UNIVERSITY OF NOVI SAD

FACULTY OF TECHNICAL SCIENCES
NOVI SAD

Miloš Simić

Micro clouds and edge computing as a service

- Ph. D. Thesis -

Supervisor
Goran Sladić, PhD, associate professor

Novi Sad, 2021.

Miloš Simić: *Micro clouds and edge computing as a service*

SERBIAN TITLE: Micro clouds and edge computing as a service

SUPERVISOR:
Goran Sladić, PhD, associate professor

LOCATION:
Novi Sad, Serbia

DATE:
September 2021

UNIVERZITET U NOVOM SADU • **FAKULTET TEHNIČKIH NAUKA**
21000 NOVI SAD, Trg Dositeja Obradoviće 6

KLJUČNA DOKUMENTACIJSKA INFORMACIJA

Redni broj, **RBR**:	
Identifikacioni broj, **IBR**:	
Tip dokumentacije, **TD**:	Monografska dokumentacija
Tip zapisa, **TZ**:	Tekstualni štampani materijal
Vrsta rada, **VR**:	Doktorska disertacija
Autor, **AU**:	Miloš Simić
Mentor, **MH**:	dr Goran Sladić, vanredni profesor
Naslov rada, **NR**:	Micro clouds and edge computing as a service
Jezik publikacije, **JP**:	engleski
Jezik izvoda, **JI**:	srpski
Zemlja publikacije, **ZP**:	Srbija
Uže geografsko područije, **UGP**:	Vojvodina
Godina, **GO**:	2021
Izdavač, **IZ**:	Fakultet tehničkih nauka
Mesto i adresa, **MA**:	Trg Dositeja Obradovića 6, 21000 Novi Sad
Fizički opis rada, **FO**: (poglavlja/strana /citata/tabela/slika/grafika/priloga)	6/159/154/13/13/0/0
Naučna oblast, **NO**:	Elektrotehničko i računarsko inženjerstvo
Naučna disciplina, **ND**:	Distribuirani sistemi
Predmetna odrednica/Ključne reči, **PO**:	distributed systems, cloud computing, microservices, software as a service, edge computing, micro clouds
UDK	
Čuva se, **ČU**:	Biblioteka Fakulteta tehničkih nauka, Trg Dositeraj Obradovića 6, 21000 Novi Sad
Važna napomena, **VN**:	

UNIVERZITET U NOVOM SADU • **FAKULTET TEHNIČKIH NAUKA**
21000 NOVI SAD, Trg Dositeja Obradoviće 6

KLJUČNA DOKUMENTACIJSKA INFORMACIJA

Izvod, **IZ**:		U sklopu disertacije izvršeno je istraživanje u oblasti razvoja bezbednog softvera. Razvijene su dve metode koje zajedno omogućuju integraciju bezbednosne analize dizajna softvera u proces agilnog razvoja. Prvi metod predstavlja radni okvir za konstruisanje radionica čija svrha je obuka inženjera softvera kako da sprovode bezbednosnu analizu dizajna. Drugi metod je proces koji proširuje metod bezbednosne analize dizajna kako bi podržao bolju integraciju spram potreba organizacije. Prvi metod je evaluiran kroz kontrolisan eksperiment, dok je drugi metod evaluiran upotrebom komparativne analize i analize studija slučaja, gde je proces implementiran u kontekstu dve organizacije koje se bave razvojem softvera.	
Datum prihvatanja teme, **DP**:			
Datum odbrane, **DO**:			
Članovi komisije, **KO**:	Predsednik:	dr Branko Milosavljević, redovni profesor, FTN, Novi Sad	
	Član:	dr Silvia Gilezan, redovni profesor, FTN, Novi Sad	
	Član:	dr Gordana Milosavljević, vanredni profesor, FTN, Novi Sad	Potpis mentora
	Član:	dr Žarko Stanisavljević, docent, ETF, Beograd	
	Član, mentor:	dr Goran Sladić, vanredni profesor, FTN, Novi Sad	

UNIVERSITY OF NOVI SAD • **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:	
Identification number, **INO**:	
Document type, **DT**:	Monograph documentation
Type of record, **TR**:	Textual printed material
Contents code, **CC**:	Ph.D. thesis
Author, **AU**:	Miloš Simić
Mentor, **MN**:	Goran Sladić, Ph.D., Associate Professor
Title, **TI**:	Micro clouds and edge computing as a service
Language of text, **LT**:	English
Language of abstract, **LA**:	Serbian
Country of publication, **CP**:	Serbia
Locality of publication, **LP**:	Vojvodina
Publication year, **PY**:	2021
Publisher, **PB**:	Faculty of Technical Sciences
Publication place, **PP**:	Trg Dositeja Obradovića 6, 21000 Novi Sad
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/	6/159/154/13/13/0/0
Scientific field, **SF**:	Electrical engineering and computing
Scientific discipline, **SD**:	Distributed systems
Subject/Key words, **S/KW**:	distributed systems, cloud computing, microservices, software as a service, edge computing, micro clouds
UC	
Holding data, **HD**:	Library of Faculty of Technical Sciences, Trg Dositeja Obradovića 6, 21000
Note, **N**:	

UNIVERSITY OF NOVI SAD • **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Abstract, **AB**:		This thesis presents research in the field of secure software engineering. Two methods are developed that, when combined, facilitate the integration of software security design analysis into the agile development workflow. The first method is a training framework for creating workshops aimed at teaching software engineers on how to perform security design analysis. The second method is a process that expands on the security design analysis method to facilitate better integration with the needs of the organization. The first method is evaluated through a controlled experiment, while the second method is evaluated through comparative analysis and case study analysis, where the process is tailored and implemented for two different software vendors.	
Accepted by the Scientific Board on, **ASB**:		11.07.2019.	
Defended on, **DE**:			
Defended Board, **DB**:	President:	Branko Milosavljević, PhD, Full Professor, FTN, Novi Sad	
	Member:	Silvia Gilezan, PhD, Full Professor, FTN, Novi Sad	
	Member:	Gordana Milosavljević, PhD, Associate Professor, FTN, Novi Sad	Menthor's signature
	Member:	Žarko Stanisavljević, PhD, Assistant Professor, ETF, Belgrade	
	Member, Mentor:	Goran Sladić, PhD, Associate Professor, FTN, Novi Sad	

Acknowledgements

Abstract

Key words: distributed systms, cloud computing, micro clouds, edge computing

Rezime

Ključ reči: distributed systms, cloud computing, micro clouds, edge computing

Table of Contents

List of Figures

List of Tables

Listings

List of Equations

List of Abbreviations

CC	Cloud computing
AWS	Amazon Web Services
IoT	Internet of Things
DS	Distributed systems
DC	Distributed computing
DC	Data center
IaaS	infrastructure as a service
PaaS	Platform as a service
SaaS	Software as a service
CaaS	Container as a service
DBaaS	Databae as a service
XaaS	Everything as a Service
P2P	Peer-to-peer
DHT	Distributed Hash Table
NoSQL	Not Only SQL

EC	Edge computing
MEC	Mobile edge computing
MCC	Mobile cloud computing
QoE	Quality of experience
QoS	Quality of service
MDCs	Micro data-centers
SoC	Separation of concerns
ES	Edge servers
CDN	Content delivery networks
SDN	Software-defined networks
VM	Virtual machine
OS	Operating system
SEC	Strong Eventual Consistency
MPST	Multiparty asynchronous session types
API	Application programming interface
CSP	Communicating Sequential Processes
IaC	Infrastructure as code
CRDTs	Conflict-free replicated datatypes

Chapter 1

Introduction

Various software systems has changed the way people communicate, learn and run businesses, and interconnected computing devices has numerous positive applications in everyday life. Over the past decade, computation and data volumes have increased significantly [1]. Augmented reality, online gaming, face recognition, autonomous vehicles, or the Internet of Things (IoT) applications produce huge volumes of data. Workloads like those require latency below a few tens of milliseconds [1]. These requirements are outside what a centralized model like the CC can offer [1]. Even small problems can contribute to large downtime of applications and servies people are depending on. Recent example is yet another outage that happend in Amazon Web Services (AWS), and as a result a large amount of intrnet become unavaleble.

The aim of this thesis is to provide formal models uppon which we implement distributed system for oragnizing cloud-like geo-distributed environments for users or CC providers to utilize, in order to minimize downtime of critical services. The whole system can be looked as a pre-cloud or pre-processing layer sending only important data to the cloud minimizing cost for users, and ensuring avelability of CC services.

In this section, we are going to give an overview of the topics, that are of significant importance for the rest of the thesis, since it is heavely based on these topis. We start by describing the general problem area

that our work addresses in Section 1.1. Sections 1.2 and 1.3 describe the theoretical background behind the problem, where we examine distributed systems (DS) and distributed computing (DC), focusing on design details, communication patters and organizational structure. In Section 1.4 we describe similar models that might be source of confusion, and how they are differnet than DS or DC and how some concepts can fit in the bigger picture. Section **??** describe different architecture and application model and how deployment can be done in large DS. In Section 1.5 we describe different virtualization methods that are used in CC for system and/or applications. In section 1.7 we describe difference between concurency and parallelism and introduce actor system, that will be used latter on in the thesis. In Section 1.8, we specify the exact problem that our work addresses and describe our hypothesis and research goals in Section 1.9. Section 1.10 present the structure of the thesis.

1.1 Problem area

Cloud centralized architecture with enormous data-centers (DCs) capacities creates an effective economy of scale to lower administration cost [2]. However, when such a system grows to its limits, centralization brings more problems than solutions [3, 4]. Despite all the CC benefits, applications and services face a serious degradation over time, due to the high bandwidth and latency [5]. This can have a huge consequence on the business and potentially human lives as well. Organizations use cloud services to avoid huge investments [6], like creating and maintaining their own DCs. They consume resources created by others [7] and pay for usage time – a pay as you go model.

Data is required to be moved to the cloud from data sources, which introduces a high latency in the system [8]. For example, Boeing 787s generates half a terabyte of data per single flight, while a self-driving car generates two petabytes of data per single drive. Bandwidth is not large enough to support such requirements [9]. Data transfer is not

the only problem: applications like self-driving cars, delivery drones, or power balancing in electric grids require real-time processing for proper decision making [9]. We might face serious issues if a cloud service becomes unavailable due to denial-of-service attack, network, or cloud failure [3].

To overcome cloud latency, research led to new computing areas, and model in which computing and storage utilities are in proximity to data sources [7]. The cloud is enhanced with new ideas for future generation applications [10].

1.2 Distributed systems

There are various definitions of DS, but we can think of DS as a systems where multiple entities can comunicate to one another in some way, but at the same time, they are able to performing some operations.

In [11, 12] Tanenbaum et al. give two interesting assumption about DS:

(**1**) "A computing element, which we will generally refer to as a node, can be either a hardware device or a software process".

(**2**) "A second element is that users (be they people or applications) believe they are dealing with a single system. This means that one way or another the autonomous nodes need to collaborate".

These two assamptions are usefull and powefull, when talking about DS. As such, in this thesis we will adopt and use them rigorously.

Three significant characteristics of distributed systems are [12]:

(**1**) **concurrency of components**, referst to ability of the DS that multiple activities are executed at the same time. These activities takes place on multiple nodes that are part of a DS.

(**2**) **independent failure of components**, this property refers to a nasty feature of DS that nodes fail independently. Thy can fail at the same time as well, but they usually fail independently for numerus reasons.

(**3**) **lack of a global clock**, this is a consequence of dealing with independent nodes. Each node have its own notion of time, and such we cannot assume that there is something like a global clock.

In [11] authors give formal definition "distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system".

When talking about DS, we usually think about computing systems that are connected via network or over the internet. But DS are not exclusiv to domain of cumputer science. They existed before computers started to enrich almost every aspect of human life. DS have been used in varios different domains such as: **telecommunication networks**, **aircraft control systems**, **industrial control systems** etc. DS are used anywhere where amout of users are growing rapidly, so that single entity can't reponse to users demands in (near) real-time.

Distributed systems (in computer science) are consists of various algorithms, techniques and trade-offs to create an illusion that set of nodes act as one. DS algorithms may include: (1) replication, (2) consensus, (3) communication, (4) storage, etc.

DS are hard to immplement because of their nature. James Gosling and Peter Deutsch both fellows at Sun Microsystems at the time crated list of problems for network applicatoins know as *8 fallacies of Distributed Systems*:

(**1**) **The network is reliable**; there will always be something that go wrong with the network — power failure, a cut cable, environment disasters etc.

(**2**) **Latency is zero**; localy latency is not an issue, but it deteriorates very quickly when you move to the internet and CC scenarios.

(**3**) **Bandwidth is infinite**; even though bandwidth is constantly getting better and better, the amount of data we try to push through it rise as well.

(**4**) **The network is secure**; Internet attack trends are showing growth, and this becomes problem even more in public CC.

(**5**) **Topology doesn't change**; network topology is usually out of user control, and network topology changes sonstantly for numerous of reasons — added or removed new devices, servers, breaks, outages etc.

(**6**) **There is one administrator**; nowdays there are numerous of administratos for web servers, databases, cache and so one, but also company collaborates with other copanies or CC provider.

(**7**) **Transport cost is zero**; we have to serialize information and send data over the wire, which takes resources and adds to the total latency. Problem here is not just latency,but that information serialization takes time and resources.

(**8**) **The network is homogeneous**; today, homogeneous network is the exception, rather then a rule. We have different servers, systems, clients that interacts. The implication of this is that we have to assume interoperability between these systems sooner or later but we must be aware of it. We might also have some proprietary protocols that might also take time to send on and they may stay without support, so we should avoid them.

These fallacies are intruduced over the dacade ago, and more than four decades since we started building DS, but the characteristics and underlying problems remains pretty the same. It is interesting fact

that designers, architects still asume that technology solves everything. This is not the case in DS, and these fallacies should not be forgoten. Because of these problems, DS are hard to implement corectly and they are hard to test and maintain.

This section will explain different aspects of DS in computing systems, that are important for future parts of the thesis. Section 1.2.1 gives more details about scalability and what it meens in modern day computer applications. Section 1.2.2 gives explanation what CC is, organizational aspects of CC as well as used models. Section 1.2.3 gives explanation what peer-to-peer networks are, and why are they important in modern DS. Section 1.2.4 gives general deffinition what mobile computing is and new ways of implementation DS.

1.2.1 Scalability

Scalability is the property of a system to handle a growing amount of work by adding resources to the system [13]. When talking about computer systems scalability can be represented in two flavors:

- **Scaling vertically** means upgradeing the hardware that computer systems are running on. Vertical scaling can increase performance to what latest hardware can offer, and here we are limited by the laws of phisics and Moor's law [14]. Tipical example that require this type of scaleing is relation dataase server. These capabilities are insufficient for moderate to big workloads.
- **Scaling horizontally** means that we scale our system by keep adding more and more computers, rather than upgrading the hardware of a single one. With this apporouch we are (almost) limitless how much we can scale. Whenever performance degrades we can simply add more computers (or nodes). These nodes are not required to be some high-end machines.

Table 1.1 summarize differences between horizontall and verticall scaling.

Feature	Scaling vertically	Scaling horizontally
Scaling	Limited	Unlimited
Managment	Easy	Comlex
Investments	Expensive	Afordable

Table 1.1: Differences between horizontall and verticall scaling.

Scaling horizontally is a preferable way for scaling DS, not because we can scale easier, or because it is significantly cheaper than vertical scaling after a certain threshold [13] but because this approuch comes with few more benefits that are esspecially important when talking about large-scale DS. Adding more nodes gives us two important properties:

- **Fault tolerance** means that applications running on multiple places at the same time, are not bound to the fail of a node, cluster or even DCs. As long as there is a copy of application running somewhere, user will get response back. As a consequence of runnin multiple copies of a service and on multiple places, we have that service is more **avalible**, that running on a single node no metter how high-end that node is. Eventually all nodes are going to break, and if we have multiple copies of the same service we have more resilient and more avalible system to serve user requests.

- **Low latency** refers to the idea that the world is limited by the speed of light. If a node running application is too far away, user will wait too long for the response to get back. If same application is running on multiple places, user request will hit node that is closest to the user.

But despite all the obvious benefits, for a DS to work properly, we need the write software in such a way that is able to run on multiple nodes, as well as that accept **failure** and deal with it. This turns out to be not an easy task.

For example users need to be aware when using DS, is related to distributed data storage systems. Storage implementations that relys on vertical scaling to ensure scalability and faulte tolerance, have one nasty feature.

This nasty feature is represented in theorem called **CAP theorem** presented by Eric Brewer [15]. Proven after insepction [16], CAP theorem states that it is impossible for a distributed data store to simultaneously provide more than two out of three guarantees:

(1) **C**onsistency, which means that all clients will see the same data at the same time, no matter which node they are connected to. Clients my not be connected to the same node since data could be replicated on many nodes in different locations.

(2) **A**vailability, which means that any client issued a request will get response back, even if one or nodes are down. DS will not interprate this situation as an exception or error. Availability is represented in percentage, and it describe how much downtime is allowed per year, This can be calculated using formula:

$$Availability = \frac{uptime}{(uptime + downtime)} \tag{1.1}$$

Industry is using measureing avelability in "class of nines". Availability class is the number of leading nines in the availability figure for a system or module [17]. This metric relates to the amount of time (per year) that a service is up and running. Table 1.2 show different classes of nine and their availability and unavailability in minutes per year (**min/year**) for some examples [17].

We can calculate availability class if we have system availability A, the system's availability class is define as [17]:

Type	Availability	Unavailability
Unmanaged	90%	50,000
Managed	99%	5,000
Well-managed	99.9%	500
Well-managed	99.9%	500
Fault-tolerant	99.99%	50
High-availability	99.999%	5
Very-high-availability	99.9999%	0.5

Table 1.2: Downtime for different classes of nines.

$$e^{\log_{10} \frac{1}{(1-A)}} \tag{1.2}$$

It is important to notice that even a 99% available system gives almost four days of downtime in a year, which is unacceptable for services like Facebook, Google, AWS etc. And when service is down, companies are loosing customers.

(3) **P**artition tolerance, which means that the cluster must continue to work despite any number of communication breakdowns between nodes in the system. It is important to state that in a distributed system, partitions can't be avoided.

Years after CAP theorem inception, Shapiro et al. prove that we can alleviate CAP theorem problems by only in some cases, and offers **Strong Eventual Consistency (SEC) model** [18]. They prove that if we can represent our data structure to be:

- **Commutative** $a * b = b * a$
- **Associative** $(a * b) * c = a * (b * c)$
- **Idempotent** $(a * a) = a$

where $*$ is a binary operation, for example: *max*, *union*, *or* we can rely on SEC properties,

1.2.2 Cloud computing

We can define cloud computing (CC) like aggregation of computing resources as a utility, and software as a service [19]. Hardware and software in big DCs provide services for user consumption over the internet [20]. Resources like CPU, GPU, storage, and network are utilities and can be used as well as released on-demand [21]. The key strength of the CC are offered services [19].

The traditional CC model provides enormous computing and storage resources elastically, to support the various applications needs. This property refers to the cloud ability to allow services, allocation of additional resources, or release unused ones to match the application workloads on-demand [22]. Services usually fall in one of three main categories:

- **Infrastructure as a service (IaaS)** allows businesses to purchase resources on-demand and as-needed instead of bying and manageing hardware themself.
- **Platform as a service (PaaS)** delivers a framework for developers to create, maintain and manage their applications. All resources are managed by the enterprise or a third-party vendor.
- **Software as a service (SaaS)** deliver applications over the internet to its users. These applications are managed by a third-party vendor, .

Figure 1.1 show difference in control and management of resources between different cloud options and on-premises solution.

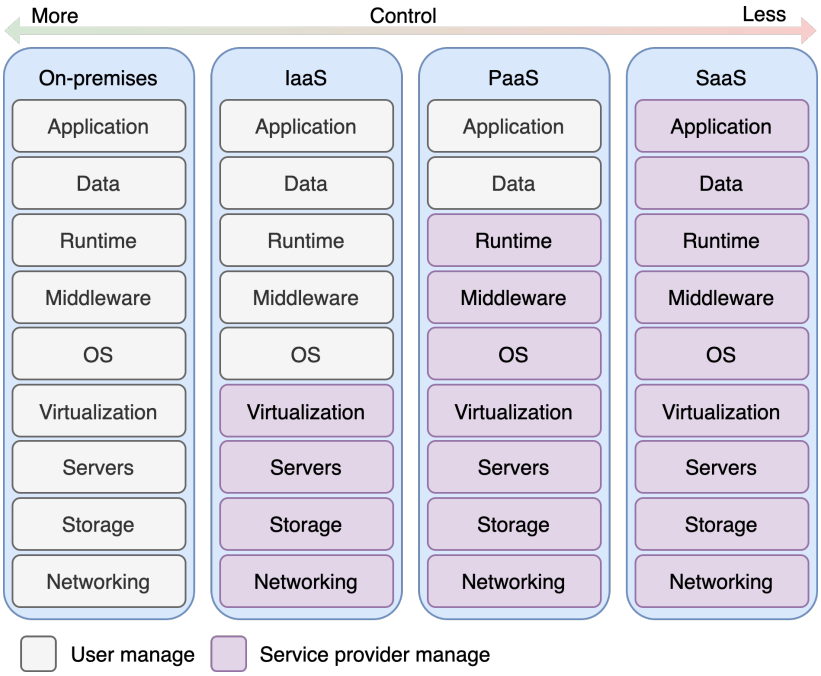

Figure 1.1: Difference between cloud options and on-premises solution.

The user can choose a single solution, or combine more of them if such a thing is required depending on preferences and needs.

By the ownership, CC can be categorized into three categories:

- **Public cloud** is type where CC is delivered over the internet, and shared across many many organizations and users. In this type of the CC, architecture is built and maintained by others. Users and organizations pay for what they use. Examples include: AWS EC2, Google App Engine, Microsoft Azure etc.

- **Private cloud** is type where CC is dedicated only to a single organization. In this type of the CC, architecture is built by organization who may offer their solution or services to the users or other organizations. These services are in domain what the

organization does, and that organization is in charge of maintanance. Examples include VMWare, XEN, KVM etc.

- **Hybrid cloud** is such environment that uses both public and private clouds. Examples include: IBM, HP, VMWare vCloud etc.

Table 1.3 show comparison of public, private and hybrid cloud capabilities.

Capabilities	**Public cloud**	**Private cloud**	**Hybrid cloud**
Data control	IT enterprise	Service Provider	Both
Cost	Low	High	Moderate
Data security	Low	High	Moderate
Service levels	IT specific	Provider specific	Aggregate
Scalability	Very high	Limited	Very high
Reliability	Moderate	Very high	Medium/High
Performance	Low/Medium	Good	Good

Table 1.3: Comparison of public, private and hybrid cloud capabilities.

In the rest of the thesis, if not stated differently when CC term is used it denotes public cloud.

CC has been the dominating tool in the past decade in various applications [7]. It is changing, evolving, and offering new types of services. Resources such as container as a service (CaaS), database as a service (DBaaS) [23] are newly introduced. The CC model gives us a few benefits. Centralization relies on the economy of scale to lower the cost of administration of big DCs. Organizations using cloud services avoid huge investments. Like creating and maintaining their own DCs. They consume resources usually created by others [7] and pay for usage time – a pay as you go model.

But centralization give us few really hard problems to solve. As already stated in section 1.1 data is required to be moved to the cloud from data sources, which introduces a high latency in the system [8].

There are few notable attempts to help data ingestion into the cloud. Remote Direct Memory Access (RDMA) protocol makes it possible to read data directly from the memory of one computer and write that data directly to the memory of another. This is done by using *specialized hardware* interface cards and switches and software as well, and operations like read, write, send, receive etc. do not go through CPU. With this caracteristivs, RDMA have low latencies and overhead, and as such reach better throughputs [24]. This new hardware may not be cheap, and not evey CC provider use them for every use-case. And this may not be enough, esspecially with ever growing amount of IoT devices and services.

Over the years there are more as a service options avalible, forming **everything as a service (XaaS)** model [25]. This model propose that any hardware or software resource can be ofered as a service to the users over the internet.

Table 1.4 shows common examples of SaaS, PaaS, and IaaS applications.

Platform type	**Common Examples**
IaaS	AWS, Microsoft Azure, Google Compute Engine
PaaS	AWS Elastic Beanstalk, Azure, App Engine
SaaS	Gmail, Dropbox, Salesforce, GoToMeeting

Table 1.4: Common examples of SaaS, PaaS, and IaaS.

CC is giving a user an illusion that he is using single machine, while the backgroud implementaion is fairly complicated and consists of various elements that are composed of countles machines. CC is tipical example of horizontally scalable system presented in 1.2.1

1.2.3 Peer-to-peer networks

Peer-to-peer (P2P) communication is a networking architecture model that partitions tasks or workloads between peers [26]. All peers are

created equally in the system, and there is no such thing as a node that is more important then others. Every Peer have a portion system resources, such as processing power, disk storage or network bandwidth, directly available to other network participants, without the need for central coordination by servers or stable hosts [26]. P2P nodes are connected and share resources without going through a separate server computer that is responsabile for routing. Figure 1.2 show difference in network topology between P2P networks and client-server architecture.

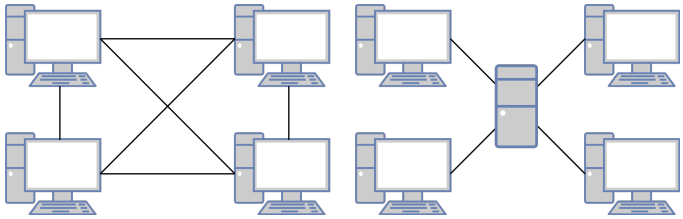

Figure 1.2: P2P network and client-server network.

Peers are creating a sense of virtual community. This community of peers can resolve a greater tasks, beyond those that individual peers can do. Yet, these tasks are beneficial to all the peers in the system [27].

Based on how the nodes are linked to each other within the overlay network, and how resources are indexed and located, we can classify networks as [28]:

- **Unstructured** do not have a particular structure by design, but they are formed by nodes that randomly form connections [29]. Their strenght and weaknes at the same time is ther lack of structure. These networks and robust when peers join and leave network. But when doing query, they must found more possible peers that have same peace of data. Tipical example of this group is a Gossip-based protocols like [30].

- **Structured** peers are organized into a specific topology, and the protocol ensures that any node can efficiently search the network for a resource. The famos type of structured P2P network is a Distributed Hash Table (DHT). These networks maintain lists of neighbors to do more efficent lookup, and as such they are not so robust when nodes join or leave the network. DHT commonly used in resource loopkup systems [31], and as efficent resource lookip management and scheduling of applications, or as an integral part of distributed storage systems and NoSQL[32] databases.

- **Hybrid** combine previous two models in various ways.

P2P networks are great tool in many arsenals, but because their unique ability to act as a server and as a client at the same time we must be aware and pay more attention to security because they are more vulnerable to exploits [33]/

1.2.4 Mobile computing

Mobile cloud computing (MCC), was the first idea that introduced task offloading [34, 35]. Heavy computation remains in the cloud. Mobile devices run small client software and interact with the cloud, over the internet using his resources.

The main problem with MCC is that the cloud is usually far away from end devices. That leads to high latency and bad quality of experience (QoE) [35]. Especially for latency-sensitive applications. Even though MCC is not that much different from the standard cloud model. We had moved a small number of tasks from the cloud. Thus opening the door for future models.

To overcome cloud latency and MCC problems, research led to new computing areas like edge computing (EC). EC is a model in which computing and storage utilities are in proximity to data sources [7]. The cloud is enhanced with new ideas for future generation applications [10].

Over the years, designs like fog [36], cloudlets [6], and mobile edge computing (MEC) [37] emerged. In this thesis, we refer to all these models as edge nodes. They all use the concept of data and computation offloading from the cloud closer to the ground [38], while heavy computation remains in the cloud because of resource availability [10].

EC models introduce small-scale servers that operate between data sources and the cloud. Typically, they have much less capabilities compared to the cloud counterparts [39]. These servers can be spread in base stations [37], coffee shops, or over geographic regions to avoid latency as well as huge bandwidth [6]. They can serve as firewalls [40] and pre-processing tier, while users get a unique ability to dynamically and selectively control the information sent to the cloud.

1.3 Distributed computing

DC can be defined as the use of a DS to solve one large problem by breaking it down into several smaller parts, where each part is computed in the individual node of the DS and coordinatio is done by passing messages to one another [12]. Computer programs that use this strategy and runs on DS are called **distributed programs** [41, 42].

Similar to CC in Section 1.2.2, to a normal user, DC systems appear as a single system similar to one he use every day on his personal computer.

DC share same fallacies to DS presented in 1.2.

In this section we are going to explain examples of distributed programs that rely on DS that are important for future parts of the thesis. Section 1.3.1 gives more details about using DS to process huge quantity of data. Section 1.3.2 gives explanation how to build scalable applications that are able to whitstand huge request rate and large amout of users.

1.3.1 Big Data

Tearm big data means that the data is unable to be handled, processed or loaded into a single machine [43]. That menas that traditional data mining methods or data analytics tools developed for a centralized processing may not be able to be applied directly to big data [44].

New tools and methos that are developed are relying on DS and one specific feature **data locality**. Data locality can be described as a process of moving the computation closer to the data, instead of moving large data to computation [45]. This simple idea, minimizes network congestion and increases the overall throughput of the system.

In 1.1 we already give two examples how huge generated data could be, and when we incude other IoT sensors and devices these numbers will just keep getting bigger [46].

On contrary to relational databases that mostly deal with structured data, big data is dealing with various kinds of data [43, 44, 45]:

- **Structured** data is kind of data that have some fixed structure and format. Tipical example of this is data stored inside table of some database. organizations susully have no huge problem extracting some kind of value out of the data.
- **Unstructured** data is kind of data where wo do not have any kind of structure at all. These data sources are heterogeneous and may containing a combination of simple text files, images, videos etc. This type of data is usully in raw format, and organizations have hard time to derive value out.
- **Semi-structured** data is kind of data that can contain both previously mentioned types of data. Example of this type of data is XML files.

Along it's share size, big data have other instantly recognizable features called **V's** of big data [47]. Name is derived from starting

letters from the other features that are describing big data. Image 1.3 show 6 V's comonly used to represent the big data.

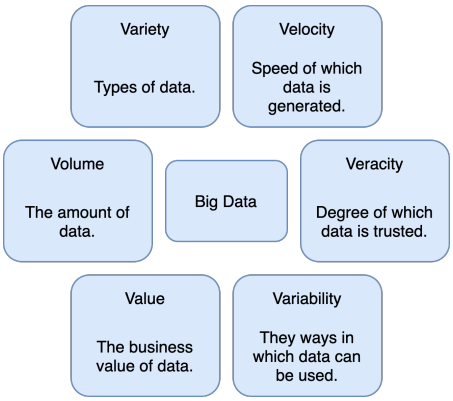

Figure 1.3: V's of Big Data.

Processing in big data systems can be represented as [48, 49]:

- **Batch processing** represents data prcessing technique that is done on huge quantity of the stored data. This type of prcessing is usually slow and requure time.
- **Stream processing** represents data processing technique that is done as data get into the system. This type of processing is usually done on smaller quantity of the data **at the time**, and it is faster.
- **Lambda architectures** represents processing technique where stream processing and handling of massive data volumes in batch are combined in a uniform manner, reducing costs in the process [49].

Big data systems, are not processing and value extracting systems. Big data systems can be separated in few categories: (1) data storage,

(2), data ingestion (3), data processing and analytics. All these system aids to properly analyze ever growing requirements [50],

Dispite promise that big data offers to derive value out of the collected data, this task is not easy to do and requre properly set up system filtering and removeing data that contains no value. To aid this idea, data could be filtered and little bit preprocessed on close to the source [51], and as such sent to data lakses [52].

1.3.2 Microservices

There is no single comperhensive deffinition of what a microservice is. Differnet people and organizations use different definition do describe them. A working definition is offered in [53] as "s microservice is a cohesive, independent process interacting via messages". Despite lack of comperhensive definition all agree on few features that come with microservices:

(**1**) they are small computer programs that are independently deployable and developed.

(**2**) they coud be developed using different languages, principles and using differend databses.

(**3**) they communicate over the network to achieve some goal.

(**4**) they are organized around business capabilities [54].

(**5**) they are implemented and mainteined by a small team.

Industry is migrating much of their applications to the cloud, because CC offers to scale their computing resources as per their usage [55]. Microservices are small loosely coupled services that follos UNIX philosophy "do one thing, and do it well" [56], and they communicate over well defined API [53].

This architecure patter is well aligned to the CC paradigm [55], contrary to previous models like monolith whose modules cannot be

executed independently [53, 57], and are not well aligned with the CC paradigm [57]. Table 1.5 summrize differences between monolith and microservie architecture.

Feature	**Monolith**	**Microservices**
Structure	Single unit	Independent services
Management	Usually easier	Add DS complexity
Scale/Update	Entire app	Per service
Error	Usually crush entire app	App continue to work

Table 1.5: Differences between horizontall and verticall scaling.

Since their inception, microservices architecture is gone through some adaptations. And modern day microservices are extended with two new models each with it's unique abilities and problems:

- **Cloud-native applications**, are specially designed applications for CC. They are distributed, elastic and horizontal scalable system by their nature, and composed of (micro)services which isolates state in a minimum of stateful components [58]. These type of applications are self-contained, could be deployed independently, and they are composed of loosely coupled microservices that are packaged in lightweight containers. They have Improved resource utilization, and they are centered around APIs.

- **Serversles applications** is computing model, where the developers need to worry only about the logic for processing client requests [59]. Logic is represented as event handler that only runs when client request is received, and billing is done only when these functions are executing [59]. **Cold start** is one of features of the severless computing, and we can define it as user requests need to wait, until new container instance is up and running before can do any processing at all. Most providers have 1–3 second cold starts, and this is important for certain types of applications where latency is concern. Cold start is

only happening when there are no *warm* containers available for the request, meaning there is no single instance to server request. Other features include: (1) simplified services development, (2) faster time to market, (3) and lower costs.

- **Service Mesh** is designed to standardize the runtime operations of applications [60]. As part of the microservices ecosystem, this dedicated communication layer can provide a number of benefits, such as: (1) observability, (2) providing secure connections, or (3) automating retries and backoff for failed requests. With these features, developers only focus on implementation of buisniss logic, while operators gain out-of-the-box traffic policies, observability, and insights from the services. Advocates of microservice movemant, nowdays recommend using service mesh architecture when running microservices in production environments.

Microservices communicate over a network to fulfil some goal using message passing technique and technology-agnostic protocols such as HTTP. They can be implemented as:

- Representational state transfer (REST) services [61], is an architectural style with set of constraints that users can create web srevices and interoperability between computer systems on the internet. It is based on HTTP routs to define resources, and used HTTP verbs to represent operations over these resources. It relys on textual based communications, and payload could be represented using $JSON$, XML, $HTML$ etc.

- Remote procedure calls (RPC) represent architectural way to design services that are able to call subroutines that are located in different places, usually on other machine. Client is calling these operations like they are located localy in his address space.

- Event-driven services, are services where communicatoin between services is done using events. Events are sent on some channel and other read messages that are received on other channel. These channels could be implemented either like message queues or message topics. Services connect to message queu or subscribe to the specific topic, and when messige arive, they can act acording to message type.

They are well aligned with text based protocols like HTTP/1 using *JSON* for example, or binary protocols such as HTTP/2 using *protobuf* and *gRPC* for example, and even new faster version like HTTP/3 over new *QUIC* protocol, designed by Google.

To enshure vider range of devices that are able to comunicate with the rest of the systems, developrs usually have a gateway into the system that is REST service, and other services could be implemented in different way.

It is important to point out, that all flavors of microservices applications rely on continuous delivery and deployment [62]. This is enabled by lightweight containers, instead of virtual machines [63], and orchestration tools such Kubernetes [64]. These concepts wll be described in more detail in Section 1.5.

Microservices architecture are good starting point especially for build as a service applications, and applications that should serve huge amount of requests and users. Esspecially with benefits of CC to pay for usage, and ability to scale parts of the system independently. Although they are not necessarily easy to implement properly. There are more and more critique to the architecture model [65]. Microservices are relying and use parts of the DS, and as such they inherit almost all problems DS has.

One particilar thing that users need to be aware of is **idempotency**. In microservices applications, developers are dealing with inconsistencies in distributed state, and their operations should be implemented as idempotent. An operation is idempotent if it will produce the same results when executed over and over again. It is a

strategy that means that operations with sade effects like creation or deletion can be called any number of times, while guaranteeing that side effects only occur once. Idempotency is term that come from mathematics, and can be represented by simple idempotency law for operation $*$ like [66]:

$$\forall x, x * x = x \tag{1.3}$$

Not all Create, Read, Update, Delete (CRUD) operations are idempotent by default. But developers need to make effort to make all of them idempotent, to prevent bad outcomes and incosistant state.

Table 1.6 show list of idempotent and non-idepotent for standard CRUD operations:

Operation	**Idempotent**	**Non-idempotent**
Create		x
Read	x	
Update	x	
Delete	x	

Table 1.6: Idempotent and non-idempotent operations.

Crate operation is not idempotent by default, but to make it idempotent there are multiple strategies how to do so. Most comon way is to create **idempotency key** that will be sent in the request, and based on that request server can decide is this operations already invoked or not. If server is already "seen" specified idempotency key, than operation is already done and we can return just responce that operation is done but no operation will be done over the state of the service or application. If server see idempotency key for the first time, that is the signal that this request is new one, and it should be done.

Idempotency key could be stored in any kind of the storage, it is not uncommon that these keys are stored in cache storage with some

time to live (TTL) policy that will automatically remove the key after specified time.

Other option that is comonly used is hasing user specified actions. This is usefull to know what part of action set is already done and what is not. This strategy is used in scenarios where we must preserve order of actions.

Best chance to success when implementing microservices architecture, is to simply follow existing patters and use existing solutions with proven quality.

1.4 Similar computing models

In this section we are going to shortly describe models that are similar to the DS, and as such they may be the source of confusion. Section 1.4.1 describes parallel computing compared to DC. Section 1.4.2 describe decentralized computing compared to DS.

1.4.1 Parallel computing

DC and parallel computing seems like models that are the same, and that may share some features like simultaneously executing a set of computations in parallel. Broadly speaking, this is not far from the truth [41]. But distinguished between the two can be presented as follows: in parallel computing all processor units have acces to the shared memory and have some way of the faster inter-process communication, while in DS and DC all processors have their own memory on their own machine and communicate over network to other nodes which is significantly slower.

These models are similar, but they are not indentical, and the kind of problems they are designed to work on are different. Figure 1.4 visually summarize the architectural differences between DC (*up*) and parallel computing (*down*).

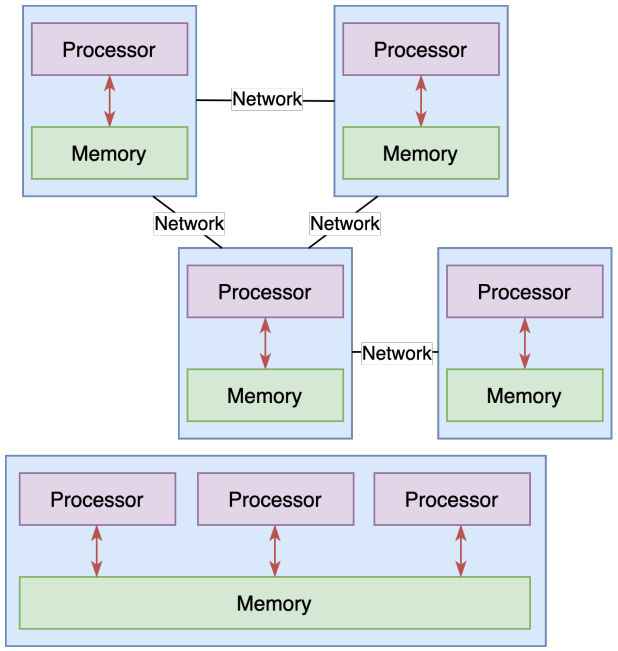

Figure 1.4: Architectural difference between DC and parallel computing.

Parallel computing is often used strategy with problems, that due to their nature or constraints must be done on multi-core machines simultaneously [67]. It is ofthen, that huge problems are divided into smaller ones, which can then be solved at the same time.

There are number of tasks that requre parallel computing like simulations, computer graphics rendering or different scenarios in scientific computing.

1.4.2 Decentralized systems

Decentralized systems are similar to DS, in technical sense they are still DS. But if we take closer look, these systems **should not** be owned by the single entity. CC for example is perfect example of DS, but it is not decentralized by it's nature. It is centralized systems by

the owner like AWS, Google, Microsoft or some other private compay because all computation needs to be moved to big DCs [8].

By today standards, when we are talking about decentralyzed systems, we usually think of blockchain or blockchain-like technology [68]. Since here we have distributed nodes, that are scattered and there is no single entity that own all these nodes. But even if this technology is run in the cloud, it is loosing the decentralized feature. This is the caveat we needs to be aware of. These systems are facing different issues, because any participent in the system might be malicious and they need to handle this case.

Nontheless, CC can and should be decentralized in a sense that some computation can happend outside of cloud big DCs, closer to the sources of data. These computation could be owned by someone else, and big cloud companies could give their own solution to this as well to relax centralization and problems that CC will have esspecially with ever growing IoT and mobile devices.

1.5 Virtualization techniques

Virtualization as a technique started long ago in time-sharing systems, to provide isolation between multiple users shareing a single system liek a mainframe computer [69].

In [70] Sharma et al. describe virtualization as technologies which provide a layer of abstraction of the physical computing resources between computer hardware systems and the software systems running on them.

Modeern virtualization diferentiate few different tools. Some of them are used as an integral part of the infrastructure for some flavors like IaaS, while others are used in different CC flavors as well as microservices packageing and distribution format, or are new and still are looking for their place. These options are:

- **Virtual machines (VM)** are the oldest tehnology of the three. In [70] Sharma et al. describe them as a self-contained operating environment consisting of guest operating system and associated applications, but independent of host operating system. VMs enable us to pack isolation and better utilization of hardware in big DCs. They are vidly used in IaaS environment [71, 72] as a base where users can install theirown operating system (OS) and required software tools and applications.

- **Containers** provide almost same functionality to VMs, but there are several subtle differences that make them a goto tool in modern develpment. Instead of the guest OS running on top of host OS, containers use tools that are in Linux kernles like *cgroups* that limits process resource usage so that single process can not starve other processes and use all the resources for himself, and *namespaces* to provide isolation and partitions kernel resources so that single process see node resources like he only exists there. Containers reduce time and footprint from development to testing to production, and they utilize even more hardware resources compared to VMs and show better performance compared to the VMs [73, 63]. Containers provide easier way to pack servies and deploy and they are esspecially used in microservices architecture and service orchestration tools like Kubernetes [64]. Google stated few times in their on-line talks that hey have used container technology for all their services, even they run VMs inside containers for their cloud platform. Even though they exist for a while, containers get popularized when companies like Docker and CoreOS developed user-friendly APIs.

- **Unikernels** are the newest addition to the virtualization space. In [74] Pavlicek define unikernels as small, fast, secure virtual machines that lack operating systems. Unikernels are comprised of source code, along with only the required system calls and

drivers. Because of their specific design, they have single process and they contains and executes what it absolutely needs to nothing more and nothing less [75]. They are advertised that new technology that will safe resources and that they are *green* [76] meaning they save both power and money. When put to the test and compared to containers they give interesting results [75, 77]. Unikernels are still new technology and they are not widly adopted yet. But they give promessing features for the future, esspecially **if** properly ported to ARM architectures, and various development languages. Unikernes will probably be used as a user applications and functions virtualization tool, because their specific architecture, esspecailly for serverless applications presened in 1.3.2.

Figure 1.5 represent architectural differences between VMs, containers and unikernels.

With every virtualization technique, ultimate goal is to pack as much applications on existing hardware as possible, so that there is no resources that are left not used — we are trying to achieve high resource utilization.

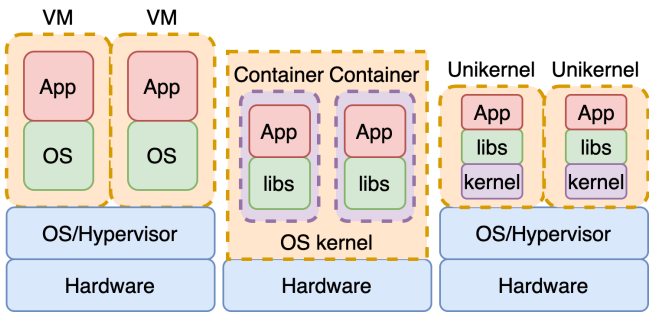

Figure 1.5: Architectural differences between VMs, containers and unikernels.

1.6 Deployment

Over the years two different approuches evolved how to deploy infrastructure and applications. The difference just get more amplified, when CC and microservices get into the picture, where frequent deployment is very common.

Here evolve new strategy to mange and deploy complicated infrastructure elements — Infrastructure as code (IaC). In his book [78] Wittig et al. describe it as a process of managing and provisioning computer data centers through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools.

Deplyments in such complex environment can be separated how they handle changes on existing infrastructure or applications on:

- **Mutable model**, is model where we have in place changes which mean that the parts of the existing infrastructure or applications get updated or changed in order to do update. In place change can produce few problems: (1) more risk, beacause in place change my not finish completly which put our infrastructure or the application in possible bad state. This is esspecially probem, if we have a lot of services and multiple copies of the same service. Possibility that our system is not on, is a lot higher, (2) high complexity, this is direct implicatoin of previous feature. Since out change might not get fully done, we can't give guarantis that our infrastructure or applicatoin is transitioned from one version to the another — change is not **descrete**, but **continues** since we might end up in some state in between where we are now and where we want to be.

- **Immutable model**, is model where we do not do any in place changes on existing infrastructure or application whatsoever. In this model, we replace it complety with new version that is updated or changed compared to previous version. Previous version

get discarded in favour of new version. Compared to the previous model, immutable deployment have: (1) less risk, since we do not change existing infrastructure or the applicatoin but we start new one with and shut down previous one. This is important esspecially in DS where everyting can fail at any tome, (2) previous property reduce complexity of mutable deployment model. This is direct implicatoin of previous feature, since we shut down and fully replace previous version with new one we get **descrete** version change and atomic deployment with dafer deployments with fast rollback and recovery processes. On the other hand, this process requre more resources **??**, since both hersions must be present on the node in order to this process is done. Second problem is data that is used by the app, we should not lost the data that app is generated. If we externalize data than this problems is resolved. We should not rely on local storage, but store that data elsewhere, esspecially when the parts of the system are volatile and changed often. Key adventage of this approuch, is avoiding downtime experienced by the end user when new features are released.

Figure 1.6 summaraze difference bewteen prevous infrastructure deplyment models.

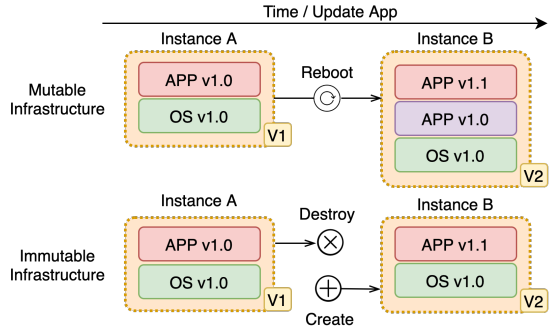

Figure 1.6: Difference bewteen mutable and immutable deplyment models

Immutability is a simple concept to understand, and simplify a lot esspecially in DS [79]. Write down some data, and ensure that it never changes. It can never be modified, updated, or deleted [80]. When this is combined with premisse that we can avoid downtime esspecially in complex DS, it is clear why immutable model is gainign more and more popularity (esspecially with arival of containers). Immutable infrastructure deployment offer few models how to deploy change on the services, even in production to test it, or switch to whole new version. These strategeis include:

- **Blue-Green deployment**, this strategy require two separate environemnts: (1) *Blue* current running version, and (2) *Green* is the new version that needs to be deployed. When we are satisfied that the green version is working properly, we can gradually reroute the traffic from the old environment to the new environment for example by modifying DNS. This strategy offers near zero downtime.

- **Canary update** is the strategy where we do direct a small number of requests to the new version — the canary. If we are satisfied with the change, we can continue to increase number of requests and monitor how service is working with increasing load, monitor for erros etc.

- **Rolling update** streategy update large environments a few nodes at a time. The setup is similar to blue-green deployment, but here we have single environment. With this strategy, new version gradually replaces the old one. But this is not the only benefit. If for whatever reason, new verrsion is not working properly on larger amount of nodes, we can always do rolling back to previous version.

With mutable infrastructure these strategies would be hard to implement, and maybe it is not possible at all.

Beside infrastructure deployemnt, there is another side that we must consider, and that is how describe these deployments. Here we can consider two different strategies:

- **Imperative**, with this option users have to write code or specific instructions step by step what specific tool need to do in order that application or infrastrucgure is properly setup. In this approuch we have a *smart* user who describe *dumb* machine what is needed to be done and in what order to achieve desired state.

- **Declarative**, with this option user have to describe end state or what is his desired state, and tool needs to figure out the way how to do this. Here we have *smart* system that will found a way how to achieve desired state, and we have user who *do not care* in what order actions need to be done — that is what system needs to do. User do not need to worry about timing, this simplify whole process and code always represents the latest state. With this type of deployment, we can offer users two different models: (1) use existing formats that are user familiar with like JSON, YAML, XML etc., or (2) create new domain specific language that users need to learn, but we might be able to optimaze description.

With introduction of *LinuxKit*, we can create Linux subsystems based around containers, that are very secure. With linuxkit, every purt of the Linux subsystem is running inside container, so we can assemble a Linux subsystem with services that are needed. As a result, systems created with LinuxKit have a smaller attack surface [81] than general purpose systems. This is important from security point of view, but also from infrastructure deployment because we can compose specific OS based around containers that we need for different purpose. And we can update, change and adopt these OS for every machine or purpose we need.

Deployment is based around changeint parts of the OS, and his services that are running inside containers. As a result, everything can be removed or replaced. It's highly portable and can work on desktops, servers, IoT, mainframes, bare metal, and virtualized systems.

1.7 Concurency and parallelism

People usually confuse these two concepts. Even they looks similar, they are different way of doing things. In his talk Rob Pike [82] give great explanation and examples on this topic. In this toke he give great deffinitions of these concepts like:

- **Concurrency** is composition of independently executing things. Concurrency is about dealing with a lot of things at once.
- **Parallelism** is simultaneous execution of multiple things. Parallelism is about doing a lot of things at once.

These things are important, esspecially when building applications and systems that should achieve very high throughput. We must build them with a good structure and a good concurrency model. These features enables possible parallelism, but with communication [82]. These ideas are based on Tony Hoare work of Communicating Sequential Processes (CSP) [83].

1.7.1 Actor model

In actor model, the main idea is based around **actors** which are small concurrent code, that communicate independently by sending messages, removing the need for lock-based synchronization [84]. This model propose similar idea like Tony Hoare in his work with CSP [83], and actors are ofttten confused with CSP. Table 1.7 give differences between actor model and CSP.

Feature	CSP	Actor model
Fault tolerance	Distributed Queue	Hierarchy of supervisors
Process identity	Anonymus	Concrete
Composition	NA	Applicable
Communication	Queue	Direct
Message passing	Sync	Async

Table 1.7: Ddifferences between actor model and CSP.

Actors do not share memory, and they are isolated by nature. Actor can create another actor/s and even watch on them in case they stop unexpectedly. And when an actor finished its job, and he is not needed anymore, it disappears. These actors can create complicated networks that are easy to understand, model and reason about and everything is based on a simple massage passing mechanism.

Every actor have a designated message box. When a message arrives, actor will test message type and do job acording to message type he received. In this way we are not dependent of lock-based synchronization that can be hard to understand, and it can cause serious problems.

Actor model is fault tolerant by design. It support crush to happend, because there is a "self heal" mechanism that will monitor actor/s, and when crash happend it will try to apply some strategy, in most cases just restart actor, but other strategies could be applied. This philosophy is really usefull, because it is hard to think about every single failure option.

1.8 Motivation and Problem Statement

In [85] Greenberg et al. point out that micro data-centers (MDCs) are used primarily as nodes in content distribution networks and other "embarrassingly distributed" applications.

One size never fits all, so the cloud should not be our final computing shift. Various models presented in 1.2.4, show possibility that computing could be done closer to the data source, to lower the latency for its clients by contacting the cloud only when needed, while heavy computation remains in the cloud because of resource availability. Send to the cloud only information that is crucial for other services or applications [51]. Not ingest everything as the standard cloud model proposes.

MDCs with a zone-based server organization is a good starting point for building EC as a service, but we need a more available and resilient system with less latency. EC originates from P2P systems [4] as sugested by López et al., but expands it into new directions and blends it with the CC. But, infrastructure deployment will not happen until the process is trivial [86]. Going to every node is tedious and time consuming. Especially when geo-distribution is taken into consideration.

A well defined system could be offered as a service, like any other resource in the CC. We can offer it to researchers and developers to create new human-centered applications. If we need more resources on one side, we can take from one pool of resources and move to another one. But on the other hand, some CC providers might choose to embed it into their own existing system, hiding unnecessary complexity, behind some communication interface or proposed application model.

The idea of small servers with heterogeneous compute, storage, and network resources, raise interesting research idea and motivation for this thesis. Taking advantage of resources organized locally as micro clouds, community clouds, or edge clouds [87] suggested by Ryden et al., to help power-hungry servers reduce traffic [88]. Contact the cloud only when needed [51]. Send to the cloud only information that is crucial for other services or applications. Not ingest everything as the standard CC model proposes.

To achieve such behavior, dynamic resource management, and device management is essential. We must perceive available resources,

configuration, and utilization [89, 37].

Traditional DCs is a well organized and connected system. On the other hand, these MDCs consist of various devices, including ones presented in 1.2.4 that are not [90]. This idea, brings us to the problem this thesis address.

EC and MDCs models lack dynamic geo-organization, well defined native applications model, and clear separation of concerns. As such they cannot be offered as a service to the users. They usually exist independently from one another, scattered without communication between them, offered by providers who mostly lock users in their own ecosystem. Co-located edge nodes should be organized locally, making the whole system and applications more available and reliable, but also extending resources beyond the single node or group of nodes, maintaining good performance to build servers and clusters [91].

This cloud extension deepens and strengthens our understanding of the CC as a whole. With the separation of concerns setup, EC native applications model, and a unified node organization, we are moving towards the idea of EC as a service.

Based on this, we define the problem through the following research questions two segments:

(1) *Can we organize geo-distributed edge nodes in a similar way to the cloud, adopted for the different environment, with clear separation of concerns and familiar applications model for users.*

(2) *Can we offer these organized nodes as a service to the developers and researchers for new human-centered applications, based on the cloud pay as you go model?*

(3) *Can we make model in such a way that is formaly correct, easy to extend, understand and reason about?*

This cloud-like extension makes the whole system and applications more available and reliable, but also extends resources beyond the single node. Satyanarayanan et al. in [40] show that MDCs can

serve as firewalls, while Simić et al., in [51] use similar idea as pre-processing tier. At the same time, users are getting a unique ability to dynamically and selectively control the information sent to the cloud. Years after its inception, EC is no longer just an idea [40] but a must-have tool for novel applications to come.

1.9 Research Hypotheses, and Goals

Based on reserach questions and motivation presented in 1.8, we derive the hypothesis around which the thesis is based. It can be summarized as follows:

- Organize EC nodes in a standard way based on cloud architecture, with adaptation for an EC geo-distributed environment. Give users the ability to organize nodes in the best possible way in some geographic areas to serve only the local population in near proximity.
- Offer it to researchers and developers to create new human-centered applications. If we need more resources on one side, we can take from one pool of resources and move to another one, or organize them any other way needed.
- Present clear separation of concerns for the future EC as a service model, and establish a well-organized system where every part has an intuitive role.
- Present unified model that supports heterogeneous EC nodes, with a set of technical requirements that nodes must fulfil, if they want to join the system.
- Present a clear application model so that users can use full potential of newly created infrastructure.

From this hypothesis, we ce derive the primary goals of this thesis, where the expected results include:

(1) *The construction of a model with a clear separation of concerns for the model influenced by cloud organization, with adaptations for a different environment. With a model for EC applications utilizing these adaptations. This addresses the first research question, and is the topic of Chapter 3.*

(2) *The constructed model is more available, resilient with less latency, and as such it can be offered to the general public as a service like any other service in the cloud. This addresses the second research question, and is the topic of Chapter 3.*

(3) *The constructed is is well described formaly, using solid mathematical theory, but also easy to extend both formaly and technicaly, easy to understand and reason about. This addresses the third research question, and is the topic of Chapter 3.*

1.10 Structure of the thesis

Throughout this introductory Chapter, we defined the motication for out work with problems that this thesis addresses and presented the necessary background informations and areas to support our work. Here we outline the rest of the thesis.

Chapter 2 presents the literature review, where we examine different aspects of existing systems and methods important for the thesis. We analyze existing nodes organizational abilities in both industry and academia frameworks and solutions to address our first research question. We further exemine platform models from industry and academia tools and frameworks to address our second research question. And last but not least, we examine current strategies to offload tasks from the cloud. All three parts address our third research question.

Chapter 3 details our model, how it is related to other research and where it connects to other existing models and solutions. We further describe our solution as well as protocols requried for such sysrtem to be implemented foramly. We give examples of how exisintg infrastructure could be used, as well as familiar application model for developers.

Chapter 4 present our implementation details, results possible applications with limitations.

Chapter 5 concludes our work and presents opportunities for further research and development.

Chapter 2

Research review

Faced real issues and limits of cloud computing, both academia, and the industry started researching and developing viable solutions. Some research is focused more on adapting existing solutions to fit EC, while others experiment with new ideas and solutions.

In this Chapter, we present the results of our research reviews addressing issues discussed earlier. in Section 2.1 we review existing nodes organizational abilities, as well platform models from industry and academia in Section 2.1. In Section 2.3 we review cloud offloading techniques. In Section 2.4 we review some applicartio models, and we give position of this thesis compared to all revied aspects in Section **??**.

2.1 Nodes organization

A zone-based organization for edge servers (ES) presented by Guo et al. in [92], gives an interesting perspective on EC in a smart vehicles application. They showed how zone-based models enable continuity of dynamic services, and reduce the connection handovers. Also, they show how to enlarge the coverage of ESs to a bigger zone, thus expanding the computing power and storage capacity of ESs. Since one of

the premises of EC is geo-distributed workloads, organizing ESs into zones and regions could potentially benefit EC.

Baktir et al. [93] explored the programming capabilities of software-defined networks (SDN). Findings show SDN can simplify the management of the network in cloud-like environment. SDN is a good candidate for networking because it hides the complexity of the heterogeneous environment from the end-users. Kurniawan et al. [94] argue about very bad scalability in centralized delivery models like cloud content delivery networks (CDN). They proposed a decentralized solution using nano DCs as a network of gateways for internet services at home [94]. These DCs are equipped with some storage as well. Authors show a possible usage of nano DCs for some large scale applications with much less energy consumption.

Ciobanu et al. [95] introduce an interesting idea called drop computing. The authors show that we can compose EC platforms ad-hoc, thus enabling collaborative computing dynamically, using a decentralized model over multilayered social crowd networks. Instead of sending requests to the cloud, drop computing employees the mobile crowd formed of nearby devices, hence enabling quick and efficient access to resources. The authors show an interesting idea of how to form a computing group ad-hoc. Forming ad-hoc platforms from crowd resources might raise a few possible concerns: (1) crowd nodes availability, and (2) offered resources. Crowd nodes might be an interesting idea as a backup option, in cases we need more computing power or storage and there are no more available resources to use.

MDCs are an interesting model and area of rapid innovation and development. Greenberg et al. [85] introduce MDCs as DCs that operate in proximity to a big population (on contrary to nano DCs that serves a lot smaller population), thus minimizing the latency and costs for end-users [96, 85], and reducing the fixed costs of traditional DCs. Minimum size of an MDCs is defined by the needs of the local population [85, 97], with agility as a key feature. Here agility means an ability to dynamically grow and shrink resources and satisfy the demands and

usage of resources from the most optimal location [85].

2.2 Platform models

Kubernetes [64] is an open-source variant of Google orchestrator Borg [98]. All workloads end in the domain of one cluster [64, 98, 99]. Rossi et al. [99] focuses on adapting Kubernetes for geo-distributed workloads using a reinforcement learning (RL) solution, to learn a suitable scaling policy from experience. Like every other machine learning implementation this could be potentially slow due to the required model training. Kubernetes is a promising solution, but it might not be the best proposal for EC. By design, Kubernetes operate in a completely different environment from EC. The second potential issue is the deployment concept that might be too complicated for EC workloads. On the other hand, there are a few ideas that are worth exploring, such as loosely coupling elements with labels and selectors. Nonetheless, researches show that adapted Kubernetes architecture works for geo-distributed workloads like EC.

Ryden et al. [87] present a platform for distributed computing with attention to user-based applications. Unlike other systems, the goal is not to implement a resource management policy, but to give users more flexibility for application development. Users implement applications using Javascript (JS) programming language, with some embedded native code for efficiency. Similar to [95], the authors use volunteer nodes to run all the workloads, with the difference that some nodes are used for storage, while others are used for calculation. Sandboxing technique protects nodes running applications from malicious code. It is an interesting idea to show how users can develop their applications and run them in an EC environment.

Lèbre et al. [100] describe a promising solution of extending OpenStack, an open-source IaaS platform for fog/edge use cases. They try to manage both cloud and edge resources using a NoSQL database.

Implementation of a massively distributed multi-site IaaS, using OpenStack is a challenging task [100]. Communication between nodes of different sites can be subject to important network latencies [100]. The major advantage is that users of the IaaS solution can continue using the same familiar infrastructure. In [101] Shao et al. present a possible MDCs structure serving only the local population, in the smart city use-case.

In [10], the Ning et al. show current open issues of EC platforms based on the literature survey. They illustrate the usage of edge computing platforms to build specific applications. In the survey, the authors outline how CC needs EC nodes to pre-process data, while EC needs massive storage and strong computing capacity of CC.

In [81] the de Guzmán et al. present solution based on Kubernetes that use Kubernetes Deployment Manifests in order to reuse successful principles from Kubernetes by creating virtual machine for each Pod using Linuxkit. Their solution is based on the immutable infrastructure pattern, and instead of containers they use the virtual machines as the unit of deployment. Authors prove that the attack surface of their system is reduced since Linuxkit only installs the minimum OS dependencies to run containers. It represent interesting usage of LinuxKit to deploy OS dependencies, and immutable infrastructure pattern, but VMs might be a bit problem for small devices, and ARM nodes as well as complex flow of Kubernetes application model. But nonetheless, it is interesting extension of Kuibernetes framework and prove that LinuxKit can be used for immutable infrastructures with custom OS.

In [102] Sami et al. show interesting platform for dynamic sevices distribtion over Fog nodes using volunteer nodes. Their platform is tuned for container placement with relevance and efficiency on volunteering fog devices, near users with maximum time availability and shortest distance. They do this *on the fly* with improved QoS.

There are few industry operating frameworks for EC, like Amazon Greengrass [103], deeply connected to the entire Amazon cloud

ecosystem, or KubeEdge [104], a lightweight extension of Kubernetes framework. These frameworks are mainly used for user-based applications, while, for instance, General Electric Predix [105] is a scalable platform used for industrial IoT applications.

2.3 Task offloading

As already mention in 1.2.4 EC nodes rely on the concept of data and computation offloading from the cloud closer to the ground [38], while heavy computation remains in the cloud because of resource availability [10].

Offloading is effective when using cloud servers but this principle introduce long latency, which some applications can't tolerate. On the other hand mobile devices and sensors do not have sufficient battery energy for task offloading [106]. The computation performance may be compromised due to insufficient battery energy for task offloading, so these devices might send their data to nearby EC nodes.

In literature, there are few platforms proposing task offloading [96, 38, 39, 35, 90, 106] to the nearby edge layer. These offloading techniques are based on different parameters, options, and techniques to put tasks to different sets of nodes in such a way that it won't drain mobile devices and sensors baterry. After compuitation is done, this edge layer send pre-processed data to the cloud for further analyse, storage et.

In [102] authors used Evolutionary Memetic Algorithm (MA) to solve their multi-objective container placement optimization problem to achieve better QoS.

2.4 Application models

Ryden et al. [87] present Nebula, a platform for distributed computing. Users develop their applications using JS only. This restriction

comes from using Google Chrome Web browser-based Native Client (NaCl) sandbox [107]. JS is a popular language at the moment, but the restriction of a single language might be a deal-breaker for some usages. If standard virtual machines become too resource-demanding, a solution using containers could provide sandboxing and bring better resource utilization.

In [86] Satyanarayanan et al. represent an interesting view on cloudlets s a "data center in a box.". They give example that cloudleta should support the wide range of users, with minimal constraints on their software. They pun emphesis on transient VM technology. The emphasis on transient VMs is beacuse cloudlet infrastructure is restored to its pristine software state after each use, without manual intervention. In the time they conduct they reserach containers might not be working solution or it might be hard to use them. By today standards, containers may even fit better, and pack more user softvare on same hardware. This may be case for the unikernels, once they reach wider adoption rate and stable products.

Various Kubernetes variants lik [104, 99], give users possibiliety to run different applications like web servers and databses even on smaler devices cretaing green DC [91].

Satyanarayanan et al. [40] propse concept of edge-native applications that will separate space into 3 tiers.Tier 1 represent various mobile and IoT devices. These devices produce a lot of data. Tier 2 represent applications running in cloudlets or other EC models, that will pre-process, fiter data before it goes further. Finaly, tier 3 represnet classic cloud applications that will accept pre-processed and filer data from previous tier. This represent interesting concept, and give wide space for users and application developement.

In [108] Beck et al. argue that applications should use message bus, because most mobile edge applications are expected to be event driven. Message bus system is an interesting, because virtualized applicatoin can subscribe to message streams, i.e., topics. And if for some reason mobile edge applications can't reach close EC server, it can always

send data to cloud. So cloud applications should be changed so so slightly, just to accept this case.

2.5 Thesis position

Different from the aforementioned work,this thesis focuses on descriptive dynamic organization of geo-distributed nodes over an arbitrary vast area that lacks in other solutions. To achieve such a task, thesis model is influenced by the CC organization, but adapted for a different environment such as EC. These adaptations are followed by clear Separation of concerns (SoC) and EC applications model. All these allow us to push the whole solution more towards EC as a service model like any other utility in the cloud.

Chapter 3

Micro clouds and edge computing as a service

This section explains the model of EC as a service compared to the traditional CC model. We present the formation of such a system with a formal model and proof of concept implementation based on the proposed model.

In Section 3.1 present configurable model used to describe our system. in Section 3.2 we present separation of concerns for our model. In Section 3.3 we discuss how this system could be offered as a service model and could be used for EC as a service. In Section 3.4 we present possible application model and how users can utilize newly created architecture fully, using existing application models. Section 3.5 present desired option for infrastructure deployment. In Section 3.6 we present why formal models are important in computer science and DS in particular, as well as model of our system with all protocols and nodes properties. Section 3.7 give repercussion of proposed model, and how it can be used as stand alone model where other features could be implemented on top of that or used as service for other, existing, systems. Finally, section 3.8 present limitation of our model.

3.1 Configurable Model Structure

Satyanarayanan et al. [40] propse architecture eparated into 3 tiers. Combained with MDCs and a zone-based server organization we get a good starting point for building EC as a service and micro cloud system, because we can extend the computing power and storage capacity serving local population. But, we need a more available and resilient system with less latency. To do that we need to extend these concepts and adopt them for different usage scenario.

If we take a look at the CC design, every part contributes to a more resilient and scalable system. Regions (or DCs) are isolated and independent from each other, and also contain resources application needs. They are usually composed of few availability zones [109]. If the one zone fails, there are still more of them to serve user requests. With some adaptations, EC could use a similar proven strategy. That strategy could be than used in applications beyound just EC.

Table 3.1 shows similar concepts between CC and edge centric computing (ECC) concepts, with accent on difference what is the physical part, and what are logical concepts.

Edge centric computing	**Cloud computing**
Topology (logical)	Cloud provider (logical)
Region (logical)	Region (physical)
Cluster (physical)	Zone (physical)

Table 3.1: Similar concepts between cloud computing and ECC.

We can combine multiple node clusters into a bigger logical concept of *region*, increasing availability and reliability of both system and applications. We are talking about geo-distributed systems, so we have a bit of a different scenario than in the standard CC model. The cloud region is a physical thing [109], while in the ECC a region could be represented in a different way. We now give a forma definition of a *region* in ECC as:

Definition 3.1.1. *In geo-distributed ECC and micro clouds, a regoin is used to describe a set of clusters over an arbitrary geographic region.*

Regions can accept or release clusters and clusters can accept or release nodes. Based on this property, we can define minimum size of an ECC region as:

Definition 3.1.2. *ECC regions are composed of at least one cluster, but can be composed of much more, so as to achieve a more resilient, scalable, and available system.*

To ensure less latency in the system, vast distance between clusters should be avoided in normal circumstances. In a CC, region extension requires physically connecting modules to the rest of the infrastructure [110].

Multiple regions form a second logical layer - *topology*. Formaly, we can define a *topology* in ECC as:

Definition 3.1.3. *In geo-distributed ECC and micro clouds, topology is a logical concept composed of a minimum one region, and could span over more regions.*

When designing a topology, especially if regions need to share information, vast distance between regions should be avoided, if possible. With these simple abstractions, we can cover any geographic region with the ability to shrink or expand clusters, regions, and topology. Separation on *clusters*, *regions*, and *topologies* is a matter of agreement and usages similar to modeling in Big Data systems [111, 112].

For example, clusters could be as wide as the whole city or small as all devices in a single household and everything in between. The city could be one region with parts of the city being organized into clusters. We can form a city topology by splitting the city into multiple regions containing multiple clusters, or we can form a country topology by splitting the country into regions, with cities being regions. We

can follow a more natural administrative division of some region, and organize resources by population usages.

Nodes inside the cluster should run some membership protocol. Gossip style protocols, like SWIM [30], could be used in conjunction with replication mechanisms [113, 114, 115] making the whole system more resilient. In [116] Simić et al. take a look from theoretical point of view on conflict free replicated data types (CRDTs) usage, to achieve SEC in EC and conclude that CRDTs could be natural fit to EC as long as we are aware of potential pitfals of CDRTs.

Single *topology* reflects one CC provider, so multiple *topologies* are forming $micro-clouds$ that are able to help CC with huge latency issues, pre-processing in huge volumes of data and relax and decentralize strict centralized CC model.

These micro-clouds have much less resources, copared to standard clouds but they are much closter to the user meaning they have much fater response. In case of storage, if data is not present in the time of user request, they can pull data from the cloud and cache it for latter.

This 3 thier acrchitecture with numerous clients in the bottom, micro clouds in the middle, and cloud on the top kinda resembles cache level architecture in CPU [117]. On lower levels we have fastest response time, since data is on the device. But at the same time, we have very limited storage capacity and processing power. As we go on upper tiers, we have more and more storage capacity and processing power, but contrary the response time is higher and higher. Esspecially when we take distance into consideration, and huge volumes of data that need to be moved to the cloud.

Figure 3.1. shows the 3 tier architecture, with the response time and resource avelability.

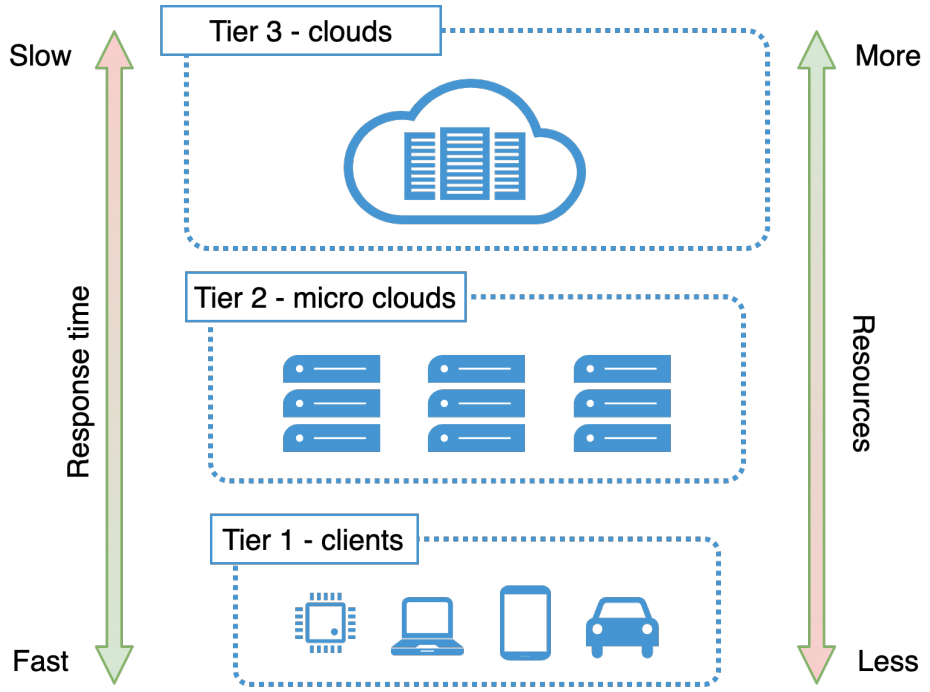

Figure 3.1: 3 tier architecture, with the response time and resource avelability

In everything as a service model [25], EC as a service fits in between CaaS and PaaS, depending on users needs.

3.2 Separation of concers

To describe physical services, Jin et al. [118] proposes three core concepts and specifies their relationships. These concepts are: (1) Devices, (2) Resources, and (3) Services.

SoC is a vital part of any system, especially if creating a platform to offer as a service. We based our SoC model for ECC as a service on these concepts, with adaptations for our use case, separated in three layers depicted in Figure 3.2.

The bottom layer consists of various devices, or data creators and services consumers. The second layer represents resources. Resources have a spatial feature and indicate the range of their hosting devices [118]. Developers at any time must know the resource utilization and spread, as well as application's state and health.

Resources represent EC nodes, and to be part of the system, a node must satisfy four simple rules:

(1) run an operating system with a file system

(2) be able to run some application isolation engine, for example a container or unikernels

(3) have available resources for utilization

(4) have stable internet connection

Services expose resources through an interface and make them available on the Internet [118]. These services respond to clients immediately if possible, or cache information [86, 119] for future requests. Services in the cloud should be able to accept pre-processed data, and they are responsible for computation and storage that is beyond the capabilities of ECC nodes. Services in the cloud should be able to take direct requests from client just in case that something catastrific happend to the micro-cloud that is close to the user.

Figure 3.2. shows the proposed SoC for every layer of the ECC as a service model.

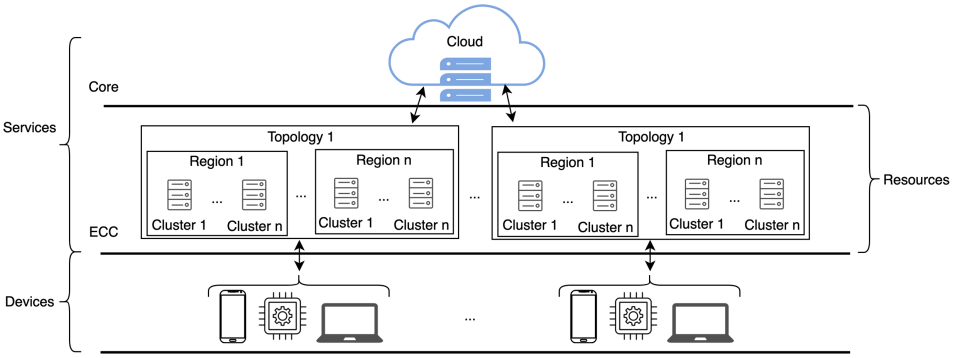

Figure 3.2: ECC as a service architecture with separation of concerns.

3.3 As a service model

Users should be able to develop their applications using two different models:

(1) **microPaaS or mPaaS**, where the platform is doing all the management and offers a simple interface for developers to deploy their applications. This model is similar to PaaS, and the only difference is that this is running in micro-cloud and synchronize with CC.

(2) **microCaaS or mCaaS**, if users require more control over resources requirements, deployment and orchestration decisions. This model is similar to CaaS, and the only difference is that this is running in micro-cloud and synchronize with CC.

Both variants should not stand alone, at least not for now. Fot that same reason we do not have microSaaS or mSaaS option, since that woul requre that whole application is running **only** in micro-clouds. But both options could be included, and part of any cloud model presented in 1.2.2, or offered separatly.

3.4 Applications Model

Traditional DCs and CC propose specially designed cloud-native applications [120], that are easier to scale, more available, and less error-prone when compared to traditional web applications [120]. Edge-native applications [40] should use the full potential of EC infrastructure, and keep good features of their cloud counterparts.

Applications may be split in front and back processing services. Front processing service is an edge-native application running inside micro cloud to minimize latency, while the back service runs in traditional cloid as a cloud-native application to leverage greater resources.

These edge-native applications will handle user requests coming to nearby MDCs, and communicate to cloud-native applications when needed. Separation like that gives developers better flexibility and large design space.

Frontend services model should be event-driven, with a subscription policy to message streams using topics [108]. The processing strategy is in the developer's hands, depending on the nature of the use-case.

Some examples may include:

(**1**) **events** that notify users if some value is above or below some defined threshold

(**2**) **stream** or processing data as it comes to the system

(**3**) **batch** does processing in predefined times over some bigger collection of data

(**4**) **other**, something that falls outside these models, or it is composition of multiple operations at once. This type should get events from some topic as they arrive, and user can define his own strategy what it needs to be done and how it needs to be pcessed. Usec can contact other existing services, and user is responsabile for optimization.

These types of applications could be implemented in may ways like those disccused in 1.3.2, or some adapted variant of those models.

Section 3.4.1 argue about how applications could be packed and deployed in the wild.

3.4.1 Packaging

Because their nature, micro clouds could be most likley composed of ARM devices. These devices in many cases are not able to run full VMs because their hardware restrictions. In recent years there are advances in VMs technology and their ability to run VMs on ARM devices. In [121] Ding et al. show such possibility to run VMs on ARM devices.

But even if VMs are fully compatable with ARM devices, we still inherite VMs large footprint, already descused in 1.5 if we try to use tham *asis*.

On contrary, containers and unikernels give us $moreforless$ same functionality but using less resources, meaning we can run more services in containers and even more in unikernels. But untill unikernels are fulkly ready to be used they will fall in nice to have category, and we should stick to containers. But even with containers we need to be aware of their limitations and pitfalls, and know that there is no *silverbullit* and there is no one solution for all scenarios.

At the moment, containers are more matured solution than unikernels, and require less resources than VMs. On top of that, there are numerous of tools already existing using containers that could be utilized. Knowing all this, at first stages of micro-clouds, containres should be option to go.

In [122] Simić et al. show benefits of using containres in large sclae edge computing systems from theoretical point of vew, by looking into architecture difference. Authors focus on differences between VMS and containers in cases where services needs to be runned on ARM devices with limited resources.

In future and when unikernels are more matured and tested, they could be used for paritical use-cases and applications, esspecially like events or serverless implementations. Containres will probably not be fully replaced, but they can co-exist with unikernels in some cases where we need more controll over running single function.

Like any other system, users can create their own variants of the systems and different flavours optimized for certen solutions. In that cases, they may favorized one solution over the other one. But in general case containers and unikernels should be prefered way to package, run and distributed user applicatoins in micro-clouds environment.

3.5 Immutable infrastructure

As described in 1.6, we have few options when it comes to setup and deply infrastructure and/or applications. This thesis propose DS model that is build with three tier architecture that should operate in geo-distributed environment we blieve that **immutable deployment** model would be good fit. It will simplify deployment process, since we want to rely on atomic operations and do not want to left misconfigured system at any level. If something like that happend, we will end up in problem, that would be hard to properly address and resolve.

Geo-distributed micro-clouds model that is described in this chaper, should operate in two levels of deployment that are build one on top of the another:

(1) **Infrastructure deployement**, update and change should be atomic and immutable. Users should do changes declaratively — mutations of the system, by telling the system what new state should be, and let the system to figure out the best way how to do user specified changes. In this category, we can account any change that is doing on the cluster, region or topology that user/s operate on. For example create of new cluster, region, topology or doing configurations of the setup system. The only change

that could possible be done by imperative strategy is updates on the nodes itself. But even for this strategy, it would be beneficial if we could usedeclarative way **if possible**. It is important to notice that **mutation** does not mean in place change, but just name of operation. This deployment strategy is reserved for operations people, but if compay or team is small any developer could do this. Developers should not be dealing with this part of the deployment.

(**2**) **Services deployment**, make sense only if previous action is takane. We must have infrastructure setup already, in order do put any sort of services (applications) into the system. As previous model, this should be done declaratively as well, and all changes dould be done immutably without any in place change. User should specify his new state or "view of the word" declaratively and let system do all the changes he wants. All user services should be packed as described in 3.4.1 because this simplifies way services are put to the nodes. When done properly, this allow operations people to do faster changes with almost zero downtime deployments with all strategies already discused in 1.6. this part of the deployemnt should be done by developers, since they done implementation and testing. They know hos much resources they need for their service, what type of service they had developed. This deployment could be done in colaboration between operations and deveopers, if compay is big or time is properly separated.

It is important to notice, that both deployments should be closley followed, for posible errors and problems so that users can act accordingly. This deployment messages, logs and traces should be stored in centralized log system, for convenient lookup, alerting and reporting.

Separation like this, simplify deployment and usage for both application development spectrums:

(i) operation people like **devops** who should be dealing with infrastructure deployment, tooling setup, applications deployment, monitoring and in general health of applications and infrastructure.

(ii) **developers** who should be dealing with development of the services, their interactions, and cloud to micro cloud and vice-verca synchrnonizations.

Only with tight colaboration with those two development roles, such complex system like one presetned in this chapter can be alive, well and servig user requests withoyt collaps.

3.6 Formal model

Ensuring reliability and correctness of any system is very difficult, and should be mathematically based. Formal methods are techniques that allow us specification and verification of complex (software and hardware) systems based on mathematics and formal logic. There are several options how to formaly describe DS: TLA+, *pi* calculus, combinational topology, asynchronous session types (MST), etc.

Unfortunately, becasue of their nature DS cannot always be formally described by any of the existing models. But if the nature of the DS that is developing is such that can be formaly described, it is recomended and beneficial. Formaly described and corect model, can save hours, days and event months of hard debugging, testing to reveal all bugs and problems in the system.

Infrastructure deployment will not happen until the process is trivial [86], hence the key is to simplify ECC management. The main problem is that going to every node is tedious and time consuming, especially in a geo-distributed environment. In such complex environemnt, formal models are of great help if we can model and prove

that protocols that system reyls on are correct. The system we propose tackles this issue using remote configuration and it relies on four formaly modeled protocols:

(**1**) **health-check** protocol informs the system about state of every node

(**2**) **cluster formation** protocol forms new clusters dinamicaly

(**3**) **idempotency check** protocol for preventing creating existing infrastructure

(**4**) **list detail** protocol shows the current state of the system to the user

These three protocols are base of the geo-distribued Infrastructure deployment.

In Section 3.6.1 we give short introduction what MST are and what properties they guarantee. Section 3.6.2 describe formal model of healh-check protocol. Section 3.6.3 describe cluster formation protocol. Section 3.6.4 describe protocol that return data based on some query.

3.6.1 Multiparty asynchronous session types

We can model communication protocols using [123], an extension of *multiparty asynchronous session types* (MPST) [124] — a class of behavioral types tailored for describing distributed protocols relying on asynchronous communications. The type specifications are not only useful as formal descriptions of the protocols, but we can also rely on a modeling-based approach developed in [123] to validate our protocols satisfies multiparty session types safety (there is no reachable error state) and progress (an action is eventually executed, assuming fairness).

The first step in modeling the communications of a system using MPST theory is to provide a *global type*, that is a high-level description of the overall protocol from the neutral point of view. Following [123], the syntax of global types is constructed by:

$$\mathsf{G} \;::=\; \{\mathsf{p} \dagger \mathsf{q}_i{:}\ell_i(\mathsf{T}_i).\mathsf{G}_i\}_{i\in I} \quad | \quad \mu\mathbf{t}.\mathsf{G} \quad | \quad \mathbf{t} \quad | \quad \mathtt{end}$$

where $\dagger \in \{\rightarrow, \twoheadrightarrow\}$ and $I \neq \emptyset$. In the above, $\{\mathsf{p} \dagger \mathsf{q}_i{:}\ell_i(\mathsf{T}_i).\mathsf{G}_i\}_{i\in I}$ denotes that *participant* p can send (resp. connects) to one of the participants q_i, for $\dagger = \rightarrow$ (resp. $\dagger = \twoheadrightarrow$), a *message* ℓ_i with the *payload* of *sort* T_i, and then the protocol continues as prescribed with G_i. $\mu\mathbf{t}.\mathsf{G}_1$ is a recursive type, and $\mathbf{t}$ is a recursive variable, while `end` denotes a terminated protocol. We assume all participants are (implicitly) disconnected at the end of each session (cf. [123]).

The advance of using approach of [123], when compared to standard MPST (e.g., [124]), is in a relaxed form of choice (a participant can choose between sending to different participants), and, $\twoheadrightarrow$, that explicitly connects two participants, hence (possibly) dynamically introducing participants in the session. Both of these features will be significant for modeling our protocols (we will return to this point).

The second step in modeling protocols by MPST is providing a syntactic projection of the protocol onto each participant as a local type, that is then used to type check the endpoint implementations. We use the definition of projection operator given in [123, Figure 1.2]. In essence, the projection of global type G onto participant p can result in $\mathsf{S}_\mathsf{p} = \mathsf{q}!\ell(\mathsf{T})\ldots$ (resp. $\mathsf{S}_\mathsf{p} = \mathsf{q}!!\ell(\mathsf{T})\ldots$) when $\mathsf{G} = \mathsf{p} \rightarrow \mathsf{q}{:}\ell(\mathsf{T})\ldots$ (resp. $\mathsf{G} = \mathsf{p} \twoheadrightarrow \mathsf{q}{:}\ell(\mathsf{T})\ldots$), and, dually, $\mathsf{S}_\mathsf{p} = \mathsf{q}?\ell(\mathsf{T})\ldots$ (resp. $\mathsf{S}_\mathsf{p} = \mathsf{q}??\ell(\mathsf{T})\ldots$) when $\mathsf{G} = \mathsf{q} \rightarrow \mathsf{p}{:}\ell(\mathsf{T})\ldots$ (resp. $\mathsf{G} = \mathsf{q} \twoheadrightarrow \mathsf{p}{:}\ell(\mathsf{T})\ldots$), while the projection operator "skips" the prefix of a global type if participant p is not mentioned neither as sender nor as receiver. Furthermore, a local type must be represented by the following syntax:

$$\mathsf{S} \;::=\; +\{\mathsf{q}_i\alpha\ell_i(\mathsf{T}_i).\mathsf{S}_i\}_{i\in I} \quad | \quad \mu\mathbf{t}.\mathsf{S} \quad | \quad \mathbf{t} \quad | \quad \mathtt{end}$$

where $\alpha \in \{!, !!\}$ or $\alpha \in \{?, ??\}$ (in which case $\mathtt{q}_i = \mathtt{q}_j$ must hold for all $i, j \in I$, to ensure consistent external choice subjects, cf. [123, Page 6.]), and $I \neq \emptyset$. Interested reader can find details in [123].

For simplicity, we consider all participants are communicating within a single private session, and that all sent messages, but not yet received, are buffered in a single queue that preserves their order.

Actually, the order is preserved only for pairs of messages having the same sender and receiver, while other pairs of massages can be swapped, since these are asynchronously independent.

3.6.2 Health-check protocol

In a clustered environment, every node has a channel where it sends metrics, as a health-check mechanism. We can use this channel or create a new one to spread actions to the nodes, for example, a cluster formation message.

Figure 3.3. shows a low-level health-check protocol between a single node and the rest of the system, involving the following participants: Node, Nodes, State, and Log.

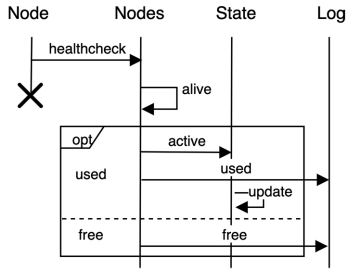

Figure 3.3: Low level health-check protocol diagram.

The participants which are included in Figure 3.3 follow the next protocol: that is described informally below:

(**1**) **Node** sends a health-check signal to the nodes service;

(**2**) **Nodes** accept health-check signals for every node, update node metrics and if node is used in some cluster, inform that cluster about the node state;

(**3**) **State** contains information about nodes in the clusters, regions and topologies;

(**4**) **Log** contains records of operations. Users can query this service.

Every node will inform the system that it exists via health-check ping. However, the rest of the system will be informed about that ping if and only if (henceforth iff) the node is used in some cluster.

In the following, we present Algorithm 1 to describe how the system will store the node data and determine if the node is free or used.

Algorithm 1: Health-check data received

```
   input: event, config
 1 if isNodeFree(event.id) then
 2 |   if exists(event.id) then
 3 |   |   renewLease(event.id, config.leaseTime);
 4 |   |   updateData(event.id, event.data);
 5 |   else
 6 |   |   leaseNewNode(event.id, config.leaseTime, event.data);
 7 |   |   saveMetrics(event.id, event.metrics);
 8 |   end
 9 else if isNodeReserved(event.id) then
10 |   updateData(event.id, event.data);
11 else
12 |   renewLease(event.id, config.leaseTime);
13 |   updateData(event.id, event.data);
14 |   saveMetrics(event.id, event.metrics);
15 |   sendNodeACK(event.id);
16 end
```

To describe servers or nodes (terms are used interchangeably) in the system formally, we can use set theory. In the beginning, the server

set S is empty, denoted with $S = \emptyset$. To determine the node state, we can use a node-id structure, for example.

Using node-id structure or some other trchnique, we can define free nodes in the systems as follows:

Definition 3.6.1. *Nodes are free iff they do not belong to any cluster.*

For example, if the received health-check message from the particular node contains only node-id, it is free, otherwise, it is not.

If we have n free nodes in the wild, denoted with s_i, where $i \in \{1, \ldots, n\}$, and they notify the system with a health-check ping that they are free, than we should add them to the server set and thus we have $S_{new} = S_{old} \cup \bigcup_{i=1}^{n} \{s_i\}$. The order in which messages arrive is not important.

Definition 3.6.2. *Nodes in the same cluster are equal as free nodes, and there are no special nodes.*

The only thing we care of is that nodes are alive and ready to accept some jobs.

Algorithm 1 describes how the system stores the node data and determine if the node is free or used. We can describe the s_i server in the server set S as a tuple $s_i = (L, R, A, I)$, where:

- L is a set of ordered key-value pairs, i.e., $L = \{(k_1, v_1), \ldots, (k_m, v_m)\}$ where $k_i \neq k_j$, for each $i, j \in \{1, \ldots, m\}$ such that $i \neq j$. L represents node labels or server-specific features. We based labels on Kubernetes [99] labels concept, which is used as an elegant binding mechanism for its components.
- R is a set of tuples $R = \{(f_1, u_1, t_1), \ldots, (f_m, u_m, t_m)\}$ representing node resources, where f_i, u_i, t_i, for $i \in \{1, \ldots, m\}$ are as follows:

  - f_i is the free resource value,
  - u_i is the used resource value, and
  - t_i is the total resource value.

- A is a set of tuples $A = \{(l_1, r_1, c_1, i_1), \ldots, (l_m, r_m, c_m, i_m)\}$, representing running applications, where l_j, r_j, c_j, i_j, for $j \in \{1, \ldots, m\}$, are as follows:
  - l_j represents labels, same way we used for node labels,
  - r_j is the resource set application requires,
  - c_j is the configuration set application requires, and
  - i_j is the general information like name, port, developer.

- I represent a set of general node information like: name, location, IP address, id, cluster id, region id, topology id, etc.

If we want to assign m (fresh) labels to the i_{th} server, we start with empty labels set $s_i[L] = \emptyset$, then we add labels to server. Therefore, we have $s_i[L]_{new} = s_i[L]_{old} \cup \bigcup_{j=1}^{m} \{(k_j, v_j)\}$.

Definition 3.6.3. *Every server from set S must have* non-empty *set of labels, but the number of labels for every server may vary.*

For the label definition, we can use arbitrary alphanumeric text for both key and value, separated with colon sign (e.g., os:linux, arch:arm, model:rpi, cpu:2, memory:16GB, disk:300GB, etc.).

Labels should be chosen carefully and agreed on upfront, but should be able to be changed if needed. Usually, labels include server resources, geolocation, and possibly some specific features that might be valuable for developers or administrators to target (e.g., SSD drive).

Following the above, we now present a formal description of the low-level health-check communication protocol (cf. Figure 3.3). The global protocol G_1 (given bellow) conforms the informal description

given at page 63: `node` connects `nodes` with *health_check* message and a payload of type T_1 that is required by the system to properly register node.

Then, depending on the received information, `nodes` **either** connects `state` with *active* message, informing the node status with a payload typed with T_2 (that contains information required to register active health-check sender), and then also connects `log` with the same message, **or** directly connects `log` informing the node is *free*.

$$\mathsf{G}_1 = \texttt{node} \twoheadrightarrow \texttt{nodes}{:}\mathit{health\_check}(\mathsf{T}_1).\begin{cases}\texttt{nodes} \twoheadrightarrow \texttt{state}{:}\mathit{active}(\mathsf{T}_2).\texttt{nodes} \twoheadrightarrow \texttt{log}{:}\mathit{used}(\mathsf{T}_2).\texttt{end}\\ \texttt{nodes} \twoheadrightarrow \texttt{log}{:}\mathit{free}(\mathsf{T}_2).\texttt{end}\end{cases}$$

Notice that in G_1 we indeed have a choice of `nodes` sending either to `state` or to `log`. Such communication pattern could not be directly modeled using standard MPST approaches. Also, notice that `state` will be introduced into the session only when receiving from `nodes`. Hence, if the session after the first ping from `node` to `nodes` proceeds with the second branch (i.e., connecting `nodes` with `log`) then `state` is not considered as stuck, as it would be in standard MPST, such as, e.g., [124], but rather idle.

Projecting global type G_1 onto participants `node`, `nodes`, `state` and `log` we obtain local types:

$$\mathsf{S}_{\texttt{node}} = \texttt{nodes}!!\mathit{health\_check}(\mathsf{T}_1).\texttt{end}$$

$$\mathsf{S}_{\texttt{nodes}} = \texttt{node}??\mathit{health\_check}(\mathsf{T}_1).+\begin{cases}\texttt{state}!!\mathit{active}(\mathsf{T}_2).\texttt{log}!!\mathit{used}(\mathsf{T}_2).\texttt{end}\\ \texttt{log}!!\mathit{free}(\mathsf{T}_2).\texttt{end}\end{cases}$$

$$\mathsf{S}_{\texttt{state}} = \texttt{nodes}??\mathit{active}(\mathsf{T}_2).\texttt{end}$$

$$\mathsf{S}_{\texttt{log}} = +\begin{cases}\texttt{nodes}??\mathit{used}(\mathsf{T}_2).\texttt{end}\\ \texttt{nodes}??\mathit{free}(\mathsf{T}_2).\texttt{end}\end{cases}$$

where, for instance, type $\mathsf{S}_{\texttt{nodes}}$ specifies `nodes` can receive the ping message from `node`, after which it will dynamically introduce either

`state` or `log` into the session, where in the former case it also connects `log` (but now with message *free*).

3.6.3 Cluster formation protocol

Another communication protocol in our system appears in the cluster formation process. Users are able to dynamically form new clusters. Here, we distinguish two different actions:

(1) The first action is a user-system communication, where the user sends a query to the system in order to obtain a list of available nodes based on the query parameters.

(2) The second action starts when the user sends a message to the system with a new specification. In this setting, the system involves participants: User, Queue, Scheduler, State, Nodes, Log, and NodesPool, that cooperate in order to dynamically form new clusters, regions or topologies, adhering to the scenario shown in Figure 3.4.

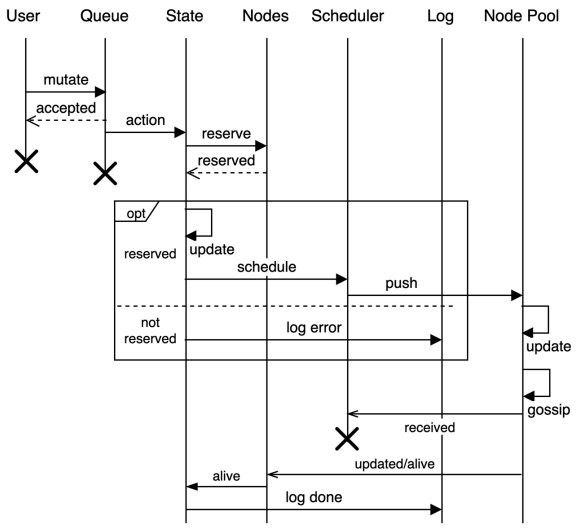

Figure 3.4: Low level cluster formation communication protocol diagram.

The participants follow the protocol that we now describe informally:

(**1**) **User** query Nodes service, based on some predefined criteria. User sends a create message to Queue, and, either gets response *ok*, or *error* if the message cannot be accepted due to missing rights or other issues. This operation is called *mutation*;

(**2**) **Queue** accepts a user message, and passes it to State. Messages are handled in FIFO (First In, First Out) order. The queue prevents system congestion, with received messages;

(**3**) **State** accepts mutation messages from Queue, and tries to store new information about cluster, region, or topology. If Nodes service is able to reserve all desired nodes, the system will store new user desired information and send a message to Scheduler to physically create clusters of desired nodes;

(**4**) **Nodes** accept messages from State. If possible, it will reserve desired nodes, otherwise it will send an error message to Log service. On a health-check message, if a node is used in some cluster, it will inform that the node is alive;

(**5**) **Scheduler** waits for a message sent from State, and pushes cluster formation messages to desired nodes;

(**6**) **Log** contains records of operations. Users can query this service to see if their tasks are finished or have any problems;

(**7**) **Nodes Pool** represents the set of n free nodes that will accept mutation messages. On message receive, every node will:

- (**i**) start gossip protocol to inform other nodes from the mutation message about cluster formation;
- (**ii**) send an event to Scheduler and Nodes service that it is alive and can receives messages;

If a user wants to get a list of free nodes, he must create a query using *the selector*, which is the set of key-value pairs desired by the user.

Algorithm 2 describes steps required to perform a proper node lookup based on a received selector value.

Algorithm 2: Nodes lookup

```
   input: query
1  Initialize: nodes ← []
2  foreach node ∈ freeNodes() do
3  |  if len(node.labels) == len(query) ∧ node.haveAll(query)
   |   then
4  |  |  nodes.append(node)
5  |  end
6  end
7  return nodes
```

We start with the empty selector $Q = \emptyset$, in which we append key-value pairs. Hence, when a user submits a set of p key-value pairs we have that $Q_{new} = Q_{old} \cup \bigcup_{i=1}^{p} \{(k_i, v_i)\}$.

Once the query is submitted, for every server in the set S, we need to check:

(1) the cardinality of the i_{th} server's set of labels and the query selector are identical in size

$$|s_i[L]| = |Q|\,, \text{ and} \tag{3.1}$$

(2) every key-value pair from query set Q is present in the i_{th} server's labels set $s_i[L]$, hence the following predicate must yield true:

$$P(Q, s_i) = \Big(\forall (k,v) \in Q \; \exists (k_j, v_j) \in s_i[L] \text{ such that } k = k_j \wedge v \leq v_j\Big) \tag{3.2}$$

The i_{th} server from the server set S will be present in the result set R, iff both rules are satisfied:

$$R = \{s_i \mid |s_i[L]| = |Q| \wedge P(Q, s_i), i \in \{1, \ldots, n\}\} \tag{3.3}$$

If the result set R is not empty, we reserve nodes for configurable time so that other users cannot see (and try to use) them, and, finally, we add reserved nodes with message data md to the task queue set $TQ_{new} = TQ_{old} \cup \{(R, md)\}$. When the task comes to execution, the task queue sends messages to every node.

Algorithm 3 describes the process required for cluster formation.

Algorithm 3: Clustering formation message

```
input: request, config
nodes ← searchFreeNodes(data.query)
reserveNodes(nodes, config.time)
pushMsgToQueue(nodes, data)
key ← saveTopologyLogicState(data)
watchForNodesACK(key)
```

Users can choose to override labels with their own or keep existing ones when including nodes in the cluster. If the node is free, or if the user did not change the node labels on cluster formation, the system will use default labels.

On message receive, the node will pick and contact a configurable subset of nodes $R_g \subset R$, and start the gossip protocol, propagating informations about nodes in the cluster (e.g, new, alive, suspected, dead, etc.). When every node inside newly formed cluster have complete set of nodes R obtained through gossiping, the cluster formation process is over.

Topology, region, or cluster formation should be done descriptively using YAML, or similar formats.

Algorithm 4 describes steps after nodes receive a cluster formation message.

Algorithm 4: Node reaction to clustering message

```
   input: event
1  switch event.type do
2      case formationMessage do
3          updateId(event.topology, event.region, event.cluster)
4          newState ← updateState(event.labels, event.name)
5          sendReceived(newState)
6          nodes ← pickGossipNodes(event.nodes)
7          startGossip(nodes)
8      end
9  end
```

In the following, we formally describe a low-level cluster formation communication protocol (cf. Figure 1.3), using the same extension of multiparty session types [123] as for the health-check protocol.

Global protocol G_2 (given below) conforms the informal description of the cluster formation protocol given at page 69. The protocol starts with `user` connecting `state` by message *query* and a payload typed with T_1 that contains user query data, and then `state` forwards the message by connecting `nodes`. Then, the protocol possibly enters into

a loop, specified with $\mu\mathbf{t}$, depending on the later choices. Further, `nodes` replies a response *resp* to `state`, that, in turn, forwards the message to `user`. The payload of the message is typed with T_2 that has response data, based on a given query. At this point, `user` sends to `state` one of three possible messages:

(1) *mutate*, and the mutation process, described with global protocol G', starts;

(2) *quit*, in which case the protocol terminates; or,

(3) *query* — this means the process of querying starts again, the query message is forwarded to `nodes` and the protocol loops, returning to the point marked with $\mu\mathbf{t}$.

The third branch is the only one in which protocol loops. Also, notice that `user` − `state` and `state` − `nodes` are connected before specifying recursion. Hence, even after a number of recursion calls these connections will be unique (thus, there is no need to disconnect them before looping).

$$\begin{aligned}\mathsf{G}_2 = &\texttt{user} \twoheadrightarrow \texttt{state}{:}\mathit{query}(\mathsf{T}_1).\texttt{state} \twoheadrightarrow \texttt{nodes}{:}\mathit{query}(\mathsf{T}_1).\\ &\mu\mathbf{t}.\texttt{nodes} \rightarrow \texttt{state}{:}\mathit{resp}(\mathsf{T}_2).\texttt{state} \rightarrow \texttt{user}{:}\mathit{resp}(\mathsf{T}_2).\\ &\begin{cases}\texttt{user} \rightarrow \texttt{state}{:}\mathit{mutate}().\mathsf{G}'\\ \texttt{user} \rightarrow \texttt{state}{:}\mathit{quit}().\texttt{end}\\ \texttt{user} \rightarrow \texttt{state}{:}\mathit{query}(\mathsf{T}_1).\texttt{state} \rightarrow \texttt{nodes}{:}\mathit{query}(\mathsf{T}_1).\mathbf{t}\end{cases}\end{aligned}$$

The mutate protocol G', activated in the first branch in G_1, starts with `user` sending `create` message to `state`, specifying also informations about new user desired state typed with T_3, and `state` replies back with *ok*. Then, `state` sends *ids* of the nodes to be reserved (specified in the payload typed with T_4) to `nodes`, that, in turn sends one of the two possible messages to `state`:

(i) *rsrvd*, denoting all nodes are reserved and the protocol proceeds as prescribed with G'', or

(ii) *error*, with error message in the payload, informing there has been unsuccessful reservation of nodes, in which case `state` connects `log` reporting the error and the protocol terminates.

$$\mathsf{G}' = \texttt{user} \to \texttt{state}{:}\mathit{create}(\mathsf{T}_3).\texttt{state} \to \texttt{user}{:}\mathit{ok}().\texttt{state} \to \texttt{nodes}{:}\mathit{ids}(\mathsf{T}_4).\begin{cases}\texttt{nodes} \to \texttt{state}{:}\mathit{rsrvd}().\mathsf{G}'' \\ \texttt{nodes} \to \texttt{state}{:}\mathit{err}(\mathsf{String}).\texttt{state} \twoheadrightarrow \texttt{log}{:}\mathit{err}(\mathsf{String}).\texttt{end}\end{cases}$$

Finally, in G'' `state` connects `sched` (Scheduler) with message *ids* and the payload that contains other data imported for mutation to be completed (typed with T_5). Then, `sched` connects `pool` (Nodes Pool) with *update* specified with T_6, after which `pool` replies back with *ok*, and connects to `nodes` sending new id's *nids* typed with T_4 (that contains successfully reserved user desired nodes). Now `nodes` notifies `state` the action was successful, that in turn connects `log` with the same message, and the protocol terminates.

$$\begin{aligned}\mathsf{G}'' = {} & \texttt{state} \twoheadrightarrow \texttt{sched}{:}\mathit{ids}(\mathsf{T}_5).\texttt{sched} \twoheadrightarrow \texttt{pool}{:}\mathit{update}(\mathsf{T}_6).\texttt{pool} \to \texttt{sched}{:}\mathit{ok}(). \\ & \texttt{pool} \twoheadrightarrow \texttt{nodes}{:}\mathit{nids}(\mathsf{T}_4).\texttt{nodes} \to \texttt{state}{:}\mathit{succ}().\texttt{state} \twoheadrightarrow \texttt{log}{:}\mathit{succ}().\texttt{end}\end{aligned}$$

We may now obtain the projections of global type G_2 onto the participants `user`, `state`, `nodes`, `log`, `pool`, and `sched`:

$$\begin{aligned}\mathsf{S}_{\mathsf{user}} = {} & \texttt{state}!!\mathit{query}(\mathsf{T}_1).\mu\mathbf{t}.\texttt{state}?\mathit{resp}(\mathsf{T}_2). \\ & + \begin{cases}\texttt{state}!\mathit{mutate}().\texttt{state}!\mathit{create}(\mathsf{T}_3).\texttt{state}?\mathit{ok}().\texttt{end} \\ \texttt{state}!\mathit{quit}().\texttt{end} \\ \texttt{state}!\mathit{query}(\mathsf{T}_1).\mathbf{t}\end{cases} \\ \mathsf{S}_{\mathsf{state}} = {} & \texttt{user}??\mathit{query}(\mathsf{T}_1).\texttt{nodes}!!\mathit{query}(\mathsf{T}_1).\mu\mathbf{t}.\texttt{nodes}?\mathit{resp}(\mathsf{T}_2).\texttt{user}!\mathit{resp}(\mathsf{T}_2). \\ & + \begin{cases}\texttt{user}?\mathit{mutate}().\texttt{user}?\mathit{create}(\mathsf{T}_3).\texttt{user}!\mathit{ok}().\texttt{nodes}!\mathit{ids}(\mathsf{T}_4).\mathsf{S}' \\ \texttt{user}?\mathit{quit}().\texttt{end} \\ \texttt{user}?\mathit{query}(\mathsf{T}_1).\texttt{nodes}!\mathit{query}(\mathsf{T}_1).\mathbf{t}\end{cases}\end{aligned}$$

where

$$\mathsf{S}' = + \begin{cases} \texttt{nodes}?rsrvd().\texttt{sched}!!ids(\mathsf{T}_5).\texttt{nodes}?succ().\texttt{log}!!succ().\texttt{end} \\ \texttt{nodes}?err(\mathsf{String}).\texttt{log}!!err(\mathsf{String}).\texttt{end} \end{cases}$$

$$\mathsf{S}_{\texttt{nodes}} = \texttt{state}??query(\mathsf{T}_1).\mu \mathbf{t}.\texttt{state}!resp(\mathsf{T}_2).$$
$$+ \begin{cases} \texttt{state}?ids(\mathsf{T}_4).+ \begin{cases} \texttt{state}!rsrvd().\texttt{end} \\ \texttt{state}!err(\mathsf{String}).\texttt{poll}??nids(\mathsf{T}_4).\texttt{state}!succ().\texttt{end} \end{cases} \\ \texttt{state}?query(\mathsf{T}_1).\mathbf{t} \end{cases}$$

$$\mathsf{S}_{\texttt{log}} = + \begin{cases} \texttt{state}??succ().\texttt{end} \\ \texttt{state}??err(\mathsf{String}).\texttt{end} \end{cases}$$

$$\mathsf{S}_{\texttt{pool}} = \texttt{sched}??update(\mathsf{T}_6).\texttt{sched}!ok().\texttt{nodes}!!nids(\mathsf{T}_4).\texttt{end}$$

$$\mathsf{S}_{\texttt{sched}} = \texttt{state}??ids(\mathsf{T}_5).\texttt{pool}!!update(\mathsf{T}_6).\texttt{pool}?ok().\texttt{end}$$

For instance, type $\mathsf{S}_{\texttt{sched}}$ specifies that participant `sched` gets included in the session only after receiving from `state` message *ids*, then `sched` connects `pool` with *update* message, after which expects to receive *ok* message and finally terminates.

We remark global type G_2 could also be modeled directly using standard MPST models (such as [124]). However, in such models the projection of G_2 onto, for instance, participant `sched` would be undefined (cf. [123]). Since we follow the approach of [123] with explicit connections, projection of G_2 onto `sched` is indeed defined as $\mathsf{S}_{\texttt{sched}}$.

3.6.4 List detail protocol

The last communication protocol in our system appears in the information retrieval process. Namely, on formed topologies, using labels, the user can specify what part of the system he wants to retrieve, for example to visualise on some dashboard. Two options are available:

(**1**) **global view** of the system — all topologies the user manages

(**2**) **specific clusters** details — complete details about specified clusters like resources utilization over time (using stored metrics information), node information, and running or stopped services.

It is important to note, that similar to the query operation, both rules (3.1) and (3.2) must be satisfied in order for information to be present in the response.

One additional information may be specified is whether the user wants a detailed view or not. If detail view information is presented in a request, the user will get a detailed view.

Figure 3.5. shows a low-level view of the list operation protocol, where users can get details about the formed system. In this setting, the system involves participants: User, State, Nodes, and Log.

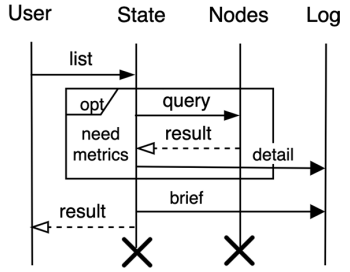

Figure 3.5: Low level view of list operation communication.

We now informally describe the participants roles in the protocol:

(**1**) **User** sends a list request to State service;

(**2**) **State** accepts the list request and the query local state based on the user selector. If a detail view is required, the state gets metrics data from Nodes service;

(**3**) **Nodes** contain node metrics data, and if required, it may send this data to State;

(**4**) **Log** contains records of all operations. Users can query this service.

Algorithm 5 describes steps after the state receives a list message.

Algorithm 5: List of current state of the system

```
   input: request
1  Initialize: data ← []
2  foreach (topology, isDetail) ∈ userData(request.query) do
3  |  if isDetail then
4  |  |  data.append(topology.collectData())
5  |  else
6  |  |  data.append(topology.data())
7  |  end
8  end
9  return data
```

Next we present a formal description of the list communication protocol (cf. Figure 1.4) by using [123]. Global type G_3 (given below) starts with `user` connecting `state` with one of the two possible messages: (1) *list*, specifying a request for a detailed view, where sort T_1 identifies which parts of the system user wants to view in details, after which `state` connects `nodes` with *query* message, with a payload of sort T_2 containing specification of which nodes need to show their metrics data, and then protocol proceeds as prescribed with G'; (2) *list**, specifies no need for a detailed view is specified, where a payload of sort T_4 denotes user specified parts of the system the user wants to view, but without greater details. In the latter case protocol follows global type G''.

$$\mathsf{G}_3 = \begin{cases} \texttt{user} \twoheadrightarrow \texttt{state}{:}\mathit{list}(\mathsf{T}_1).\texttt{state} \twoheadrightarrow \texttt{nodes}{:}\mathit{query}(\mathsf{T}_2).\mathsf{G}' \\ \texttt{user} \twoheadrightarrow \texttt{state}{:}\mathit{list}^*(\mathsf{T}_4).\mathsf{G}'' \end{cases}$$

Global type G' starts with `nodes` replying to `state` *result* message and a payload identifying parts of the system user wants to see in greater detail typed with T_3. Then, `state` connects `log` with `details`

and also sends `result` to `user`, and finally terminates. In G'', `state` also connects `log` with *brief* and a payload typed with T_5 identifying parts of the system user wants to see without greater detail. Then, `state` replies to `user` with `result` message, and the protocol terminates.

$$\begin{aligned}\mathsf{G}' =& \texttt{nodes} \to \texttt{state}{:}\mathit{result}(\mathsf{T}_3).\texttt{state} \twoheadrightarrow \texttt{log}{:}\mathit{detail}(\mathsf{T}_3).\\ & \texttt{state} \to \texttt{user}{:}\mathit{result}(\mathsf{T}_3).\texttt{end}\\ \mathsf{G}'' =& \texttt{state} \twoheadrightarrow \texttt{log}{:}\mathit{brief}(\mathsf{T}_5).\texttt{state} \to \texttt{user}{:}\mathit{result}(\mathsf{T}_5).\texttt{end}\end{aligned}$$

Same as for the health-check and the cluster formation protocols, here we also present the projections of global type G_3, modeling the list protocol, onto participants `user`, `state`, `nodes`, and `log`:

$$\begin{aligned}\mathsf{S}_{\texttt{user}} =& + \begin{cases}\texttt{state}!!\mathit{list}(\mathsf{T}_1).\texttt{state}?\mathit{result}(\mathsf{T}_3).\texttt{end}\\ \texttt{state}!!\mathit{list}^*(\mathsf{T}_4).\texttt{state}?\mathit{result}(\mathsf{T}_5).\texttt{end}\end{cases}\\ \mathsf{S}_{\texttt{state}} =& + \begin{cases}\texttt{user}??\mathit{list}(\mathsf{T}_1).\texttt{nodes}!!\mathit{query}(\mathsf{T}_2).\mathsf{S}'\\ \texttt{user}??\mathit{list}^*(\mathsf{T}_4).\texttt{log}!!\mathit{brief}(\mathsf{T}_5).\texttt{user}!\mathit{result}(\mathsf{T}_5).\texttt{end}\end{cases}\end{aligned}$$

where

$$\begin{aligned}\mathsf{S}' =& \texttt{nodes}?\mathit{result}(\mathsf{T}_3).\texttt{log}!!\mathit{detail}(\mathsf{T}_3).\texttt{user}!\mathit{result}(\mathsf{T}_3).\texttt{end}\\ \mathsf{S}_{\texttt{nodes}} =& \texttt{state}??\mathit{query}(\mathsf{T}_2).\texttt{state}!\mathit{result}(\mathsf{T}_3).\texttt{end}\\ \mathsf{S}_{\texttt{log}} =& + \begin{cases}\texttt{state}??\mathit{detail}(\mathsf{T}_3).\texttt{end}\\ \texttt{state}??\mathit{brief}(\mathsf{T}_5).\texttt{end}\end{cases}\end{aligned}$$

For instance, type $\mathsf{S}_{\texttt{log}}$ specifies `log` gets included in the session only after receiving from `state`, either message *detail*, or message *brief*, and then terminates.

Similarly as for G_2, we remark G_3 could also be modeled using standard MPST (e.g., [124]), but again the projection types would be undefined, while following the approach of [123] with explicit connections, we have obtained all valid projections.

3.6.5 Idempotency

Mutate operation should be atomic, immutable and idempotent. User can specify the same topology details, but in different order. We must ensure that new cluster formation protocol should **not** be initiated, if user change order of regions, clusters, nodes or labels in one or more node/s. If mutate operation fails, for whatever possible reason but infrastructure is created, or same infrastructure already exists, user can get message that infrastructure is formed. If user change the number of labels per node, nodes per cluster, clusters per region **only** at that case we should start new protocol.

But since we have different scenario then standard write to storage, and we do not specify steps how operations should be done, to implement idempotance correctly we have to do it a little bit differently. First of all, we must ensure that structure and operation that we are going to do over that structure are idempotent.

Idempotent structure can be represented as a tuple of topology name and set of data like $S = (Name, Data)$. $Name$ could be used for faster lookup, while $Data$ can be represebted as a set, because most of the set operations are idempotent, as described in SEC. $Data$ set could be represented in the two ways:

(**1**) **flat keyspace**, with this opetion all data could be part of the same set, and distinguishment could be achieved usnig *prefix* identity for example: region1_cluster1, cluster1_node1, node1_label1 etc.

(**2**) **hierarhical keyspace**, with this option we can create nested data-structures of elements for example set of regions, where every region is set of clusters, where every cluster is set of nodes etc. We can go deep as long as we want, but restriction is that every structure **must** be idempotent. So we can use set of sets of sets and so on.

If we have cached idempotency data for user request, and if he tries to send same request we can test idempotency using set operation **intersection**.

Definition 3.6.4. *Intersection of two sets x and y $x \cap y$ is an idempotent operation, becasue $x \cap x$ is always equal to x. This means that the idempotency law 1.3 $\forall x,\ x \cap x = x$ is always true.*

If we have stored user request on cluster formation protocol, and we receive new request with the same name, **than** we can take intersection of two sets. If we get the same set, that means that this action is already done because of the definition 3.6.4. Otherwise request represents the new action, and new cluster formation protocol.

We can make this test a litle bit faster. If we first test does the same topology name is already present in idempotency store, that lookup will spare us of unnecessary comperiosson on sets. We can go little bit further, and use CRDTs and SEC to store and replicate deta on copies of the services that are required for idempotency test.

Figure 3.6 show zoomed view in the *State* participent from figure 3.4, and idempotency check communication.

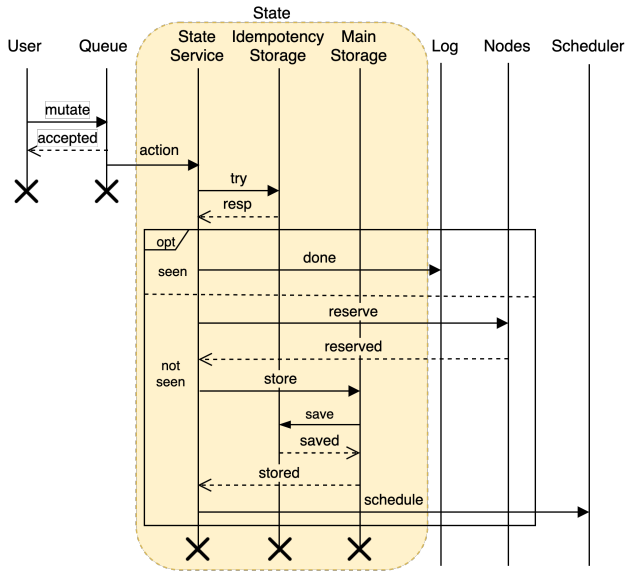

Figure 3.6: Low level view of idempotency check communication.

The participants follow the communication that we now describe informally:

(1) **User** sends a list request to State service (same as 3.6.3);

(2) **Queue** accepts the list request and the query local state based on the user selector. If a detail view is required, the state gets metrics data from Nodes service (same as 3.6.3);

(3) **State service** service that is wrapper aroud system main storage. Interacts with main storage in oreder to create new topologies, regions or clusters, or to get data from main storages about same entities.

(4) **Idempotency storage** contains idempotent set for all **already** formed topologies, regions and clusters.

(5) **Main storage** contains records about desired state for all formed topologies, regions and clusters.

(**6**) **Log** contains records of operations. Users can query this service to see if their tasks are finished or have any problems (same as 3.6.3);

(**7**) **Nodes** accept messages from State. If possible, it will reserve desired nodes, otherwise it will send an error message to Log service. On a health-check message, if a node is used in some cluster, it will inform that the node is alive (same as 3.6.3);

(**8**) **Scheduler** waits for a message sent from State, and pushes cluster formation messages to desired nodes (same as 3.6.3);

When testing is request idempotent we must have in mind that stored structure could be (1) flat keyspace and (2) hierarhical keyspace both options are valid as log as structure is idempotent. Algorithm that will test structure idempotency must be able to test both options.

Algorithm 6 describe steps requred to test if operation is done before.

Algorithm 6: Mutate idempotency check

```
1  function idempotent(stored, topology):
2      foreach data ∈ topology do
3          if isNotSet(data) then
4              if stored.intersection(data) = stored then
5                  return true
6              return false
7          else
8              idempotent(stored, data)
   input: request
9  id ← seenBefore(request.payload.name)
10 if id = null then
11     return true;
12 else
13     stored ← storage.findRequest(id)
14     topology ← request.payload.topology
15     return idempotent(stored, topology)
16 end
```

Next we present a formal description of the idempotency check protocol (cf. Figure 3.6) by using [123].

$$\mathsf{G}_4 = \begin{cases} \mathtt{user} \to \mathtt{state}{:}\mathit{quit}().\mathtt{end} \\ \mathtt{user} \to \mathtt{state}{:}\mathit{create}(\mathsf{T}_3).\mathtt{state} \to \mathtt{user}{:}\mathit{ok}().\mathsf{G}' \end{cases}$$

$$G' = \mathtt{service} \to \mathtt{istorage}{:}\mathit{try}(\mathsf{T}_3).\mathtt{istorage} \to \mathtt{service}{:}\mathit{resp}().$$
$$\begin{cases} \mathtt{state} \to \mathtt{log}{:}\mathit{done}(\mathsf{String}).\mathtt{end} \\ \mathtt{state} \to \mathtt{nodes}{:}\mathit{ids}(\mathsf{T}_4).\mathtt{nodes} \to \mathtt{state}{:}\mathit{rsrvd}().\mathsf{G}'' \end{cases}$$

$$\begin{aligned}\mathsf{G}'' = &\texttt{service} \rightarrow \texttt{mstorage}{:}\mathit{store}(\mathsf{T}_3).\texttt{mstorage} \rightarrow \texttt{istorage}{:}\mathit{save}(\mathsf{T}_3).\\ &\texttt{istorage} \rightarrow \texttt{mstorage}{:}\mathit{saved}().\texttt{mstorage} \rightarrow \texttt{service}{:}\mathit{stored}().\\ &\texttt{service} \rightarrow \texttt{sched}{:}\mathit{ids}(\mathsf{T}_5).\texttt{end}\end{aligned}$$

3.7 Repercussion

Model presented in this chapter, have two possible repercussions:

(**1**) **Stand alone**, the proposed model can serve as a base layer for future ECC as a service implementation. On top of it, we can implement other services and features like scheduling, storage, applications, management, monitoring, etc. As such it could be valiable option in the CC.

(**2**) **Integration**, the proposed model could be integrated with existing systems like Kubernetes, OpenShift, or cloud provider infrastructure since they all operate over the cluster. This is possible, with some small infrastructure changes and adaptations because — the communication should implemented via standard interfaces like HTTP and JSON, the integrations should be relatively easy to achieve. The proposed model could be used as a geo-distributed description and/or an organization tool.

3.8 Limitations

Since the perfect model never existed, model that is proposed in this thesis have some limitations, that we must be aware of. Either to work on improvements, or use the model as is. When talking about small scale servers and micro-clouds, we must be aware of the few things.

(**1**) not all companies and organizations will be able to deploy them, due to the high investments required [6]. Similar to the cloud where massive DCs are built by a few companies and used by many others. These small servers could be deployed by government authorities or large cloud companies for their own needs. The general public can use them, similarly to the cloud — pay as you go, model.

(**2**) there is no guarantiee that existing public cloud providers will allow nodes that are not built, resigned or deployed by them. If we are building private cloud, thant we can make different decision. One way to resolve this issue is that whole platform become open-source, so that public cloud providers can engage into development, and eventually use tham as a solution.

(**3**) These small scale servers must be out of reach of people, and protected in some way so that not everyone have access to them. Some degree of phisical security mut be implemented.

(**4**) places where these small scale servers will be deployd must have stable internet connection, and ability to integrate SDN or other similar techologies, so that complex network topologies could be implemented properly.

(**5**) these servers can have some open architecture or could be custom built by other providers. In both cases they must be able to satisfy rules that are presented in 3.2.

Chapter 4

Implementation

In this chapter, we are going to give more details about framework implemented based on formal model and architecture specification given i previous chapter. Section 4.1 framework architecture, andd system implementation details. Section 4.3 describe few possible senarios and applications that could utilize micro-clouds platform. In Section 4.2 we present results of our experiments.

4.1 Framework

We have implemented a proof of concept system based on the proposed, model using the microservice architecture with services that have distinct role and purpose to the entire system. These services are:

- **Gateway**, this purpose is to export services feature to the rest of the world. Gateway is designed as a REST service, accepting *JSON* style messages, so that various clients can communicate to the system. When request arrive at gateway, if request is valid it will pass request to the rest of the system. It communicate to rest of the services to check if user exists, does he have proper

rights for actions that he send, and if not return proper message and do not propagete it to the rest of the system.

- **Authentification & Authorization**, sole purpose of this service is to store users and their credentials. This servic will validate idoes user exists in the system, and does he have certen rights to performe some specific operation. Users that often use the system will be stored in the cache layer of the sevice, so that on next request his actions are done faster. Users that not use the system that often will not be stored in the cache, until first use. After that, if user do not use system in some time, he will expire from the cache.

- **Queues**, purpose of this service is to prevent huge request load to the system, and to accept more user requests. When user submit any **mutation** operation — operation that change state of the system, thse opreations will be put in the queue. User can create their own queues, to prevent long lines for specific tasks. For example, user can crete queues for specific tasks, and use them only for those tasks, while other queues could be general purpos queues. On system start, evey user will start with one queue — **default**. When doing mutations on different parts of the system, user can specify in matadata which queue he wnats task to go to.

- **Nodes**, this service stores and maintain informations about registered nodes in the system. All node hardware and software details will be stored in this here. This service is also responsabile for storeing metrics data, accept health-check requests from nodes, and inform rest of the system that used node is alive.

- **State**, is heart of the system. This service store all informations about architecture, clusters, regions and topologies. When new cluster/region/topology is creted, this service will setup watchers for nodes, so that if node is not send healtch-check signal for some

time, that node will be declared dead. This is important, so that at any moment we must know state of the clusters and their utilization. This service as well will cache frequently used nodes data, so that on next request node lookup is faster since we can have huge number of nodes, topologies clusters and informations about them. To prevent data loss, this service will first store a copy of operation before attepting any mutation of the system.

- **Log**, is responsabile for storeing all log an trace data from every service. Here user can check are all jobs done, or is there some error and possible why error happend to resolve it, or fix for the next time. From user point of view, this is **read only** service and from sysem view this is **write only** service.

- **Command push**, purpose of this service is to push commands and operations to the nodes user desired. This service implement token bucket rate limiting algorithm to prevent constant data push to the nodes. As State service, this service will store a copy of operation information localy before attempting any push to the nodes. This information will be deleted, once operation reach all desided nodes, and all of them response with ACK message.

All services, are higly customizable and all have their own configuratoin file that could be changed and adopted. All these services are easy to extend to support new operations and informations about nodes, regions, clusters, topologies and latter on applications as well.

The system operates with four commands, where three of them follow formal models described in previous chapter. Last command is simple command to list logs for every user.

In the rest of this section we will see details about all operations, as well as used technologies to implement whole system. We will describe possible applicatinos and give future directions for applicatoin development. System architecture is shown in Figure 4.1.

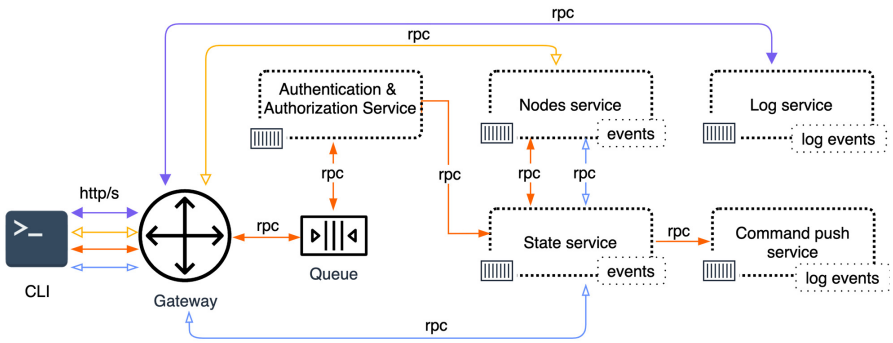

Figure 4.1: Proof of concept implemented system.

4.1.1 Technologies

All services are implemented using the Go programming language, because his well-known tooling, support for developing system software, web based applications, small binaries but also great concurency model and ability to build binaries for almost any architecture without any changes in code. All services rely on go inplicit *interface* implementation mechanism. Every technology or component used in the system can be swaped for some other, as long as that component implement *interface* fully. Framework is developed in such a way that is relativly easy to extend, or switch and use different components and technologies.

As a main storage layer for our system, we used etcd, a popular open-source key-value database, that show good performance, and it is mostly used for configiration data. Metrics data are stored in the open-source time-series database InfluxDB.

Communication between microservices is implemented in RPC manner using gRPC, and Protobuf as a message definition. gRPC and Protobuf are open-source tools designed by Google to be scalable, interoperable, and available for general purposes. Communication between nodes and the system is carried out using NATS, an open-source messaging system. Health checking and action push to nodes are implemented over NATS in a publish-subscribe manner.

Chacheing layer for every service is implemented using Redis key-value, in memory database. It is important to notice, that all concrete tools that are used, are easy swapable for some other as long as they implement proper interface.

All communication with the outside world, is done in REST manner using JSON encoded messages over HTTP. To communicate with the platform, we have developed a simple command-line interface (CLI) application also using the Go programming language that sends JSON encoded messages over HTTP to the system.

Every service is packed in a container, and for this purpose Docker containers are used. To achieve automatic orchestration, and self-heal and up-time, all services that are packed in containres, are runnin inside Kubernetes.

Every service will log details abot his usage and calls, as well as traces how requests are going. Log data is stored outside the service and container, and informations will be sent to centralized log storage on every t seconds specified by user. Sending interval could be changed and addopted using configuration file for every service independently.

Log data will be stored in the two levels:

(**1**) **system level**, this data is generated by the system, and could be viewd by administrators of the system **only**. Non operations people in the team or devops, but by developers of the system and providers of the system.

(**2**) **user level** that stores informations about user requests that **only** users can see. This type of data will not be visible to the developers of the system, and only users that created these logs will be able to see them.

Log storage could be searched to see general state of the sytem, and informations about user requests and state of their requests.

4.1.2 Node daemon

Every node runs a simple daemon program implemented using the Go programming language, as an actor system (Ref. secion 1.7.1). Actor system is developed for this purpose. When a message arrives, the proper actor will react based on the message type, or discard if the type is not supported.

Extending such system is reather easy, because users need to simply write new actor and logic that goes with him and register it to the system.

When daemon start, he will first read configuration file to do proper setup, and then will contact actor system to start all the actors.

System messages will be send to the daemon, but he will not react to these messages. Daemon is not able to communicate with any actor dyrectly. All communication goes trough actor system who is responsabile to pass messages to the actors. Actor system at this point have only tree existing actors:

(1) **cluster actor**, this actor react on messages about new cluster formation, but he is also responsabile to contact rest of the system that message has arrived.

(2) **internal actor**, this actor react on messages from other actors in order to update daemon state. For example on new cluster creation, this actor will update daemon id, name, labels etc.

(3) **health actor**, this actor will periodicaly sent healtch-check data to the sytem about node state, utilization etc. This actor will communicate to the rest of the hardware to get proper informations, to collect logs from node and send all that data to the system.

The actor system will monitor these actors, so in order that any of them crush for whatever reason, actor system will restart them. Listing 4.1 show actor system hierarchy of existing actors.

```
\StarSystem
  +--\TopologyActor
  +--\HealthcheckActor
  +--\InternalActor
```

Listing 4.1: Actor system hierarchy.

Before daemon starts, the user needs to specify some parameters for proper configuration like:

(1) identifier

(2) name

(3) labels

(4) health-check details

(5) system informations

Configuration can be done easly using YAML configuration file. Listing 4.2 show simple YAML configuration file for deamon process.

```
star:
  version: v1
  nodeid: node1
  name: noname
  flusher: address
  healthcheck:
    address: address
    topic: health
    interval: 1m
  labels:
    os: linux
    disk: flash
    arch: arm
    model: rpi
```

```
15      memory: 4GB
16      storage: 120GB
17      cpu: 1
18      cores: 4
```

Listing 4.2: Daemon configiration file

Based on the configuration file, the daemon will start a background health-check mechanism, and it will subscribe to the system, using an identifier as a subscription topic. The background thread will contact the system repeatedly using a contact interval time, specified in the configuration file.

On every health-check, the node will send labels, name, id, and metrics to the system (e.g., CPU utilization, total, used, free ram or disk, etc.). The specified labels will be used when the user is querying for available nodes, while node id will be used for reservation when forming a cluster.

On first start of the deamon when node is free, user can specify whathever node id he wants. Once, node is a part of the cluster, node id will be updated to match that. Node id update will happend on cluster formation process.

4.1.3 Mutate

Mutate operation (orange arrows in Figure 4.1.) change the system state by creating, editing, or deleting clusters, regions, and topologies. When a user wants to perform a mutation over the system, a desired state needs to be specified using a YAML file.

The users specify which nodes are forming the cluster. Optionally, users can also specify labels and names on the node level, and retention period on the cluster level. The retention period is used to describe how long metrics are going to be kept. When the retention period expires, the metrics data will be deleted or moved to another location. Users can target a specific system queue, by adding a metadata part in the configuration file. With this ability, users can have specific queues

just for the mutation to avoid long waiting times if other queues are full. When forming a topology, users can assign a name and labels to the entire topology. These parameters will be used when the user wants to query all topologies to get full information about regions, clusters, and nodes inside a topology

```
constellations:
  version: v1
  kind: Topology
  metadata:
    taskName: default
    queue: default
  payload:
    name: MyTopology
    selector:
      labels:
        l1: v1
        l2: v2
        l3: v3
    topology:
      region1:
        cluster1:
          retention:
            period: 24h
          machineid1:
            name: node1
          machineid2:
            name: node2
          machineid3:
            name: node3
            selector:
              labels:
                os: linux
                arch: arm
```

```
29                    model: rpi
30                    storage: 100GB
31                    memory: 1GB
```

Listing 4.3: Example of mutate file using YAML.

4.1.4 List

This command show the current state of the system for the registered user (blue arrows in Figure 4.1.). Using labels, the user can specify what part of the system he wants to see. He can get a global view of the system - all topologies he manages, or details about a single topology (i.e., regions, clusters, and nodes in a single topology). Users can specify a more detailed view of a single cluster, meaning the users will get information about what resources the cluster has, but also resource utilization over time (using stored metrics information);

4.1.5 Query

This operation showing all or filtered free nodes registered to the system (yellow arrows in Figure 4.1.). When a user wants to get information about free nodes, they need to submit a selector value which is composed of multiple key-value pairs. The selector will be used to compare labels of every free node existing in the system. The nodes satisfying the rules 3.3 defined in Section 3.6.3 will be in the result.

Querying process is little bit changed from one presented in Section 3.6.3, and the only change is addition of the *Gateway* service that will pass requests into the system and to prevent overflow of requests. This change **does not** affect or valiate formal model presented before.

Figure 4.2 show communication diagram for query action.

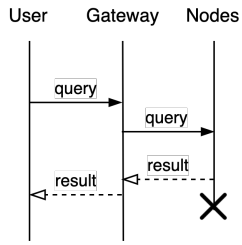

Figure 4.2: Low level communication protocol diagram of query operation.

4.1.6 Logs

This operation showing stored logs and traces to the user (purple arrows in Figure 4.1.). Here, the user can see the state of his operations and actions.

4.2 Results

In our tests, we have used:

- 9 Raspberry Pi 3+ Model B with 1GB LPDDR2 SDRAM, 16GB SDCard storage, and 1.4GHz Cortex-A53 64-bit SoC running Raspbian Linux, a Debian-based operating system.
- 3 Beagle bone black devices with 512MB DDR3 RAM, 4GB 8-bit eMMC on-board flash storage, and 1GHz ARM Cortex-A8 running a version of Linux Debian operating system.

as test heterogenious nodes for creating clusters, regions, and topologies.

Using Go tooling we were able to build daemon problam without changes and dissiminate on all nodes.

We have run tests on different configurations and different nodes clusters using these nodes. All nodes were connected on the WIFI network, and experiments were conducted in a controlled environment.

Experimental research was realized in the laboratory of the Department of Informatics at the Faculty of Technical Sciences in Novi Sad. The laboratory of the Department of Informatics is equipped with adequate computer, communication and software equipment on which the set goals in this thesis can be fully realized.

4.3 Applications

This research focus on a platform with geo-distributed edge nodes can be organized dynamically into the micro data-centers and regions based on the cloud model, but adapted for a different environment. This middle layer helps the power-hungry servers reduce traffic by serving the nearby population requests and possibly syncing their data without expensive consensus protocols [55]. These micro data-centers or micro clouds also increase the availability and reliability of the cloud services, and as such could be offered as a service to users.

With clear separation of concerns and a familiar application model for the users it opens possibilities for new human-centered applications, for example, area traffic control.

If we imagine a scenario where a new catastrophic event (earthquake, tsunami, war, terrorist attack, pandemic, etc.) is in the human population, an increasing amount of ambulance vehicles must be routed to hospitals fast, in the city area for example. The traffic control system suddenly needs more resources to continue operating properly. On top of that, if the healthcare system is like the one presented in [56, 57], we can monitor patient health in real-time [58] and transfer data to the healthcare platform.

This gives health workers crucial information about patients on their arrival. Resource vise, this scenario is relatively easy to solve if using the cloud, as we increase resource demand. The main advance of

EC, when compared to the cloud only approach, is the acceleration of the communication speed. In our scenario, the cloud could bring huge latency, while EC originates from peer to peer systems [11], serving only local population needs, minimizing potentially huge round-trip time of the cloud. Furthermore, our model expands peer to peer systems into new directions and blends them with the cloud to allow novel human-centered, cloud-like applications.

For applications like self-driving cars, delivery drones, or power balancing in electric grids require real-time processing for proper decision making, or any other application that future developers may develop.

Users are getting a new platform as a blank canvas, and there is no limitation in what applications they can develop. Integrating hardware and/or software even more, connecting sensors and things around us and eventually an operating system that will be capable of running city/state infrastructure without human intervention.

Chapter 5

Conclusion

In Section 5.1 we give summary of contributions for this thesis, while in Section 5.2 we present future work.

5.1 Summary of contributions

5.2 Future work

Bibliography

[1] M. Chiang, T. Zhang, Fog and iot: An overview of research opportunities, IEEE Internet Things J. 3 (6) (2016) 854–864. doi:10.1109/JIOT.2016.2584538.
URL https://doi.org/10.1109/JIOT.2016.2584538

[2] M. F. Bari, R. Boutaba, R. P. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, M. F. Zhani, Data center network virtualization: A survey, IEEE Commun. Surv. Tutorials 15 (2) (2013) 909–928. doi:10.1109/SURV.2012.090512.00043.
URL https://doi.org/10.1109/SURV.2012.090512.00043

[3] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, K. J. Eliazar, Why does the cloud stop computing? lessons from hundreds of service outages, in: M. K. Aguilera, B. Cooper, Y. Diao (Eds.), Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016, ACM, 2016, pp. 1–16. doi:10.1145/2987550.2987583.
URL https://doi.org/10.1145/2987550.2987583

[4] P. G. López, A. Montresor, D. H. J. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. P. Barcellos, P. Felber, E. Rivière, Edge-centric computing: Vision and challenges, Comput. Commun. Rev. 45 (5) (2015) 37–42. doi:10.1145/2831347.

2831354.
URL https://doi.org/10.1145/2831347.2831354

[5] M. B. A. Karim, B. I. Ismail, M. Wong, E. M. Goortani, S. Setapa, L. J. Yuan, H. Ong, Extending cloud resources to the edge: Possible scenarios, challenges, and experiments, in: International Conference on Cloud Computing Research and Innovations, ICCCRI 2016, Singapore, Singapore, May 4-5, 2016, IEEE Computer Society, 2016, pp. 78–85. doi:10.1109/ICCCRI.2016.20.
URL https://doi.org/10.1109/ICCCRI.2016.20

[6] S. A. Monsalve, F. G. Carballeira, A. Calderón, A heterogeneous mobile cloud computing model for hybrid clouds, Future Gener. Comput. Syst. 87 (2018) 651–666. doi:10.1016/j.future.2018.04.005.
URL https://doi.org/10.1016/j.future.2018.04.005

[7] M. Satyanarayanan, The emergence of edge computing, Computer 50 (1) (2017) 30–39. doi:10.1109/MC.2017.9.
URL https://doi.org/10.1109/MC.2017.9

[8] S. K. A. Hossain, M. A. Rahman, M. A. Hossain, Edge computing framework for enabling situation awareness in iot based smart city, J. Parallel Distributed Comput. 122 (2018) 226–237. doi:10.1016/j.jpdc.2018.08.009.
URL https://doi.org/10.1016/j.jpdc.2018.08.009

[9] J. Cao, Q. Zhang, W. Shi, Edge Computing: A Primer, Springer Briefs in Computer Science, Springer, 2018. doi:10.1007/978-3-030-02083-5.
URL https://doi.org/10.1007/978-3-030-02083-5

[10] H. Ning, Y. Li, F. Shi, L. T. Yang, Heterogeneous edge computing open platforms and tools for internet of things, Future Gener. Comput. Syst. 106 (2020) 67–76. doi:10.1016/

j.future.2019.12.036.
URL https://doi.org/10.1016/j.future.2019.12.036

[11] M. van Steen, A. S. Tanenbaum, A brief introduction to distributed systems, Computing 98 (10) (2016) 967–1009. doi:10.1007/s00607-016-0508-7.
URL https://doi.org/10.1007/s00607-016-0508-7

[12] A. S. Tanenbaum, M. van Steen, Distributed systems - principles and paradigms, 2nd Edition, Pearson Education, 2007.

[13] A. B. Bondi, Characteristics of scalability and their impact on performance, in: Second International Workshop on Software and Performance, WOSP 2000, Ottawa, Canada, September 17-20, 2000, ACM, 2000, pp. 195–203. doi:10.1145/350391.350432.
URL https://doi.org/10.1145/350391.350432

[14] J. L. Gustafson, Moore's Law, Springer US, Boston, MA, 2011, pp. 1177–1184. doi:10.1007/978-0-387-09766-4_81.
URL https://doi.org/10.1007/978-0-387-09766-4_81

[15] E. A. Brewer, Towards robust distributed systems., in: Symposium on Principles of Distributed Computing (PODC), 2000.
URL http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[16] S. Gilbert, N. A. Lynch, Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, SIGACT News 33 (2) (2002) 51–59. doi:10.1145/564585.564601.
URL https://doi.org/10.1145/564585.564601

[17] J. Gray, D. P. Siewiorek, High-availability computer systems, Computer 24 (9) (1991) 39–48. doi:10.1109/2.84898.
URL https://doi.org/10.1109/2.84898

[18] M. Shapiro, N. M. Preguiça, C. Baquero, M. Zawirski, Conflict-free replicated data types, in: X. Défago, F. Petit, V. Villain (Eds.), Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings, Vol. 6976 of Lecture Notes in Computer Science, Springer, 2011, pp. 386–400. doi:10.1007/978-3-642-24550-3_29.
URL https://doi.org/10.1007/978-3-642-24550-3_29

[19] W. Vogels, A head in the clouds the power of infrastructure as a service, in: Proceedings of the 1st Workshop on Cloud Computing and Applications, 2008.

[20] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia, Above the clouds: A berkeley view of cloud computing, Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley (Feb 2009).
URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html

[21] Q. Zhang, L. Cheng, R. Boutaba, Cloud computing: state-of-the-art and research challenges, J. Internet Serv. Appl. 1 (1) (2010) 7–18. doi:10.1007/s13174-010-0007-6.
URL https://doi.org/10.1007/s13174-010-0007-6

[22] M. D. de Assunção, A. D. S. Veith, R. Buyya, Distributed data stream processing and edge computing: A survey on resource elasticity and future directions, J. Netw. Comput. Appl. 103 (2018) 1–17. doi:10.1016/j.jnca.2017.12.001.
URL https://doi.org/10.1016/j.jnca.2017.12.001

[23] P. M. Mell, T. Grance, Sp 800-145. the nist definition of cloud computing, Tech. rep., Gaithersburg, MD, USA (2011).

[24] D. Cohen, T. Talpey, A. Kanevsky, U. Cummings, M. Krause, R. Recio, D. Crupnicoff, L. Dickman, P. Grun, Remote direct memory access over the converged enhanced ethernet fabric: Evaluating the options, in: K. Bergman, R. Brightwell, F. Petrini, H. Bubba (Eds.), 17th IEEE Symposium on High Performance Interconnects, HOTI 2009, New York, New York, USA, August 25-27, 2009, IEEE Computer Society, 2009, pp. 123–130. doi:10.1109/HOTI.2009.23.
URL https://doi.org/10.1109/HOTI.2009.23

[25] Y. Duan, G. Fu, N. Zhou, X. Sun, N. C. Narendra, B. Hu, Everything as a service (xaas) on the cloud: Origins, current and future trends, in: C. Pu, A. Mohindra (Eds.), 8th IEEE International Conference on Cloud Computing, CLOUD 2015, New York City, NY, USA, June 27 - July 2, 2015, IEEE Computer Society, 2015, pp. 621–628. doi:10.1109/CLOUD.2015.88.
URL https://doi.org/10.1109/CLOUD.2015.88

[26] R. Schollmeier, A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications, in: R. L. Graham, N. Shahmehri (Eds.), 1st International Conference on Peer-to-Peer Computing (P2P 2001), 27-29 August 2001, Linköping, Sweden, IEEE Computer Society, 2001, pp. 101–102. doi:10.1109/P2P.2001.990434.
URL https://doi.org/10.1109/P2P.2001.990434

[27] H. M. N. D. Bandara, A. P. Jayasumana, Collaborative applications over peer-to-peer systems-challenges and solutions, Peer Peer Netw. Appl. 6 (3) (2013) 257–276. doi:10.1007/s12083-012-0157-3.
URL https://doi.org/10.1007/s12083-012-0157-3

[28] M. Kamel, C. M. Scoglio, T. Easton, Optimal topology design for overlay networks, in: I. F. Akyildiz, R. Sivakumar, E. Ekici, J. C. de Oliveira, J. McNair (Eds.), NETWORKING 2007. Ad

Hoc and Sensor Networks, Wireless Networks, Next Generation Internet, 6th International IFIP-TC6 Networking Conference, Atlanta, GA, USA, May 14-18, 2007, Proceedings, Vol. 4479 of Lecture Notes in Computer Science, Springer, 2007, pp. 714–725. doi:10.1007/978-3-540-72606-7_61.
URL https://doi.org/10.1007/978-3-540-72606-7_61

[29] I. Filali, F. Bongiovanni, F. Huet, F. Baude, A survey of structured P2P systems for RDF data storage and retrieval, Trans. Large Scale Data Knowl. Centered Syst. 3 (2011) 20–55. doi:10.1007/978-3-642-23074-5_2.
URL https://doi.org/10.1007/978-3-642-23074-5_2

[30] A. Das, I. Gupta, A. Motivala, SWIM: scalable weakly-consistent infection-style process group membership protocol, in: 2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings, IEEE Computer Society, 2002, pp. 303–312. doi:10.1109/DSN.2002.1028914.
URL https://doi.org/10.1109/DSN.2002.1028914

[31] I. Stoica, R. T. Morris, D. R. Karger, M. F. Kaashoek, H. Balakrishnan, Chord: A scalable peer-to-peer lookup service for internet applications, in: R. L. Cruz, G. Varghese (Eds.), Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 27-31, 2001, San Diego, CA, USA, ACM, 2001, pp. 149–160. doi:10.1145/383059.383071.
URL https://doi.org/10.1145/383059.383071

[32] N. Leavitt, Will nosql databases live up to their promise?, Computer 43 (2) (2010) 12–14. doi:10.1109/MC.2010.58.
URL https://doi.org/10.1109/MC.2010.58

[33] Q. H. Vu, M. Lupu, B. C. Ooi, Peer-to-Peer Computing - Principles and Applications, Springer, 2010. doi:10.1007/

978-3-642-03514-2.
URL https://doi.org/10.1007/978-3-642-03514-2

[34] N. Fernando, S. W. Loke, J. W. Rahayu, Mobile cloud computing: A survey, Future Gener. Comput. Syst. 29 (1) (2013) 84–106. doi:10.1016/j.future.2012.05.023.
URL https://doi.org/10.1016/j.future.2012.05.023

[35] L. Lin, X. Liao, H. Jin, P. Li, Computation offloading toward edge computing, Proc. IEEE 107 (8) (2019) 1584–1607. doi:10.1109/JPROC.2019.2922285.
URL https://doi.org/10.1109/JPROC.2019.2922285

[36] F. Bonomi, R. A. Milito, P. Natarajan, J. Zhu, Fog computing: A platform for internet of things and analytics, in: N. Bessis, C. Dobre (Eds.), Big Data and Internet of Things: A Roadmap for Smart Environments, Vol. 546 of Studies in Computational Intelligence, Springer, 2014, pp. 169–186. doi:10.1007/978-3-319-05029-4_7.
URL https://doi.org/10.1007/978-3-319-05029-4_7

[37] S. Wang, X. Zhang, Y. Zhang, L. Wang, J. Yang, W. Wang, A survey on mobile edge networks: Convergence of computing, caching and communications, IEEE Access 5 (2017) 6757–6779. doi:10.1109/ACCESS.2017.2685434.
URL https://doi.org/10.1109/ACCESS.2017.2685434

[38] A. Khune, S. Pasricha, Mobile network-aware middleware framework for cloud offloading: Using reinforcement learning to make reward-based decisions in smartphone applications, IEEE Consumer Electron. Mag. 8 (1) (2019) 42–48. doi:10.1109/MCE.2018.2867972.
URL https://doi.org/10.1109/MCE.2018.2867972

[39] M. Chen, Y. Hao, Y. Li, C. Lai, D. Wu, On the computation offloading at ad hoc cloudlet: architecture and service modes,

IEEE Commun. Mag. 53 (6-Supplement) (2015) 18–24. doi:10.1109/MCOM.2015.7120041.
URL https://doi.org/10.1109/MCOM.2015.7120041

[40] M. Satyanarayanan, G. Klas, M. D. Silva, S. Mangiante, The seminal role of edge-native applications, in: E. Bertino, C. K. Chang, P. Chen, E. Damiani, M. Goul, K. Oyama (Eds.), 3rd IEEE International Conference on Edge Computing, EDGE 2019, Milan, Italy, July 8-13, 2019, IEEE, 2019, pp. 33–40. doi:10.1109/EDGE.2019.00022.
URL https://doi.org/10.1109/EDGE.2019.00022

[41] R. de Vera Jr., Review of: Distributed systems: An algorithmic approach (2nd edition) by sukumar ghosh, SIGACT News 47 (4) (2016) 13–14. doi:10.1145/3023855.3023860.
URL https://doi.org/10.1145/3023855.3023860

[42] G. Andrews, , parallel, and distributed programming, Addison-Wesley, 2000.
URL http://books.google.com.br/books?id=npRQAAAAMAAJ

[43] D. Fisher, R. DeLine, M. Czerwinski, S. M. Drucker, Interactions with big data analytics, Interactions 19 (3) (2012) 50–59. doi:10.1145/2168931.2168943.
URL https://doi.org/10.1145/2168931.2168943

[44] C.-W. Tsai, C.-F. Lai, H.-C. Chao, A. V. Vasilakos, Big data analytics: a survey, Journal of Big Data 2 (1) (2015) 21. doi:10.1186/s40537-015-0030-3.
URL https://doi.org/10.1186/s40537-015-0030-3

[45] Z. Guo, G. C. Fox, M. Zhou, Investigation of data locality in mapreduce, in: 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012, Ottawa, Canada, May 13-16, 2012, IEEE Computer Society, 2012, pp.

419–426. doi:10.1109/CCGrid.2012.42.
URL https://doi.org/10.1109/CCGrid.2012.42

[46] P. G. Sarigiannidis, T. Lagkas, K. Rantos, P. Bellavista, The big data era in iot-enabled smart farming: Re-defining systems, tools, and techniques, Comput. Networks 168 (2020). doi:10.1016/j.comnet.2019.107043.
URL https://doi.org/10.1016/j.comnet.2019.107043

[47] R. Patgiri, A. Ahmed, Big data: The v's of the game changer paradigm, in: J. Chen, L. T. Yang (Eds.), 18th IEEE International Conference on High Performance Computing and Communications; 14th IEEE International Conference on Smart City; 2nd IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2016, Sydney, Australia, December 12-14, 2016, IEEE Computer Society, 2016, pp. 17–24. doi:10.1109/HPCC-SmartCity-DSS.2016.0014.
URL https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0014

[48] Y. Kumar, Lambda architecture - realtime data processing, Ph.D. thesis (01 2020). doi:10.13140/RG.2.2.19091.84004.

[49] M. Kiran, P. Murphy, I. Monga, J. Dugan, S. S. Baveja, Lambda architecture for cost-effective batch and speed big data processing, in: 2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015, IEEE Computer Society, 2015, pp. 2785–2792. doi:10.1109/BigData.2015.7364082.
URL https://doi.org/10.1109/BigData.2015.7364082

[50] T. R. Rao, P. Mitra, R. Bhatt, A. Goswami, The big data system, components, tools, and technologies: a survey, Knowl. Inf. Syst. 60 (3) (2019) 1165–1245. doi:10.1007/s10115-018-1248-0.
URL https://doi.org/10.1007/s10115-018-1248-0

[51] M. Simic, M. Stojkov, G. Sladic, B. Milosavljević, Edge computing system for large-scale distributed sensing systems, in: Edge computing system for large-scale distributed sensing systems, 2018.

[52] J. E. Marynowski, A. O. Santin, A. R. Pimentel, Method for testing the fault tolerance of mapreduce frameworks, Comput. Networks 86 (2015) 1–13. doi:10.1016/j.comnet.2015.04.009.
URL https://doi.org/10.1016/j.comnet.2015.04.009

[53] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina, Microservices: yesterday, today, and tomorrow, CoRR abs/1606.04036 (2016). arXiv:1606.04036.
URL http://arxiv.org/abs/1606.04036

[54] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, N. M. Josuttis, Microservices in practice, part 1: Reality check and service design, IEEE Softw. 34 (1) (2017) 91–98. doi:10.1109/MS.2017.24.
URL https://doi.org/10.1109/MS.2017.24

[55] S. Li, H. Zhang, Z. Jia, Z. Li, C. Zhang, J. Li, Q. Gao, J. Ge, Z. Shan, A dataflow-driven approach to identifying microservices from monolithic applications, J. Syst. Softw. 157 (2019). doi:10.1016/j.jss.2019.07.008.
URL https://doi.org/10.1016/j.jss.2019.07.008

[56] L. Krause, Microservices: Patterns and Applications: Designing Fine-Grained Services by Applying Patterns, Lucas Krause, 2015.
URL https://books.google.rs/books?id=dd5-rgEACAAJ

[57] O. Al-Debagy, P. Martinek, A comparative review of microservices and monolithic architectures, CoRR abs/1905.07997

(2019). arXiv:1905.07997.
URL http://arxiv.org/abs/1905.07997

[58] N. Kratzke, P. Quint, Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study, J. Syst. Softw. 126 (2017) 1–16. doi:10.1016/j.jss.2017.01.001.
URL https://doi.org/10.1016/j.jss.2017.01.001

[59] G. Adzic, R. Chatley, Serverless computing: economic and architectural impact, in: E. Bodden, W. Schäfer, A. van Deursen, A. Zisman (Eds.), Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017, ACM, 2017, pp. 884–889. doi:10.1145/3106237.3117767.
URL https://doi.org/10.1145/3106237.3117767

[60] W. Li, Y. Lemieux, J. Gao, Z. Zhao, Y. Han, Service mesh: Challenges, state of the art, and future research opportunities, in: 13th IEEE International Conference on Service-Oriented System Engineering, SOSE 2019, San Francisco, CA, USA, April 4-9, 2019, IEEE, 2019. doi:10.1109/SOSE.2019.00026.
URL https://doi.org/10.1109/SOSE.2019.00026

[61] P. Adamczyk, P. H. Smith, R. E. Johnson, M. Hafiz, REST and web services: In theory and in practice, in: E. Wilde, C. Pautasso (Eds.), REST: From Research to Practice, Springer, 2011, pp. 35–57. doi:10.1007/978-1-4419-8303-9_2.
URL https://doi.org/10.1007/978-1-4419-8303-9_2

[62] A. Balalaie, A. Heydarnoori, P. Jamshidi, Microservices architecture enables devops: Migration to a cloud-native architecture, IEEE Software 33 (3) (2016) 42–52. doi:10.1109/MS.2016.64.

[63] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An updated performance comparison of virtual machines and linux containers,

in: 2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, March 29-31, 2015, IEEE Computer Society, 2015, pp. 171–172. doi:10.1109/ISPASS.2015.7095802.
URL https://doi.org/10.1109/ISPASS.2015.7095802

[64] B. Burns, B. Grant, D. Oppenheimer, E. A. Brewer, J. Wilkes, Borg, omega, and kubernetes, Commun. ACM 59 (5) (2016) 50–57. doi:10.1145/2890784.
URL https://doi.org/10.1145/2890784

[65] J. Soldani, D. A. Tamburri, W. van den Heuvel, The pains and gains of microservices: A systematic grey literature review, J. Syst. Softw. 146 (2018) 215–232. doi:10.1016/j.jss.2018.09.082.
URL https://doi.org/10.1016/j.jss.2018.09.082

[66] G. Grätzer, B. Davey, R. Freese, B. Ganter, M. Greferath, P. Jipsen, H. Priestley, H. Rose, E. Schmidt, S. Schmidt, et al., General Lattice Theory: Second edition, Birkhäuser Basel, 2002.
URL https://books.google.rs/books?id=SoGLVCPuOz0C

[67] G. S. Almási, A. Gottlieb, Highly parallel computing (2. ed.), Addison-Wesley, 1994.

[68] S. Leible, S. Schlager, M. Schubotz, B. Gipp, A review on blockchain technology and blockchain projects fostering open science, Frontiers Blockchain 2 (2019) 16. doi:10.3389/fbloc.2019.00016.
URL https://doi.org/10.3389/fbloc.2019.00016

[69] S. Crosby, D. Brown, The virtualization reality, ACM Queue 4 (10) (2006) 34–41. doi:10.1145/1189276.1189289.
URL https://doi.org/10.1145/1189276.1189289

[70] S. Sharma, Y. Park, Virtualization: A review and future directions executive overview, American Journal of Information Technology 1 (2011) 1–37.

[71] E. E. Absalom, S. M. Buhari, S. B. Junaidu, Virtual machine allocation in cloud computing environment, Int. J. Cloud Appl. Comput. 3 (2) (2013) 47–60. doi:10.4018/ijcac.2013040105.
URL https://doi.org/10.4018/ijcac.2013040105

[72] C. Yang, K. Huang, W. C. Chu, F. Leu, S. Wang, Implementation of cloud iaas for virtualization with live migration, in: J. J. Park, H. R. Arabnia, C. Kim, W. Shi, J. Gil (Eds.), Grid and Pervasive Computing - 8th International Conference, GPC 2013 and Colocated Workshops, Seoul, Korea, May 9-11, 2013. Proceedings, Vol. 7861 of Lecture Notes in Computer Science, Springer, 2013, pp. 199–207. doi:10.1007/978-3-642-38027-3_21.
URL https://doi.org/10.1007/978-3-642-38027-3_21

[73] K.-T. Seo, H.-S. Hwang, I. Moon, O.-Y. Kwon, B. jun Kim, Performance comparison analysis of linux container and virtual machine for building cloud, 2014.

[74] R. Pavlicek, Unikernels: Beyond Containers to the Next Generation of Cloud, O'Reilly Media, 2016.
URL https://books.google.rs/books?id=qfDXuQEACAAJ

[75] T. Goethals, M. Sebrechts, A. Atrey, B. Volckaert, F. D. Turck, Unikernels vs containers: An in-depth benchmarking study in the context of microservice applications, in: 8th IEEE International Symposium on Cloud and Service Computing, SC2 2018, Paris, France, November 18-21, 2018, IEEE, 2018, pp. 1–8. doi:10.1109/SC2.2018.00008.
URL https://doi.org/10.1109/SC2.2018.00008

[76] R. Pavlicek, The next generation cloud: Unleashing the power of the unikernel, USENIX Association, Washington, D.C., 2015.

[77] M. Plauth, L. Feinbube, A. Polze, A performance survey of lightweight virtualization techniques, in: F. D. Paoli, S. Schulte, E. B. Johnsen (Eds.), Service-Oriented and Cloud Computing - 6th IFIP WG 2.14 European Conference, ESOCC 2017, Oslo, Norway, September 27-29, 2017, Proceedings, Vol. 10465 of Lecture Notes in Computer Science, Springer, 2017, pp. 34–48. doi:10.1007/978-3-319-67262-5_3.
URL https://doi.org/10.1007/978-3-319-67262-5_3

[78] A. Wittig, M. Wittig, Amazon Web Services in Action, Manning Publications, 2018.
URL https://books.google.rs/books?id=-LRotAEACAAJ

[79] P. Helland, Immutability changes everything, Commun. ACM 59 (1) (2016) 64–70. doi:10.1145/2844112.
URL https://doi.org/10.1145/2844112

[80] M. Perry, The Art of Immutable Architecture: Theory and Practice of Data Management in Distributed Systems, Apress, 2020.
URL https://books.google.rs/books?id=Ea9tzQEACAAJ

[81] P. C. de Guzmán, F. Gorostiaga, C. Sánchez, i2kit: A tool for immutable infrastructure deployments based on lightweight virtual machines specialized to run containers, CoRR abs/1802.10375 (2018). arXiv:1802.10375.
URL http://arxiv.org/abs/1802.10375

[82] R. Pike, Concurrency is not parallelism, waza conference (2013).
URL https://blog.golang.org/waza-talk

[83] C. A. R. Hoare, Communicating sequential processes, Commun. ACM 21 (8) (1978) 666–677. doi:10.1145/359576.359585.
URL https://doi.org/10.1145/359576.359585

[84] C. Hewitt, Actor model for discretionary, adaptive concurrency, CoRR abs/1008.1459 (2010). arXiv:1008.1459.
URL http://arxiv.org/abs/1008.1459

[85] A. G. Greenberg, J. R. Hamilton, D. A. Maltz, P. Patel, The cost of a cloud: research problems in data center networks, Comput. Commun. Rev. 39 (1) (2009) 68–73. doi:10.1145/1496091.1496103.
URL https://doi.org/10.1145/1496091.1496103

[86] M. Satyanarayanan, P. Bahl, R. Cáceres, N. Davies, The case for vm-based cloudlets in mobile computing, IEEE Pervasive Comput. 8 (4) (2009) 14–23. doi:10.1109/MPRV.2009.82.
URL https://doi.org/10.1109/MPRV.2009.82

[87] M. Ryden, K. Oh, A. Chandra, J. B. Weissman, Nebula: Distributed edge cloud for data intensive computing, in: 2014 IEEE International Conference on Cloud Engineering, Boston, MA, USA, March 11-14, 2014, IEEE Computer Society, 2014, pp. 57–66. doi:10.1109/IC2E.2014.34.
URL https://doi.org/10.1109/IC2E.2014.34

[88] M. Hirsch, C. Mateos, A. Zunino, Augmenting computing capabilities at the edge by jointly exploiting mobile devices: A survey, Future Gener. Comput. Syst. 88 (2018) 644–662. doi:10.1016/j.future.2018.06.005.
URL https://doi.org/10.1016/j.future.2018.06.005

[89] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami, Internet of things (iot): A vision, architectural elements, and future directions, Future Gener. Comput. Syst. 29 (7) (2013) 1645–1660. doi:10.1016/j.future.2013.01.010.
URL https://doi.org/10.1016/j.future.2013.01.010

[90] C. Jiang, X. Cheng, H. Gao, X. Zhou, J. Wan, Toward computation offloading in edge computing: A survey, IEEE Access 7

(2019) 131543–131558. doi:10.1109/ACCESS.2019.2938660.
URL https://doi.org/10.1109/ACCESS.2019.2938660

[91] R. V. Aroca, L. M. G. Gonçalves, Towards green data centers: A comparison of x86 and ARM architectures power efficiency, J. Parallel Distributed Comput. 72 (12) (2012) 1770–1780. doi:10.1016/j.jpdc.2012.08.005.
URL https://doi.org/10.1016/j.jpdc.2012.08.005

[92] H. Guo, L. Rui, Z. Gao, A zone-based content pre-caching strategy in vehicular edge networks, Future Gener. Comput. Syst. 106 (2020) 22–33. doi:10.1016/j.future.2019.12.050.
URL https://doi.org/10.1016/j.future.2019.12.050

[93] A. C. Baktir, A. Ozgovde, C. Ersoy, How can edge computing benefit from software-defined networking: A survey, use cases, and future directions, IEEE Commun. Surv. Tutorials 19 (4) (2017) 2359–2391. doi:10.1109/COMST.2017.2717482.
URL https://doi.org/10.1109/COMST.2017.2717482

[94] I. Kurniawan, H. Febiansyah, J. Kwon, Cost-Effective Content Delivery Networks Using Clouds and Nano Data Centers, Vol. 280, 2014, pp. 417–424. doi:10.1007/978-3-642-41671-2_53.

[95] R. Ciobanu, C. Negru, F. Pop, C. Dobre, C. X. Mavromoustakis, G. Mastorakis, Drop computing: Ad-hoc dynamic collaborative computing, Future Gener. Comput. Syst. 92 (2019) 889–899. doi:10.1016/j.future.2017.11.044.
URL https://doi.org/10.1016/j.future.2017.11.044

[96] C. Shi, K. Habak, P. Pandurangan, M. H. Ammar, M. Naik, E. W. Zegura, COSMOS: computation offloading as a service for mobile devices, in: J. Wu, X. Cheng, X. Li, S. Sarkar (Eds.), The Fifteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing, MobiHoc'14, Philadelphia, PA, USA, August 11-14, 2014, ACM, 2014, pp. 287–296.

doi:10.1145/2632951.2632958.
URL https://doi.org/10.1145/2632951.2632958

[97] N. Abbas, Y. Zhang, A. Taherkordi, T. Skeie, Mobile edge computing: A survey, IEEE Internet Things J. 5 (1) (2018) 450–465. doi:10.1109/JIOT.2017.2750180.
URL https://doi.org/10.1109/JIOT.2017.2750180

[98] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, J. Wilkes, Large-scale cluster management at google with borg, in: L. Réveillère, T. Harris, M. Herlihy (Eds.), Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015, ACM, 2015, pp. 18:1–18:17. doi:10.1145/2741948.2741964.
URL https://doi.org/10.1145/2741948.2741964

[99] F. Rossi, V. Cardellini, F. L. Presti, M. Nardelli, Geo-distributed efficient deployment of containers with kubernetes, Comput. Commun. 159 (2020) 161–174. doi:10.1016/j.comcom.2020.04.061.
URL https://doi.org/10.1016/j.comcom.2020.04.061

[100] A. Lèbre, J. Pastor, A. Simonet, F. Desprez, Revising openstack to operate fog/edge computing infrastructures, in: 2017 IEEE International Conference on Cloud Engineering, IC2E 2017, Vancouver, BC, Canada, April 4-7, 2017, IEEE Computer Society, 2017, pp. 138–148. doi:10.1109/IC2E.2017.35.
URL https://doi.org/10.1109/IC2E.2017.35

[101] Y. Shao, C. Li, Z. Fu, L. Jia, Y. Luo, Cost-effective replication management and scheduling in edge computing, J. Netw. Comput. Appl. 129 (2019) 46–61. doi:10.1016/j.jnca.2019.01.001.
URL https://doi.org/10.1016/j.jnca.2019.01.001

[102] H. Sami, A. Mourad, Dynamic on-demand fog formation offering on-the-fly iot service deployment, IEEE Trans. Netw. Serv. Manag. 17 (2) (2020) 1026–1039. doi:10.1109/TNSM.2019.2963643.
URL https://doi.org/10.1109/TNSM.2019.2963643

[103] A. Kurniawan, Learning AWS IoT: Effectively manage connected devices on the AWS cloud using services such as AWS Greengrass, AWS button, predictive analytics and machine learning, Packt Publishing, 2018.
URL https://books.google.rs/books?id=7NRJDwAAQBAJ

[104] Linux Foundation, KubeEdge, https://kubeedge.io/ (accessed November 7, 2020).

[105] General Electric, GE. Predix, https://www.ge.com/digital/iiot-platform/ (accessed November 7, 2020).

[106] Y. Mao, J. Zhang, K. B. Letaief, Dynamic computation offloading for mobile-edge computing with energy harvesting devices, IEEE J. Sel. Areas Commun. 34 (12) (2016) 3590–3605. doi:10.1109/JSAC.2016.2611964.
URL https://doi.org/10.1109/JSAC.2016.2611964

[107] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, N. Fullagar, Native client: a sandbox for portable, untrusted x86 native code, Commun. ACM 53 (1) (2010) 91–99. doi:10.1145/1629175.1629203.
URL https://doi.org/10.1145/1629175.1629203

[108] M. Beck, M. Werner, S. Feld, T. Schimper, Mobile edge computing: A taxonomy, in: The Sixth International Conference on Advances in Future Internet (AFIN 2014), 2014.
URL https://www.researchgate.net/publication/267448582_Mobile_Edge_Computing_A_Taxonomy

[109] F. R. de Souza, C. C. Miers, A. Fiorese, M. D. de Assunção, G. P. Koslovski, QVIA-SDN: towards qos-aware virtual infrastructure allocation on sdn-based clouds, J. Grid Comput. 17 (3) (2019) 447–472. doi:10.1007/s10723-019-09479-x.
URL https://doi.org/10.1007/s10723-019-09479-x

[110] J. R. Hamilton, An architecture for modular data centers, in: CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings, www.cidrdb.org, 2007, pp. 306–313.
URL http://cidrdb.org/cidr2007/papers/cidr07p35.pdf

[111] K. Sonbol, Ö. Özkasap, I. Al-Oqily, M. Aloqaily, Edgekv: Decentralized, scalable, and consistent storage for the edge, J. Parallel Distributed Comput. 144 (2020) 28–40. doi:10.1016/j.jpdc.2020.05.009.
URL https://doi.org/10.1016/j.jpdc.2020.05.009

[112] J. Wang, D. Crawl, I. Altintas, W. Li, Big data applications using workflows for data parallel computing, Comput. Sci. Eng. 16 (4) (2014) 11–21. doi:10.1109/MCSE.2014.50.
URL https://doi.org/10.1109/MCSE.2014.50

[113] C. Li, J. Bai, Y. Chen, Y. Luo, Resource and replica management strategy for optimizing financial cost and user experience in edge cloud computing system, Inf. Sci. 516 (2020) 33–55. doi:10.1016/j.ins.2019.12.049.
URL https://doi.org/10.1016/j.ins.2019.12.049

[114] E. Cau, M. Corici, P. Bellavista, L. Foschini, G. Carella, A. Edmonds, T. M. Bohnert, Efficient exploitation of mobile edge computing for virtualized 5g in EPC architectures, in: 4th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering, MobileCloud 2016, Oxford, United Kingdom, March 29 - April 1, 2016, IEEE Computer Society, 2016, pp.

100–109. doi:10.1109/MobileCloud.2016.24.
URL https://doi.org/10.1109/MobileCloud.2016.24

[115] W. Yu, C.-L. Ignat, Conflict-Free Replicated Relations for Multi-Synchronous Database Management at Edge, in: IEEE International Conference on Smart Data Services, 2020 IEEE World Congress on Services, Beijing, China, 2020.
URL https://hal.inria.fr/hal-02983557

[116] M. Simic, M. Stojkov, G. Sladic, B. Milosavljević, Crdts as replication strategy in large-scale edge distributed system: An overview, in: CRDTs as replication strategy in large-scale edge distributed system: An overview, 2020.

[117] A. Farshin, A. Roozbeh, G. Q. M. Jr., D. Kostic, Make the most out of last level cache in intel processors, in: G. Candea, R. van Renesse, C. Fetzer (Eds.), Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019, ACM, 2019, pp. 8:1–8:17. doi:10.1145/3302424.3303977.
URL https://doi.org/10.1145/3302424.3303977

[118] X. Jin, S. Chun, J. Jung, K. Lee, Iot service selection based on physical service model and absolute dominance relationship, in: 7th IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2014, Matsue, Japan, November 17-19, 2014, IEEE Computer Society, 2014, pp. 65–72. doi:10.1109/SOCA.2014.24.
URL https://doi.org/10.1109/SOCA.2014.24

[119] Y. Yao, B. Xiao, W. Wang, G. Yang, X. Zhou, Z. Peng, Real-time cache-aided route planning based on mobile edge computing, IEEE Wirel. Commun. 27 (5) (2020) 155–161. doi:10.1109/MWC.001.1900559.
URL https://doi.org/10.1109/MWC.001.1900559

[120] D. Gannon, R. S. Barga, N. Sundaresan, Cloud-native applications, IEEE Cloud Comput. 4 (5) (2017) 16–21. doi:10.1109/MCC.2017.4250939.
URL https://doi.org/10.1109/MCC.2017.4250939

[121] J. hung Ding, C. jung Lin, P. hao Chang, C. hao Tsang, W. chung Hsu, Y. ching Chung, Armvisor: System virtualization for arm, in: In Proceedings of the Ottawa Linux Symposium (OLS, 2012, pp. 93–107.

[122] M. Simić, M. Stojkov, G. Sladic, B. Milosavljević, M. Zarić, On container usability in large-scale edge distributed system, in: On container usability in large-scale edge distributed system, 2019.

[123] R. Hu, N. Yoshida, Explicit connection actions in multiparty session types, in: M. Huisman, J. Rubin (Eds.), Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Vol. 10202 of Lecture Notes in Computer Science, Springer, 2017, pp. 116–133. doi:10.1007/978-3-662-54494-5_7.
URL https://doi.org/10.1007/978-3-662-54494-5_7

[124] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, in: G. C. Necula, P. Wadler (Eds.), Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008, ACM, 2008, pp. 273–284. doi:10.1145/1328438.1328472.
URL https://doi.org/10.1145/1328438.1328472