stays hidden. Scaling in the geographical dimension is achieved by adding more EDs with the services deployed on them. This also means that the MMS scales up and out in the size dimension. On the other hand, the monolith only scales up. However, deploying new instances of the MMS components is slightly troublesome, as the IPs of the other services have to be added to the services code. Administrative scalability was achieved with Kubernetes orchestratio. With the query options that users are able to change, the service acts accordingly to user expectations. Each service is small and easy to change, but at their current state, they are difficult to integrate into other systems.

During the testing of the measurement process, it was noticed that connecting a RPi to the Monsoon Power Monitor increased latencies. This is most likely due to the Power Monitor's voltage output not being high enough, making the RPi's processor speed vary.

The latency evaluation shows that lightweight monoliths and distributed microservices deployed on low-resource EDs at the network edge are able to handle service requests concerning small numbers of APs in reasonable time. While the monolith is faster for small amounts of data, the distributed microservice provides better performance for large amounts of data. In this case, the MMS surpassed the monolith's execution speed at 106 APs, which corresponds to approximately 8000 movement actions. Such results indicate that distributed services should be able to handle the massive volume of data generated by IoT better than monoliths.

The gap in latency between the monolith and the MMS for low numbers of APs is likely caused by latencies added by data transmission. Based on the latency measurements it was also noticed that intermediate results should not be added to the map as they arrive. Instead, they should be added at the same time to reduce the time used saving the image. Comparing the gathered results to the

ones in [45] shows that with code optimization, RPis are able to provide almost as low latencies as EDs with more computational resources for small numbers of APs. Latencies for the RPis also behave very similarly to the EDs used in [45].

Lowering the latencies of the MMS was attempted by adding the parallelization of services, but based on the collected results, this did not work as expected, as using the chunk size of 1000 is faster than the others.

The energy consumption measurements show that the distributed MMS consumes more energy than the monolith. This is no surprise, as the MMS uses four RPis, which makes the MMS consume four times more energy when idling, and approximately 3.6 times more when processing data. However, this is acceptable if large amounts of data are being analyzed, as the distributed MMS processes the data for a shorter time than the monolith.

From the individual services, the monolith and the visualizer consumed the most energy for 4 APs. For 218 APs, the monolith, MTA, and probability calculation consumed approximately as much energy, making them the most energy-consuming services for large amounts of data. MTA and probability calculation energy usage also trends upwards as computation increases. In all cases, the UI used the least energy.

The goal of this thesis, testing the feasibility of distributed microservices for edge computing, was achieved, as the service execution latencies and energy consumption were able to be collected. However, as R and RPis were used, their significance depends on implementation. Services deployed on resource-rich EDs or created with other programming languages than R, e.g. can have different results.

In comparison with previous work [49], microservices were used to test the feasibility of a proposed architecture containing a cloud, fog, and edge layer. Containerized microservices deployed on RPis at the edge were shown, similarly to this thesis, to provide lower latencies with increased variance. The fast activation times of containers observed in [32] were also noticed during the measurement process of this thesis.

5. CONCLUSION

In this thesis, the feasibility of distributed microservices for IoT edge computing was tested for two important workload allocation optimization metrics, i.e. latency and energy consumption. Testing was done with a real-world prototype of a monolithic user mobility analysis service and a distributed Mobility MicroService. Both service versions provided refined information on user trajectories and probabilities for movements between APs of the panOULU Wi-Fi network.

The collected results showed that centralized monoliths provide better QoS for small amounts of data and distributed microservices for large amounts of data. Overall the distributed microservices version consumed significantly more energy than the monolith. The latencies were also compared to results previously gathered in [45].

6. REFERENCES

[1] Gubbi J., Buyya R., Marusic S. & Palaniswami M. (2013) Internet of Things (IoT): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7), pp. 1645–1660.

[2] Network World (2016). URL: https://www.networkworld.com/article/3147892/one-autonomous-car-will-use-4000-gb-of-dataday.html. Accessed 25.8.2019.

[3] Leppänen T. & Riekki J. (2019) Energy Efficient Opportunistic Edge Computing For the Internet of Things. In: *Web Intelligence*, 17(3), IOS Press, pp. 209–227.

[4] Shi W., Cao J., Zhang Q., Li Y. & Xu L. (2016) Edge computing: Vision and Challenges. *IEEE Internet of Things Journal* 3(5), pp. 637–646.

[5] Satyanarayanan M. (2017) The Emergence of Edge Computing. *Computer* 50(1), pp. 30–39.

[6] Yousefpour A., Fung C., Nguyen T., Kadiyala K., Jalali F., Niakanlahiji A., Kong J. & Jue P. (2019) All One Needs to Know About Fog Computing and Related Edge Computing Paradigms: A Complete Survey. *Journal of Systems Architecture*.

[7] IBM IT Infrastructure Blog (2018). URL: https://www.ibm.com/blogs/systems/ibm-power9-scale-out-servers-deliver-more-memory-better-price-performance-vs-intel-x86/. Accessed 26.8.2019.

[8] Pahl C. & Lee B. (2015) Containers and clusters for edge cloud architectures–a technology review. In: *3rd international conference on future internet of things and cloud*, IEEE, pp. 379–386.

[9] Dragoni N., Giallorenzo S., Lafuente A.L., Mazzara M. M.F., Mustafin R. & Safina L. (2017) Microservices: Yesterday, Today, and Tomorrow. In: *Present and ulterior software engineering*, pp. 195–216. Springer.

[10] Krylovskiy A., Jahn M. & Patti E. (2015) Designing a smart city internet of things platform with microservice architecture. In: *2015 3rd International Conference on Future Internet of Things and Cloud*, IEEE, pp. 25–30.

[11] Ashton K. (2009) That 'internet of things' thing. *RFID journal* 22(7), pp. 87–114.

[12] Rose K., Eldridge S. & Chapin L. (2015) The internet of things: An overview. *The Internet Society (ISOC)*, 80 p.

[13] Lin J., Yu W., Zhang N., Yang X., Zhang H. & Zhao W. (2017) A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications. *IEEE Internet of Things Journal* 4(5), pp. 1125–1142.

[14] Mell P. & Grance T. (2011), The NIST definition of cloud computing.

[15] Van Steen M. & Tanenbaum A.S. (2017) Distributed Systems. Maarten van Steen Leiden, The Netherlands, third ed. URL: distributed-systems.net.

[16] Hoffman P. & Woods D. (2010) Cloud Computing: The Limits of Public Clouds for Business Applications. In: *IEEE Internet Computing* 14(6), pp. 90–93.

[17] Armbrust M., Fox A., Griffith R., Joseph A.D., Katz R., Konwinski A., Lee G., Patterson D., Rabkin A., Stoica I. & Zahari M. (2010) A view of cloud computing. *Communications of the ACM*, 53(4).

[18] openEDGEcomputing (2019). URL: https://www.openedgecomputing.org/about/. Accessed 28.8.2019.

[19] Lin L., Li P., Liao X., Jin H. & Zhang Y. (2018) Echo: An edge-centric code offloading system with quality of service guarantee. *IEEE Access* 7, pp. 5905–5917.

[20] Bondi A.B. (2000) Characteristics of scalability and their impact on performance. In: *Proceedings of the 2nd international workshop on Software and performance*, ACM, pp. 195–203.

[21] Weinstock C.B. & Goodenough J. (2006), On system scalability.

[22] Conti M., Giordano S., May M. & Passarella A. (2010) From opportunistic networks to opportunistic computing. *IEEE Communications Magazine* 48, pp. 126–139.

[23] Mirri S., Prandi C., Salomoni P., Callegati F., Melis A. & Prandini M. (2016) A service-oriented approach to crowdsensing for accessible smart mobility scenarios *Mobile Information Systems*, article no. 2821680.

[24] Leppänen T., Savaglio C., Lovén L., Russo W., Di Fatta G., Riekki J. & Fortino G. (2018) Developing Agent-Based Smart Objects for IoT Edge Computing: Mobile Crowdsensing Use Case. In: *International Conference on Internet and Distributed Computing Systems*, Springer, pp. 235–247.

[25] Yu W., Liang F., He X., Hatcher W.G., Lu C., Lin J. & Yang X. (2017) A survey on the edge computing for the Internet of Things. *IEEE access* 6, pp. 6900–6919.

[26] Campbell S. & Jeronimo M. (2006), An introduction to virtualization. *Published in "Applied Virtualization", Intel*, pp. 1-15.

[27] Tao Z., Xia Q., Hao Z., Li C., Ma L., Yi S. & Li Q. (2019) A Survey of Virtual Machine Management in Edge Computing. *Proceedings of the IEEE*, 107(8) , pp. 1482–1499. IEEE.

[28] Ismail B.I., Goortani E.M., Ab Karim M.B., Tat W.M., Setapa S., Luke J.Y. & Hoe O.H. (2015) Evaluation of docker as edge computing platform. In: *2015 IEEE Conference on Open Systems (ICOS)*, IEEE, pp. 130–135.

[29] What is Kubernetes. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. Accessed 29.9.2019.

[30] Ha K., Abe Y., Eiszler T., Chen Z., Hu W., Amos B., Upadhyaya R., Pillai P. & Satyanarayanan M. (2017) You can teach elephants to dance: agile vm handoff for edge computing. In: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*.

[31] Yi S., Hao Z., Zhang Q., Zhang Q., Shi W. & Li Q. Lavea: Latency-aware video analytics on edge computing platform. In: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, ACM, p. 15.

[32] Morabito R. (2017) Virtualization on internet of things edge devices with container technologies: a performance evaluation. *IEEE Access* 5, pp. 8835–8850. IEEE.

[33] Butzin B., Golatowski F. & Timmermann D. (2016) Microservices approach for the internet of things. In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE, pp. 1–6.

[34] Cerny T., Donahoo M.J. & Trnka M. (2018) Contextual understanding of microservice architecture: current and future directions. *ACM SIGAPP Applied Computing Review* 17, pp. 29–45. ACM.

[35] Haselböck S., Weinreich R., Buchgeher G. & Kriechbaum T. (2018) Microservice design space analysis and decision documentation: A case study on api management. In: *2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA)*, IEEE, pp. 1–8.

[36] Amaral M., Polo J., Carrera D., Mohomed I., Unuvar M. & Steinder M. (2015) Performance evaluation of microservices architectures using containers. In: *2015 IEEE 14th International Symposium on Network Computing and Applications*, IEEE, pp. 27–34.

[37] Zhou P., Zhang W., Braud T., Hui P. & Kangasharju J. (2019) Enhanced Augmented Reality Applications in Vehicle-to-Edge Networks. In: *2019 22nd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, IEEE, pp. 167–174.

[38] Cheng Z., Caverlee J., Lee K. & Sui D.Z. (2011) Exploring millions of footprints in location sharing services. In: *Fifth International AAAI Conference on Weblogs and Social Media*.

[39] Noulas A., Scellato S., Lathia N. & Mascolo C. (2012) Mining user mobility features for next place prediction in location-based services. In: *2012 IEEE 12th international conference on data mining*, IEEE, pp. 1038–1043.

[40] Kostakos V., Ojala T. & Juntunen T. (2013) Traffic in the smart city: Exploring city-wide sensing for traffic control center augmentation. *IEEE Internet Computing* 17(6), pp. 22–29.

[41] R (2019). URL: https://www.r-project.org/. Accessed 29.8.2019.

[42] Rahman M.A., Salih Q.M., Asyhari A.T. & Azad S. (2019), Traveling distance estimation to mitigate unnecessary handoff in mobile wireless networks. *Annals of Telecommunications*, pp. 1-10.

[43] Johnson S.B., Nath P.S. & Velmurugan T. (2013) An optimized algorithm for vertical handoff in heterogeneous wireless networks. In: *2013 IEEE Conference on Information and Communication Technologies*, IEEE, pp. 1206–1210.

[44] Kahle D. & Wickham H. (2013) ggmap: Spatial visualization with ggplot2. The R Journal 5, pp. 144–161. URL: https://journal.r-project.org/archive/2013-1/kahle-wickham.pdf. Accessed 29.8.2019.

[45] Leppänen T., Savaglio C., Lovén L., Järvenpää T., Ehsani R., Peltonen E., Fortino G. & Riekki J. (2019) Edge-based Microservices Architecture for Internet of Things: Mobility Analysis Case Study. Accepted.

[46] Monsoon Solutions Inc. URL: https://www.msoon.com/lvpm-software-download. Accessed 30.8.2019.

[47] Raspberry Pi Power Supply. URL: https://www.raspberrypi.org/documentation/hardware/raspberrypi/power/README.md. Accessed 30.8.2019.

[48] Installation failed on Raspberry PI 3 - Debian Stretch - R 3.5.1. URL: https://github.com/r-lib/later/issues/73. Accessed 2.9.2019.

[49] Alam M., Rufino J., Ferreira J., Ahmed S.H., Shah N. & Chen Y. (2018) Orchestration of microservices for iot using docker and edge computing. *IEEE Communications Magazine* 56, pp. 118–123. IEEE.