UNIVERSITY OF NOVI SAD

FACULTY OF TECHNICAL SCIENCES NOVI SAD

Miloš Simić

Dynamic formation of the distributed micro clouds

- 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 Obradovića 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**:	Dinamičko formiranje mikro okruženja u računarstvu u oblaku
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)	5/200/138/9/17/0/0
Naučna oblast, **NO**:	Elektrotehničko i računarsko inženjerstvo
Naučna disciplina, **ND**:	Distribuirani sistemi
Predmetna odrednica/Ključne reči, **PO**:	distribuirani sistemi, računarstvo u oblaku, višestruko računarstvo u oblaku, mikroservisi, softver kao servis, ivično računarstvo, mikro računarstvo u oblaku, veliki podaci.
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 Obradovića 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	

Obrazac **Q2.HA.06-05**- Izdanje 1

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**:	Dynamic formation of the distributed micro clouds
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/	5/200/138/9/17/0
Scientific field, **SF**:	Electrical engineering and computing
Scientific discipline, **SD**:	Distributed systems
Subject/Key words, **S**/**KW**:	distributed systems, cloud computing, multi cloud, microservices, software as a service, edge computing, micro clouds, big data.
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**:			
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

Cloud computing is facing some serious latency issues due to huge volumes of data that need to be transferred from the place where data is generated to the cloud. For some types of applications, this is not acceptable.

One of the possible solutions to this problem is the idea to brings cloud services closer to the edge of the network, where data originates. This idea is called edge computing, and it is advertised that a dramatically reduces the network latency as a bridge that links the users and the clouds, and as such it makes the foundation for future interconnected applications.

Edge computing is a relatively new area of research and still faces many challenges like geo-organization and a clear separation of concerns, but also remote configuration, well defined native applications model, and limited node capacity. Because of these issues, edge computing is hard to be offered as a service, for future real-time user-centric applications.

This thesis presents the dynamic organization of geo-distributed edge nodes into micro data-centers and forming micro-clouds to cover any arbitrary area and expand capacity, availability, and reliability. We use a cloud organization as an influence with adaptations for a different environment with a clear separation of concerns, and native applications model that can leverage the newly formed system.

We argue that the presented model can be integrated into existing solutions or used as a base for the development of future systems.

Furthermore, we give a clear separation of concerns for the proposed model. With the separation of concerns setup, edge-native applications model, and a unified node organization, we are moving towards the idea of edge computing as a service, like any other utility in cloud computing.

The first chapter of this thesis, gives an introduction to the area of distributed systems, narrowing it down to only parts that are important for further understanding of the other chapters and the rest of the thesis in general.

The second chapter, show related work from different areas that are connected or that influenced this thesis. This chapter as well shows what is the current state of the art in industry and academia, but also describes the position of this thesis compared to the related research.

The third chapter proposes a model that is influenced by cloud computing architectural organizations but adopted for a different environment. We present how we can separate the geographic area into micro data-centers that are zonally organized to serve the local population and, and form them dynamically. This chapter also gives formal models for all protocols used for the creation of such a system with separation of concerns, applications models, and present limitations of this thesis.

The fourth chapter presents an implemented framework that is based on the model described in chapter three. We describe the architecture and in detail, every operation framework can do with limitations of the framework.

The fifth and the last chapter concludes this thesis and presents future work that should be done.

Key words: distributed systems, cloud computing, multi cloud, microservices, software as a service, edge computing, micro clouds, big data.

Rezime

Razni softverski sistemi promenili su način na koji ljudi komuniciraju, uče i vode posao, a međusobno povezani računarski sistemi imaju brojne pozitivne primene u svakodnevnom životu. Tokom protekle decenije, količina proračuna i obim podataka su se znatno povećali [1], što je za posledicu imalo sve veću upotrebu distriburanih sistema da bi se ti poslovi mogli obaviti uspešno.

Proširena stvarnost, igre preko mreže, automatksko prepoznavanje lica, autonomna vozila ili aplikacije Internet stvari (IoT) proizvode izuzetno velike koliˇine podataka. Ovakva optereénja zahtevaju kašnjenje ispod nekoliko desetina milisekundi [1]. Ovi zahtevi su izvan onoga što centralizovani računarski modeli (npr. poput računarstva u oblaku) moge da ponude [1].Čak i mali problemi mogu dovesti do velikog zastoja u aplikacijama i uslugama od kojih ljudi zavise.

Primer koji se desio nedavno, je još jedan u nizu prekida koji se dogodio na Amazon Web Services (AWS) platformi. Ovim je platforma bila nedostupna korisnicima i aplikacijama, a kao rezultat nedostupnosti platforme velika količina aplikacija i servisa koji se izvršavaju preko interneta postaje nedostupna korisnicima. Da bi razumeli kako smo došli do tog problema, prvo moramo razumeti šta je zapravo računarstvo u oblaku.

Računarstvo u oblaku možemo da definišemo kao agregaciju računarskih resursa koji se mogu ponuditi korisnicima, kroz uslužni softver [2]. Hardver i softver u velikim centrima za obradu podataka pružaju usluge korisnicima preko Interneta [3]. Resursi poput CPU-a,

GPU-a, skladišta podataka i mreže su resursi, i mogu se koristiti ili ne i to na zahtev i po potrebi korisnika [4].

Ključna snaga računarstva u oblaku su servisi koji su ponuđeni korisnicima kao usluge [2]. Tradicionalni model računarstva u oblaku pruža ogromne procesne i skladišne resurse i to po potrebi, na zahtev korisnika — elastično, kako bi podržao različite potrebe aplikacija. Ovo svojstvo se odnosi na sposobnost ovog modela da dozvoli alokaciju dodatnih resursa ili oslobađpostojećih, da bi se podudarali sa radnim intereć enjima aplikacije i to na zahtev [5].

Problem nastaje (između ostalog) kada je potrebno da se (veliki) podaci prebace sa izvora u oblak. Ovaj proces dovodi do velike latencije ili kašnjenja u sistemu [6]. Na primer, Boeing 787s generiše pola terabajta podataka po jednom letu, dok autonomni automobil generiše dva petabajta podataka tokom samo jedne vožnje. Međutim, propusni opseg nije dovoljno velik da bi podržao takve zahteve [7]. Prenos podataka nije jedini problem sa čime se računarsto u oblaku susreće. Aplikacije kao što su autonomni automobili, bespilotne letelice ili balansiranje snage u električnim mrežama zahtevaju obradu podatka u realnom vremenu da bi ispravno donosili odluke, i reagovali na promene [7]. Suočavamo se sa ozbiljnim problemima, ako usluga u oblaku postane nedostupna zbog napada milicioznih korisnika — hakera, ili prostog kvara na mreži [8].

Centralizovana arhitektura računarstva u oblaku sa ogromnim kapacitetima centara za obradu podataka, stvara efikasnu ekonomiju obima. Ovom strategiom se dolazi do smanjenja administrativnih trok̆ova celokupnog sistema [9]. Međutim, kada takav sistem dodđe do svojih granica, centralizacija donosi više problema nego što ih može rešiti [8, 10]. Uprkos svim prednostima ovog modela, servisi i usluge vremenom se suočavaju sa ozbiljnom degradaciom kvaliteta odziva i performansi, usled velike propusnosti i kašnjenja [11].

To može dovesti do nesagledivih posledica po biznis, ali potencijalno i ljudske živote. Organizacije koriste usluge u oblaku kako bi izbegle velike infrastrukturalne investicije [12], poput pravljenja i održavanja sopstvenih centara za obradu podataka. Oni koriste resurse koje su obezbedili drugi [13], i plaćaju shodno tome koliko vremenski koriste usluge – a pay as you go model.

Cilj ove teze je predstavi upotrebu formalnih modela na osnovu kojih možemo opisati, formalno verifikovati protokole i implementirati radni okvir za distribuirani sistem koristeći geografski rasprostranjena okruženja, nalik računarstvu u oblaku. Opisani sistem mogu da koriste ne samo obični korisnici, veći pružaoci usluga računarstva u oblaku mogu ga integrisai u svoje servise kako bi se minimalizovao zastoj kritičnih sistemskih segmenata. Čitav sistem možemo posmatrati kao skup mikro oblaka ili sloj obrade, koji u oblak šalje samo važne podatke smajujuć i troškove korisnicima, ali i obezbeđujući veću dostupnost usluga računarstva u oblaku.

Distribuirane softverske sisteme nije jednostavno implementirati niti modelovati. Problem često nastaje zbog problema u komunikaciji čvorova preko mreže koja nije sigurna ni pouzdana. Poruke mogu da kasne ili da ne stignu nikada. Takođe, čvorovi u sistemu mogu da prestanu da rade potpuno nasumično stvarajući dodatne komplikacije. James Gosling i Peter Deutsch, tadašnji saradnici iz Sun Microsistems, kreirali su listu problema za mrežne aplikacije poznate kao *8 zabluda distribuiranih sistema*:

(1) **Mreža je pouzdana**; uvek će se nešto katastrofalno desiti sa mrežom koja je prilično nepouzdana - prekid napajanja, prekid kabla, katastrofe u okruženju itd;

(2) **Latencija ne postoji**; lokalno kašnjenje nije problem, ali se situacija vrlo brzo pogoršava kada pređemo na komunikaciju preko Interneta i scenario gde se koristi izuzetno kompleksna mrežna komunikacija računarstva u oblaku.

(3) **Propusnost je beskonačna**; iako se širina propusnog opsega stalno povećava i sve je bolja i bolja, isto tako raste i količina podataka koju pokušavamo da prebacimo na obradu ili skladištenje.

(4) **Mreža je sigurna**; Trendovi internet napada pokazuju izuzetno veliki rast napda, a ovo još više postaje problem u računarstvu u oblaku javnog tipa.

(5) **Topologija se ne menja**; mrežna topologija je obično izvan kontrole korisnika, a topologija mreže se stalno menja iz brojnih razloga - dodati ili uklonjeni novi uređaji, serveri, prekidi u komunikaciji itd.

(6) **Postoji samo jedan administrator**; danas postoje brojni administrativi za veb servere, baze podataka, keš memoriju i slično, ali takođe kompanije sarađuje sa drugim kompanijama ili pružaocima usluga računarstva u oblaku.

(7) **Troškovi transporta ne postoje**; ovo nikako nije tačno iz prostog razloga što moramo serializovati informacije i podatke koje saljemo, što troši resurse i povećava ukupnu kašnjenje. Ovde nije problem samo u kašnjenju, već u tome što svaka serializacija informacija zahteva dodatno vreme i dodatne resurse.

(8) **Mreža je homogena**; danas je homogena mreža izuzetak, a ne pravilo. Imamo različite servere, sisteme, klijente koji komuniciraju. Implikacija ovoga je da pre ili kasnije moramo pretpostaviti da nam je potrebna interoperabilnost između ovih sistema. Mogli bismo da imamo i neke zaštićene protokole koji nisu javno dostupni koji bi takođe mogli trošiti dodatno vreme, pri tome oni mogu ostati bez podrške, pa bismo ih trebali izbegavati.

Ove zablude su definisane pre više od deceniju, i više od četiri decenije otkako smo počeli da gradimo DS, ali karakteristike i osnovni problemi ostaju prilično isti. Zanimljiva je činjenica da dizajneri, arhitekte i dalje pretpostavljaju da tehnologija sve rešava. To nije slučaj u DS-u i ove zablude ne bi trebalo zaboraviti. Zbog ovih problema, DS je teško korektno primeniti, a teško ih je testirati i održavati.

Programeri i dizajneri distribuiranih sistema i dan danas zaborave na ove zablude i to dovodi o velikih problema. Način da se to u ranim fazama otkrije je korišćenje formalnih, matematički zasnovanih metoda. Ove metode čine razne tehnike koje služe za specifikaciju i verifikaciju kompleksnih sistema koje su zasnovane na matematičkim i logičkim principima.

Da bi se prevazišlo kašnjenje u oblaku, istraživanje je dovelo do novih računarskih oblasti poput ivičnog računarstva (EC). EC je model u kome se procesne i skladišne mogućnosti računarstva u oblaku, u blizini izvora podataka [13]. Računarstvo u oblaku je prošireno novim mogućnostima što dovodi do novih ideja za aplikacije buduće generacije [14]. Tokom godina pojavili su se razni modeli poput fog računarstva [15], cloudleta [12] i mobilnih ivičnih računara (MEC) [16]. U ovom radu sve ove modele nazivamo ivičnim čvorovima.

Svi pomenuti modeli koriste koncept prenosa skladišnih i procesnih mogučnosti, iz oblaka bliže izvorima podataka [17] dok su zahtevnije obrade ostaju u oblaku iz vrlo prostog razloga – dostupnosti znatno većkoličine resursa [14]. EC modeli uvode male servere koji se arhitekturalno nalaze između izvora podataka i oblaka. Tipično je za ove servere da imaju mnogo manje mogućnosti u poređenju sa servera u oblaku [18].

Prednost malih servera je u tome što se oni mogu naći skoro bilo gde, na primer u baznim stanicama [16], gradskim centralama, kafićima ili rasprostranjeni po geografskim regionima a sve to, kako bi se izbeglo kašnjenje, kao i povećala propusnost [12]. Oni mogu poslužiti kao zaštitni zidovi [19] ili kao nivo obrade pre nego što podaci budu poslati u oblak. Sa druge strane, korisnici dobijaju jedinstvenu mogućnost dinamičke i selektivne kontrole informacija koje bivaju poslate u oblak. Još jenda prednost ovih servera predstavili su su Aroca [20] i saradnici, gde su njihovi rezultati pokazali da mali serveri zadržavaju dobre perfomanse prilikom pokretanja servera i klasterskog okruženja. Malo slabije performanse su pokazali u slučaju trenutno dostupnih skladišta podataka, ali to može biti podsticaj da se polje istrživanja skladišta podataka dopuni novim modelima optimizovanim za male servere.

Jedna opcija će teško moćda odgovara potrebama svih aplikacija u budućnosti, tako da računarstvo u oblaku ne bi trebalo da bude naša granica i jedina opcija. Razni modeli nastali na bazi malih servera, pokazuju mogućnost da se obrada podataka može obaviti bliže izvorištu, a sve u cilju smanjenja kašnjenje zahteva klijentskih aplikacija primenom malih servera, dok teški proračuni mogu ostati u oblaku zbog veće dostupnosti resursa. Slediti ideju, da u oblak poslati samo informacije koje su ključne za druge usluge ili aplikacije [21], a ne sve kako predlaže standardni model oblaka. Ideja malih servera sa različitim računskim, skladišnim i mrežnim resursima pokreće zanimljive istrazživaćke ideje i motivacija su za ovu tezu. Koristeći resurse koji su organizovani lokalno kao mikro oblaci, oblaci zajednice ili ivični oblaci [22] predlažu Riden i saradnici.

Suočeni sa stvarnim problemima i problemima koji mogu nastati u doglednoj budućnosti ali i ograničenjima računarstva u oblaku u trenutnoj izvedbi, akademska zajednica, kao i industrija počeli su da istražuju i razvijaju održiva rešenja. Neka istraživanja su više usredsređena na prilagođavanje postojećih rešenja da odgovaraju EC, dok druga eksperimentišu sa novim idejama i rešenjima.

U svom radu [23] Greenberg i saradnici ističu da se mikro centri za obradu podataka (MDC) koriste prvenstveno kao čvorovi u mrežama za distribuciju sadržaja i drugim "sramotno distribuiranim" aplikacijama. MDC su zanimljiv model u području brzih inovacija i razvoja. Greenberg i saradnici [23] uvode MDC-ove kao centar za obradu podataka koji rade u blizini velike populacije, smanjujući fiksne troškove tradicionalnih centara za obradu podataka. Minimalna veličina MDC-a definisana je potrebama lokalnog stanovništva [23, 24], prižajući agilnost kao ključnu karakteristiku. Ovde agilnosti znači sposobnost dinamičkog rasta i smanjenja potrebe za resursima i upotrebe resursa sa optimalne lokacije [23].

Zonska organizacija malih servera koju su predstavili Guo i saradnici [25] u primeni kod pametnih vozila, daje zanimljivu perspektivu o

EC. Autori su pokazali kako modeli zasnovani na zonama omogućavaju kontinuitet dinamičkih usluga i smanjuju primopredaju veze. Takođe, su pokazali kako da se poveća pokrivenost malim serverima na veću zonu, čime se proširuju računarska snaga i kapacitet skladištenja podataka.

EC potiče iz peer-to-peer sistema [10] kako su to pretpostavili López i saradnici, ali ga proširuje u novim pravcima i pruža mogućnost integracije sa računarstvom u oblaku.

U som radu Kurniavan i saradnici [26] su pokazali vrlo lošu skalabilnost u centralizovanim modelima mreža za isporuku sadržaja u oblaku (CDN). Predložili su decentralizovano rešenje koristeći nano centre za obradu podataka sačinjenu od mrežnih uređaja u kuć [26]. Ovi DC-ovi su takođe opremljeni sa nešto skladišta. Autori su pokazali mogucú upotrebu nano centara za obradu podataka za neke velike primene, sa mnogo manjom potrošnjom energije.

MDC-ovi sa zonskom organizacijom servera dobra su polazna osnova za izgradnju EC-a koja može biti ponuđna kao servis i mikro računarstva u oblaku, jer možemo proširiti računarsku snagu i skladišni kapacitet koji opslužuju lokalno stanovništvo. Da bi to postigli, potreban nam je dostupniji i elastičniji sistem sa manje kašnjenja.

Ako pogledamo dizajn CC, svaki deo doprinosi otpornijem i skalabilnom sistemu. Regioni ili centri za obradu podataka (DC) su izolovani i nezavisni jedni od drugih, a takođe sadrže resurse koji su potrebni aplikacijama za nesmetan rad. Sčinjeni su od nekoliko dostupnih zona [27]. Ako zona iz bilo kog razloga postane nedostupna, ima ih još koje mogu da opslože korisničke zahteve. Uz neke adaptacije, EC i mikro računarsto u oblaku bi mogla da koristi vrlo sličnu strategiju.

Čvorove možemo grupisati u klastere, a više klastera čvorova u veću logičku celinu – “region”, povećavajući dostupnost i pouzdanost sistema kao i njegovih aplikacija. Kada pričamo o EC i mikro računarstvu u oblaku, često mislimo na geografski rasprostranjene distribuirane sistemime, tako da imamo malo drugačiji scenario nego u standardnom CC modelu.

Koncept “regiona” u računarstvu oblaka je fizički element [27], dok se u mikro računarstvu u oblaku region može koristiti za opisivanje skupova klastera čvorova, preko proizvoljne geografske oblasti.

Regioni se sastoje od najmanje jednog klastera, ali mogu se sastojati i od više njih, tako da se postigne otporniji, skalabilniji i dostupniji sistem. Da bi se osiguralo manje kašnjenje u sistmu, u normalnim okolnostima treba izbegavati veliku udaljenost između klastera. U tradicionalnom modelu računarstva u oblaku, proširenje regiona zahteva fizičko povezivanje modula sa ostatkom infrastrukture [28].

U mikro računarstvu u oblaku, regioni mogu prihvatiti nove, ili osloboditi postojeće klastere ali i klasteri mogu prihvatiti nove ili osloboditi postojeće čvorove.

Više regiona čine sledeći logički sloj – “topologiju”. Topologija se sastoji od najmanje jednog regiona, a može se prostirati i na više regiona. Prilikom dizajniranja topologije, posebno ako regioni trebaju da dele informacije ili treba na neki način da sarađuju, treba izbegavati veliku udaljenost između regiona, ako je to moguće.

Ovim vrlo jednostavnim konceptima, možemo da pokrijemo bilo koju geografsku oblast sa sposobnošću da smanjimo ili proširimo postojeće klastere, regione pa čak i topologije. Organizacija klastera, regiona i topologija u mikro računarstvu u oblaku je isključivo stvar dogovora, i kao takva slična je modeliranju u sistemima velikih podataka [29, 30].

Na primer, klasteri mogu biti veliki kao čitav jedan grad ili mali kao svi uređaji u jednom domaćinstvu i sve između ova dva ekstrema. Grad bi mogao predstavljati jedan region, sa delovima grada koji su organizovani u klastere. Topologiju grada možemo formirati tako što ćemo grad podeliti na više regiona, koji sadrže više klastera. Topologiju države možemo formirati tako što ćemo ćemo je podeliti na regione, pri čemu su gradovi regioni.

Čvorovi unutar svakog klastera treba da izvršavaju neki od protokol-a za održavanje klastera. Neki od *Gossip* protokola poput *SWIM*-a[31], mogu se koristiti u saradnji sa mehanizmima replikacije podataka [32, 33, 34] čineći ceo sistem otpornijim na potencijalne greške.

U modelu koji opisuje sve resurse kao usluge [35], EC i mikro računarstvo u oblaku kao usluga se nalazi između CaaS i PaaS, u zavisnosti od potreba korisnika.

Dobro definisan sistem mogao bi se ponuditi kao usluga korisnicima, kao i bilo koji drugi resurs raůnarstva u obaku. Možemo ga ponuditi istraživačima i programerima da naprave nove aplikacije usmerene više ka raznim potrebama ljudi. Ako nam je potrebno više resursa na jednoj strani, možemo uzeti jedanu količinu resursa i premestiti gde nam ti resursi trebaju. Sa druge strane, kompanije koje pružaju usluge računarstva u oblaku mogu da integrisati model u svoj postojeći sistem, skrivajući nepotrebnu složenost iza nekog komunikacionog interfejsa ili predloženog modela aplikacije.

Da bi se postiglo takvo ponašanje, neophodno je dinamičko upravljanje resursima i upravljanje uređajima. Moramo uvek imati dostupne informacije o resursima, konfiguraciju i zauzetosti čvorova [36, 16]. Tradicionalni centri za obradu podataka predstavljaju dobro organizovan i povezan sistem. Sa druge strane, MDC-ovi se sastoje od razliǐtih uređaja koji nisu [37]. Ova ideja nas dovodi do problema sa kojim se bavi ova teza.

EC i MDC modelima nedostaje jasna dinamička organizacija geografski raspoređnenih čvorova, dobro definisan model matičnih aplikacija i jasno razdvajanje nadležnosti. Kao takvi ne mogu se ponuditi kao usluga korisnicima. EC obično postoje nezavisno jedni od drugih, rasuti su bez međusobne komunikacije i saradnje, a nude ih pružaoci usluga, koji korisnike uglavnom zaključavaju u sopstveni ekosistem. Grupisani čvorovi treba da budu organizovani lokalno, čineći kompletan sistem i aplikacije dostupnijim i pouzdanijim, ali takođe proširujući resurse izvan pojedinačnog čvora ili male grupe čvorova, održavajući dobre performanse za izgradnju servera i klastera [20].

Da bi opisao fizičke usluge, Jin [38] i saradnici predlažu tri osnovna koncepta i precizira njihove odnose. Ovi koncepti su: (1) uređaji, (2) resursi i (3) servisi. Podela nadležnosti je bitan deo svakog sistema, posebno ako se stvara platformu koja se nudi kao usluga. Model podele nadležnosti koji ova teza predlaže zasnovan je na ovim konceptima, prilagođen za drugačiji slučaj korišćenja, i podeljen u tri sloja što se može videti na slici 3.2.

Donji sloj čine različiti uređaji ili kreatori podataka i korisnici usluga odnosno servisa. Drugi sloj predstavlja resurse. Resursi imaju prostornu karakteristiku i ukazuju na mogućnosti njihovih hosting uređaja za obradu odnosno skladištenje podataka [38]. Programeri u bilo kom trenutku moraju znati iskorišćenost resursa, kao i stanje i dostupnost aplikacija.

Resursi predstavljaju EC čvorove, i da bi čvor bio deo sistema, on mora zadovoljiti četiri jednostavna pravila:

(1) moraju biti sposobni da pokrenu operativni sistem sa sistemom datoteka;

(2) moraju biti u mogućnosti da pokrene neki alat za izolaciju aplikacija, na primer kontejnere ili unikernele;

(3) moraju imati dostupne resurse za korišćenje (npr. CPU, GPU, disk itd.) ;

(4) moraju imaju stabilnu internet vezu;

Servisi pružaju resurse putem definisanog interfejsa i čine ih dostupnim preko Interneta [38]. Servisi odmah odgovaraju na klijentske zahteve, ako je to moguće, ili mogu da skladište pronađenu informaciju za neke buduće zahteve [39, 40]. Servisi koji se izvrvšavaju u oblaku treba da budu u stanju da prihvate unapred obrađene podatke i odgovorne su za obradu i skladištenje podataka čiji kapacitet prevazilazi mogućnosti EC čvorova.

Ovo proširenje računarstva u oblaku produbljuje i jača naše dosadasnje razumevanje oblasti u celini. Razdvajanjem nadležnosti, modela matičnih aplikacija i objedinjenem organizacije čvorova, idemo ka ideji EC-a kao usluge, i mogucnosti dinamičkog pravljenja mikro oblaka koji bi mogli da obrade podatke na samom njihovom izvoru.

Međutim, infrastrukture neće biti postavljena sve dok proces podešavanja i korišćenja ne bude trivijaln [39]. Odlazak od čvora do čvora je dosadan i dugotrajan proces. Naročito kada se uzme u obzir geografska rasprostranjenost čvorova.

Sistem koji je predložen u ovoj tezi, rešava problem pomoću podešvanja i formiranja prethodno definisanih elemenata, i oslanja se na tri protokola:

(1) **provera stanja čvora** protokol obaveštava sistem o stanju svakog čvora;

(2) **formiranje klastera** protokol formira nove klastere, regione i topologije;

(3) **pregled detalja** protokol prikazuje trenutno stanje sistema korisniku kroz razne nivoe detalja;

Da bismo formalno jednostavno opisali servere ili cvorove (pojmovi se koriste naizmenično) u sistemu, možemo koristiti teoriju skupova. Prethodno definisane protokole možemo formalno modelirati koristći [41], proširenje *multiparty asynchronous session types* (MPST) [42] — klasa tipova ponašanja skrojena za opisivanje distribuiranih protokola oslanjajući se na asinhrone komunikacije.

Ova matematička teorija nije samo korisna kao formalni opisi protokola, već se možemo iskoristiti i mogućnosti teorije za verifikaciju da li naši protokoli zadovoljavaju MPST sigurnost (nema dostupnog stanja greške) i napredak (akcija se na kraju izvršava, pod pretpostavkom poštenja).

Proces modelovanja se odvija u dva koraka:

(**1**) **Prvi korak** u modeliranju komunikacija sistema pomoću MPST teorije je da pružimo *globalni tip*, to je globalni opis celokupnog protokola sa neutralnog tačka posmatranja.

(**2**) **Drugi korak** u modeliranju komunikacija sistema pomoću MPST teorije je pružanje sintaksičke projekcije protokola na svakog učesnika kao lokalni tip, koji se zatim koristi za proveru tipa i implementacije krajnje tačke.

Na osnovu ovih ideja, definišemo problem koji ova teza obrađuje kroz sledeća tri istraživačka pitanja:

(**1**) *Da li možemo da organizujemo geografski distribuirane čvorove na sličan način kao računarstvo u oblaku, adaptiran za različito okruženje, sa jasnom podelim nadležnosti, i poznatim modelom razvoja aplikacija za korisnike?*

(**2**) *Da li možemo da ponudimo ovako organizovane čvorove kao uslugu programerima i istraživačima za budućaplikacije usmerene više ka ljudima, a zasnovane na poznatom modelu – pay as you go?*

(**3**) *Da li možemo da napravimo model na takav način da je formalno ispravan, lak za proširivanje, razumevanje i obrazloěnje?*

Ako su prethogna istraživačka pitanja potvrdna, onda proširenje nalik na oblak čini čitav sistem, ali i same aplikacije koje bi se izvršavale u njemu dostupnijim i pouzdanijim, ali takođe proširuje resurse van granica pojedinačnog čvora. Satianaraianan i saradnici u svom radu [19] pokazuju da MDC-ovi mogu poslužiti kao zaštitni zidovi, dok Simić i saradnici, u svom radu [21] koriste sličnu ideju kao nivo obrade podatka na njihovom mestu nastanka – izvoru. Istovremeno, korisnici dobijaju jedinstvenu mogućnost dinamičkog i selektivnog upravljanja informacijama koje se šalju u oblak. Godinama nakon svog osnivanja, EC više nije samo ideja [19], većneophodan alat za nove aplikacije koje dolaze.

Na osnovu prethodno definisanih istraživačkih pitanja i motivacije , izvodimo hipoteze oko kojih se temelji teza, a koje se mogu rezimirati na sledeći način:

(1) **Hipoteza:** *Moguće je organizovati čvorove na standardni način zasnovan na arhitekturi zasnovanu na računarstvu u oblaku, prilagođen drugačijem geografski rasprostranjenim okruženju. Dati korisnicima mogućnost da na najbolji mogući način organizuju čvorove po raznim geografskim oblastima kako bi opsluživali samo lokalno stanovništvo u neposrednoj blizini.*

(2) **Hipoteza:** *Moguće je ponudite dobijeni model istraživačima i programerima da kreiraju nove aplikacije usmerene više ka ljudima. Ako nam je potrebno više resursa na jednoj strani, možemo uzeti određneu količinu resursa i premestiti na mesto gde su oni potrebni, ili ih organizovati na bilo koji drugi željeni način.*

(3) **Hipoteza:** *Moguće je predstaviti jasnu podelu nadležnosti za budući sistem koji bi bio pružen korisnicima kao uslugu, i uspostaviti dobro organizovan sistem u kojem svaki deo ima intuitivnu i jasnu ulogu.*

(4) **Hipoteza:** *Moguće je predstaviti objedinjeni model koji podržava heterogene čvorove, sa jasnim setom tehničkih zahteva koje budući čvorovi moraju ispuniti, ako žele da postanu deo sistema.*

(5) **Hipoteza:** *Moguće je predstaviti jasan aplikativni model intuitivan korisnicima, kako bi mogli da iskoriste puni potencijal novonastale infrastrukture.*

Iz prethodno definisanih hipoteza izvodimo primarne ciljeve ove teze, gde očekivani rezultati uključuju:

(1) *Konstrukcija modela sa jasnom podelom nadležnost, po ugledu na organizaciju računarstva u oblaku, prilagođeno drugačijem*

okruženju izvršavanja, sa jasnim aplikativnim modelom koji će moć i da iskoristi novu, adaptoranu arhitekturu. Ovo se odnosi na prvo istraživačko pitanje, a tema je poglavlja 3.

(2) *Definisani model je dostupniji, elastičan sa manje kačnjenja u poređenju sa pojedinačnim malim serverima, i kao takav se široj javnosti može ponuditi kao usluga, kao bilo koja druga usluga u oblaku. Ovo se odnosi na drugo istraživačko pitanje i tema je poglavlja 3.*

(3) *Konstruisani model je dobro opisan formalno, koristeći čvrstu matematičku osnovu, ali takođe i lako proširiv i formalni i tehnički, lak je za razumevanje i obrazloženje. Ovo se odnosi na treće istraživačko pitanje i tema je poglavlja 3.*

Ova teza predstavlja moguće rešenje za organizaciju geografski rasprostranjenih mikro-oblaka sa EC čvorovima, uz dodatak nekoliko dokazanih apstrakcija iz računarstva u oblaku poput zona i regiona.

Ove apstrakcije omogućavaju pokrivanje bilo kog geografskog područja i daju dostupniji i pouzdaniji sistem. Organizacija i reorganizacija ovih apstrakcija vrši se opisom zeljenog stanja bez direknog slanja komandi sistemu, a njihova veličina određuje se potrebama stanovništva.

Predstavili smo preslikavanje računarstva u oblaku na EC, i uz to smo prikazali formalni model sistema, sa jasnim modelom i sistemom podela nadležnosti i matičnim modelom aplikacije za budući EC koji bi bio ponuđen kao usluga.

Teza takođe prikazuje implementirano rešenje koristeći prethodno opisane koncepte i formalne modele. Implementirano rešenje može koristiti kao samostalno rešenje gde se kasnije mogu dodati potrebni podsistemi, ali takođe pruža mogućnost integracije u postojeća rešenja. Dati su primeri domena gde bi sistem mogao da se koristi zajdno sa primerima aplikacija gde bi korisnici imali benefit. Teza je organizovana u pet poglavlja.

U **poglavlju** 1 dali smo kratki uvod u temu distribuiranih sistema, sa fokusom na područja koja su važna za razumevanje ove teze.

Pokazali smo šta su distribuirani sistemi ili bar opšti konsenzus kako neki sistem možemo opisati ili posmatrati kao distribuirani. Predstavili smo probleme koje ovi sistemi stvaraju i zašto ih je tako teško implementirati, koristiti i održavati.

Takođe smo predstavili nekoliko primera distribuiranih računarskih aplikacija koje možemo koristiti da efikasno iskoristimo veliki broj čvorova u distribuiranom sistemu. Dalje smo predstavili šta je skalabilnost i zašto je ona važna za distribuirane sisteme, sa nekoliko primera organizacionih mogućnosti poput peer-to-peer mreža i protokola za opis grupa ili zajednica čvorova koji sarađuju, a koji su važni u distribuiranom okruženju iz različitih razloga. Dali smo primere raznih varijanti računarstva u oblaku koje možemo iskoristiti za svoje potrebe.

Zatim smo opisali nekoliko tehnika virtuelizacije koje se mogu koristiti za pakovanje i raspoređivanje kako aplikacija tako i infrastrukture. Prikazali smo razne tehnike bitne za raspoređivanje aplikacija i infrasktruture u okruženju računarstva u oblaku, ali i razliku između distriburanih sistema i nekoliko modela koji se često smatraju distribuiranim poput paralelnog raǐ decentralizovanog računarstva.

Na kraju smo dali opis motivacije za ovavku tezu sa izvedenim istraživackim pitanjima i hipotezama na koja želim da odgovorimo ovom tezom.

U **poglavlju** 2 smo prikazale sliņe radove koje su radili razni drugi istraživači ili kompanije. Isto kao i kod prethognog pogavlja, opet se fokusiramo samo na stvari koje su povezane sa ovom tezom.

Prikazali smo različite platforme, gde ljudi menjaju ili prilagođavaju postojeća rešenja kao što su Kubernetes ili OpenStack da bi ih prilagodili da rade u oblastima poput ivičnog računarstva i mobilnog računarstva. Dalje smo predstavili implementacije nekoliko platformi koje koriste čvorove koje su korisnici ponudili na dobrovoljnoj bazi, da bi se izvršila nekakva obrada ili skladištenje podataka na njima kao na primer drop computing i sistemima poput Nebule, između ostalih.

Pokazali smo kako čvorovi mogu biti organizovani po geografskim područijima na zone ali kako mikro centri za obradu podataka mogu da pomognu računarstvu u oblaku da služi zahteve lokalnog stanovništva koje se nalazi u neposrednoj blizini isth. Dalje smo olisali različite tehnike kako se zadaci sa mobilnih uređaja mogu prebaciti na ivične čvorove, ali takođe i različite modele primene koji bi mogli iskoristiti ove tehnike.

Na kraju ovog poglavlja, predstavljamo gde je mesto ove teze u poređđenju sa drugim sličnim modelima i drugim sličnim istraživanjima.

Poglavlje 3 predstavlja srž ove teze. Razdvajamo sve najvažnije aspekte koje treba da imamo kako bismo pomogli CC-u u vezi sa problemima sa kašnjenjem, velikim podacima sa ogromnim obimima podataka posebno u doba mobilnih uređaja i IoT-a.

Predloženi model zasnovan je na MDC-ima koji su zonsko organizovani i koji će opsluživati lokalno stanovništvo i stanovništvo u blizini. Predstavljamo model koji se zasniva na računarstvu u oblaku, ali je prilagođen za drugačiji scenario i slučajeve korišcénja.

Pokazali smo kako možemo dinamički da formiramo nove klastere, regione i topologije i kako ih možemo koristiti zajedno sa mobilnih uređajima i aplikacijama poput internet stvari (IoT). Ovaj novoformirani sistem ili sistem sistema oslanja se na jasan model podele nadležnosti, usvojen iz postojećih istraživanja i prilagođen za novu troslojnu arhitekturu. Formirani model može da služi kao sloj za obradu podatka na na samom izvorištu ili skoro na samom izvorištu, kao zaštitni zid ili sloj za zaštitu privatnosti. Predstavljeni sistem je izuzetno prilagodljiv u različitim dimenzijama, odnosno potrebama i zahtevima.

Predstavljeni model može biti ogroman kao cela država, ili malen kao pojedinačno domaćinstvo i sve između toga. Veličina klastera je stvar dogovora, i mogućnost je izbora. Takođe smo predstavili kako programeri mogu da iskoriste novu infrastrukturu, i koji sve modeli aplikacija mogu postojati, a takođe kako administratori mogu rasporediti

razvijene servise na novoformiranu infrastrukturu.

Pred kraj obog poglavlja, predstavljamo posledice ovog modela i kako se isti može koristiti kao sastavni deo postojećih sistema, na primer kao skladište informacija o čvorovima ili se može koristiti kao novi model gde možemo razviti nove podsisteme i aplikacije. Predstavili smo protokole za stvaranje takvog sistema i modeliramo ih koristeći formalne matematičke metode ili knkretno, teoriju asinhronih tipova sesija. Sistem sledi formalni model i lako ga je proširiti, kako formalno toko i praktično.

Na samom kraju pogavlja dajemo ograničenja ovog sistema sistema i ujedno ove teze, i stvari kojih moramo biti svesni, **ako** takva tehnologija bude korišćena u realnim situacijama.

U **poglavlju** 4, predstavljamo implementirani okvir zasnovan na znanju i istraživanijma skupljenim iz prethodnih poglavlja. Takođe smo detaljno opisali operacije koje se mogu obaviti u radnom okruženju, i kako se implementirani model uklapa i gde mu je mesto u prethodno opisanom modelu podele nadležnosti.

Dalje predstavljamo rezultate naših eksperimenata u kontrolisanom okruženju, kao i to koja su ograničenja implementiranog radnog okvira u trenutnoj fazi razvoja. Takođe smo opisali i moguće primene ovog sistema, i gde bi ovaj model mogao da se koristi kada bi ušao u upotrebu.

Poglavlje 5 predstavlja poslednje poglavlje ove teze. U ovom poglavlju zaključili smo tezu sa onim što je urađeno, šta može da se uradi u budućnosti u vidu budućih pravaca istraživanja.

Ključ reči: distribuirani sistemi, računarstvo u oblaku, višestruko računarstvo u oblaku, mikroservisi, softver kao servis, ivično računarstvo, mikro računarstvo u oblaku, veliki podaci.

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
DCs	Data centers
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
ECC	Edge-centric 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
RPC	Remote procedure call

Chapter 1

Introduction

Various software systems have changed the way people communicate, share information, learn, and run businesses. The interconnected computing devices have numerous positive applications in everyday life. In the past decade or so, the volumes of data that is collected, stored, and computed are grown dramatically [1]. New age applications that might include augmented reality, massive online gaming, face recognition, autonomous drones, and vehicles, or the Internet of Things (IoT) produce enormous amounts of different kinds of data. Such workloads require that latency is below a few tens of milliseconds [1], or even less. These requirements fall just right outside of what a standard centralized model like cloud computing (CC), for example, could offer [1]. Even the smallest problems can contribute to largely unplanned downtime of applications and services people and other services may depend on. A most recent example is yet another outage that was happened to the Amazon Web Services (AWS), and as a result, a large amount of internet becomes unavailable.

This thesis aims to provide formal models based on whom we can model and implement distributed systems (DS) for organizing cloud-like geo-distributed environments for users or CC providers. We can look at the whole system as a micro-cloud or pre-processing layer. The responsibility of such a system is for sending only necessary and data to the cloud. This strategy reduces cost for users and ensuring the availability of CC services. Such a system could lead to lowering the downtime of critical 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 heavily based on these topics. 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 patterns, and organizational structure. In Section 1.5 we describe similar models that might be a source of confusion, and how they are different than DS or DC, and how some concepts can fit in the bigger picture. Section 1.7 describe different architecture and application model and how deployment can be done in large DS. In Section 1.6 we describe different virtualization methods that are used in CC for systems and/or applications. In section 1.8 we describe the difference between concurrency and parallelism and introduce an actor system, that will be used later on in the thesis. In Section 1.9, we specify the exact problem that our work addresses and describe our hypothesis and research goals in Section 1.10. Section 1.11 present the structure of the thesis.

1.1 Problem area

To lower the administration cost, cloud providers create an effective economy of scale [9] by housing data-centers (DCs) with huge capacities. However, such a model does not come with the cost. When such a system grows to its physical limits, a centralized model brings more harm than good [8, 10]. Despite all the benefits that centralization can provide, it is inevitable for the CC services to suffer from serious problems [11]. Over time, and due to the high bandwidth and latency, these services face degradation that we cannot overlook. This serious services degradation can have an enormous consequence on the human business and potentially lives as well [43]. To avoid large investments [12], like creating and maintaining their own DCs, organizations use cloud services created by others [13]. They consume resources and pay for their usage time. This model is known as – pay as you go model.

CC requires data transfer to the DCs from data sources. This operation is problematic because it creates a high latency in the system [6]. We can observe few examples like data collection from planes and autonomous cars. Boeing 787s per single flight generates half a terabyte of data, while a self-driving car generates two petabytes of data per single drive. On the other hand, bandwidth is not large enough to support such requirements [7]. Data transfer is not the only problem CC is facing. Some applications require real-time processing for proper decision-making [7]. For example, self-driving cars, delivery drones, or power balancing in electric grids. Such applications might face serious issues if a cloud service becomes unavailable due to whatever reson [8].

Over the years, research led to new computing areas and model in which computing and storage utilities are in proximity to data sources [13]. To overcome cloud latency issues centralized CC model is enhanced with some new ideas [14].

1.2 Distributed systems

There are various definitions of DS, but we can think of DS as a system where multiple entities can communicate to one another in some way, but at the same time, they can perform some operations. In [44, 45] Tanenbaum et al. give two interesting assumptions 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 assumptions are useful and powerful when talking about DS. As such, in this thesis, we will adopt and use them rigorously.

Three significant characteristics of distributed systems are [45]:

(1) **concurrency of components**, refers to the ability of the DS that multiple activities are executed at the same time. These activities take 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. They can fail at the same time as well, but they usually fail independently for numerous reasons.

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

In [44] 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 is not exclusive to the domain of computer science. They existed before computers started to enrich almost every aspect of human life. DS have been used in various different domains such as: **telecommunication networks**, **aircraft control systems**, **industrial control systems** etc. DS are used anywhere where the number of users is growing rapidly so that a single entity cannot respond to the demands in (near) real-time.

Distributed systems (in computer science) are consists of various algorithms, techniques, and trade-offs to create an illusion that a set of nodes act as one. DS algorithms may include: (1) replication, (2) consensus, (3) communication, (4) storage, (5) processing, (6) membership etc.

DS is hard to implement because of its nature. James Gosling and Peter Deutsch both fellows at Sun Microsystems at the time created a list of problems for network applications know as *8 fallacies of Distributed Systems*:

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

(2) **Latency is zero**; locally 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 a problem even more in public CC.

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

(6) **There is one administrator**; nowadays there are numerous administrators for web servers, databases, cache and so one, but also company collaborates with other companies 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. The problem here is not just latency, but that information serialization takes time and resources.

(8) **The network is homogeneous**; today, a homogeneous network is the exception, rather than a rule. We have different

servers, systems, clients that interact. 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 introduced over a decade ago, and more than four decades since we started building DS, but the characteristics and underlying problems remain pretty the same. It is interesting fact that designers, architects still assume that technology solves everything. This is not the case in DS, and these fallacies should not be forgotten. Because of these problems, DS is hard to implement correctly and they are hard to test and maintain.

1.2.1 Scalability

Scalability is the property of a system to handle a growing amount of work by adding resources to the system [46]. When talking about computer systems scalability can be represented in two flavors:

- **Scaling vertically** means upgrading the hardware that computer systems are running on. Vertical scaling can increase performance to what the latest hardware can offer, and here we are limited by the laws of physics and Moor's law [47]. A typical example that requires this type of scaling is a relation database 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 approach, we are (almost) limitless on 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) [46], but because this approach comes with few more benefits that are especially 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 the application running somewhere, the user will get a response back. As a consequence of running multiple copies of a service and on multiple places, we have that service is more **avalible**, that running on a single node no matter 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 a more resilient and more available 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, the user will wait too long for the response to get back. If the same application is running in multiple places, the user request will hit the node that is closest to the user.

But despite all the obvious benefits, for a DS to work properly, we need the writing 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, which is related to distributed data storage systems. Storage implementations that rely on vertical scaling to ensure scalability and fault tolerance, have one nasty feature.

This nasty feature is represented in theorem called **CAP theorem** presented by Eric Brewer [48], and proven after inspection by Gilbert et a. [49]. The CAP theorem states that it is impossible for a distributed data store to simultaneously provide more than two out of three guarantees shown in Figure 1.1.

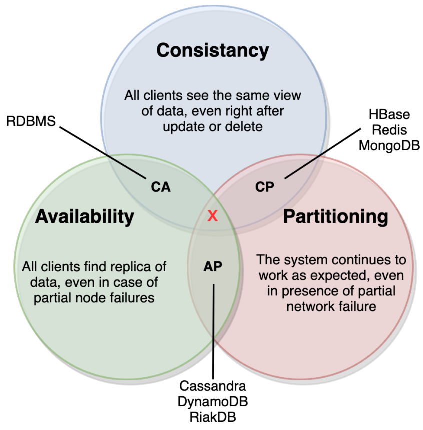
Consistancy
All clients see the same view of data, even right after update or delete
RDBMS
HBase
Redis
MongoDB
CA
CP
X
Availability
All clients find replica of data, even in case of partial node failures
AP
Partitioning
The system continues to work as expected, even in presence of partial network failure
Cassandra
DynamoDB
RiakDB

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

(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 may 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 a response back, even if one or nodes are down. DS will not interpret this situation as an exception or error. Availability is represented in percentage, and it describes how much downtime is allowed per year, This can be calculated using formula:

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

The industry is using measuring availability in "class of nines". Availability class is the number of leading nines in the availability figure for a system or module [50]. This metric relates to the amount of time (per year) that 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 [50].

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.

We can calculate availability class if we have system availability A, the system's availability class is defined as [50]:

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

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 losing 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 cannot be avoided.

Years after CAP theorem inception, Shapiro et al. prove that we can alleviate CAP theorem problems, but only in some cases, and offers **Strong Eventual Consistency (SEC) model** [51]. 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

Vogels et al. describe CC as an "aggregation of computing resources as a utility, and software as a service" [2]. Big DCs provide hardware and software services for their users over the internet [3]. Cloud providers offer various resources like CPU, GPU, storage, and network as utilities that can be used and released on-demand [4]. The key strength of the CC is reflected in the offered services [2]. To support the various application needs, the traditional CC model provides enormous computing and storage resources elastically. This property refers to the cloud ability to allow services to allocate additional resources or release unused ones to match the application workloads on-demand [5]. 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 buying and managing 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 1.2 show the difference in control and management of resources between different cloud options and on-premises solutions.

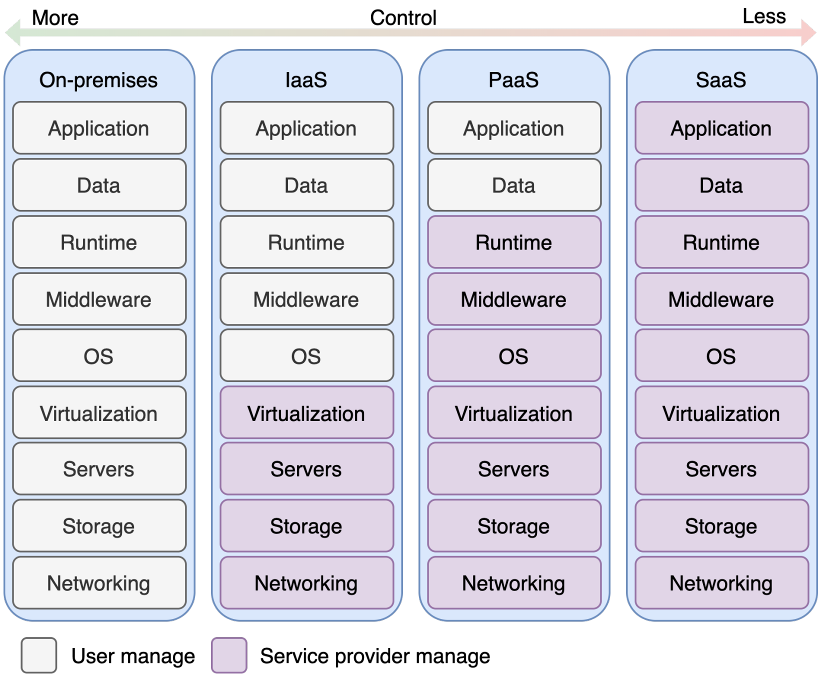

Figure 1.2: Difference between cloud options and on-premises solution.

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 a type where CC is delivered over the internet and shared across many many organizations and users. In this type of 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 a type where CC is dedicated only to a single organization. In this type of CC, architecture is built by an organization that may offer their solution or services to the users or other organizations. These services are in the domain of what the organization does, and that organization is in charge of maintenance. Examples include VMWare, XEN, KVM, etc.
- **Hybrid cloud** in such an 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 [13]. It is changing, evolving, and offering new types of services. Resources such as container as a service (CaaS), database as a service (DBaaS) [52] 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 [13] and pay for usage time – pay as you go model.

But centralization gives 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 [6].

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 reading, write, send, receive, etc. do not go through the CPU. With these characteristics, RDMA has low latencies and overhead, and as such reaches better throughputs [53]. This new hardware may not be cheap, and not every CC provider uses them for every use-case. And this may not be enough, especially with the ever-growing amount of IoT devices and services.

Over the years there are more service options available, forming **everything as a service (XaaS)** model [35]. This model proposes that any hardware or software resource can be offered as a service to the users over the internet.

Table 1.4 shows common examples of SaaS, PaaS, and IaaS applications.
In recent years there is one extension of CC from a series standpoint called **multi-cloud** [54, 55].

Platform	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.

A multi-cloud environment is a such environment where an enterprise uses more than one cloud platform, with at least two or more public cloud providers, that each delivers a specific application or service. A multi-cloud can be comprised of any model presented on page 12. This model relies on the possibility that if one cloud provider fails for whatever reason, the next one will be able to serve user requests.

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

1.2.3 Membership protocol

A the start of this section we introduced DS, and we present two interesting assumptions by Tanenbaum et al. [44, 45]. If we take one more look at the **(2)** assumption, we will see that users of the DS whether they are users or applications perceive DS as a single unit. Inside this single unit, nodes need to collaborate, so that they are able to do various kinds of tasks.

The most basic of all these tasks is that nodes need to know which group they belong to, and who are their peers in the group they will collaborate with. This might sound like a trivial idea, but when we include 8 fallacies of the DS 1.2 into the equation, things start to be not so trivial, after all. In the setup where nodes are connected over the local network or internet, and they need to communicate things will go wrong for various reasons.

To resolve the problem that nodes need to know who are their group peers, a membership protocol comes to help. These protocols need to ensures that each process of one group updates its local list of **non-faulty** members of the group, and when a new process joins or leaves the group, the local list for every process needs to be updated. This is the most basic idea behind membership protocols.

Processes in the group of nodes in a group will ping each other in different ways, and using different strategies to figure out which nodes are dead and which are alive. There are few existing algorithms that do this job, and they are (usually) based on the way epidemics spread or how gossip is spread in a human population. Because of this feature, these algorithms are usually called *Gossip* style protocols.

Every membership protocol has some properties that will ensure efficiency and scalability:

(1) **Completeness**, this property must ensure that every failure in the system is detected.

(2) **Accuracy**, in an ideal world, there should be no mistakes when detecting failures. But In a real-life scenario, we need to reduce false positives as much as we can.

(3) **Failure detection speed**, all failures needs to be treated as fast as possible, in order to remove the node from the group and reschedule the tasks from dead node to alive ones.

(4) **Scale**, with this property we must ensure that the network load that is generated should be distributed equally between all processes in the group.

The easiest idea to implement this protocol would be **heartbeating** technique where process P_i will send a heartbeat message to all his peers in the group or **multicast**. After some time if process P_j did not receive a heartbeat message from P_i, it will mark him as failed. This idea is easy to understand, and implement but the downsides are that his process is not that **scalable**, especially for large groups, and this will introduce huge network traffic.

To resolve this problem, Das et al. [31] introduced **S**calable **W**eakly-consistent **I**nfection-style Process Group **M**embership protocol, or **SWIM** for short. This protocol divides the membership problem into two parts:

(1) **Failure detection**, this component works so that one node will select a random node in the group, and it will send him *ping* message, expecting *ack* message in return — **direct ping**. If such message is not received, he will pick n nodes to probe through a $ping - req$ message — **indirect ping**. If this fails, the node will be marked as *suspected*, and it will be marked as *dead* after some timeout. If the node gets alive, he will ping some other node and he will get back into the group. Figure 1.3 show message passing in **direct** ($left$), and **indirect** ($right$) ping in SWIM protocol.

(2) **Information dissemination**, with previous strategy, we can disseminate information by **piggybacking** the data on multiple messages (*ping*, $ping - req$ and *ack*), and avoid using the multicast solution.

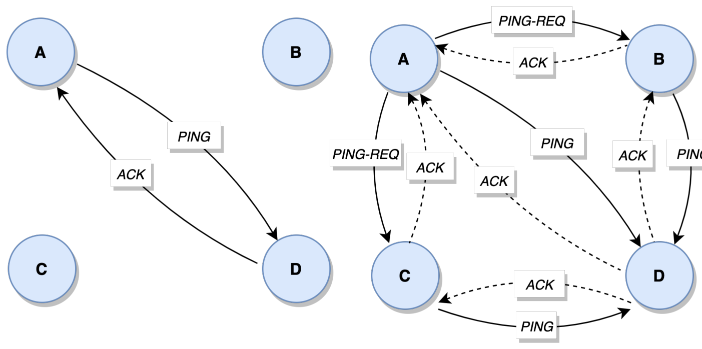

Figure 1.3: Direct and indirect ping in SWIM protocol.

Over the years, researchers found ways to improve the protocol for example Dadgar et al. present Lifeguard protocol [56] for more accurate failure detection, and there are other implementations to fine-tune the SWIM, but the base idea is still there. Today SWIM or SWIM-like protocols are standard membership protocols whenever we are doing some node clustering.

1.2.4 Mobile computing

The first idea that introduced task offloading from the cloud [57, 58] was Mobile cloud computing (MCC). The mobile devices run small client software and interact with the cloud over the internet, while heavy computation remains in the cloud.

The cloud is usually far away from end devices because DCs are built on specific locations in the world to target as many users nearby as possible. This sparse deployment will most likely lead to high latency, and bad quality of experience (QoE) [58] for most users. Latency-sensitive applications especially will have a hard time. As a model, MCC is not much different from the standard CC model. The good thing is that we relaxed the cloud a little bit, and We had moved a small number of tasks from the cloud. But this model opens the door for the next-generation models.

The development led to new computing areas like edge computing (EC). EC is a next-generation model where computing and storage resources are in proximity to data sources [13]. This idea might overcome cloud latency issues and known MCC problems. The main strength of the EC lays in the CC enhancements with new processing ideas, for the next-generation use-cases [14].

EC brought few different models over the years. Models like fog [15], cloudlets [12], and mobile edge computing (MEC) [16] emerged. This thesis will refer to all these models as edge nodes. Different EC models rely on the concept of data and computation offloading from the cloud closer to the ground [17]. Only heavy computation remains in the cloud because of more available resources [14], compared to edge nodes.

EC models introduced small-scale servers that operate between data sources and the cloud. These small-scale servers have much fewer capabilities compared to the cloud servers [18]. To avoid latency and huge bandwidth [12], EC nodes can be dispersed in various locations, for example, base stations [16], coffee shops, or over arbitrary geographic regions.

1.3 Distributed computing

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 coordination is done by passing messages to one another [45]. Computer programs that use this strategy and runs on DS are called **distributed programs** [59, 60].

Similar to CC in Section 1.2.2, to a normal user, DC systems appear as a single system similar to one he uses every day on his personal computer. DC shares the same fallacies to DS presented in 1.2.

1.3.1 Big Data

Term big data means that the data is unable to be handled, processed, or loaded into a single machine [61]. That means that traditional data mining methods or data analytics tools developed for centralized processing may not be able to be applied directly to big data [62].

New tools and methods 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 [63]. This simple idea minimizes network congestion and increases the overall throughput of the system.

In 1.1 we already give two examples of how huge generated data could be, and when we include other IoT sensors and devices these numbers will just keep getting bigger [64].

On contrary to relational databases that mostly deal with structured data, big data is dealing with various kinds of data [61, 62, 63]:

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

Along with the share size, big data have other instantly recognizable features called **V's** of big data [65]. Name is derived from starting letters from the other features that are describing big data. Image 1.4 show 6 V's commonly used to represent the big data.

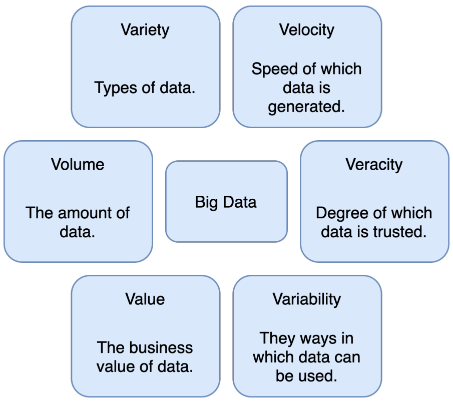
Variety
Types of data.
Velocity
Speed of which data is generated.
Volume
The amount of data.
Big Data
Veracity
Degree of which data is trusted.
Value
The business value of data.
Variability
They ways in which data can be used.

Figure 1.4: V's of Big Data.

Processing in big data systems can be represented as [66, 67]:

- **Batch processing** represents a data processing technique that is done on a huge quantity of the stored data. This type of processing is usually slow and requires time.
- **Stream processing** represents a data processing technique that is done as data get into the system. This type of processing is usually done on a smaller quantity of the data **at the time**, and it is faster.
- **Lambda architectures** represents a processing technique where stream processing and handling of massive data volumes in a batch are combined in a uniform manner, reducing costs in the process [67].

Big data systems, are not processing and value extracting systems. Big data systems can be separated into few categories: (1) data storage, (2), data ingestion (3), data processing, and analytics. All these systems aids to properly analyze ever-growing requirements [68],

Despite a promise that big data offers to derive value out of the collected data, this task is not easy to do and requires properly set up system filtering and removing data that contains no value. To aid this idea, data could be filtered and a little bit preprocessed on close to the source [21], and as such sent to data lakses [69].

1.3.2 Microservices

There is no single comprehensive definition of what a microservice is. Different people and organizations use different definitions to describe them. A working definition is offered in [70] as “s microservice is a cohesive, independent process interacting via messages”. Despite the lack of a comprehensive definition, all agree on few features that come with microservices:

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

(2) they could be developed using different languages, principles, and using different databases.

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

(4) they are organized around business capabilities [71].

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

’The industry is migrating much of their applications to the cloud because CC offers to scale their computing resources as per their usage [72]. Microservices are small loosely coupled services that follow UNIX philosophy “do one thing and do it well” [73], and they communicate over well defined API [70].

This architecture pattern is well aligned to the CC paradigm [72], contrary to previous models like monolith whose modules cannot be executed independently [70, 74], and are not well aligned with the CC paradigm [74]. Table 1.5 summarize differences between the monolith and microservices 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 its inception, microservices architecture is gone through some adaptations. And modern-day microservices are extended with two new models each with its unique abilities and problems:

- **Cloud-native applications** are specially designed applications for CC. They are distributed, elastic, and horizontally scalable systems by their nature, and composed of (micro)services that isolate state in a minimum of stateful components [75]. 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 a computing model, where the developers need to worry only about the logic for processing client requests [76]. Logic is represented as an event handler that only runs when a client request is received, and billing is done only when these functions are executing [76]. **Cold start** is one of the features of serverless computing, and we can define it as user requests need to wait until a 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 a 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 [77]. As part of the microservices ecosystem, this dedicated communication layer can provide several 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 the implementation of business logic, while operators gain out-of-the-box traffic policies, observability, and insights from the services. Advocates of the microservice movement, nowadays recommend using service mesh architecture when running microservices in production environments.

Previous models are not explicitly different, they all can be viewed as cloud-native applications. The enumeration is given for the sake of pointing out their different models and aspects of working.

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

- Representational state transfer (REST) services [78], is an architectural style with a set of constraints that users can create web services 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 relies on textual based communications, and payload could be represented using $JSON$, XML, $HTML$ etc.

- Remote procedure calls (RPC) represent an architectural way to design services that can call subroutines that are located in different places, usually on another machine. The client is calling these operations like they are located locally in his address space.

- Event-driven services are services where communication between services is done using events. Events are sent on some channel and other read messages that are received on another channel. These channels could be implemented either like message queues or message topics. Services connect to message queue or subscribe to the specific topic, and when messages arrive, they can act according to the 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. HTTP 3 is the latest version of the conventional and trusted HTTP protocol. It is very similar to HTTP 2, but it also offers a few important new features. Table 1.6 show important difference between versions of HTTP protocol.

Feature	**HTTP1**	**HTTP2**	**HTTP3**
Transport	text	binary	binary
Parallelism	No	Yes	Yes
Protocol	TCP	TCP	QUIC
Space	OS level	OS level	User level
Server push	No	Yes	Yes
Compression	Data	Data/Headers	Data/Headers

Table 1.6: Idempotent and non-idempotent operations.

To ensure a wider range of devices that can communicate with the rest of the systems, developers usually have a gateway into the system that is REST service, and other services could be implemented differently.

It is important to point out, that all flavors of microservices applications rely on continuous delivery and deployment [79]. This is enabled by lightweight containers, instead of virtual machines [80], and orchestration tools such Kubernetes [81]. These concepts will be described in more detail in Section 1.6.

Microservices architecture is a good starting point especially for building as a service applications model, and applications that should serve a huge amount of requests and users. Especially with the benefits of CC to pay for usage, and the ability to scale parts of the system independently. Although they are not necessarily easy to implement properly. There is more and more critique to the architecture model [82]. Microservices are relying upon and use parts of the DS, and as such, they inherit almost all problems DS has.

One particular thing that users need to be aware of is **idempotency**. In microservices applications, developers are dealing with inconsistencies in the 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 side effects like creation or deletion can be called any number of times while guaranteeing that side effects only occur once. Idempotency is a term that comes from mathematics, and can be represented by simple idempotency law for operation $*$ like [83]:

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

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 inconsistent states. Table 1.7 show list of idempotent and non-idempotent for standard CRUD operations:
Crate operation is not idempotent by default, but to make it idempotent there are multiple strategies for how to do so. The most common way is to create **idempotency key** that will be sent in the request, and based on that request server can decide if this operation is already invoked or not. If a server is already “seen” specified idempotency key than operation is already done and we can return just response that operation is done but no operation will be done over the state of the

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

Table 1.7: Idempotent and non-idempotent operations.

service or application. If the server sees the idempotency key for the first time, that is the signal that this request is a new one, and it should be done.

Idempotency key could be stored in any kind of 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 a specified time.

Another option that is commonly used is hashing user specified actions. This is useful to know what part of the action set is already done and what is not. This strategy is used in scenarios where we must preserve the order of actions.

The best chance to succeed when implementing a microservices architecture is to simply follow existing patterns and use existing solutions with proven quality.

1.3.3 Log aggregation

Logging is an integral part of any real-world computing system. In case of errors, fails or misbehavior of the system we can gain some insight into what causes failure or what set of parameters in which circumstances.

With this operation, developers can store various information, that will provide more details for those who are investigating the failure. One thing we must be aware to not store any sensitive pieces of information in the log because this can cause a bunch of problems. Another thing we must be aware of is that we do not log too much and too often to slow down the business logic and execution of the function.

In monolithic applications logging is a little bit easier to implement, because we have the whole application state in one place. When we come to the field of DS and microservices, our state is scattered across multiple elements or services. The solution for this problem is to use a centralized logging service that aggregates logs from each service [84]. This is beneficial because the users can search and analyze the logs as a whole state of the system. To do this properly the log must be stored very reliably [85]. The users can than configure log server for some alerts that are triggered when certain messages appear in the logs. The log of DS usually does not contain enough information to regenerate the timeline of execution, and this is one reason that logs in DS are so hard to interpret [84].

To resolve this problem of DS execution timeline, Google develops a new technique called **tracing** [86]. The trace represents a single execution timeline or execution of one request. Trace will create a tree, and the tree is used to establish order. Every node in the tree represents a unit of work and it is called span. A tree unites all the elements needed to carry out an originating request. In every span or unit of work, we can attach more details about that particular execution element.

DS logging and tracing, represent the important role of any system, and as such, it should not be neglected especially in the DS environment. Every user request should be traced and logged from an infrastructure perspective, but we should allow users to store logs from their applications.

1.4 Distribution Models

The role of distribution models is to determine the responsibility for the request, or to answer the fundamental question "who is in charge" for a specific request. There are two ways to answer this question: (1) all nodes in the system, or (1) single node in the system.

1.4.1 Peer-to-peer

Peer-to-peer (P2P) communication is a networking architecture model that partitions tasks or workloads between peers [87]. All peers are created equally in the system, and there is no such thing as a node that is more important than others.

Every Peer has a portion of 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 [87]. P2P nodes are connected and share resources without going through a separate server computer that is responsible for routing.

Figure 1.5 show difference in network topology between P2P networks (*left*) and client-server architecture (*right*).

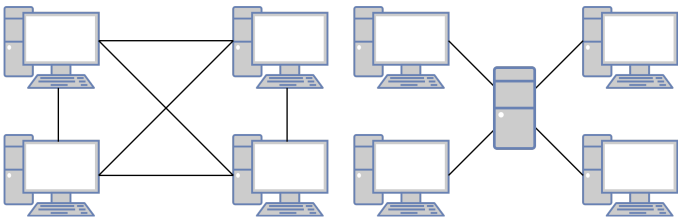
Figure 1.5: P2P network and client-server network.

Peers are creating a sense of virtual community. This community of peers can resolve greater tasks, beyond those that individual peers can do. Yet, these tasks are beneficial to all the peers in the system [88]. When a request comes to such a network, a node that accepted request is usually called **coordinator**, because he then is trying to found the right peer to send a request to.

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 [89]:

- **Unstructured** do not have a particular structure by design, but they are formed by nodes that randomly form connections [90]. Their strength and weakness at the same time is the lack of structure. These networks and robust when peers join and leave the network. But when doing a query, they must found more possible peers that have the same piece of data. A typical example of this group is a Gossip-based protocol like [31].
- **Structured** peers are organized into a specific topology, and the protocol ensures that any node can efficiently search the network for a resource. The famous type of structured P2P network is a Distributed Hash Table (DHT). These networks maintain lists of neighbors to do a more efficient lookup, and as such, they are not so robust when nodes join or leave the network. DHT commonly used in resource lookup systems [91], and as efficient resource lookup management and scheduling of applications, or as an integral part of distributed storage systems and NoSQL[92] databases.
- **Hybrid** combine the previous two models in various ways.

P2P networks are a great tool in many arsenals, but because of 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 [93].

1.4.2 Master-slave

In the master-slave architecture, there is one node that is in charge – **master**. This node accepts requests, and we usually do not communicate to the rest of the nodes or **slaves**. The master node is usually better and more expensive or even specialized hardware such as RAID drives to lower the crash probability. The cluster can also be configured with a **standby** master, and this node is continually updated from the master node.

But no matter how specialized hardware master runs on, it is prone to fail for various reasons, so he is a **single point of failure**. If crush happened, then standby master could continue to the server as a master, or new **leader election** protocol [94] is initiated to pick a new master node.

The master node is responsible for processing any updates to that data. If the master fails, then the slaves can still handle **read** requests. Failure of the standby master node, to take over from the master node is a real problem if we want to achieve a high-availability system.

Figure 1.6 show difference between mater-slave ($left$) and peer-to-peer ($right$) request handling.

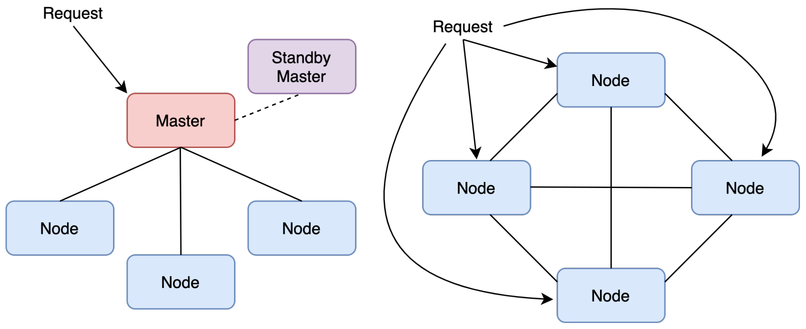

Figure 1.6: Handling requests master-slave and peer-to-peer

Using the right distribution model usually depends on the business requirements. High availability requires a P2P network because no single point of failure. If we could manage data using batch jobs that run in off-hours, then the simpler master-slave model might be the solution.

1.5 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.

1.5.1 Parallel computing

DC and parallel computing seem 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 [59].

Distinguished between the two can be presented as follows: in parallel computing, all processor units have access to the shared memory and have some way of the faster inter-process communication, while in DS and DC all processors have their memory on their machine and communicate over the network to other nodes which are significantly slower.

These models are similar, but they are not identical, and the kinds of problems they are designed to work on are different. Figure 1.7 visually summarize the architectural differences between DC (*up*) and parallel computing (*down*).

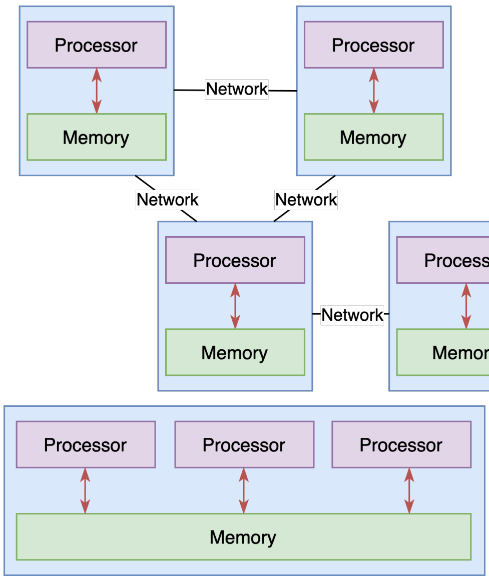

Figure 1.7: Architectural difference between DC and parallel computing.

Parallel computing has often used the strategy with problems, that due to their nature or constraints must be done on multi-core machines simultaneously [95]. It is often, that some big problems are divided into smaller ones, which can then be solved at the same time.

Several tasks require parallel computing like simulations, computer graphics rendering, or different scenarios in scientific computing.

1.5.2 Decentralized systems

Decentralized systems are similar to DS, in a technical sense, they are still DS. But if we take a closer look, these systems **should not** be owned by a single entity. CC, for example, is a perfect example of DS, but it is not decentralized by its nature. It is a centralized system by the owner like AWS, Google, Microsoft, or some other private company because all computation needs to be moved to big DCs [6].

By modern standards, when we are talking about decentralized systems, we usually think of blockchain or blockchain-like technology [96]. Since here we have distributed nodes, that are scattered and there is no single entity that owns all these nodes. But even if this technology is run in the cloud, it is losing the decentralized feature. This is the caveat we need to be aware of. These systems are facing different issues because any participant in the system might be malicious and they need to handle this case.

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

1.6 Virtualization techniques

Virtualization as a technique started long ago in time-sharing systems, to provide isolation between multiple users sharing a single system like a mainframe computer [97].

In [98] Sharma et al. describe virtualization as technologies that provide a layer of abstraction of the physical computing resources between computer hardware systems and the software systems running on them.

Modern virtualization differentiates 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 packaging and distribution format, or are new and still are looking for their place. These options are:

- **Virtual machines (VM)** are the oldest technology of the three. In [98] Sharma et al. describe them as a self-contained operating environment consisting of guest operating system and associated applications, but independent of the host operating system. VMs enable us to pack isolation and better utilization of hardware in big DCs. They are widely used in IaaS environment [99, 100] as a base where users can install their own operating system (OS) and required software tools and applications.

- **Containers** provide the almost same functionality to VMs, but there are several subtle differences that make them a goto tool in modern development. Instead of the guest OS running on top of host OS, containers use tools that are in a Linux kernel 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 [101, 80]. Containers provide an easier way to pack services and deploy and they are especially used in microservices architecture and service orchestration tools like Kubernetes [81]. Google stated few times in their on-line talks that they 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** is the newest addition to the virtualization space. In [102] 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 a single process and they contain and execute what it absolutely needs to nothing more and nothing less [103]. They are advertised that new technology that will save resources and that they are *green* [104] meaning they save both power and money. When put to the test and compared to containers they give interesting results [103, 105]. Unikernels are still a new technology and they are not widely adopted yet. But they give promising features for the future, especially **if** properly ported to ARM architectures, and various development languages. Unikernels will probably be used as a user applications and functions virtualization tool, because of their specific architecture, especially for serverless applications presented in 1.3.2.

With every virtualization technique, the ultimate goal is to pack as many applications on existing hardware as possible, so that there are no resources that are left not used – we are trying to achieve high resource utilization. Figure 1.8 represent architectural differences between VMs, containers, and unikernels.

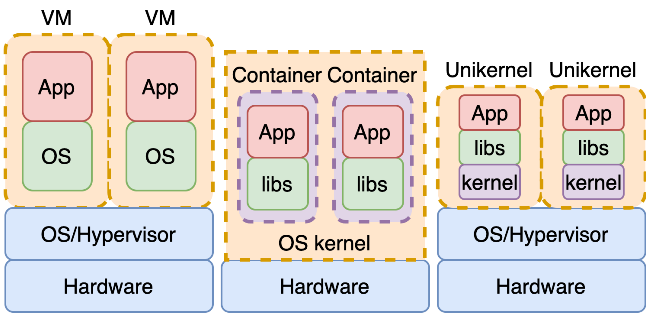

Figure 1.8: Architectural differences between VMs, containers and unikernels.

1.7 Deployment

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

Here evolve a new strategy to manage and deploy complicated infrastructure elements – Infrastructure as code (IaC). In his book [106] 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.

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

- **Mutable model**, is a model where we have in place changes which mean that the parts of the existing infrastructure or applications get updated or changed to do an update. In place, change can produce few problems: (1) more risk because in-place change may not finish which puts our infrastructure or the application in a possible bad state. This is especially a problem if we have a lot of services and multiple copies of the same service. The possibility that our system is not on is a lot higher, (2) high complexity, this is a direct implication of the previous feature. Since our change might not get fully done, we cannot give a guarantee that our infrastructure or application is transitioned from one version to 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 a model where we do not do any in-place changes on existing infrastructure or application whatsoever. In this model, we replace it completely with a new version that is updated or changed compared to the previous version. The previous version gets discarded in favor of the new version. Compared to the previous model, immutable deployment has: (1) less risk, since we do not change existing infrastructure or the application but we start a new one with and shut down the previous one. This is important especially in DS where everything can fail at any time, (2) previous property reduces the complexity of the mutable deployment model. This is a direct implication of the previous feature, since we shut down and fully replace the previous version with the new one we get **descrete** version change and atomic deployment with deferring deployments with fast rollback and recovery processes. On the other hand, this process requires more resources [107], since both versions must be present on the node for this process is done. The second problem is data that is used by the application, we should not lose the data that the application is generated. If we externalize data then this problem is resolved. We should not rely on local storage but store that data elsewhere, especially when the parts of the system are volatile and changed often. The key advantage of this approach is avoiding downtime experienced by the end-user when new features are released.

Figure 1.9 summarize difference bewteen prevous infrastructure deplyment models.

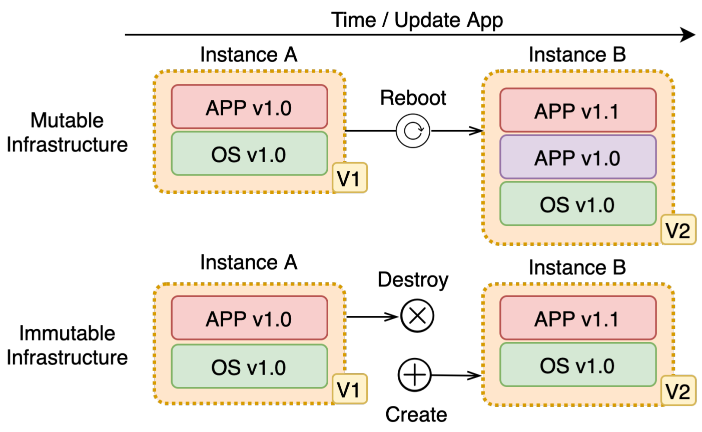

Figure 1.9: Difference bewteen mutable and immutable deplyment models

Immutability is a simple concept to understand and simplify a lot especially in DS [107]. Write down some data, and ensure that it never changes. It can never be modified, updated, or deleted [108]. When this is combined with the promise that we can avoid downtime especially in complex DS, it is clear why the immutable model is gaining more and more popularity (especially with the arrival of containers). Immutable infrastructure deployment offers few models on how to deploy change on the services, even in production to test it or switch to the whole new version. These strategies include:

- **Blue-Green deployment**, this strategy require two separate environments: (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 the number of requests and monitor how the service is working with increasing load, monitor for errors, etc.

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

With mutable infrastructure, these strategies would be hard to implement, and maybe it is not possible at all. Besides infrastructure deployment, there is another side that we must consider, and that is how to 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 on what specific tool need to do so that the application or infrastructure is properly setup. In this approach, we have a *smart* user who describes *dumb* machine what is needed to be done and in what order to achieve the desired state.
- **Declarative**, with this option user, has to describe the end state or what is his desired state, and the tool needs to figure out the way how to do this. Here we have *smart* system that will found a way how to achieve the desired state, and we have user who*do not care* in what order actions need to be done — that is what the system needs to do. Users do not need to worry about timing, this simplifies the whole process and the 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 a new domain-specific language that users need to learn, but we might be able to optimize description.

With the introduction of *LinuxKit*, we can create Linux subsystems based around very secure containers. With linuxkit, every part of the Linux subsystem is running inside the 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 [109] than general-purpose systems. This is important from the security point of view, but also infrastructure deployment because we can compose specific OS based around containers that we need for a different purpose. And we can update, change and adopt these OS for every machine or purpose we need.

Deployment is based around changing 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.

1.8 Concurrency and parallelism

People usually confuse these two concepts. Even they look similar, they are a different way of doing things. In his talk, Rob Pike [110] give great explanation and examples on this topic. In his talk, he gives great definitions of these concepts like:

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

These things are important, especially 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 enable possible parallelism, but with communication [110]. These ideas are based on Tony Hoare work of Communicating Sequential Processes (CSP) [111].

1.8.1 Actor model

An 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 [112]. This model proposes a similar idea to Tony Hoare in his work with CSP [111], and actors are often confused with CSP. Table 1.8 give differences between actor model and CSP.

Actors do not share a memory, and they are isolated by nature. An actor can create another actor/s and even watch on them in case they stop unexpectedly. And when an actor finished his job, and he is not

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

Table 1.8: Ddifferences between actor model and CSP.

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 message passing mechanism.

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

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

1.9 Motivation and Problem Statement

In their work [23] 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 rarely suits all needs, so the CC should not be our final computing shift. Various models presented in 1.2.4 show a promising possibility of how computing could be done closer to the data sources, to lower the latency for its clients by contacting the cloud only when needed. While the heavy computation could remain in the cloud, because of more available resources. Send to the cloud only information that is crucial for other services or applications [21]. Not ingest everything as the standard cloud model proposes.

A zonally-based organization of servers combined with MDCs shows a great possibility for building micro-clouds and EC as a service. To achieve such a behavior, we need a few more abstractions and layers, to make the whole system more available, resilient, and with less latency. By their nature, EC originates from P2P systems [10] as suggested by López et al., but expands it into new directions and blends it with the CC. This very interesting, because we already have much research and knowledge available for P2P systems that we could use for inspiration. But one extension of the P2P system leads to geo-distributed deployments, and going from node to node is a time-consuming process. Satyanarayanan et al. stated that infrastructure deployment will not happen until the whole process is relatively easy [39].

If we assume that we have a well-defined system and the previous task is solved so that we can easily deploy infrastructure and operate with it, we will have a system that could be offered like any other resource in the CC – *as a service*. Such a system or service could be offered to various types of users, from researchers to developers to create new types of applications. A well-defined system that is easy to operate with, will be able to move resources from one place to another with no problem. We also must be aware, that some cloud providers might choose to integrate the system into their existing CC platform to reduce the load or avoid bottlenecks and single points of failure [113].

The idea of small-scale servers introduced by EC, with heterogeneous, compute, storage, and network resources, raise interesting research ideas and it is the main motivation for this thesis. Taking advantage of resources organized locally as micro clouds, community clouds, or edge clouds [22] suggested by Ryden et al., to help power-hungry servers reduce traffic [114]. Contact the cloud only when needed [21]. 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. At any given time, we must know available resources, configuration, and utilization [36, 16] of the system. Traditional DCs is a well organized and connected system, built upon years of experience. On the other hand, these MDCs consist of various devices, including ones presented in 1.2.4 that are not [37]. This idea brings us to the problem this thesis address.

Currently, existing EC and MDCs models lack dynamic well defined geo-organization structure, native applications model, and a clear separation of concerns model. As such they cannot be offered as a service to the users, and it is hard to form micro-clouds on them. Usually, these systems exist independently from one another, scattered without any communication between them. Providers who build and maintain these systems, usually lock users in their ecosystem frequently integrate tightly with their cloud services giving users small or even no other options. Nearby EC nodes could be organized locally, making the whole system more available and reliable, but at the same time extending resources beyond the single node or group of nodes, maintaining good performance to build servers and clusters [20].

This cloud extension strengthens our understanding of not just DS but also CC as a system and field of research. With EC native applications model, separation of concerns well defined, and a unified node organization strategy, we are moving towards the idea of EC as a service and micro-clouds. Based on this, we define the problem this thesis is trying to solve, through the three research questions:

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

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

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

This micro-cloud model makes both system and applications more available and reliable, while resources are extended further beyond the single node or even MDCs. IN his work [19] Satyanarayanan et al. show that MDCs can serve as firewalls, while users get a unique ability to dynamically and selectively control their information that will be uploaded to the cloud. Simić et al., in [21] use a similar idea as a pre-processing tier for the cloud. Years after its inception, EC is no longer just an idea [19] but a must-have tool for novel applications to come.

1.10 Research Hypotheses, and Goals

Based on research questions presented on a page 43, and motivation presented in a section 1.9, we derive the hypotheses around which we base this thesis. We can summarize them as follows:

(1) **Hypothesis:** *It is possible to organize EC nodes in a standard way based on cloud architecture, adapted for EC geo-distributed environment –* ***micro-clouds****. Giving users the unique ability to organize nodes, descriptively, in the best possible way to serve the only population nearby.*

(2) **Hypothesis:** *It is possible to offer a newly formed micro-cloud model to researchers and developers as a service based on the cloud pay as you go model to create new human-centered applications. But sustain the ability to rearrange resources as needed.*

(3) **Hypothesis:** *It is possible to form a clear separation of concerns for the future micro-cloud model (EC as a service model) and establish a well-organized system with an intuitive role for every part.*

(4) **Hypothesis:** *It is possible to present a unified model that will be able to support heterogeneous small-scale servers (EC nodes). This unified model will rely on a set of technical requirements that nodes must fulfill to join the system.*

(5) **Hypothesis:** *It is possible to present a clear and familiar application model so that users can use the full potential of newly created infrastructure but familiarly and intuitively.*

From the previously defined hypotheses, we can 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 model is described formally well, using solid mathematical theory, but also easy to extend both formally and technically, easy to understand and reason about. This addresses the third research question and is the topic of Chapter 3.*

1.11 Structure of the thesis

Throughout this introductory Chapter 1, we defined the motivation for our work, with problems that this thesis addresses. Furthermore, we presented the necessary background details, information, and areas required for understanding and support the rest of the thesis. 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 organizational abilities for nodes in both industry and academia frameworks and solutions to address our first research question. We further examine 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 required for such a system to be implemented formally. We give examples of how existing infrastructure could be used, as well as familiar application model for developers.

Chapter 4 presents implementation details of a framework developed to test hypotheses defined earlier in this chapter, but also a model and formally defined protocols defined in 3. This chapter also shows results after conducting experiments, current limitations of the implemented system, and possible applications that could benefit from such a system.

Chapter 5 is the final chapter, and it concludes the work of this thesis, outlines the opportunities and directions for further research and development in this area.

Chapter 2

Research review

In this chapter, we present the results of our research reviews addressing issues and limits of CC discussed earlier. Both academia, and the industry researching and developing viable solutions to help cloud in the future.

Few direction are feasible: (1) focusing on adapting existing solutions to fit EC model, (2) experiment and developd new ideas and solutions to maybe fit more the nature of EC, and (3) try to combine both ideas to cover more research area.

in Section 2.1 we review existing nodes organizational abilities. in Section 2.1 we review platform models from both industry and academia. In Section 2.3 we review cloud offloading techniques. In Section 2.4 we review some applications models. Section **??** conclude this chapter, and give the position of this thesis, compared to research previously reviewed.

2.1 Nodes organization

Guo et al. [25], gives a promising model based on a zone-based organization of edge nodes in the smart vehicles application. The authors show how zone-based nodes organization enables continuity of dynamic services and reduces the connection handovers. They prove it is possible to enlarge the coverage of edge servers to a bigger zone, but at the same time, we could expand the computing power and storage capacity of edge servers. EC should be based on the geo-distributed workloads, and this zone-based organization could benefit the EC model in various ways.

In their research [115], Baktir et al. explored the capabilities of software-defined networks (SDN) from a programming stand point. Their findings show that SDN can be used to simplify the management of the network in a cloud-like environment. Networking is such a complex environment is not an easy task to achieve. The authors show how SDN hides the complexity of the heterogeneous environment from the end-users. As such, SDNs represents a good candidate for networking tasks in complex, cloud-like environments.

Sayed et al. in their work [43] show that EC systems will perform actions before connecting to the cloud and how they are easier to integrate with other wireless networks like mobile ad-hoc networks (MANETs), vehicular ad-hoc networks (VANETs), intelligent transport systems (ITSs) and the IoT to mitigate network-related and computational problems.

Content delivery networks (CDN) in centralized delivery models like CC have bad scalability, as Kurniawan et al. [26] argue in their research. To overcome these centralized problems and bad scalability, the authors proposed a different solution, a decentralized solution. To achieve such tasks, authors were using a network of gateways equipped with some storage as well, for internet services at home [26] forming even smaller DCs – nano DCs. Authors present a possible usage for these nano DCs in some large scale applications with much less energy consumption than traditional DCs.

In their paper [116], Ciobanu et al. introduces an interesting idea called *drop computing*. The authors show the possibility for ad-hoc EC platform composition using a decentralized model over multilayered social crowd networks. This idea gives us the ability for collaborative computing that can be formed dynamically. The authors present an idea that we can form a computing group ad-hoc by employing the nearby devices in the mobile crowd, that is fully capable of quick and efficient access to resources, instead of sending requests to the cloud. Crowd nodes could also be an interesting idea for backup nodes, in cases we need more computing power or storage and there are no more available resources to use. Forming platforms from crowd resources and ad-hoc, raise a few concerns: (1) crowd nodes availability, and (2) offered resources.

Greenberg et al. [23] introduce the idea of MDCs as DCs that operate in proximity to a big population compared to nano DCs that serves a lot smaller population, for example, a single household. MDCs are an interesting model and area of rapid innovation and development, and because they are close to some population, they are minimizing the costs and the latency for end-users [117, 23], Their minimum size is defined by the needs of the local population [23, 24], as such, they are reducing traditional DCs fixed costs. The main feature that MDCs are relying on is agility. The authors describe agility as MDCs ability to dynamically grow and shrink resources to satisfy the resource demands and usage from the most optimal location [23]. In [118] Shao et al. present a possible MDCs structure serving only the local population, in the smart city use-case.

2.2 Platform models

Kubernetes (k8s for short) [81] is a system originally developed by Google influenced by their orchestrator platform called Borg [119]. Various other companies joined in developing this system, and now it is de facto standard in the cloud environment for microservices and cloud-native applications. By design, Kubernetes is not intended to operate in a geo-distributed environment, because he operates on a single cluster [81, 119, 120]. As such it might not be best suited for

EC and geo-distributed micro-clouds. Nonetheless, it is a super valuable tool in the CC because it enables health checking, restarting, and orchestration at scale. Another potential problem with Kubernetes is his relatively complicated deployment concept that might be too complicated for EC workloads. It is developed to connect microservices across the CC environment. But we could use them as he is, a cloud-native orchestrator to run the master process for EC and micro-clouds and also cloud-native applications that will accept streams coming from micro-cloud applications. Kubernetes relies upon a lot of stuff to work properly. So we should not ignore existing technologies and ideas that are worth exploring, such as loosely coupling elements with labels and selectors for example.

In their research, Rossi et al. [120] present a solution based on Kubernetes. Authors adapted Kubernetes for workloads that are geo-distributed and they had used reinforcement learning (RL) techniques, to learn a suitable scaling policy from past experience. There are potential downsides to this approach. The first might be that machine learning implementation could be potentially slow due to the required model training, but also we need to somehow describe what is good and what is a bad decision in order that some algorithm to learn. This might be a problematic thing, especially in urgent situations. The second potential problem is that, even though Kubernetes is a promising solution and de-facto standard in the CC environment, as previously stated it might not be the best proposal for EC and geo-distributed micro-cloud environment. Despite all these potential problems, researchers show great work in adopting Kubernetes architecture to work for geo-distributed workloads.

Ryden et al. [22] presents an interesting platform for distributed computing, that is more oriented towards user-based applications. Compared to other similar systems. their goal was not to develop a solution that will do resource management policy, on the contrary, their focus is more oriented to give flexibility to the users for application development. Users can develop their applications using exclusively

Javascript (JS) programming language, with some embedded native code for a more efficient solution. The authors rely on a bunch of volunteer nodes to run all the tasks and applications, similar to work presented in [116]. The main difference between these two solutions is that Ryden et al. make a split on which nodes are storage nodes, and which nodes are used for calculation and processing tasks. Their application environment is protected from malicious code, using sandboxing techniques. This presents an interesting work to show how users can develop applications and run them in an EC environment.

In [121] Lèbre et al. show an interesting solution based on extending and adopting the OpenStack system. OpenStack is a free and open standard cloud computing IaaS platform for CC use cases in both public and private clouds. The authors tried to manage both cloud and edge resources using a NoSQL database. Massively distributed multi-site IaaS, using OpenStack is a challenging task [121] to implement, because the communication between nodes of different sites can be subject to important network latencies [121]. On the other hand, if it could be done properly we are gaining one major advantage that users of the IaaS solution can continue using the same familiar infrastructure for both cloud and edge/fog use-cases.

Based on the literature survey, the Ning et al. presents current open issues of EC platforms [14]. In their work, authors focus on different aspects of EC systems, and they outline the importance of EC and CC tight collaboration. The CC needs to be unloaded and EC nodes could provide data pre-processing. On the other hand, EC needs massive storage and a strong computing capacity of CC. The authors illustrate the usage of edge computing platforms to build specific applications and conclude that with CC and EC integration, both sides will benefit.

In [109] the de Guzmán et al. present solution based on Kubernetes that use Kubernetes Deployment Manifests to reuse successful principles from Kubernetes by creating a 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 represents interesting usage of LinuxKit to deploy OS dependencies and immutable infrastructure patterns, but VMs might be a bit problem for small devices, and ARM nodes as well as the complex flow of the Kubernetes application model. Nonetheless, it is an interesting extension of the Kubernetes framework and proves that LinuxKit can be used for immutable infrastructures with custom OS.

In [122] Sami et al. show an interesting platform for dynamic services distribution 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.

Besides academy efforts, the industry as well introduced a few interesting platforms and frameworks for EC. For example, Amazon introduced their framework Greengrass [123] that can run on various hardware to do some processing. Amazon turns to the option that their framework is deeply connected to the rest of the AWS cloud ecosystem. KubeEdge [124] is a lightweight extension of the Kubernetes framework, to operate in an EC environment. But same as regular Kubernetes, all workloads are done in the domain of a single cluster which might not be the best solution for geo-distributed micro-clouds. Both Greengrass and KubeEdge are frameworks that are mainly used for user-based applications. On the other side, we have General Electric with their Predix [125] platform. Predix 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 [17], while heavy computation remains in the cloud because of resource availability [14].

Offloading is an effective strategy when using cloud services. Relying only on cloud services may be prone to introduce long latency, which some applications cannot tolerate. On the other hand, mobile devices and sensors do not have sufficient battery energy for task offloading [126]. 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 [117, 17, 18, 58, 37, 126] 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 battery. After the computation is done, this edge layer sends pre-processed data to the cloud for further analysis, storage, etc.

When using task offloading techniques, it is very important to have good QoS that users can rely on. In [122] authors used Evolutionary Memetic Algorithm (MA) to solve their multi-objective container placement optimization problem to achieve better QoS.

2.4 Application models

Sayed et al. in their work [43] describe that EC follows a decentralized architecture model and that data processing at the edge of the network, thus it enables nodes to make autonomous decisions. So the applications written for EC can perform actions locally before connecting to the cloud at all. This will have some benefits like reducing network overhead issues as well as the security and privacy issues.

As we already mention on page 50, Ryden et al. [22] presents an interesting user-oriented platform for distributed computing called Nebula. In this section, we are going to dissect their research but from a different angle. Nebula allows users to develop their applications using JS exclusively, due to the usage of Google Chrome Web browser-based Native Client (NaCl) sandbox [127] that can run JS code only. Restriction on a single language might be a problem for some users and use-cases even though JS is a popular language at the moment. On the other hand, virtual machines tend to be too resource-demanding packing stuff that might not be needed, so a solution using containers or unikernels might provide better resource utilization and pack more services per node than virtual machines.

In [39] Satyanarayanan et al. represent an interesting view on cloudlets s a “data center in a box.”. They give an example that cloudlets should support a wide range of users, with minimal constraints on their software. They put emphasis on transient VM technology. The emphasis on transient VMs is because cloudlet infrastructure is restored to its pristine software state after each use, without manual intervention. In the time they conduct their research containers might not be working solution or it might be hard to use them. By modern standards, containers may even fit better, and pack more user software on the same hardware. This may be the case for the unikernels, once they reach a wider adoption rate and stable products.

Various Kubernetes variants lik [124, 120], give users the possibility to run different applications like web servers and databases even on smaller devices creating green DC [20].

Satyanarayanan et al. [19] propose the concept of edge-native applications that will separate space into 3 layers or tiers. Tier (1) represents various mobile, IoT devices autonomous vehicles, etc, and these devices produce a lot of data. Tier (2) represent applications running in cloudlets or other EC models, that will be able to do some pre-processing, or data filtering before it goes further. Finally, tier (3) represents classic cloud applications that will accept pre-processed and filtered data from the previous tier do more processing, react on some values, or store for future use. This idea represents an interesting concept and gives wide space for users and application development.

In [128] Beck et al. argue that applications should use message bus, streams, or topics because most mobile or edge applications are expected to be event-driven. The message bus system is an interesting proposition because the virtualized applications can subscribe to message streams, i.e., topics, and act only when data arrive. Applications might not be alive the whole time. And if for some reason mobile edge applications cannot reach a close EC server, it can always send data to the cloud. So cloud applications should be changed so slightly, just to cover this edge case.

In [113], Jararweh et al. show how integration between EC with CC principles will create more complex services and applications at the edge of the network opening new possibilities for applications to reduce the load on the centralized cloud model but also avoid bottlenecks and single points of failure.

2.5 Thesis position

In the previous sections, we described different aspects of EC and integration with the CC, influential research, interesting concepts, and implementations.

The focus of this thesis is a little bit different from the aforementioned work. With this thesis, we want to make the connection between CC and EC stronger represented as a system that can descriptively and dynamically organize geo-distributed nodes that already exist over an arbitrary vast area into one coherent system. This approach is not fully addressed in other solutions.

This thesis is influenced by the organization of CC and their big DCs but adapted for a different environment such as EC and micro-clouds, but also influenced on work described in previous sections.

Adaptations that are required for such tasks must be followed by a clear Separation of concerns (SoC) model and intuitive applications model so that users can fully use new-formed infrastructure properly.

All these allow will us to push the whole solution more towards EC as a service and micro-cloud model that can help CC with latency issues with new-age applications.

Chapter 3

Micro clouds

In this section, we are going to present a core idea of this thesis from point of view how such a model can be formed and described, and what is important so that such a system operates properly. We will describe a micro-service model, based on the CC organizational model. The presented model will be able to support EC as a service, amongst other applications and use-cases.

Throughout this section, we are going to rely on research presented in chapter 1 and chapter 2 and make a connection with the existing CC model.

In Section 3.1 we present a high overview of the system. We present an architecture that is influenced by the standard CC model but adopted for a different environment. In Section 3.2 we introduce the separation of concerns model, for previously defined system architecture. In Section 3.3 we present a possible application model and how users can utilize newly created architecture fully. Application models are based on existing development models, so that transition is fairly easy. In Section 3.4 we discuss how this system could be offered as a service, and what options can be offered to the potential users for developing their applications. Section 3.5 presents desired option for infrastructure and application deployment, and why immutability is important in a micro-cloud geo=distributed environment. In

Section 3.6 we present why formal models are important in computer science and DS in particular, as well as formal models of all protocols used in system formation and operation. Section 3.7 show access patterns that could be used in micro-clouds and geo=distributed environments. Section 3.8 gives repercussion of the proposed model, and how it can be used as a stand-alone model where other features could be implemented on top of that or used as service for other, existing, systems. Finally, we conclude this chapter wi section 3.9 present limitations of our model.

3.1 Configurable Model Structure

In his work [19] Satyanarayanan et al. propose a new architecture pattern and separation into the three tiers, where every tier or layer has a distinct role. This idea is a very powerful one because we can now split and optimize applications into parts for a specific role in the global picture. On the other hand, too many moving parts meaning more problems, and the whole system is potentially more prone to errors and failures.

If we take the previous model and combine it with MDCs and a zonally-based server organization we get a good starting point for building micro-cloud infrastructure, and EC as a service. This extension is an interesting move because we can then extend the computing power and storage capacity that is serving the nearby population. This base model is just a starting point that is promising but to make a fully functional model, we need to make it more available and resilient with less latency. To achieve such behavior, we need to extend these concepts and adapt them for different usage scenarios but never losing a geo-distributed idea from our vision.

To extend the system in a new direction, we can look for some inspiration in existing systems that are proven and working. This is especially important in DS, so we want to lower down the complexity and avoid known problems by sticking to existing models and algorithms. When we observer the CC design, we can see that every single part in that system, contributes to a more resilient and scalable system. On the high level, the CC architecture is separated into few building blocks that make the whole system lot easier to understand, maintain and operate.

The first building block of CC architecture is a cluster, and a cluster can be defined as a set of nodes or servers that operate as a single unit to achieve some goal. This is where resources are, and this is where user applications are running on. The next building block that consists of multiple clusters is called Regions (or DCs). Regions are isolated and independent from each other, and they contain resource application needs. These resources come in form of clusters. Regions are usually composed of a few availability zones [27]. These zones are the defense against the fail. If for whatever reason, one zone fails or goes offline, there are still more of them to serve user requests, we have better availability, scalability, and resilience.

If we now observer micro-clouds as geo-distributed systems, we can use a similar model, with some adaptations. Rely on a similar proven strategy, do not reinvent the wheel, but adopt for the different use-case.

Table 3.1 present similar concepts between CC and edge-centric computing (ECC) concepts. Here we want to put an accent on the difference between the physical logical concepts in the two models.

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.

Since we are talking about geo-distributed systems, our scenario is a little bit different than one used in the standard CC model. We can still combine multiple EC nodes into clusters, that is what MDCs already propose. If we want to go a little bit further, we can then combine multiple node clusters into a next bigger logical concept of *region*. A region will increase the availability and reliability of both the system and applications.

But the region in CC and ECC is not fully the same thing. In the standard CC model, the region is a physical thing or element of the existing infrastructure [27], while in the ECC a region could be view differently, not as a physical element but rather as a logical element. We can now give a formal definition of a *region* in ECC as:

Definition 3.1.1. *In a geo-distributed environment like ECC and micro-clouds, a concept of region is used to describe a set of clusters (that could be) scattered over an arbitrary geographic region. Regions are fully capable to accept or release clusters in the same way that clusters can accept or release nodes.*

In MDCs, a cluster is as big as the population nearby that is using it [23]. If we combine this with the previous definition than we can formally define the minimum size of an ECC region as:

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

With the previous definition, we have to be careful to not introduce huge latency in the system. To lower the region latency, the vast distances between clusters should be strongly avoided in normal circumstances. To extend the CC region, we need to bring new nodes and connect them physically to the rest of the infrastructure [28], while in the ECC we should extend the region just by changing the definition to whom specific cluster belongs.

Multiple regions should be able to form a second logical layer – *topology*. In the geo-distributed systems such as ECC or micro-clouds, we can formally define a *topology* as:

Definition 3.1.3. *In geo-distributed systems, topology represents the highest logical concept that is composed of a minimum of one region and could span over multiple regions. Topology is fully capable to accept new or release the existing regions.*

With these simple abstractions, we can easily cover any geographic region with the ability to shrink or expand clusters, regions, and topologies. Size and formation of *clusters*, *regions*, and *topologies* should be a matter of need, agreement, and usages of nearby population similar to the size of MDCs [23], and modeling in Big Data systems [29, 30]. This separation and organization give us one interesting feature. We can follow a more natural administrative division of some region, and organize resources by population usages.

The organization of these concepts should be fully optional. So for example, we could fit clusters in an interval of nano-DCs [26] and MDCs [23] or as wide as the whole city or small as all devices in a single household and everything in between. Formaly, we can represent the size of some cluster c like:

$$c \in [nanoDCs, MDCs] \tag{3.1}$$

If we go a little bit further, we can represent the city as one region, where parts of the city are organized into clusters. We can even form a city topology by splitting the city into multiple regions containing multiple clusters, and ultimately or we can form a country topology by splitting the country into regions, with cities being clusters.

Nodes that belong to the same cluster should run some form of membership protocol presented in 1.2.3. Gossip style protocols, like SWIM [31] (cf. page 16), are standard in the cloud environment. The same strategy could be applied here, used in conjunction with replication mechanisms [32, 33, 34] making the whole system more resilient.

Replication could be used not only in nodes inside the cluster, but we can also replicate data in clusters inside the region and even in regions inside topology. This property should exist, but it should be controlled by users depending on how resilient and available the system he wants and needs. In [129] Simić et al. take a look from a theoretical point of view on CRDTs usage, to achieve SEC in EC. the authors conclude that CRDTs could be a natural fit to EC as long as we are aware of the potential pitfalls of CRDTs.

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

These micro-clouds have much fewer resources, compared to standard clouds but they are much closer to the user meaning they have a much faster response. In the case of storage, if data is not present at the time of user request, they can pull data from the cloud and cache it for later.

Three-tier architecture with numerous clients in the bottom, micro clouds in the middle, and cloud on the top kinda resemble cache level architecture in CPU [130].

On lower levels, we have the 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 the upper tiers, we have more and more storage capacity and processing power, but contrary the response time is higher and higher. Especially when we consider distance, and huge volumes of data that need to be moved to the cloud.

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

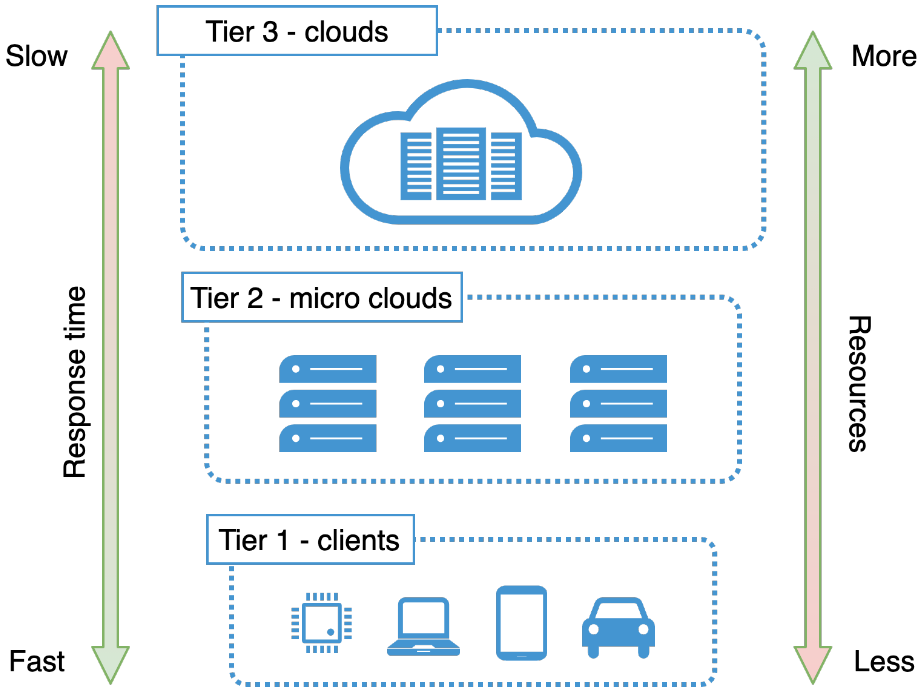

Figure 3.1: 3 tier architecture, with the response time and resource avelability

In everything as a service model [35], ECC as a service fits in between CaaS and PaaS, depending on the user needs.

3.2 Separation of concers

In his work, Jin et al. [38] introduces three core concepts to fully describe physical services, and specifies their relationships. Concepts that authors propose are: (1) Devices, (2) Resources, and (3) Services. This separation is interesting because we can rely on it in a geo-distributed environment, to describe SoC for micro-clouds.

One of the most important of every system is the SoC model. This is especially important if we are going to create a platform to offer as a service. With some adaptations, we can base our SoC model on concepts proposed by Jin et al. in three layers as depicted in Figure 3.2.

The layer on the very bottom of the three-tier architecture (cf. Figure 3.2) consists of various devices. These devices are important because they represent main *data creators* but at the same time, they are main *services consumers*.

The layer in the middle (cf. Figure 3.2) represents resources or EC nodes. These resources have a very important spatial feature, and as such, they indicate the range of their hosting devices [38]. This means that the developers at any given time must know the topology of the system, the resource spread, and utilization across clusters. Besides that main information, users must know the state and health of every application. . If some EC node desire to be part of the system, a node must obey four simple rules:

(1) a node must run an operating system with a usable file system;

(2) a node must be able to run some isolation engine for applications, for example, containers or unikernels;

(3) a node must have available resources for utilization so that we can run applications or store data;

(4) a node must have a stable internet connection;

One thing that is important to notice, is that all nodes in clusters, regions, and topologies are **equal** and there are **no special nodes**. Every node that joins the system must be able to store and process information.

Last but certainly not least layer represent services. These services expose resources through some interface and make them available over the internet [38]. These services respond to the client requests immediately, if possible, or cache information [39, 40] for future use. Unlike the previous two layers, services have one specific feature and span over two tiers of the system (cf. Figure 3.2):

(i) Services that exist in the micro-cloud, and they are responsible for filtering and data pre-processing before send it to the cloud. Or cache information that was not available on a previous user request for some future use.

(ii) Services in the standard cloud that 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 the clients in a case when something catastrophic happened to the micro-cloud that is close to the user, and he is not able anymore to accept user requests.

This kind of services separation creates new application model that we present in detal in the secion 3.3.

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

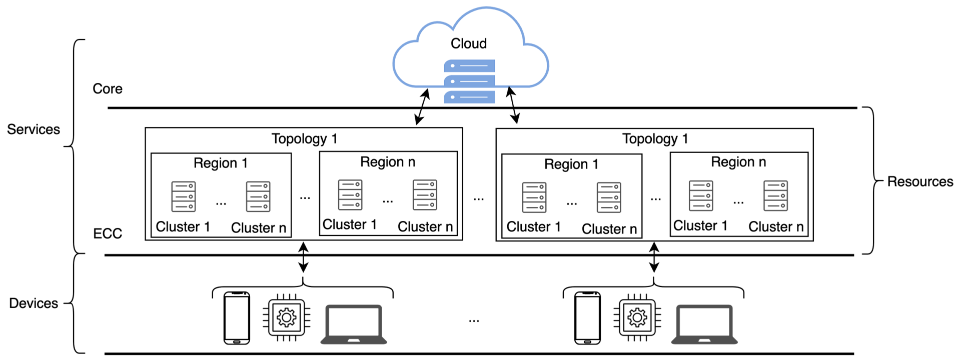

Figure 3.2: ECC as a service architecture with separation of concerns.

3.3 Applications Model

Modern-day applications that should run in the cloud are advised to follow the cloud-native model [131]. With this approach, we get applications that are easier to scale, more available, and less error-prone when compared to traditional web applications [131].

Satyanarayanan et al. present the edge-native applications model [19] that should use the full potential of EC infrastructure, and keep good features of their cloud counterparts.

When we introduced SoC for the micro-cloud model (cf. page 64), we described services that have one specific feature, and that is that these services span over two tiers of the system. This information is crucial for application development because we want to use the full potential of formed infrastructure, and keep good features of their cloud counterparts.

To satisfy the previously defined SoC model (cf. page 64), we can split existing applications into the **front** and **back** processing services. The front processing service is an edge-native application running inside some previously formed cluster to minimize latency, while the back service runs in the traditional cloud as a cloud-native application to leverage greater resources.

The front processing service will handle user requests coming to the nearby cluster, and communicate with the back processing services when needed to synchronize the pieces of information, cache data, or pass filtered or pre-processed data. Separation like that gives developers better flexibility and large design space when creating and optimizing their workloads.

With this model, we venture even deeper into understanding and applying the concept of **data locality** (cf. page 18). Since we have front and back processing services, we are committed to doing processing data closer to their source, instead of moving data to the cloud. We reduced latency and save some users money on storing and processing unnecessary data.

3.3.1 Execution models

Frontend services model should be packed in some standard way 1.6 and deployed in the wild, as an event-driven application, with the subscription policy to message streams, or infinite sequences [132] using topics [128]. The processing strategy is in the developer's hands, depending on the nature of the use-case.

If a service is subscribed to some stream using a topic, for example, it is natural that events appear in their stream. There are two strategies for building a large-scale time-ordered event system:

(1) **Fan out on write**, with this strategy, every service has some form of inbox, and when an event appears for some topic, that data is copied to every service that is subscribed to that topic. With these options, reads are fast, but writes are not. The more subscribers to the topic, the longer it takes to persist all updates.

(2) **Fan in on read**, with this strategy, every topic has a sort of outbox where it can store data. When services read their streams, the system will read the recent data from the outboxes. Writes are fast, and used storage is minimal, but reads are difficult because to do this properly on a request-response deadline is not a trivial task to do.

to implement those ideas without complicated synchronization, CRDTs could be used 1.2.1. Companies like Soundcloud and Bet365 are already using them for the same or similar tasks.

Some examples of applications 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) **daemons** or processing that are doing some tasks or executions in the background without explicit user intervention, usually their execution could be time defined but it is not mandatory;

(5) **services** or applications that would operate over standard request-response model or some variation of that protocol. For example, the NATS messaging system has a request-reply form implemented over topics;

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

These types of applications could be implemented in many ways like those discussed in section 1.3.2, or some adapted variant of those models, and should have a clear communication interface offered to users or applications and other services.

Users should be given the option to develop their applications in various available languages, not forcing them to use a strict one. But we must advise users about outcomes of their choice. For example, some languages might be slower or use more resources than others due to virtual machine execution, or some other tooling that is required to be started as well. The second important thing would be that users can do proper logging and tracing of their services. If something fails, the user must be able to go over the logs and traces to found problems.

3.3.2 Packaging options

Because of their nature, micro-clouds could be most likely be composed of ARM devices. These devices in many cases are not able to run full VMs because of their hardware restrictions. In recent years there are advances in VMs technology and its ability to run VMs on ARM devices. In [133] Ding et al. show such possibility to run VMs on ARM devices.

But even if VMs are fully compatible with ARM devices, we still inherit VMs large footprint, already discussed in section 1.6 if we try to use them $as is$.

On the contrary, containers, and unikernels give us $more for less$ same functionality but using fewer resources, meaning we can run more services in containers and even more in unikernels. But until unikernels are fully ready to be used they will fall in nice to have a 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 $silver bullit$ and there is no one solution for all scenarios.

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

In [134] Simić et al. show benefits of using containers in large scale edge computing systems from the theoretical point of view, by looking into architecture difference. Authors focus on differences between VMs and containers in cases where services need to run on ARM devices with limited resources.

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

Like any other system, users can create variants of the systems and different flavors optimized for certain solutions. In that cases, they may favor one solution over the other one. But in general case containers and unikernels should be the preferred way to package, run, and distributed user applications in a micro-clouds environment.

3.4 As a service model

Depends on their needs, users should be able to develop their applications using familiar models. Depending on how much control user require over the process of service scheduling, resource selection, etc. we can distinguish two 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 should synchronize with CC.

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

(3) **microSaaS or mSaaS**, with this option, we do not synchronize data with the CC, but all processing and storage is done in the micro-cloud. As such, this option should be used vigilantly, or not used **at all**.

It is important to note, that both variants **mCaaS** and **mPaaS** should **not** stand alone, at least not for now. We do not advise using **mSaaS** option for now, since that would require that the whole application is running **only** in the micro-clouds. In the future, when EC nodes gain more power and storage, this might change.

Both options **mCaaS** and **mPaaS** could be included, and part of any cloud model (cf. section 1.2.2) or offered separately.

3.5 Immutable infrastructure

As described in section 1.7, we have few options when it comes to setup and deploys infrastructure and/or applications. This thesis proposes a DS model that is built with a three-tier architecture that should operate in the geo-distributed environment we believe that **immutable deployment** model would be a good fit. It will simplify the deployment process since we want to rely on atomic operations and do not want to left the misconfigured system at any level. If something like that happens, we will end up in a problem, that would be hard to properly address and resolve.

Geo-distributed micro-clouds model that is described in this chapter should operate in two levels of deployment that are built one on top of the other:

(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 figure out the best way how to do user-specified changes. In this category, we can account for any change that is doing on the cluster, region, or topology that user/s operate on. For example create the new cluster, region, topology, or doing configurations of the setup system. The only change that could be done by imperative strategy updates on the nodes themselves. But even for this strategy, it would be beneficial if we could use declarative way **if possible**. It is important to notice that **mutation** does not mean in place change, but just the name of the operation. This deployment strategy is reserved for operations people, but if the company 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 the previous action is taken. We must have infrastructure setup already, to put any sort of services (applications) into the system. Like the previous model, this should be done declaratively as well, and all changes should be done immutably without an in-place change. The user should specify his new state or "view of the word" declaratively and let the system do all the changes he wants. All user services should be packed as described in section 3.3.2 because this simplifies the way services are put to the nodes. When done properly, this allows operations people to do faster changes with almost zero downtime deployments with all strategies already discussed in section 1.7. this part of the deployment should be done by developers since they did implementation and testing. They know how many resources they need for their service, what type of service they had developed. This deployment could be done in collaboration between operations and developers if a company is big or time is properly separated.

It is important to notice, that both deployments should be closely followed, for possible errors and problems so that users can act accordingly. These deployment messages, logs, and traces [86] should be stored in a 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 set up, applications deployment, monitoring, and in the general health of applications and infrastructure.

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

Only with tight collaboration with those two development roles, such a complex system like one presented in this chapter can be alive, well, and serving user requests without collapse.

It is important to note that every action in the system should be logged and traced properly. Since we are dealing with multi-tier architecture chance that something will fail is increased. Logs and traces should be available to operations people who are responsible for the infrastructure maintenance. At the same time, actions should be traced by the company that offers these micro-clouds. For security reason, any of these logs and traces should not be visible to others **but** responsible individuals, and the level of details and personal pieces of information should be different.

3.6 Formal model

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

Unfortunately, because of their nature DS cannot always be formally described by any of the existing techniques. There are a lot of variables that could influence this. But if the nature of the DS that is developing is such that can be formally described, it is recommended and beneficial. A formally described and correct model can save hours, days, and even months of hard debugging, testing to reveal all bugs and problems in the system that may only happen in some specific circumstances that are hard to initiate.

Infrastructure deployment will not happen overnight, and it might take years. It might not be started at all until the whole process is trivial [39], and this is complicated task [113]. Because of those properties, the key problem that needs to be resolved is how to simplify ECC or micro-cloud management. The naive approach would require going to every node and do it manually. This process is super tedious and time-consuming, especially if we consider a geo-distributed environment.

In such a complex environment, formal models are of great help if we can model and prove that protocols that the system relies on are correct. The system we propose tackles this issue using remote configuration and it relies on four formally modeled protocols:

(1) **health-check protocol** informs the system about state of every node (cf. Section 3.6.2)

(2) **cluster formation protocol** forms new clusters dinamicaly (cf. Section 3.6.3)

(3) **idempotency check protocol** prevents system from creating existing infrastructure (cf. Section 3.6.4)

(4) **list detail protocol** shows the current state of the system to the user (cf. Section 3.6.5)

These three protocols are base on the geo-distributed Infrastructure deployment.

3.6.1 Multiparty asynchronous session types

Communication protocols that operate from node to the system can be modeled using [41], an extension of *MPST* [42] – a class of behavioral types specifically designed for describing distributed protocols that rely on asynchronous communications.

The type specifications give us one additional benefit. They are not only useful to formally describe protocols, but we can also rely on them for a modeling-based approach developed in [41] to validate our protocols are they satisfy multiparty session types safety (there is no reachable error state) and progress (an action is eventually executed, assuming fairness).

The modeling process is done in two steps.

(1) **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 [41], the syntax of global types are constructed by:

$$\mathsf{G} \ ::= \ \{\mathsf{p} \dagger \mathsf{q}_i{:}\ell_i(\mathsf{T}_i).\mathsf{G}_i\}_{i \in I} \quad | \quad \mu\mathbf{t}.\mathsf{G} \quad | \quad \mathbf{t} \quad | \quad \mathtt{end} \tag{3.2}$$

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 $\mathtt{end}$ denotes a terminated protocol. We assume all participants are (implicitly) disconnected at the end of each session (cf. [41]).

The advance of using approach of [41], when compared to standard MPST (e.g., [42]), 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).

(2) **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 [41, Figure 1.5]. 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 $\mathsf{G} = \mathsf{q} \to \mathsf{p}{:}\ell(\mathsf{T})\ldots$ (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:

$$\mathsf{S} ::= +\{\mathsf{q}_i\alpha\ell_i(\mathsf{T}_i).\mathsf{S}_i\}_{i\in I} \quad | \quad \mu\mathsf{t}.\mathsf{S} \quad | \quad \mathsf{t} \quad | \quad \mathtt{end} \tag{3.3}$$

where $\alpha \in \{!, !!\}$ or $\alpha \in \{?, ??\}$ (in which case $\mathsf{q}_i = \mathsf{q}_j$ must hold for all $i, j \in I$, to ensure consistent external choice subjects, cf. [41, Page 6.]), and $I \neq \emptyset$. Interested reader can find details in [41].

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

Nodes that exist in the clustered environment usually have a channel where they can send metrics and other data in a form of a health-check mechanism. This channel could be used this channel to reach nodes to send some actions to them, for example, a cluster formation message.

In figure 3.3. we can see a low-level health-check protocol between a single node and the rest of the system. This process involves the following participants: Node, Nodes, State, and Log.

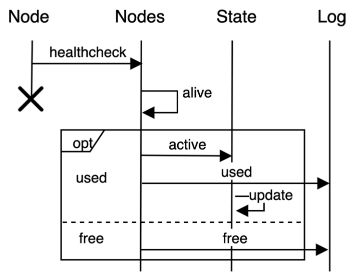

Figure 3.3: Low level health-check protocol diagram.

These participants included in Figure 3.3 follow the health-check protocol that is informally described below:

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

(2) **Nodes** accept health-check signals from 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.

Nodes service will be informed about node existence on his health-check ping. However, the system state will not be updated, changed, or even informed about this ping if the node is not used in some cluster.

In the following algorithm 1 we describe steps required by the system to determine if the node is free or used and how node information is stored.

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 formally describe servers or nodes (terms are used interchangeably) properties in the system, we could use set theory. In the beginning, the system will have an empty server set S denoted with $S = \emptyset$. To determine node state, we could use existing node properties. One approach may be that we use a node-id structure, for example.

Whatever approach we use, formally we can define free nodes as follows:

Definition 3.6.1. *Nodes are free if and only if (henceforth iff) they do not belong to any cluster.*

If we take node-id, for example, when the new health-check message from the particular node is received node-id structure will determine the node state. When the node is free, he will home some random or user-defined id, while when he is used in some cluster, the node-id structure will reflect this.

If we, for example, have n free nodes in the wild, this can be denoted with s_i, where $i \in \{1, \ldots, n\},$. If all of them send the health-check ping to the system, we need to determine their state. If a node s_i is free, we should add it to the server set, and thus we have:

$$S_{new} = S_{old} \cup \bigcup_{i=1}^{n} \{s_i\} \tag{3.4}$$

IIt is important to notice, that the order in which messages arrive is not important. The only thing that is important is that every message eventually comes into the system. We can now formally define node roles in the system like:

Definition 3.6.2. *All nodes in the system are equal, no matter are they part of some cluster or are not. One node can be used in one cluster only.*

The previous definition gives us a strong background in the further formal definition of the system because the only thing that we should care about is that node is alive and well and that he is ready to accept some jobs.

We described in algorithm 1 how the system stores the node data, but also how to determine if the node is free or used using the node-id structure. Every node s_i in the system that is part of the server set S could be described 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 [120] 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

$$s_i[L]_{new} = s_i[L]_{old} \cup \bigcup_{j=1}^{m} \{(k_j, v_j)\} \tag{3.5}$$

We can now formally define the number of labels per single server s_i in the system like:

Definition 3.6.3. *Every node from the server set S* ***must have*** non-empty *set of labels. The number of labels for every server s_i in the server set S may vary.*

Since labels are an important part of the system (more in future sections), they should be picked carefully and agreed on upfront. It should be possible also to change them if such a thing is required.

Labels should stick out some distinctive features of the node, that might be valuable for developers or administrators to target. For example, server resources, server features (e.g., SSD drive), geolocation, etc. They can be created as follows:

Definition 3.6.4. *Labels are created using arbitrary long alphanumeric text, for both keys and values, separated by colon sign. For example os:linux, arch:arm, model:rpi, cpu:*2*, memory:*16*GB, disk:*300*GB, etc.*

Following all things presented above, we can now give a formal description for 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 77: `node` connects `nodes` with *health_check* message and a payload of type T_1 required by the system to properly register node into the system.

Based on the received information, `nodes` **either** connects `state` with *active* message, informing the node status alongside payload typed with T_2 (containing informations required by the system to properly 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 patterns are impossible to be modeled using just 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., [42], but rather idle.

Projecting global type G_1 onto participants `node`, `nodes`, `state` and `log` we can than get local types as follows:

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

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

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

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

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 that the system relies on, appears in the cluster formation process, where users can form new clusters dynamically. Here, we distinguish two different actions:

(1) The first action is user-system communication. Here user sends query parameters to the system to obtain a list of available nodes that satisfy specified query parameters.

(2) The second action is a little bit more complicated than the previous one, and it starts when the user sends a message to the system with a new assembly specification. In this setting, the system involves participants: User, Queue, Scheduler, State, Nodes, Log, and NodesPool. These participants need to cooperate to successfully form new clusters, regions, or topologies dynamically, adhering to the scenario shown in Figure 3.4.

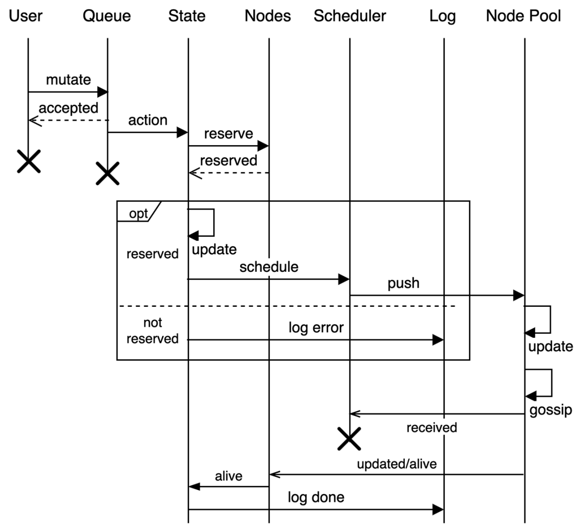

Figure 3.4: Low level cluster formation communication protocol diagram.

These participants included in Figure 3.4 follow the cluster formation protocol that is informally described below:

(1) **User** query Nodes service, based on some predefined query parameters. User sends a new creation message to Queue. User either get response *ok* if the message is accepted 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. the queue will also test if the specified message has been already handled before over idempotency check;

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

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

(5) **Scheduler** waits for a message sent from State, and physically carry on cluster formation by pushing cluster formation messages to the chosen nodes;

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

(7) **Nodes Pool** represents the set of n free nodes that will accept mutation messages. When a node receives the message, he will follow some predefined steps:

   (i) start gossip protocol to inform other nodes from the mutation message that they should form a cluster;

(ii) when cluster formation is done, send an event to Scheduler and Nodes that node is alive and can receive messages. The cluster formation is done, when all nodes have complete list of nodes that should form the cluster;

If a user wants to get a list of free nodes in the system, he must create a query using *the selector*, which is the set of key-value pairs where he can describe what type of nodes he desires. In algorithm 2, we describe steps that are 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

$$Q_{new} = Q_{old} \cup \bigcup_{i=1}^{p} \{(k_i, v_i)\} \tag{3.6}$$

Once the user submits query selector to the system with desired attributes, for every server in the set S, we need to check two things:

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

$$|s_i[L]| = |Q|\,, \text{ and} \tag{3.7}$$

(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:

$$P(Q, s_i) = \Big(\forall(k, v)\in Q\, \exists(k_j, v_j)\in s_i[L] \text{ such that } k = k_j \wedge v \leq v_j\Big) \tag{3.8}$$

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

$$R = \{s_i \mid |s_i[L]| = |Q| \wedge P(Q, s_i), i \in \{1, \ldots, n\}\} \tag{3.9}$$

If the result set R is not the empty set, we then reserve nodes for configurable time so that other users cannot see, and try to use them. Finally, we add reserved nodes with message data md to the task queue set:

$$TQ_{new} = TQ_{old} \cup \{(R, md)\}. \tag{3.10}$$

When the task comes to execution, the task queue will send messages to every node that is specified. In algorithm 3, we describe the steps required for cluster formation.

Algorithm 3: Clustering formation message

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

Users are free to override existing node labels with their labels or keep predefined ones when including nodes in the cluster. If the node is free, or the user did not change the node labels on cluster formation, the system will use default labels like node geo-location, resources, operating system, architecture, etc.

When the node receives the cluster formation message, he will automatically pick and contact a configurable subset of nodes $R_g \subset R$, and start the gossip protocol, propagating pieces of information about nodes in the cluster (e.g, new, alive, suspected, dead, etc.). When every node inside the newly formed cluster has a 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.

In the algorithm 4, we describe the required 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 1.4). We are using the same extension of MPSTs [41] used for the health-check protocol.

Global protocol G_2 (given below) conforms the informal description of the cluster formation protocol given at page 83. 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, we can notice that `user` – `state` and `state` – `nodes` are connected before specifying recursion. Hence, even after several recursion calls, these connections will be unique. So it is not required to disconnect them before looping.

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

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.

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

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.

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

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

$$\begin{aligned}\mathsf{S}_{\mathsf{user}} =& \texttt{state}!!\mathit{query}(\mathsf{T}_1).\mu\mathbf{t}.\texttt{state}?\mathit{resp}(\mathsf{T}_2).\\ &+\begin{cases}\texttt{state}!\mathit{mutate}().\texttt{state}!\mathit{create}(\mathsf{T}_3).\texttt{state}?\mathit{ok}().\mathsf{end}\\ \texttt{state}!\mathit{quit}().\mathsf{end}\\ \texttt{state}!\mathit{query}(\mathsf{T}_1).\mathbf{t}\end{cases}\end{aligned}$$

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

where

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

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

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

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

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

For instance, type $\mathsf{S}_{\mathtt{sched}}$ 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 that global type G_2 could also be modeled directly by using standard MPST models (such as [42]). However, in such models, the projection of G_2 onto, for instance, participant `sched` would be undefined (cf. [41]). Since we follow the approach of [41] with explicit connections, projection of G_2 onto `sched` is indeed defined as $\mathsf{S}_{\mathtt{sched}}$.

3.6.4 Idempotency check protocol

Mutate operation should be atomic, immutable, and idempotent. The user can specify the same topology details but in a different order, for example. We must ensure that the new cluster formation protocol should **not** be initiated, if the user change order of regions, clusters, nodes, or labels in one or more node/s.

If mutate operation fails, for whatever possible reason but the infrastructure is created, or the same infrastructure already exists, the user should get a message that infrastructure is formed. If the user changes the number of labels per node, nodes per cluster, clusters per region **only** at that case we should start a new protocol.

But since we have a different scenario than standard write to storage, and we do not specify steps on how operations should be done, to implement idempotency 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.

The 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 represented 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 a set of clusters, where every cluster is a set of nodes, etc. We can go deep as long as we want, but the restriction is that every structure **must** be idempotent. So we can use a set of sets and so on. With this option, we must test that idempotency is not violated throughout the hierarchy.

If we have cached idempotency data for user requests, and if a user tries to send the same request again, we can then test idempotency using set operation **intersection** because the intersection is an idempotent operation following the next proof.

Proof. 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*. $\square$

If we have stored user request on cluster formation protocol, and we receive a new request with the same name, **than** we can take the 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, the request represents the new action, and new cluster formation protocol.

We can make this test a little bit faster, by choosing the proper storage structure. If we first test does the same topology name is already present in the idempotency store, that lookup will spare us unnecessary comparison on sets. For example, data structures like $Hashtables$ store element as pair of $key - value$ and offers time and space complexity for lookups $\mathcal{O}(1)$ [135] *on average*. We can go even a little bit further and use CRDTs and SEC to store and replicate data on copies of the services that are required for the idempotency test.

Figure 3.5 show zoomed view in the *State* participent from figure 3.4, and idempotency check communication.

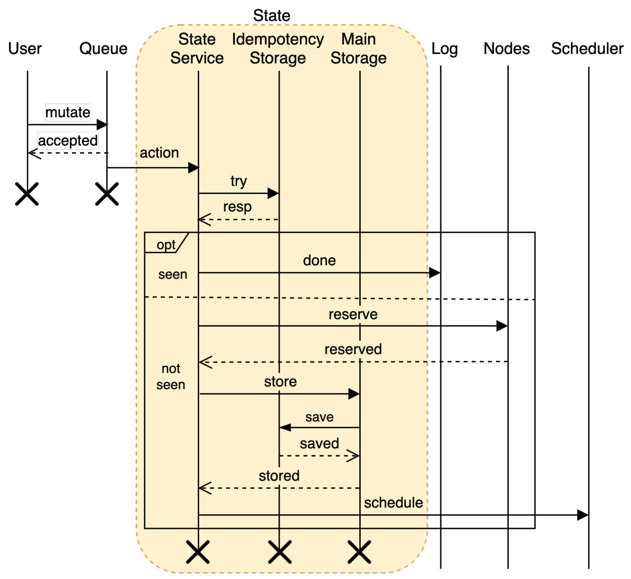

Figure 3.5: Low level view of idempotency check communication.

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 the 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) **hierarchical keyspace** both options are valid as long as **structure is idempotent** and we can do **idempotent operation** over that structure. For example, *set* as a data structure and *intersection* are good candidates. The algorithm that will test structure idempotency must be able to test both options. Algorithm 5 describe steps required to test if the operation is done before.

Algorithm 5: Mutate idempotency check

```
1  function idempotent(stored, topology):
2      foreach $data \in topology$ 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 3.5) by using [41]. The global protocol G_3 (given bellow) conforms the informal description given at page 93. At this point, the `user` can choose one of two possible messages:

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

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

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

The mutate protocol G', activated in the first branch in G_3, 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*. This process is the same as cluster formation protocol (cf. section (**1**)). Now we start specific communication protocol for idempotency check so `service` sends payload T_3 to `istorage` to test is this request *seen* before. The `istorage` responds with payload T_6 to `service`, and based on *Boolean* response `service` can do one of two things:

(1) `service` sends message to `log` and this process terminates; or,

(2) `service` sends *ids* of the nodes to be reserved (specified in the payload typed with T_4) to `nodes`;

same as cluster formation protocol (cf. section (**1**)). For simplification, we can assume that all nodes are reserved, and now idempotency data store global protocol G'', starts;

$$\begin{aligned} G' = {} & \texttt{service} \twoheadrightarrow \texttt{istorage}{:}\mathit{try}(\mathsf{T}_3). \\ & \texttt{istorage} \rightarrow \texttt{service}{:}\mathit{resp}(\mathsf{Boolean}). \\ & \qquad \begin{cases} \texttt{service} \twoheadrightarrow \texttt{log}{:}\mathit{done}(\mathsf{String}).\mathsf{end} \\ \texttt{service} \twoheadrightarrow \texttt{nodes}{:}\mathit{ids}(\mathsf{T}_4).\texttt{nodes} \rightarrow \texttt{service}{:}\mathit{rsrvd}().\mathsf{G}'' \end{cases} \end{aligned}$$

The idempotency store protocol G'', starts with `service` sending mutation payload T_3 to `mstorage`. Then `mstorage` sends same payload to `istorage`. When data is saved for future testing, `istorage` respond back to `mstorage`, and finally `mstorage` respond back to `service`. At this poing user payload T_3 is stored in both main storage and idempotency storage for future testing. Protocol continue with `state` connects `sched` (Scheduler) with message *ids* and the payload that contains other data imported for mutation to be completed (typed with T_5), and the rest of cluster formation protocol may continue.

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

We may now obtain the projections of global type G_3 onto the participants user, service, istorage, mstorage, log, nodes: and sched:

$$\mathsf{S}_{\mathsf{user}} = + \begin{cases}\mathtt{service}!\mathit{quit}().\mathsf{end}\\ \mathtt{service}!\mathit{create}(\mathsf{T}_3).\mathtt{service}?\mathit{ok}().\mathsf{end}\end{cases}$$

$$\mathsf{S}_{\mathsf{service}} = + \begin{cases}\mathtt{user}?\mathit{quit}().\mathsf{end}\\ \mathtt{user}?\mathit{create}(\mathsf{T}_3).\mathtt{user}!\mathit{ok}().\mathsf{S}'\end{cases}$$

where

$$\begin{aligned}\mathsf{S}' = &\mathtt{istorage}!!\mathit{try}(\mathsf{T}_3).\mathtt{istorage}?\mathit{resp}(\mathsf{Boolean}).\\ &+\begin{cases}\mathtt{log}!!\mathit{done}(\mathsf{String}).\mathsf{end}\\ \mathtt{nodes}!!\mathit{ids}(\mathsf{T}_4).\mathtt{nodes}?\mathit{rsrvd}().\mathsf{S}''\end{cases}\end{aligned}$$

$$\begin{aligned}\mathsf{S}_{\mathtt{mstorage}} = \ &\mathtt{service}??\mathit{store}(\mathsf{T}_3).\mathtt{istorage}!!\mathit{save}(\mathsf{T}_3).\\ &\mathtt{istorage}?\mathit{saved}().\mathtt{service}!\mathit{stored}().\mathsf{end}\end{aligned}$$

$$\begin{aligned}\mathsf{S}'' = &\mathtt{mstorage}!!\mathit{store}(\mathsf{T}_3).\mathtt{mstorage}?\mathit{stored}().\\ &\mathtt{sched}!!\mathit{ids}(\mathsf{T}_5).\mathsf{end}\end{aligned}$$

$$\begin{aligned}\mathsf{S}_{\mathtt{istorage}} = &\mathtt{service}??\mathit{try}(\mathsf{T}_3).\mathtt{service}!\mathit{resp}(\mathsf{Boolean}).\\ &\mathtt{mstorage}??\mathit{save}(\mathsf{T}_3).\mathtt{mstorage}!\mathit{saved}().\mathsf{end}\end{aligned}$$

$$\mathsf{S}_{\mathtt{log}} = \mathtt{service}??\mathit{done}(\mathsf{String}).\mathtt{end}$$

$$\mathsf{S}_{\mathtt{nodes}} = \mathtt{service}??\mathit{ids}(\mathsf{T}_4).\mathtt{service}!\mathit{rsrvd}().\mathtt{end}$$

Similarly, as for G_2, we remark G_3 could also be modeled using standard MPST (e.g., [42]), but again the projection types would be undefined while following the approach of [41] with explicit connections, we have obtained all valid projections.

3.6.5 List detail protocol

The final communication protocol in our system appears in the information retrieval process. Using labels, the user can specify what part of the system he wants to retrieve, namely, on formed topologies. This protocol could be useful, for example, if the user wants to visualize his topologies, regions, or clusters on some dashboard and monitor for some changes, alerts, etc. This protocol comes with two available options:

(1) **global view** of the system – all topologies the user manages. This will return just basic information about regions and clusters and their utilization.

(2) **specific clusters** details – in-depth details for specified clusters like resources utilization over time (using stored metrics information), node information, configuration data, and running or stopped services.

It is important to note, that similarly to the query operation (defined previously), both rules (3.7) and (3.8) **must** be satisfied for information to be presented in the response. The user can specify one additional information in the list request, and that is whether or not the user wants a detailed view or not. If such information is presented in the request, the user will get a detaild view back.

In the figure 3.6, we show a low-level view of the list operation protocol, where users can get details about the formed system. This setting involves the next participants: User, State, Nodes, and Log.

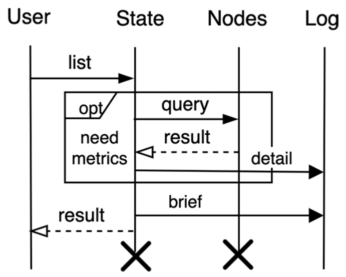

Figure 3.6: Low level view of list operation communication.

Now we can informally describe the roles of the participants in the protocol:, shown in figure 3.6

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

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

(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.

In algorithm 6, we describe steps that are required after the state receives a list message.

Algorithm 6: 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 1.7) by using [41]. Global type G_4 (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''.

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

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.

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

$$\mathsf{G}'' = \mathtt{state} \twoheadrightarrow \mathtt{log}{:}\mathit{brief}(\mathsf{T}_5).\mathtt{state} \to \mathtt{user}{:}\mathit{result}(\mathsf{T}_5).\mathtt{end}$$

Same as for the health-check and the cluster formation protocols, here we also present the projections of global type G_4, modeling the list protocol, onto participants `user`, `state`, `nodes`, and `log`:

$$\mathsf{S}_{\mathtt{user}} = + \begin{cases}\mathtt{state}!!\mathit{list}(\mathsf{T}_1).\mathtt{state}?\mathit{result}(\mathsf{T}_3).\mathtt{end}\\ \mathtt{state}!!\mathit{list}^*(\mathsf{T}_4).\mathtt{state}?\mathit{result}(\mathsf{T}_5).\mathtt{end}\end{cases}$$

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

where

$$\mathsf{S}' = \mathtt{nodes}?\mathit{result}(\mathsf{T}_3).\mathtt{log}!!\mathit{detail}(\mathsf{T}_3).\mathtt{user}!\mathit{result}(\mathsf{T}_3).\mathtt{end}$$

$$\mathsf{S}_{\mathtt{nodes}} = \mathtt{state}??\mathit{query}(\mathsf{T}_2).\mathtt{state}!\mathit{result}(\mathsf{T}_3).\mathtt{end}$$

$$\mathsf{S}_{\mathtt{log}} = + \begin{cases}\mathtt{state}??\mathit{detail}(\mathsf{T}_3).\mathtt{end}\\ \mathtt{state}??\mathit{brief}(\mathsf{T}_5).\mathtt{end}\end{cases}$$

For instance, type $\mathsf{S}_{\mathtt{log}}$ 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 and G_3 we remark G_4 could also be modeled using standard MPST (e.g., [42]), but again the projection types would be undefined, while following the approach of [41] with explicit connections, we have obtained all valid projections.

3.7 Access pattern

n section 3.3, we already discuss system access patterns from the applications point of view, using streams and topics. In this section, we are going to venture into the dashboard and system access described in 1.4.

To access the system, requests are going over some *master process*. This process is not responsible for any sort of synchronization, agreement, or something like that, but just to show dashboards and various system details since the cloud is available anywhere in the world, while micro-cloud should serve the local population.

One question that could come to mind is, what if some cloud provider offering micro-cloud functionality, or running our master process goes down, should the rest of the micro-clouds go to *read-only* mode and only accept **read** requests?

All communication is not exclusive to go over that master process, but if the user is nearby to micro-cloud should be able to initiate commands directly to clusters, regions, and topologies. But for dashboards and full pieces of information about his micro-clouds cloud would be a better solution, just because of more available resources.

If some cloud provider goes down for whatever reason, we should foresee this and to resolve it we could use multiple cloud providers using *Multi-Cloud Computing* [54, 55] so that one cloud provider is a master process and many others are backup in case the whole cloud provider goes down.

Here inevitably we have some state synchronization, and here we could rely on SEC and CRDTs to do synchronization without some expensive coordination between providers.

This is not generally a problem if applications that are running in micro-clouds are not dependent on some process, location, sensors of services, etc. If we this is the case then we can connect micro-clouds in sort of P2P network and give them some logic to just route request to the proper location on the globe.

This might not be that fast, since even light is affected by the distance, but we would be able to issue requests to cluster in a different part of the world.

3.8 Repercussion

The model presented in this chapter, have three possible execution 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 a viable 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 be 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) **Combination**, this approach can be done over multi-cloud principles. Some cloud tasks could be offloaded to the nearest micro-cloud.

3.9 Limitations

Since the perfect model never existed, the model that is proposed in this thesis has 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 a few things.

(1) We must be aware that not all organizations will be able to deploy micro-clouds, due to the high initial investments required [12]. We can rely on government authorities, large cloud providers, or other big companies to build the initial infrastructure for their own needs, and lease it to others similar to the cloud. The general public can use them, similarly to the cloud – pay as you go, model.

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

(3) These small-scale servers must be out of reach of people and protected in some way so that not everyone has access to them. Some degree of physical security must be implemented.

(4) The places where these small scale servers will be deployed must have a stable internet connection, and the ability to integrate SDN or other similar technologies, 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.

(6) Splitting the processing into two parts and the possibility that users can be responsible for micro-clouds may raise some legal concerns. Either to develop interesting applications, use them as a firewall or simply use them as a privacy level for data there must be a legal agreement that might not be that easy to achieve.

Chapter 4

Implementation

In this chapter, we are going to give more details about the framework implemented based on the formal model and architecture specification given in the previous chapter.

Section 4.1 framework architecture, and system implementation details, and framework limits. In Section 4.2 we present implementation details about framework operations. Section 4.4 describe few possible senarios and applications that could utilize micro-clouds platform. In Section 4.3 we present results of our experiments.

4.1 Framework

In this section, we are going to introduce an implemented proof-of-concept framework based on the model proposed in the previous chapter 3. The framework is called **Constellations** or **c12s**[1] for short because it is strongly influenced by nature and the neverending number of galaxies that the universe is (not only) composed of. Similarly,

[1]https://github.com/c12s

we are trying to create a universe of clusters that will serve humanity to help them with their day-to-day tasks. The framework is it is open-source, and it is implemented 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 the request arrives at the gateway, if the request is valid it will pass the request to the rest of the system. It communicate to the rest of the services to check if the user exists, does he have proper rights for actions that he sends, and if not return the proper message and do not propagate it to the rest of the system.

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

- **Queues**, the purpose of this service is to prevent huge request load to the system and to accept more user requests. When the user submits any **mutation** operation – an operation that changes the state of the system, these operations will be put in the queue. User can create their queues, to prevent long lines for specific tasks. For example, users can create queues for specific tasks, and use them only for those tasks, while other queues could be general-purpose queues. On system start, every user will start with one queue — **default**. When doing mutations on different parts of the system, the user can specify in metadata which queue he wants the task to go to. This service implements a token bucket rate-limiting algorithm [136] to prevent congestion of the system.

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

- **State**, is the heart of the system. This service stores all information about architecture, clusters, regions, and topologies. When a new cluster/region/topology is created, this service will setup watchers for nodes, so that if the node does not send the health-check signal for some time, that node will be declared dead. This is important so that at any moment we must know the state of the clusters and their utilization. This service as well will cache frequently used nodes data, so that on the next request node lookup is faster since we can have a huge number of nodes, topologies clusters, and pieces of information about them. To prevent data loss, this service will first store a copy of the operation before attempting any mutation of the system.

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

- **Command push**, the purpose of this service is to push commands and operations to the nodes user desired. This service implements a token bucket rate-limiting algorithm [136] to prevent constant data push to the nodes. Similar to the state service, this service will store a copy of operation information locally before attempting any push to the nodes. This information will be deleted, once the operation reaches all decided nodes and all of the responses with the acknowledge (ACK) message.

All services are highly customizable and all have their configuration file that could be changed and adopted. All these services are easy to extend to support new operations and pieces of information about nodes, regions, clusters, topologies, and latter on applications, configurations, namespaces, etc.

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

Regarding 3.7, the framework operates as a master process is running in Kubernetes, and through that process, all commands are issued. Future applications can communicate with nodes as stated in 3.3. They do not require communication with the master process at all, they can communicate with the formed cluster using topics and streams.

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

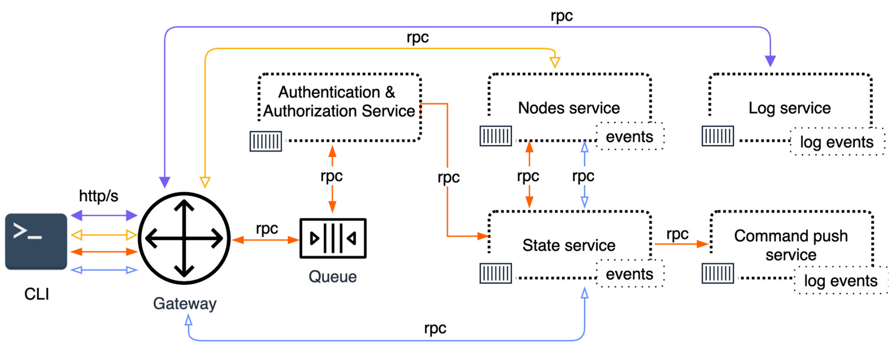

Figure 4.1: Proof of concept implemented system.

4.1.1 Technologies

All services are implemented using the Go[2] programming language, because of his well-known tooling, support for developing system software, web-based applications, small binaries but also great concurrency model, and ability to build binaries for almost any architecture without any code changes. All services rely on Go implicit *interface* implementation mechanism. Every technology or component used in the system can be swapped for some other, as long as that component implement *interface* fully. The framework is developed in such a way that is relatively easy to extend, or switch and use different components and technologies.

As the main storage layer for our system, we used etcd[3], a popular open-source key-value database, that shows good performance, and it is mostly used for configuration data. Metrics data are stored in the open-source time-series database InfluxDB[4].

Communication between microservices is implemented in RPC manner using gRPC[5], and Protobuf[6] 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[7] in a publish-subscribe manner.

Caching layer for every service is implemented using Redis[8] key-value, in-memory database. It is important to notice, that all concrete

[2]https://golang.org/
[3]https://etcd.io/
[4]https://www.influxdata.com/
[5]https://grpc.io/
[6]https://developers.google.com/protocol-buffers
[7]https://nats.io/
[8]https://redis.io/

tools that are used, are easily swappable for some other as long as they implement a proper interface.

All communication with the outside world is done in a 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[9] containers are used. To achieve automatic orchestration, and self-heal, and up-time, all services that are packed in containers, are running inside Kubernetes.

Every service will log details about his usage and calls, as well as traces how requests are going. Log data is stored outside the service and container, and pieces of information will be sent to centralized log storage on every *t* seconds specified by the user. Sending intervals could be changed and adopted using the 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 viewed 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 the general state of the system, and pieces of information about user requests and the state of their requests.

[9] https://www.docker.com/

4.1.2 Node daemon

Every node runs a simple daemon program implemented using the Go programming language, as an actor system (Ref. secion 1.8.1). The actor system is developed for this purpose. When a message arrives, the proper actor will react based on the message type, or discard it if the type is not supported.

Extending such a system is rather easy because users need to simply write a new actor and logic that goes with them and register it to the system.

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

System messages will be sent to the daemon, but he will not react to these messages. Daemon is not able to communicate with any actor directly. All communication goes through the actor system who is responsible to pass messages to the actors. The actor system at this point has only four 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 to messages from other actors to update the daemon state. For example on new cluster creation, this actor will update daemon id, name, labels, etc.

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

(4) **gossip actor**, this actor will communicate with other peers in the group using SWIM protocol techniques.

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

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

Listing 4.1: Actor system hierarchy.

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

(1) **identifier**, represent unique identifier of the node. When a node is not part of some cluster this can be whatever the user wants. Once the node is part of some cluster, the identifier will be updated, and it is not advised to change it manually afterward.

(2) **name**, represent name of the node. This property also can be changed when a node is part of the cluster, otherwise, it can be whatever we want.

(3) **labels** represent the specific features of the node. Labels are used to query for free nodes, and there is no formal restriction of what they can or can not be. This property can be changed when the node is part of some cluster. It is advised, that as labels we put some specific features of the node that might be beneficial for the user who is looking for nodes to create new cluster/s.

(4) **health-check details**, here we have pieces of information to control the health-check mechanism. Since nodes communicate with the rest of the system via publish-subscribe events, we must specify the address of the rest of the system and the topic name, where we publish our pieces of information.

(5) **system information**, represent basic information for a node to know how to contact the rest of the system. We should specify the system address, so that node knows where to send messages, and where are messages are coming from. We can also specify the version of the system we are trying to contact. System version could be used to support **backward compatibility** if we have multiple API versions of the system running at the same time or some period.

Configuration can be done easily using the YAML configuration file. Listing 4.2 show simple YAML configuration file for daemon 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
    memory: 4GB
    storage: 120GB
    cpu: 1
    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, names, IDs, 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 the node id will be used for reservation when forming a cluster.

At the first start of the daemon when the node is free, the user can specify whatever node id he wants. Once, the node is a part of the cluster, the node id will be updated to match that. Node id update will happen on the cluster formation process.

4.1.3 Separation of concerns

IThe implemented framework follow the clear SoC model, presented in 3.2. Since the presented model consists of three components, the implemented framework is dealing only with **resources** 3.2 part of the SoC.

His job is to organize nodes into clusters, regions, and topologies, to make them communicate and expose their resources to the upper layer of SoC for utilization. The upper layer will run services on these resources, to collect data from data creators and process them as requested.

The upper layer must have set up the infrastructure to do any processing or storage. This middle layer is the binding element between the layers.

4.1.4 Limitations

The framework at the current state has some limitations that we **must address**. As shown in figure 4.1, for purpose of testing the system, and to give the users any way to interact with the system a CLI application is implemented. This is a good option for initiating commands to the system. The problem with this approach is that showing logs, and topologies might be limiting for users, especially if they monitor and supervise multiple topologies with regions and clusters.

For monitoring, a tool with UI would be a better solution and it can show more details. For a small amount of data when topologies are relatively small, CLI could be used. But for some real-world applications desktop and/or Web-based UI will be a much better solution, and CLI can be used for fast lookup on specific pieces of information about clusters and nodes, for example.

The current implementation of the queueing system is **limiting** because if we want to extend the system with new queues we need to shut down the whole service, and that is not the best solution. But since the goal of this thesis is not queue management, but micro-cloud formation, protocols, and formal modeling.

But since the purpose of this thesis is not Human-computer interaction, this should be the topic of the future work 5.2.

4.2 Operations

In this section, we are going to describe all implemented operations in the framework and present specific details about every individual operation.

4.2.1 Query

The query operation is used to show all free nodes or filtered free nodes that are registered into the system, to the user (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. These key-value pairs can be any alphanumeric set of symbols for both keys and values. Based on that key-value pair system will do the query of the free nodes.

The selector will be used as a search mechanism to compare the labels of every free node that exists in the system. The nodes that are satisfying the rules 3.9 defined in Section 3.6.3 will be present in the result.

This operation is done before the formation of new topologies, regions, or clusters — mutation of the system. The user first needs to get a list of free nodes, then he can choose nodes that are best suited for him and try to form topologies, regions, or clusters of them.

The querying process is a little bit changed from one presented in Section 3.6.3. The only change that is made is the addition of the *Gateway* service that will pass requests into the system and prevent overflow of requests. This change **does not** affect or validate the formal model presented before. Since the added service does not interfere with the process of searching nodes or changes to the system.

Figure 4.2 shows a communication diagram for the query action, with the addition of the Gateway service.

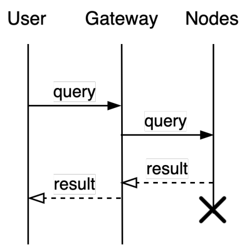

Figure 4.2: Low level communication protocol diagram of query operation.

4.2.2 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, the desired state needs to be specified **declaratively** using a YAML file and submitted to the system. When the state is submitted, the system will do the rest of the job to ensure that the state desired by the user is reached.

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.

When forming a topology, users can assign a name and label 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

Mutate operation is **immutable** which mean that there will be no in-place changes to the existing state. If a user wants to do any update, he needs to specify a full new state that will replace the existing state. This operation is **atomic** as well, which mean that a whole new state must be able to replace the existing state. If this happens, the system will replace the old state with the new state. The main storage that stores configuration data is a key-value store implemented using an etcd database. The key that will be used to store the configuration data is the whole path of topology, regions, cluster, node, while the value represents the state desired by the user. Listing 4.3 show structure for key-value pair that is stored in the main database, where on top we can see a structure of the **key**, and below it we can see the structure of the **value**. This kind of structure is similar to OS file-system data organizations of files and folders.

```
\topology\region\cluster\node
  +--\labels
  +--\informations
  +--\resources
  +--\status
```

Listing 4.3: Structure of stored key-value element.

We store as well one additional information about labels **index** value. This index is used for a faster query of elements when doing labels comparison. To find elements we can query in a similar way like searching files and folder structure. The etcd in newer version **do not allow** hierarchical keyspace, but what they allow is **ranged** query by some **prefix**. This is useful as well because we can still get all regions in topology, clusters in the region, or nodes in the cluster if we know to which group they belong.

Mutate operation is not idempotent by nature, but the whole process behind it make mutate an idempotent operation. Whether the user tries to create already existing infrastructure by changing the order of regions clusters or labels in the nodes, or if he for whatever reason did not get confirmation that infrastructure is created an existing infrastructure will not be created. This is done in such a way, that *State* service (Figure 4.1) will keep the information set about created infrastructure. This information is implemented as a **flat keyspace** set, that have information. On every mutation request, the system will test if such a topology already exists. If such topology already exists user will get confirmation that his infrastructure is created. If such topology do not exists, a new cluster formation protocol 3.6.3 will be initiated. The mutation confirmation is followed by a unique id which the user can use to query steps, logs, and traces that are done in process of working towards the state desired by the user. Logs service can give the user complete details about his task state using that unique id.

When creating topologies, the user is free to give whatever name he wants for every region, cluster, and node. The only restriction is that name should be the alphanumeric set of characters. Listing 4.4 shows an example of a user-defined state that forms the topology of one region with one cluster that contains three nodes, with a retention period of 24 hours. The whole topology will have the same set of labels, but *node*3 redefines this rule by specifying its own.

```
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
                model: rpi
                storage: 100GB
                memory: 1GB
```

Listing 4.4: Example of mutate file using YAML.

After all, nodes that should form a cluster, acknowledge cluster formation message, they will inform the system that the message is received, and they will start the process of cluster formation. This process is done by using SWIM [31], a Gossip style protocol. When every node has a complete list of his peers that should be in the cluster, the process of cluster formation is done.

On the next health-check message, every node will send his **id** that is telling the system that he is part of some cluster. This kind of messages, will be used in the *State* service to validate that cluster is alive and well, or that some nodes (or all), are dead or down if **id** is not received.

And Gossip style protocols (like SWIM) could be used in the future to propagate information in the cluster, without explicitly ping every node in the cluster.

4.2.3 Queueing

When doing mutation, users can target a specific system queue, by adding a metadata part in the configuration file. With this ability, users can aim for specific queues just for the mutation to avoid long waiting times if other queues are full. Currently, the system does not have any limitations, restrictions, or logic that will specify which queues are used for what or give them special rules or permissions. This can be viewed as a "gentleman agreement", that in one team, operations users can proclaim specific queues like *mutatation* queues used maybe for specific topologies, and later on for other operations as well.

The queue service starts only the $default$ queue exists. Adding a new queue to the system is implemented using the configuration file, for convenience only. Listing 4.5 show an example of queue service with two additional queues with specifications of their parameter needed for token bucket opeation **??**. Also, we can see configuration pieces of information for instrumentation of a single service, and the same configuration is implemented for all specified services shown in Figure 4.1.

```
blackhole:
  db: address
  queue:
    myqueue1:
      namespace: mynamespace
      retry:
        delay: linear
        doubling: 1
        limit: 5000
      maxWorkers: 4
      capacity: 4
      tokens: 0
      fillInterval:
        interval: 1
        rate: s
    myqueue2:
      namespace: default
      retry:
        delay: exp
        doubling: 2
        limit: 50000
      maxWorkers: 5
      capacity: 5
      tokens: 0
      fillInterval:
        interval: 1
        rate: s
  instrument:
    address: address
    stopic: topic
    collect: interval
    location: location
```

Listing 4.5: Structure of stored key-value element.

4.2.4 List

The list command shows the current state of the system for the logged user (blue arrows in Figure 4.1.). Logged user is able **only** to see infrastructure he created or he is maintaining. All other infrastructures, created by other users will not be visible to the users that are not creators or maintainers.

To view his infrastructures, the user can specify what part of the system he wants to see using a set of labels or list of key-value pairs which will be used by the system to determine what the user wants to see. This process is similar to **selector** shown in the query 4.2.1 operation.

There are two levels of details that user can specify:

(1) **global view** of the syste, or all topologies he manages with just basic informations like resoource utilization, number of regions clusters and nodes.

(2) **detail view**, 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) and so on.

Both options can be useful if operations people need different details levels for different topologies.

4.2.5 Logs

The logs operation showing stored logs and traces to the user (purple arrows in Figure 4.1.). Same as previous operations, the user needs to be registered and logged in to be able to perform this action. With this action, the user can see the state of his operations and actions. The user can choose between two options for querying his logs:

(1) **get**, for this option user, need to provide a unique task id that is given to the user when he creates a mutate operation. With this option, the user will get a full list of steps, traces, and logs collected over the time the system was setting up his desired state.

(2) **list**, with this option user can specify **selector** using list of key-value pairs in a similar way to query 4.2.1 and mutate 4.2.2 to filter only parts of the traces that contains same labels as selector does.

This action is implemented in some basic aspects, as this action can return a huge amount of data that require some better visualization than CLI.

4.3 Results

The framework described in the previous sections has put the test. For ease of testing, we have used few ARM-based nodes that easy to move from place to place. The test should be conducted on the heterogeneous nodes, to see how the system will react to different architectures and resources. 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 heterogeneous nodes for creating clusters, regions, and topologies

4.3.1 Experiment

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

We have run tests on different configurations and different nodes clusters using these nodes. All nodes were connected on the network, and experiments were conducted in a controlled environment. Nodes that should be part of the same claster are connected on same network for easier experiments.

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.

Our experiment started with separating nodes into groups of **three**. This number is chosen because in DS, an odd number is used in cases when we need to reach some agreement and we need major majority like consensus, for example. This is not important for purpose of this thesis, we could pick any number, membership protocol do not makes a difference if ther are three or four or eleven nodes in the cluster.

After nodes are separated,, for every node we created configuration file setting up default parameters for every property node daemon requred 4.1.2. After all services are up and running and, we turned nodes on, and health-check protocol 3.6.2 started at uprfront defined time, that we set and for test purposes it was *1 minute* interval. Logs, resources and other node details are set to *15 seconds* interval, so between health-check intervals, daemon will collect resource informations four times before sending it to the rest of the system.

For convenient testing all nodes have same set of labels, and as labels we choose OS name, OS version, architecture version, node name and some basics detail about resources of the nodes.

After some time, we were able to see all nodes registered in the system. When all nodes are sending health-check ping for some time and we have stable system, we issued a mutate operation creating clusters of nodes that are logicaly close ot each other, and we initiated cluster formation protocol 3.6.3. After the protocol is done, we ended up with four clusters as we intended. We tried to initiate same command again to test idempotency check 3.6.4, and we get message that clusters already exists.

To increase capacitity, we extended clusters by creating new ones using *three clusters* with *four nodes*. We created new new mutate file, and initiate new mutate command. After some time, we saw that one cluster is down and that we now have *three clusters* with *four-nodes*, as we intended. After successful creation of new clusters, we dleted down cluster.

Last test was to test health-check protocol once again, we disconeted one random node from the power supply, and since that node ping was missing, the system was able to detected what node is *down*. This concluds our experiment.

4.4 Applications

This research focuses on a platform with geo-distributed edge nodes that 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. 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.

Let us consider the scenario in which such a system is implemented and a new catastrophic event (earthquake, tsunami, war, terrorist attack, pandemic, etc.) struck the human population. An increasing amount of ambulance vehicles must be routed to hospitals fast. Suddenly the traffic control system needs more resources to continue operating properly. We can then organize our resources differently to accommodate such situations. On the way to the hospital, we can monitor patient health in real-time [137] and store it in some healthcare platform [138, 139]. giving the health workers crucial information on ambulance arrival.

Such a scenario is relatively easy to solve if using the cloud resource vise. As we increase resource demand, the cloud can provide us with more resources. The main advance of EC, when compared to the cloud-only approach, is the acceleration of the communication speed. n the scenario previously discussed, the cloud could bring huge latency, while EC originates from peer to peer systems [10], 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 applications.

If we imagine that we put sensors on a specific group of users and/or places in the city/state and monitor them in real-time. We can process these streams of data directly close to where they are, where they are moving and going. We could monitor air pollution for example, and make decisions in real-time to suggest users to not walk in some are especially if they have some known respiratory problem.

Geo-distributed nodes represent a great idea to do any kinda monitoring and processing especially for natural phenomenons and do alert as soon as probes, sensors, or other things detect even the slightest changes. 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. Content delivery networks, content sharing could be implemented to serve content to the users faster than going over the cloud since micro clouds should serve the only local population that is nearby.

Chapter 5

Conclusion

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

5.1 Summary of contributions

In this thesis we have presents a possible solution to how to organize geo-distributed EC nodes into micro-clouds that will be able to serve requests from populations nearby. We have introduced an extension of MDCs based on proven abstractions from cloud computing like zones and regions but adopted for different usage scenarios.

These easy to understand, yet powerful abstractions with slight adaptations, allowed us to cover any arbitrary vast geographic area, and yield a more available and reliable system forming the micro-cloud model. These abstractions are easy to organize and reorganize, and micro-cloud size is determined by the population needs descriptively.

We had also presented, the cloud to ECC mapping showing differences between two architecture models. Furthermore, we give a formal model of the system and its protocols used to form such a system, with clear SoC and native application model for future micro-clouds infrastructure and service development. The thesis also presents a proof of concept implementation and discusses integration into existing solutions, limitation of such a system with few applications that could be used in.

In Chapter 1 we gave a short introduction to the topic of distributed systems, with a focus only on the areas that are important for the understanding of this thesis.

We show what distributed systems are or at least consensus how we can describe some systems as distributed. We present problems these systems create, and why they are so hard to implement and maintain.

We had also presented few distributed computing applications, that we can use to employ nodes in the distributed system. Further on we show what is scalability and why it is important for distributed systems, with few organizational ideas like peer-to-peer and membership that protocol that is important in a distributed environment for various reasons.

Then we described virtualization techniques that can be used to pack and deploy applications and infrastructure, how to implement various deployment techniques especially in the CC environment, but also the difference between DS and few models that are usually confused as distributed like parallel computing and decentralized computing.

A the end of this chapter we present the motivation for this thesis and hypotheses, combine with goals, research questions this thesis is built on.

In Chapter 2, we show related work done by various other researchers or companies, and again we focus, only on things that are related to this thesis.

We show different platform options, where people change the existing solutions like Kubernetes or OpenStack to made them work in areas like edge computing and mobile computing. Further, we show implementations of few new platforms to use volunteer nodes to do some computation and storage on them with drop computing and systems like Nebula for example.

We show how nodes can be organized to split some geographic area into zones, and show how MDCs can help CC to serve requests from the local population. Different offloading techniques are used today, how to offload tasks from mobile devices closer to fog or edge nodes, but also various application models that could harness these offloading techniques and nodes organizations.

In the end, we present the position of this thesis, compared to other similar models.

In Chapter 3, we present the heart and soul of this thesis. We dissect all important aspects that we need to have to help CC with latency issues, Big Data with huge volumes of data especially in the age of mobile devices and IoT.

Our model is based around MDCs that are zonally organized, that will serve only local populations and populations nearby. We present a model that is based on CC but adopted for different scenarios and use cases.

We show how we can dynamically form new clusters, regions, and topologies and how we can use them in the new age of mobile devices and IoT. These newly formed systems or system od systems will have clear the SoC, adopted from existing research to three-tier architecture. The formed model will serve as a pre-processing layer, firewall, or privacy layers, and it is adjustable in various dimensions.

The presented model can be as huge as the whole state, or small as a single household and all in between. This is a matter of agreement and a matter of choice. We present how developers can use this new infrastructure and what possible models of applications could exist, and how operations can deploy developed services onto existing infrastructure.

In the end, we present the repercussions of this model, and how can be used as an integral part of existing systems to serve as topology storage or can be used as a new model where we can develop new subsystems and applications on top. We present protocols for the creation of such a system and model them using asynchronous session types. The system follows a formal model, and it is easy to extend.

In the end, we give limitations of such system and things we must be aware of, **if** such technology is going to be used in real-life scenarios.

In the last Chapter 4, we present an implemented framework that is based on knowledge compiled from previous chapters. We also present in detail operations that could be done in the framework, where it fits in the presented SoC model.

Further, we present the results of our experiments in a controlled environment, what are the limitations of the framework at the current stage, and possible applications, and where this model could be used and beneficial.

In this chapter, the last chapter of the thesis we have concluded the thesis with what is done, what can be done in the future in form of future work.

5.2 Future work

Our work on the micro-clouds is at an early stage and leaves many open questions. As part of our current and future work, we are planning to extend the proposed model in different directions. Future work might be separated into three options:

(1) features that operations and devops users can benefit from;

(2) features developers might benefits from;

(3) infrastructure features, that both previous groups can benefits from;

For the first group, the first thing that should be implemented is remote cluster management, using configurations, security credentials, and actions over nodes in one or multiple clusters. On formed clusters, the users should be able to do remote configurations that nodes and/or applications can use and set up the data without going from node to node.

The system should also be extending with namespaces for usage in environments with many users in multiple teams – multi-tenant environments. Namespaces provide the separation on virtual clusters, running on the same physical hardware. Speaking about multi-tenancy, we are also planning to implement role-based access control integrate with authentification and authorization services. With the addition of controlling different users with quotas, using rate limiting and resource limiting.

We are also planning to implement a full architecture and applications monitoring, alerting, and reporting that would be helpful to any administrator of such a system. We might also consider rethinking networking and making network isolation so that once formed topologies can communicate within themselves, and a possibility to specify strategies of communication with other topologies.

Queueing system mentioned in the section 4.2.3 should be extended so that users and/or operations people can easily add new queues and possibly assign a role for them.

We should extend the access pattern so that users can issue commands to micro-clouds directly instead of going over the cloud master process only. Here we need to implement synchronization in multi-cloud deployment.

For the last part in this operations section, we should also think about continuous integration, deployment, and delivery of services onto the infrastructure, and as well as various UI dashboards that can be customized to present different aspects of the system.

Another direction for future work is the implementation which developers could benefit from. The first thing that should be implemented here is the complete application's framework so that users can start developing services that can do something. We should also implement a framework and maybe domain-specific language for the use case where users just want to pre-process the data before sending it to the cloud, on more convenient than writing the whole application.

Users can develop their applications with different models:

(1) **mPaaS**, where the platform is doing all the management and offers a simple interface for developers to deploy their applications;

(2) **mCaaS**, if users require more control over resources requirements, deployment and orchestration decisions;

(3) **mSaaS**, users can develop their solutions only using micro-clouds, but this is not advised at the moment;

The second would be file system and database APIs that users can use to store their data. We should also provide an interface for extensions so that others can create their databases following different models from which developers can benefits, but also integrating existing ones.

The last part of the future would be extending the current system with tunable replicating strategies for the data, in case that any part of the topology fails for whatever reason, data would not be lost. Furthermore, we should provide tunable CC synchronization models that could be used.

We should implement a scheduling system for user-developed applications so that we can put applications into the formed architecture. And last but not least, we are planning to add several security layers to protect a system in general from malicious users.

We should investigate compression methods to reduce data stored and sent via the network. These tests should be conducted on ARM devices with existing methods, or maybe we can create a ground for new compression methods and techniques.

This thesis in section 3.8, stated that this model could be integrated into existing solutions. Our efforts should go as well on integrating this system with existing solutions so that they can benefits from this hierarchical and geo-distributed nodes organization in the same way or almost the same way as to stand-alone solution would.

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] 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.

[3] 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

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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

[19] 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

[20] 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

[21] 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.

[22] 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

[23] 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

[24] 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

[25] 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

[26] 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.

[27] 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

[28] 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

[29] 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

[30] 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

[31] 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

[32] 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

[33] 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

[34] 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

[35] 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

[36] 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

[37] 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

[38] 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

[39] 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

[40] 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

[41] 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

[42] 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

[43] H. El-Sayed, S. Sankar, M. Prasad, D. Puthal, A. Gupta, M. Mohanty, C. Lin, Edge of things: The big picture on the integration of edge, iot and the cloud in a distributed computing environment, IEEE Access 6 (2018) 1706–1717. doi:10.1109/ACCESS.2017.2780087.
URL https://doi.org/10.1109/ACCESS.2017.2780087

[44] 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

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

[46] 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

[47] 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

[48] 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

[49] 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

[50] 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

[51] 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

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

[53] 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

[54] J. Hong, T. Dreibholz, J. A. Schenkel, J. A. Hu, An overview of multi-cloud computing, in: L. Barolli, M. Takizawa, F. Xhafa, T. Enokido (Eds.), Web, Artificial Intelligence and Network Applications - Proceedings of the Workshops of the 33rd International Conference on Advanced Information Networking and Applications, AINA Workshops 2019, Matsue, Japan, March 27-29, 2019, Vol. 927 of Advances in Intelligent Systems and Computing, Springer, 2019, pp. 1055–1068. doi:10.1007/978-3-030-15035-8_103.
URL https://doi.org/10.1007/978-3-030-15035-8_103

[55] D. Ardagna, Cloud and multi-cloud computing: Current challenges and future applications, in: M. A. Babar, H. Paik, M. Chetlur, M. Bauer, A. M. Shariflloo (Eds.), 7th IEEE/ACM International Workshop on Principles of Engineering Service-Oriented and Cloud Systems, PESOS 2015, Florence, Italy, May 23, 2015, IEEE Computer Society, 2015, pp. 1–2. doi:

10.1109/PESOS.2015.8.
URL https://doi.org/10.1109/PESOS.2015.8

[56] A. Dadgar, J. Phillips, J. Currey, Lifeguard: Local health awareness for more accurate failure detection, in: 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, DSN Workshops 2018, Luxembourg, June 25-28, 2018, IEEE Computer Society, 2018, pp. 22–25. doi:10.1109/DSN-W.2018.00017.
URL http://doi.ieeecomputersociety.org/10.1109/DSN-W.2018.00017

[57] 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

[58] 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

[59] 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

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

[61] 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

[62] 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

[63] 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

[64] 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

[65] 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

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

[67] 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

[68] 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

[69] 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

[70] 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

[71] 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

[72] 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

[73] 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

[74] 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

[75] 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

[76] 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

[77] 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

[78] 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

[79] 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.

[80] 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

[81] 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

[82] 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

[83] 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

[84] I. Beschastnikh, P. Wang, Y. Brun, M. D. Ernst, Debugging distributed systems, Commun. ACM 59 (8) (2016) 32–37. doi:10.1145/2909480.
URL https://doi.org/10.1145/2909480

[85] D. S. Daniels, A. Z. Spector, D. S. Thompson, Distributed logging for transaction processing, in: U. Dayal, I. L. Traiger (Eds.),

Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987, ACM Press, 1987, pp. 82–96. doi:10.1145/38713.38728.
URL https://doi.org/10.1145/38713.38728

[86] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, C. Shanbhag, Dapper, a large-scale distributed systems tracing infrastructure, Tech. rep., Google, Inc. (2010).
URL https://research.google.com/archive/papers/dapper-2010-1.pdf

[87] 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

[88] 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

[89] 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

[90] 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

[91] 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

[92] 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

[93] 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

[94] E. Korach, S. Kutten, S. Moran, A modular technique for the design of efficient distributed leader finding algorithms, ACM Trans. Program. Lang. Syst. 12 (1) (1990) 84–101. doi:10.1145/77606.77610.
URL https://doi.org/10.1145/77606.77610

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

[96] 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

[97] 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

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

[99] 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

[100] 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

[101] 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.

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

[103] 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

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

[105] 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

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

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

[108] 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

[109] 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

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

[111] 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

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

[113] Y. Jararweh, A. Doulat, O. AlQudah, E. Ahmed, M. Al-Ayyoub, E. Benkhelifa, The future of mobile cloud computing: Integrating cloudlets and mobile edge computing, in: 23rd International Conference on Telecommunications, ICT 2016, Thessaloniki, Greece, May 16-18, 2016, IEEE, 2016, pp. 1–5. doi:10.1109/ICT.2016.7500486.
URL https://doi.org/10.1109/ICT.2016.7500486

[114] 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

[115] 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

[116] 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

[117] 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

[118] 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

[119] 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

[120] 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

[121] 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

[122] 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

[123] 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

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

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

[126] 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

[127] 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

[128] 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

[129] 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.

[130] 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

[131] 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

[132] J. J. M. M. Rutten, Behavioural differential equations: a coinductive calculus of streams, automata, and power series, Theor. Comput. Sci. 308 (1-3) (2003) 1–53. doi:10.1016/S0304-3975(02)00895-2.
URL https://doi.org/10.1016/S0304-3975(02)00895-2

[133] 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.

[134] 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.

[135] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 3rd Edition, MIT Press, 2009.
URL `http://mitpress.mit.edu/books/introduction-algorithms`

[136] K. Mathews, C. Kramer, R. Gotzhein, Token bucket based traffic shaping and monitoring for wlan-based control systems, in: 28th IEEE Annual International Symposium on Personal, Indoor, and Mobile Radio Communications, PIMRC 2017, Montreal, QC, Canada, October 8-13, 2017, IEEE, 2017, pp. 1–7. `doi:10.1109/PIMRC.2017.8292201`.
URL `https://doi.org/10.1109/PIMRC.2017.8292201`

[137] M. Al-Khafajiy, T. Baker, C. Chalmers, M. Asim, H. Kolivand, M. Fahim, A. Waraich, Remote health monitoring of elderly through wearable sensors, Multim. Tools Appl. 78 (17) (2019) 24681–24706. `doi:10.1007/s11042-018-7134-7`.
URL `https://doi.org/10.1007/s11042-018-7134-7`

[138] A. A. Omar, M. Z. A. Bhuiyan, A. Basu, S. Kiyomoto, M. S. Rahman, Privacy-friendly platform for healthcare data in cloud based on blockchain environment, Future Gener. Comput. Syst. 95 (2019) 511–521. `doi:10.1016/j.future.2018.12.044`.
URL `https://doi.org/10.1016/j.future.2018.12.044`

[139] M. Simic, G. Sladic, B. Milosavljević, A case study iot and blockchain powered healthcare, in: A Case Study IoT and Blockchain powered Healthcare, 2017.

Biography

This is simple text