UNIVERSITY OF NOVI SAD

FACULTY OF TECHNICAL SCIENCES
NOVI SAD

Miloš Simić

Micro clouds and edge computing as a service

- Ph. D. Thesis -

Supervisor
Goran Sladić, PhD, associate professor

Novi Sad, 2021.

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

SERBIAN TITLE: Micro clouds and edge computing as a service

SUPERVISOR:
Goran Sladić, PhD, associate professor

LOCATION:
Novi Sad, Serbia

DATE:
September 2021

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

KLJUČNA DOKUMENTACIJSKA INFORMACIJA

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

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

KLJUČNA DOKUMENTACIJSKA INFORMACIJA

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

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

KEY WORDS DOCUMENTATION

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

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

KEY WORDS DOCUMENTATION

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

Acknowledgements

Abstract

Edge computing brings cloud services closer to the edge of the network, where data originates and dramatically reduces network latency of cloud. It is a bridge linking clouds and the users making the foundation for novel interconnected applications.

However, edge computing still faces many challenges like remote configuration, well defined native applications model, and limited node capacity. It lacks geo-organization and clear separation of concerns. As such 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, and we present a model for edge applications utilizing these adaptations.

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 give introduction to 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 show what is the current state of the art in industry and academia, but also describe positoin of this thesis compred to the related reserach.

The third chapter, propose model that is influenced with cloud computing architectural organizations, but adopted for different environemtn. We presetn how we can separate geographic area into micro data-centers that are zonaly organzied to serve local population and, and forme them dinamicaly. This chapter also give formal models for all protocols used for creation of such a system with separation of concerns, applications models and present limitations of this thesis.

The forth chapter present implemented framweork that is based on model described in chapter three. We describe architecture and in detail every operation framework is able to do with limitations of the framework.

The fith and the last chapter conclud this thesis, and present future work that should be done.

Key words: distributed systems, cloud computing, micro clouds, edge computing, platform, as a service model.

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 koliine podataka. Ovakva optereénja zahtevaju kašnjenje ispod nekoliko desetina milisekundi [1]. Ovi zahtevi su izvan onoga što centralizovani računarski modeli, 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 opterećenjima 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 čime se dolazi do smanjenja administrativnih troškova celokupnog sistema [9]. Međutim, kada takav sistem dođe 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 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 [27] 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 [27]. 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 [111]. Ako zona iz bilo kog razloga postane nedostupna, ima ih još koje mogu da opsluže 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 [111], 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 [112].

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 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 [113, 114].

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 protokola za održavanje klastera. Neki od *Gossip* protokola poput *SWIM*-a[41], mogu se koristiti u saradnji sa mehanizmima replikacije podataka [115, 116, 117] čineći ceo sistem otpornijim na potencijalne

greške.

U modelu koji opisuje sve resurse kao usluge [40], 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 raŭnarstva 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 [28, 16]. Tradicionalni centri za obradu podataka predstavljaju dobro organizovan i povezan sistem. Sa druge strane, MDC-ovi se sastoje od različitih uređaja koji nisu [29]. 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. Obično postoje nezavisno jedni od drugih, rasuti bez međusobne komunikacije, 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].

Ovo proširenje računarstva u oblaku produbljuje i jača naše razumevanje oblasti u celini. Razdvajanjem nadležnosti, modela matičnih aplikacija i objedinjenem organizacije čvorova, idemo ka ideji EC-a kao usluge.

Međutim, infrastrukture neće biti postavljena sve dok proces podešavanja i korišćenja ne bude trivijaln [26]. 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 [125], proširenje *multiparty asynchronous session types* (MPST) [126] — 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 protokola pomoću MPST-a 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 obrazloĕnje?*

Ako su prethogna istraživačka pitanja potvrdna, onda takvo 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 Simi 'c 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 moĝu 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 raĭ 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 korišćénja.

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: distributed systems, cloud computing, micro clouds, edge computing, platform, as a service model.

Table of Contents

List of Figures

List of Tables

Listings

List of Equations

List of Abbreviations

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

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

Chapter 1

Introduction

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

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

In this section, we are going to give an overview of the topics, that are of significant importance for the rest of the thesis, since it is heavely based on these topis. We start by describing the general problem area that our work addresses in Section 1.1. Sections 1.2 and 1.3 describe the theoretical background behind the problem, where we examine distributed systems (DS) and distributed computing (DC), focusing on design details, communication patters and organizational structure. In Section 1.5 we describe similar models that might be source of confusion, and how they are differnet than DS or DC and how some concepts can fit in the bigger picture. Section 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 system and/or applications. In section 1.8 we describe difference between concurency and parallelism and introduce actor system, that will be used latter 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

Cloud centralized architecture with enormous data-centers (DCs) capacities creates an effective economy of scale to lower administration cost [9]. However, when such a system grows to its limits, centralization brings more problems than solutions [8, 10]. Despite all the CC benefits, applications and services face a serious degradation over time, due to the high bandwidth and latency [11]. This can have a huge consequence on the business and potentially human lives as well. Organizations use cloud services to avoid huge investments [12], like creating and maintaining their own DCs. They consume resources created by others [13] and pay for usage time – a pay as you go model.

Data is required to be moved to the cloud from data sources, which introduces a high latency in the system [6]. For example, Boeing 787s

generates half a terabyte of data per single flight, while a self-driving car generates two petabytes of data per single drive. Bandwidth is not large enough to support such requirements [7]. Data transfer is not the only problem: applications like self-driving cars, delivery drones, or power balancing in electric grids require real-time processing for proper decision making [7]. We might face serious issues if a cloud service becomes unavailable due to denial-of-service attack, network, or cloud failure [8].

To overcome cloud latency, research led to new computing areas, and model in which computing and storage utilities are in proximity to data sources [13]. The cloud is enhanced with new ideas for future generation applications [14].

1.2 Distributed systems

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

In [30, 31] Tanenbaum et al. give two interesting assumption about DS:

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

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

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

Three significant characteristics of distributed systems are [31]:

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

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

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

In [30] authors give formal definition "distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system".

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

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

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

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

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

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

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

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

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

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

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

These fallacies are intruduced over the dacade ago, and more than four decades since we started building DS, but the characteristics and underlying problems remains pretty the same. It is interesting fact that designers, architects still asume that technology solves everything. This is not the case in DS, and these fallacies should not be forgoten. Because of these problems, DS are hard to implement corectly and they are hard to test and maintain.

1.2.1 Scalability

Scalability is the property of a system to handle a growing amount of work by adding resources to the system [32]. When talking about computer systems scalability can be represented in two flavors:

- **Scaling vertically** means upgradeing the hardware that computer systems are running on. Vertical scaling can increase performance to what latest hardware can offer, and here we are limited by the laws of phisics and Moor's law [33]. Tipical example that require this type of scaleing is relation dataase server. These capabilities are insufficient for moderate to big workloads.

- **Scaling horizontally** means that we scale our system by keep adding more and more computers, rather than upgrading the hardware of a single one. With this apporouch we are (almost) limitless how much we can scale. Whenever performance degrades we can simply add more computers (or nodes). These nodes are not required to be some high-end machines.

Table 1.1 summarize differences between horizontall and verticall scaling.

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 [32] but because this approuch

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

Table 1.1: Differences between horizontall and verticall scaling.

comes with few more benefits that are esspecially important when talking about large-scale DS. Adding more nodes gives us two important properties:

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

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

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

For example users need to be aware when using DS, is related to distributed data storage systems. Storage implementations that relys

on vertical scaling to ensure scalability and faulte tolerance, have one nasty feature.

This nasty feature is represented in theorem called **CAP theorem** presented by Eric Brewer [34], and proven after insepction by Bilbert et a. [35]. 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.

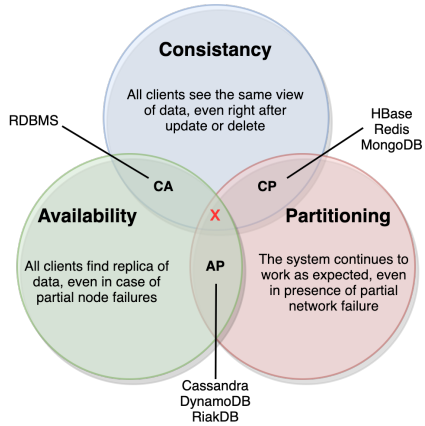

Figure 1.1: Difference between cloud options and on-premises solution.

(**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 my not be connected to the same node since data could be replicated on many nodes in different locations.

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

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

Industry is using measureing avelability in "class of nines". Availability class is the number of leading nines in the availability figure for a system or module [36]. This metric relates to the amount of time (per year) that a service is up and running. Table 1.2 show different classes of nine and their availability and unavailability in minutes per year (**min/year**) for some examples [36].

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 define as [36]:

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

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

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

Years after CAP theorem inception, Shapiro et al. prove that we can alleviate CAP theorem problems, but only in some cases, and offers **Strong Eventual Consistency (SEC) model** [37]. They prove that if we can represent our data structure to be:

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

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

1.2.2 Cloud computing

We can define cloud computing (CC) like aggregation of computing resources as a utility, and software as a service [2]. Hardware and software in big DCs provide services for user consumption over the internet [3]. Resources like CPU, GPU, storage, and network are utilities and can be used as well as released on-demand [4]. The key strength of the CC are offered services [2].

The traditional CC model provides enormous computing and storage resources elastically, to support the various applications needs. This property refers to the cloud ability to allow services, allocation of additional resources, or release unused ones to match the application workloads on-demand [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 bying and manageing hardware themself.
- **Platform as a service (PaaS)** delivers a framework for developers to create, maintain and manage their applications. All resources are managed by the enterprise or a third-party vendor.
- **Software as a service (SaaS)** deliver applications over the internet to its users. These applications are managed by a third-party vendor, .

Figure 1.2 show difference in control and management of resources between different cloud options and on-premises solution.

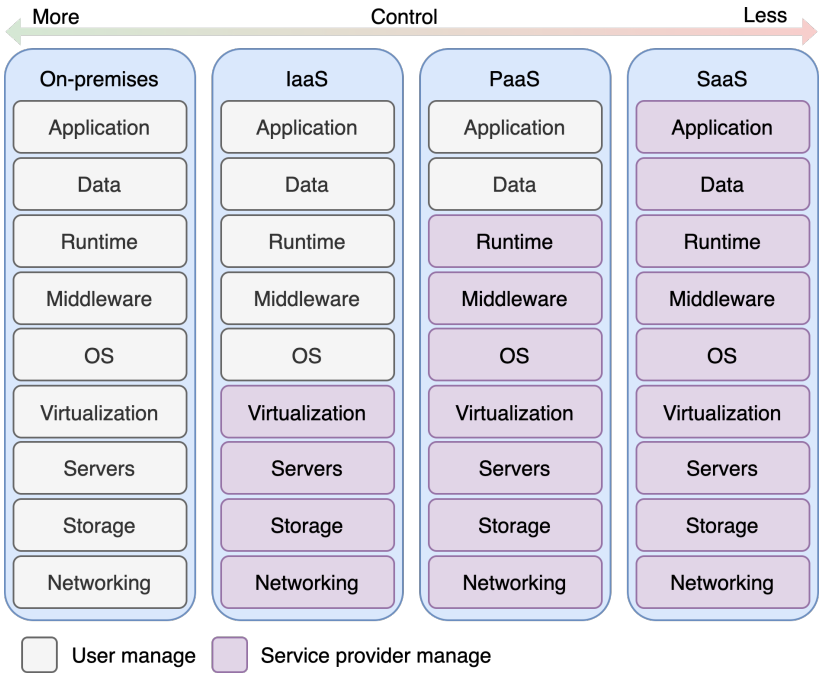

Figure 1.2: Difference between cloud options and on-premises solution.

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 type where CC is delivered over the internet, and shared across many many organizations and users. In this type of the CC, architecture is built and maintained by others. Users and organizations pay for what they use. Examples include: AWS EC2, Google App Engine, Microsoft Azure etc.
- **Private cloud** is type where CC is dedicated only to a single organization. In this type of the CC, architecture is built by organization who may offer their solution or services to the users or other organizations. These services are in domain what the organization does, and that organization is in charge of maintanance. Examples include VMWare, XEN, KVM etc.
- **Hybrid cloud** is such environment that uses both public and private clouds. Examples include: IBM, HP, VMWare vCloud etc.

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

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

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

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

CC has been the dominating tool in the past decade in various applications [13]. It is changing, evolving, and offering new types of services. Resources such as container as a service (CaaS), database as a service (DBaaS) [38] 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 – a pay as you go model.

But centralization give us few really hard problems to solve. As already stated in section 1.1 data is required to be moved to the cloud from data sources, which introduces a high latency in the system [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 read, write, send, receive etc. do not go through CPU. With this caracteristivs, RDMA have low latencies and overhead, and as such reach better throughputs [39]. This new hardware may not be cheap, and not evey CC provider use them for every use-case. And this may not be enough, esspecially with ever growing amount of IoT devices and services.

Over the years there are more as a service options avalible, forming **everything as a service (XaaS)** model [40]. This model propose that any hardware or software resource can be ofered as a service to the users over the internet.

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

CC is giving a user an illusion that he is using single machine, while the backgroud implementaion is fairly complicated and consists

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

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

of various elements that are composed of countles machines. CC is tipical example of horizontally scalable system presented in 1.2.1

1.2.3 Membership protocol

A the start of this section we introduced DS, and we present two interesting assumptions by Tanenbaum et al. [30, 31]. If we take one more look at the **(2)** assumption, we will see that user of the DS whether they are users or applications perceive DS as a single element. Inside this single elements nodes need to colaborate, so that they are albe to do various kinds of tasks.

Most basic of all these tasks, is that nodes needs to know which group they belog to, and who are their peers in the group they will colaborate with. This might sound as 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 protocols come to help. These protocols needs to ensures that each process of one group updates his local list of **non-faulty** members of the group, and when a new process joins or leaves the group, local list for every process needs to be updated. This is the most basic idea behind membership protocols.

Processes in the group, or 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 does this job, and they are (usually) based on the way epidemics spread or how gossip is spreaded in population. Because of this feature these algorithms are usually called *Gossip* style protocols.

Every membership protocol have 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 ideal world, there should be no mistakes when detecting failures. But In real life scenario, we need to reduce false positives as much as we can.

(**3**) **Failure detection speed**, all failures needs to be tected 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.

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

To resolve this problem, Das et al. [41] 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 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 fail, node will be marked as *suspected*, and it will be marked as *dead* after some timeout. If node get 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.

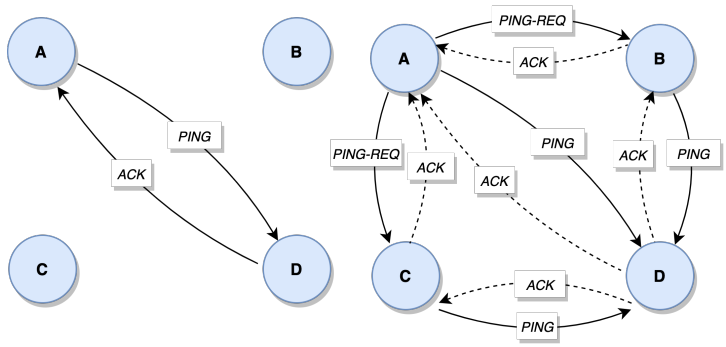

Figure 1.3: Direct and indirect ping in SWIM protocol.

Over the years, reserchers foud ways to improve the protocol for example Dadgar et al. present Lifeguard protocol [42] for more accurate failure detection, and there are other implementations to fine tune the SWIM, but base idea is still there. Today SWIM or SWIM-like protocols are standard membership procol whenever we are doing some node clustering.

1.2.4 Mobile computing

Mobile cloud computing (MCC), was the first idea that introduced task offloading [43, 44]. Heavy computation remains in the cloud. Mobile devices run small client software and interact with the cloud, over the internet using his resources.

The main problem with MCC is that the cloud is usually far away from end devices. That leads to high latency and bad quality of experience (QoE) [44]. Especially for latency-sensitive applications. Even though MCC is not that much different from the standard cloud model. We had moved a small number of tasks from the cloud. Thus opening the door for future models.

To overcome cloud latency and MCC problems, research led to new computing areas like edge computing (EC). EC is a model in which computing and storage utilities are in proximity to data sources [13]. The cloud is enhanced with new ideas for future generation applications [14].

Over the years, designs like fog [15], cloudlets [12], and mobile edge computing (MEC) [16] emerged. In this thesis, we refer to all these models as edge nodes. They all use the concept of data and computation offloading from the cloud closer to the ground [17], while heavy computation remains in the cloud because of resource availability [14].

EC models introduce small-scale servers that operate between data sources and the cloud. Typically, they have much less capabilities compared to the cloud counterparts [18]. These servers can be spread in base stations [16], coffee shops, or over geographic regions to avoid latency as well as huge bandwidth [12]. They can serve as firewalls [19] and pre-processing tier, while users get a unique ability to dynamically and selectively control the information sent to the cloud.

1.3 Distributed computing

DC can be defined as the use of a DS to solve one large problem by breaking it down into several smaller parts, where each part is computed in the individual node of the DS and coordinatio is done by passing messages to one another [31]. Computer programs that use this strategy and runs on DS are called **distributed programs** [45, 46].

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

1.3.1 Big Data

Tearm big data means that the data is unable to be handled, processed or loaded into a single machine [47]. That menas that traditional data mining methods or data analytics tools developed for a centralized processing may not be able to be applied directly to big data [48].

New tools and methos that are developed are relying on DS and one specific feature **data locality**. Data locality can be described as a process of moving the computation closer to the data, instead of moving large data to computation [49]. This simple idea, minimizes network congestion and increases the overall throughput of the system.

In 1.1 we already give two examples how huge generated data could be, and when we incude other IoT sensors and devices these numbers will just keep getting bigger [50].

On contrary to relational databases that mostly deal with structured data, big data is dealing with various kinds of data [47, 48, 49]:

- **Structured** data is kind of data that have some fixed structure and format. Tipical example of this is data stored inside table of some database. organizations susully have no huge problem extracting some kind of value out of the data.

- **Unstructured** data is kind of data where wo do not have any kind of structure at all. These data sources are heterogeneous and may containing a combination of simple text files, images, videos etc. This type of data is usully in raw format, and organizations have hard time to derive value out.

- **Semi-structured** data is kind of data that can contain both previously mentioned types of data. Example of this type of data is XML files.

Along it's share size, big data have other instantly recognizable features called **V's** of big data [51]. Name is derived from starting letters from the other features that are describing big data. Image 1.4 show 6 V's comonly used to represent the big data.

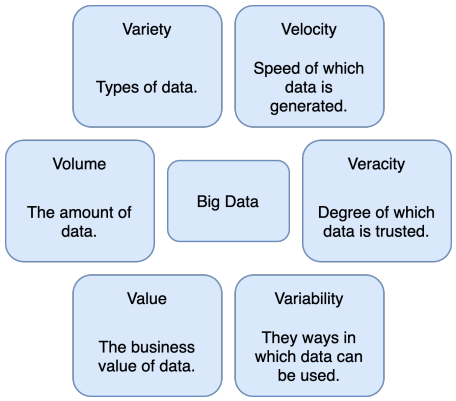

Figure 1.4: V's of Big Data.

Processing in big data systems can be represented as [52, 53]:

- **Batch processing** represents data prcessing technique that is done on huge quantity of the stored data. This type of prcessing is usually slow and requure time.

- **Stream processing** represents data processing technique that is done as data get into the system. This type of processing is usually done on smaller quantity of the data **at the time**, and it is faster.

- **Lambda architectures** represents processing technique where stream processing and handling of massive data volumes in batch are combined in a uniform manner, reducing costs in the process [53].

Big data systems, are not processing and value extracting systems. Big data systems can be separated in few categories: (1) data storage, (2), data ingestion (3), data processing and analytics. All these system aids to properly analyze ever growing requirements [54],

Dispite promise that big data offers to derive value out of the collected data, this task is not easy to do and requre properly set up system filtering and removeing data that contains no value. To aid this idea, data could be filtered and little bit preprocessed on close to the source [21], and as such sent to data lakses [55].

1.3.2 Microservices

There is no single comperhensive deffinition of what a microservice is. Differnet people and organizations use different definition do describe them. A working definition is offered in [56] as “s microservice is a cohesive, independent process interacting via messages”. Despite lack of comperhensive definition all agree on few features that come with microservices:

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

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

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

(**4**) they are organized around business capabilities [57].

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

Industry is migrating much of their applications to the cloud, because CC offers to scale their computing resources as per their usage [58]. Microservices are small loosely coupled services that follos UNIX philosophy "do one thing, and do it well" [59], and they communicate over well defined API [56].

This architecure patter is well aligned to the CC paradigm [58], contrary to previous models like monolith whose modules cannot be executed independently [56, 60], and are not well aligned with the CC paradigm [60]. Table 1.5 summrize differences between monolith and microservie architecture.

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

Table 1.5: Differences between horizontall and verticall scaling.

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

- **Cloud-native applications**, are specially designed applications for CC. They are distributed, elastic and horizontal scalable system by their nature, and composed of (micro)services which isolates state in a minimum of stateful components [61]. These type of applications are self-contained, could be deployed independently, and they are composed of loosely coupled microservices

that are packaged in lightweight containers. They have Improved resource utilization, and they are centered around APIs.

- **Serversles applications** is computing model, where the developers need to worry only about the logic for processing client requests [62]. Logic is represented as event handler that only runs when client request is received, and billing is done only when these functions are executing [62]. **Cold start** is one of features of the severless computing, and we can define it as user requests need to wait, until new container instance is up and running before can do any processing at all. Most providers have 1–3 second cold starts, and this is important for certain types of applications where latency is concern. Cold start is only happening when there are no *warm* containers available for the request, meaning there is no single instance to server request. Other features include: (1) simplified services development, (2) faster time to market, (3) and lower costs.

- **Service Mesh** is designed to standardize the runtime operations of applications [63]. As part of the microservices ecosystem, this dedicated communication layer can provide a number of benefits, such as: (1) observability, (2) providing secure connections, or (3) automating retries and backoff for failed requests. With these features, developers only focus on implementation of buisniss logic, while operators gain out-of-the-box traffic policies, observability, and insights from the services. Advocates of microservice movemant, nowdays recommend using service mesh architecture when running microservices in production environments.

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

- Representational state transfer (REST) services [64], is an architectural style with set of constraints that users can create web

srevices and interoperability between computer systems on the internet. It is based on HTTP routs to define resources, and used HTTP verbs to represent operations over these resources. It relys on textual based communications, and payload could be represented using $JSON$, XML, $HTML$ etc.

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

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

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

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

It is important to point out, that all flavors of microservices applications rely on continuous delivery and deployment [65]. This is enabled by lightweight containers, instead of virtual machines [66],

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.

and orchestration tools such Kubernetes [67]. These concepts wll be described in more detail in Section 1.6.

Microservices architecture are good starting point especially for build as a service applications, and applications that should serve huge amount of requests and users. Esspecially with benefits of CC to pay for usage, and ability to scale parts of the system independently. Although they are not necessarily easy to implement properly. There are more and more critique to the architecture model [68]. Microservices are relying and use parts of the DS, and as such they inherit almost all problems DS has.

One particilar thing that users need to be aware of is **idempotency**. In microservices applications, developers are dealing with inconsistencies in distributed state, and their operations should be implemented as idempotent. An operation is idempotent if it will produce the same results when executed over and over again. It is a strategy that means that operations with sade effects like creation or deletion can be called any number of times, while guaranteeing that side effects only occur once. Idempotency is term that come from mathematics, and can be represented by simple idempotency law for operation $*$ like [69]:

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

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

Table 1.7 show list of idempotent and non-idepotent for standard CRUD operations:

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

Table 1.7: Idempotent and non-idempotent operations.

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

Idempotency key could be stored in any kind of the storage, it is not uncommon that these keys are stored in cache storage with some time to live (TTL) policy that will automatically remove the key after specified time.

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

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

1.4 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 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 [70]. All peers are created equally in the system, and there is no such thing as a node that is more important then others. Every Peer have a portion system resources, such as processing power, disk storage or network bandwidth, directly available to other network participants, without the need for central coordination by servers or stable hosts [70]. P2P nodes are connected and share resources without going through a separate server computer that is responsabile for routing. Figure 1.5 show difference in network topology between P2P networks ($left$) and client-server architecture ($right$).

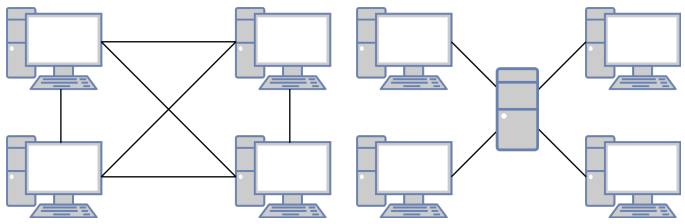

Figure 1.5: P2P network and client-server network.

Peers are creating a sense of virtual community. This community of peers can resolve a greater tasks, beyond those that individual peers can do. Yet, these tasks are beneficial to all the peers in the system [71]. When request come to such network, node that accepted

request is usually called **coordinator**, because he then is trying to found the right peer to send 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 [72]:

- **Unstructured** do not have a particular structure by design, but they are formed by nodes that randomly form connections [73]. Their strenght and weaknes at the same time is ther lack of structure. These networks and robust when peers join and leave network. But when doing query, they must found more possible peers that have same peace of data. Tipical example of this group is a Gossip-based protocols like [41].

- **Structured** peers are organized into a specific topology, and the protocol ensures that any node can efficiently search the network for a resource. The famos type of structured P2P network is a Distributed Hash Table (DHT). These networks maintain lists of neighbors to do more efficent lookup, and as such they are not so robust when nodes join or leave the network. DHT commonly used in resource loopkup systems [74], and as efficent resource lookip management and scheduling of applications, or as an integral part of distributed storage systems and NoSQL[75] databases.

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

P2P networks are great tool in many arsenals, but because their unique ability to act as a server and as a client at the same time we must be aware and pay more attention to security because they are more vulnerable to exploits [76].

1.4.2 Master-slave

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

But no metter how specialized hardware master runs on, it is prone to fail for varios reasons, so he is a **single point of failure**. If crush happend, than standby master could continue to server as a master, or new **leader election** protocol [77] is initiated to pick new master node.

Master node is responsible for processing any updates to that data. If the master fails, than 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 high-availability system.

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

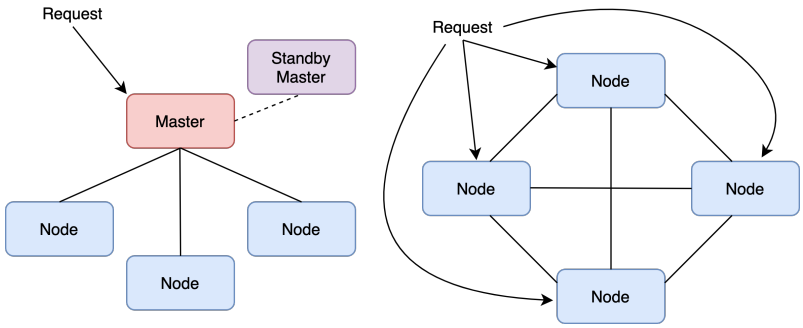

Figure 1.6: Handling requests master-slave and peer-to-peer

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 seems like models that are the same, and that may share some features like simultaneously executing a set of computations in parallel. Broadly speaking, this is not far from the truth [45].

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

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

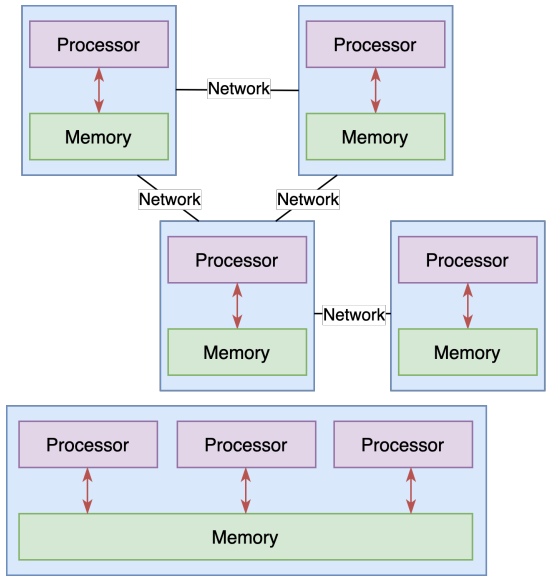

Figure 1.7: Architectural difference between DC and parallel computing.

Parallel computing is often used strategy with problems, that due to their nature or constraints must be done on multi-core machines simultaneously [78]. It is ofthen, that huge problems are divided into smaller ones, which can then be solved at the same time.

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

1.5.2 Decentralized systems

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

the owner like AWS, Google, Microsoft or some other private compay because all computation needs to be moved to big DCs [6].

By today standards, when we are talking about decentralyzed systems, we usually think of blockchain or blockchain-like technology [79]. Since here we have distributed nodes, that are scattered and there is no single entity that own all these nodes. But even if this technology is run in the cloud, it is loosing the decentralized feature. This is the caveat we needs to be aware of. These systems are facing different issues, because any participent in the system might be malicious and they need to handle this case.

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

1.6 Virtualization techniques

Virtualization as a technique started long ago in time-sharing systems, to provide isolation between multiple users shareing a single system liek a mainframe computer [80].

In [81] Sharma et al. describe virtualization as technologies which provide a layer of abstraction of the physical computing resources between computer hardware systems and the software systems running on them.

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

- **Virtual machines (VM)** are the oldest tehnology of the three. In [81] Sharma et al. describe them as a self-contained operating environment consisting of guest operating system and associated applications, but independent of host operating system. VMs enable us to pack isolation and better utilization of hardware in big DCs. They are vidly used in IaaS environment [82, 83] as a base where users can install theirown operating system (OS) and required software tools and applications.

- **Containers** provide almost same functionality to VMs, but there are several subtle differences that make them a goto tool in modern develpment. Instead of the guest OS running on top of host OS, containers use tools that are in Linux kernles like *cgroups* that limits process resource usage so that single process can not starve other processes and use all the resources for himself, and *namespaces* to provide isolation and partitions kernel resources so that single process see node resources like he only exists there. Containers reduce time and footprint from development to testing to production, and they utilize even more hardware resources compared to VMs and show better performance compared to the VMs [84, 66]. Containers provide easier way to pack servies and deploy and they are esspecially used in microservices architecture and service orchestration tools like Kubernetes [67]. Google stated few times in their on-line talks that hey have used container technology for all their services, even they run VMs inside containers for their cloud platform. Even though they exist for a while, containers get popularized when companies like Docker and CoreOS developed user-friendly APIs.

- **Unikernels** are the newest addition to the virtualization space. In [85] Pavlicek define unikernels as small, fast, secure virtual machines that lack operating systems. Unikernels are comprised of source code, along with only the required system calls and

drivers. Because of their specific design, they have single process and they contains and executes what it absolutely needs to nothing more and nothing less [86]. They are advertised that new technology that will safe resources and that they are *green* [87] meaning they save both power and money. When put to the test and compared to containers they give interesting results [86, 88]. Unikernels are still new technology and they are not widly adopted yet. But they give promessing features for the future, esspecially **if** properly ported to ARM architectures, and various development languages. Unikernes will probably be used as a user applications and functions virtualization tool, because their specific architecture, esspecailly for serverless applications presened in 1.3.2.

Figure 1.8 represent architectural differences between VMs, containers and unikernels.

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

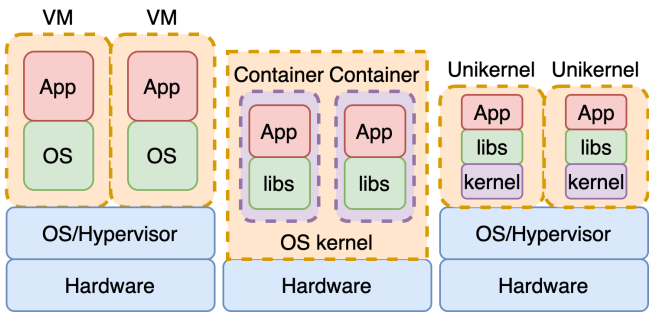

Figure 1.8: Architectural differences between VMs, containers and unikernels.

1.7 Deployment

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

Here evolve new strategy to mange and deploy complicated infrastructure elements — Infrastructure as code (IaC). In his book [89] Wittig et al. describe it as a process of managing and provisioning computer data centers through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools.

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

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

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

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

Figure 1.9 summaraze difference bewteen prevous infrastructure deplyment models.

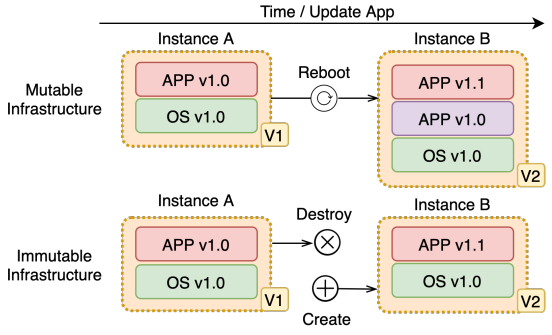

Figure 1.9: Difference bewteen mutable and immutable deplyment models

Immutability is a simple concept to understand, and simplify a lot esspecially in DS [90]. Write down some data, and ensure that it never changes. It can never be modified, updated, or deleted [91]. When this is combined with premisse that we can avoid downtime esspecially in complex DS, it is clear why immutable model is gainign more and more popularity (esspecially with arival of containers). Immutable infrastructure deployment offer few models how to deploy change on the services, even in production to test it, or switch to whole new version. These strategeis include:

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

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

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

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

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

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

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

With introduction of *LinuxKit*, we can create Linux subsystems based around containers, that are very secure. With linuxkit, every purt of the Linux subsystem is running inside container, so we can assemble a Linux subsystem with services that are needed. As a result, systems created with LinuxKit have a smaller attack surface [92] than general purpose systems. This is important from security point of view, but also from infrastructure deployment because we can compose specific OS based around containers that we need for different purpose. And we can update, change and adopt these OS for every machine or purpose we need.

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

1.8 Concurency and parallelism

People usually confuse these two concepts. Even they looks similar, they are different way of doing things. In his talk Rob Pike [93] give great explanation and examples on this topic. In this toke he give great deffinitions of these concepts like:

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

These things are important, esspecially when building applications and systems that should achieve very high throughput. We must build them with a good structure and a good concurrency model. These features enables possible parallelism, but with communication [93]. These ideas are based on Tony Hoare work of Communicating Sequential Processes (CSP) [94].

1.8.1 Actor model

In actor model, the main idea is based around **actors** which are small concurrent code, that communicate independently by sending messages, removing the need for lock-based synchronization [95]. This model propose similar idea like Tony Hoare in his work with CSP [94], and actors are ofttten confused with CSP. Table 1.8 give differences between actor model and CSP.

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

Table 1.8: Ddifferences between actor model and CSP.

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

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

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

1.9 Motivation and Problem Statement

In [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 never fits all, so the cloud should not be our final computing shift. Various models presented in 1.2.4, show possibility that computing could be done closer to the data source, to lower the latency for its clients by contacting the cloud only when needed, while heavy computation remains in the cloud because of resource availability. Send to the cloud only information that is crucial for other services or applications [21]. Not ingest everything as the standard cloud model proposes.

MDCs with a zone-based server organization is a good starting point for building EC as a service, but we need a more available and resilient system with less latency. EC originates from P2P systems [10] as sugested by López et al., but expands it into new directions and blends it with the CC. But, infrastructure deployment will not happen until the process is trivial [26]. Going to every node is tedious and time consuming. Especially when geo-distribution is taken into consideration.

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

The idea of small servers with heterogeneous compute, storage, and network resources, raise interesting research idea and motivation for this thesis. Taking advantage of resources organized locally as micro clouds, community clouds, or edge clouds [22] suggested by Ryden et al., to help power-hungry servers reduce traffic [96]. 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. We must perceive available resources,

configuration, and utilization [28, 16]. Traditional DCs is a well organized and connected system. On the other hand, these MDCs consist of various devices, including ones presented in 1.2.4 that are not [29]. This idea, brings us to the problem this thesis address.

EC and MDCs models lack dynamic geo-organization, well defined native applications model, and clear separation of concerns. As such they cannot be offered as a service to the users. They usually exist independently from one another, scattered without communication between them, offered by providers who mostly lock users in their own ecosystem. Co-located edge nodes should be organized locally, making the whole system and applications more available and reliable, but also extending resources beyond the single node or group of nodes, maintaining good performance to build servers and clusters [20].

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

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

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

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

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

This cloud-like extension makes the whole system and applications more available and reliable, but also extends resources beyond the single node. Satyanarayanan et al. in [19] show that MDCs can

serve as firewalls, while Simić et al., in [21] use similar idea as pre-processing tier. At the same time, users are getting a unique ability to dynamically and selectively control the information sent to the cloud. Years after its inception, EC is no longer just an idea [19] but a must-have tool for novel applications to come.

1.10 Research Hypotheses, and Goals

Based on reserach questions and motivation presented in 1.9, we derive the hypotheses around which the thesis is based. It can be summarized as follows:

(**1**) **Hypothesis:** *It is possible to organize EC nodes in a standard way based on cloud architecture, with adaptation for an EC geo-distributed environment. Give users the ability to organize nodes in the best possible way in some geographic areas to serve only the local population in near proximity.*

(**2**) **Hypothesis:** *It is possible to offer it to researchers and developers to create new human-centered applications. If we need more resources on one side, we can take from one pool of resources and move to another one, or organize them any other way needed.*

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

(**4**) **Hypothesis:** *It is possible to present unified model that supports heterogeneous EC nodes, with a set of technical requirements that nodes must fulfil, if they want to join the system.*

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

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

1.11 Structure of the thesis

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

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

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

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

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

Chapter 2

Research review

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

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

2.1 Nodes organization

A zone-based organization for edge servers (ES) presented by Guo et al. in [25], gives an interesting perspective on EC in a smart vehicles application. They showed how zone-based models enable continuity of dynamic services, and reduce the connection handovers. Also, they show how to enlarge the coverage of ESs to a bigger zone, thus expanding the computing power and storage capacity of ESs. Since one of

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

Baktir et al. [97] explored the programming capabilities of software-defined networks (SDN). Findings show SDN can simplify the management of the network in cloud-like environment. SDN is a good candidate for networking because it hides the complexity of the heterogeneous environment from the end-users.

Kurniawan et al. [27] argue about very bad scalability in centralized delivery models like cloud content delivery networks (CDN). They proposed a decentralized solution using nano DCs as a network of gateways for internet services at home [27]. These DCs are equipped with some storage as well. Authors show a possible usage of nano DCs for some large scale applications with much less energy consumption.

Ciobanu et al. [98] introduce an interesting idea called drop computing. The authors show that we can compose EC platforms ad-hoc, thus enabling collaborative computing dynamically, using a decentralized model over multilayered social crowd networks. Instead of sending requests to the cloud, drop computing employees the mobile crowd formed of nearby devices, hence enabling quick and efficient access to resources. The authors show an interesting idea of how to form a computing group ad-hoc. Forming ad-hoc platforms from crowd resources might raise a few possible concerns: (1) crowd nodes availability, and (2) offered resources. Crowd nodes might be an interesting idea as a backup option, in cases we need more computing power or storage and there are no more available resources to use.

MDCs are an interesting model and area of rapid innovation and development. Greenberg et al. [23] introduce MDCs as DCs that operate in proximity to a big population (on contrary to nano DCs that serves a lot smaller population), thus minimizing the latency and costs for end-users [99, 23], and reducing the fixed costs of traditional DCs. Minimum size of an MDCs is defined by the needs of the local population [23, 24], with agility as a key feature. Here agility means an ability to dynamically grow and shrink resources and satisfy the demands and

usage of resources from the most optimal location [23].

2.2 Platform models

Kubernetes [67] is an open-source variant of Google orchestrator Borg [100]. All workloads end in the domain of one cluster [67, 100, 101]. It is a great asset in CC to make cloud native applications, health checking, restarting and orchestration at scale. By design, Kubernetes operate in a completely different environment from EC. The second potential issue is the deployment concept that might be too complicated for EC workloads. On the other hand, there are a few ideas that are worth exploring, such as loosely coupling elements with labels and selectors.

Rossi et al. [101] focuses on adapting Kubernetes for geo-distributed workloads using a reinforcement learning (RL) solution, to learn a suitable scaling policy from experience. Like every other machine learning implementation this could be potentially slow due to the required model training. Kubernetes is a promising solution, but it might not be the best proposal for EC. Nonetheless, researches show that adapted Kubernetes architecture works for geo-distributed workloads like EC.

Ryden et al. [22] present a platform for distributed computing with attention to user-based applications. Unlike other systems, the goal is not to implement a resource management policy, but to give users more flexibility for application development. Users implement applications using Javascript (JS) programming language, with some embedded native code for efficiency. Similar to [98], the authors use volunteer nodes to run all the workloads, with the difference that some nodes are used for storage, while others are used for calculation. Sandboxing technique protects nodes running applications from malicious code. It is an interesting idea to show how users can develop their applications and run them in an EC environment.

Lèbre et al. [102] describe a promising solution of extending OpenStack, an open-source IaaS platform for fog/edge use cases. They try

to manage both cloud and edge resources using a NoSQL database. Implementation of a massively distributed multi-site IaaS, using OpenStack is a challenging task [102]. Communication between nodes of different sites can be subject to important network latencies [102]. The major advantage is that users of the IaaS solution can continue using the same familiar infrastructure. In [103] Shao et al. present a possible MDCs structure serving only the local population, in the smart city use-case.

In [14], the Ning et al. show current open issues of EC platforms based on the literature survey. They illustrate the usage of edge computing platforms to build specific applications. In the survey, the authors outline how CC needs EC nodes to pre-process data, while EC needs massive storage and strong computing capacity of CC.

In [92] the de Guzmán et al. present solution based on Kubernetes that use Kubernetes Deployment Manifests in order to reuse successful principles from Kubernetes by creating virtual machine for each Pod using Linuxkit. Their solution is based on the immutable infrastructure pattern, and instead of containers they use the virtual machines as the unit of deployment. Authors prove that the attack surface of their system is reduced since Linuxkit only installs the minimum OS dependencies to run containers. It represent interesting usage of LinuxKit to deploy OS dependencies, and immutable infrastructure pattern, but VMs might be a bit problem for small devices, and ARM nodes as well as complex flow of Kubernetes application model. But nonetheless, it is interesting extension of Kuibernetes framework and prove that LinuxKit can be used for immutable infrastructures with custom OS.

In [104] Sami et al. show interesting platform for dynamic sevices distribtion over Fog nodes using volunteer nodes. Their platform is tuned for container placement with relevance and efficiency on volunteering fog devices, near users with maximum time availability and shortest distance. They do this *on the fly* with improved QoS.

There are few industry operating frameworks for EC, like Amazon Greengrass [105], deeply connected to the entire Amazon cloud ecosystem, or KubeEdge [106], a lightweight extension of Kubernetes framework. These frameworks are mainly used for user-based applications, while, for instance, General Electric Predix [107] 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 effective when using cloud servers but this principle introduce long latency, which some applications can't tolerate. On the other hand mobile devices and sensors do not have sufficient battery energy for task offloading [108]. 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 [99, 17, 18, 44, 29, 108] to the nearby edge layer. These offloading techniques are based on different parameters, options, and techniques to put tasks to different sets of nodes in such a way that it won't drain mobile devices and sensors baterry. After compuitation is done, this edge layer send pre-processed data to the cloud for further analyse, storage et.

In [104] authors used Evolutionary Memetic Algorithm (MA) to solve their multi-objective container placement optimization problem to achieve better QoS.

2.4 Application models

Ryden et al. [22] present Nebula, a platform for distributed computing. Users develop their applications using JS only. This restriction comes from using Google Chrome Web browser-based Native Client (NaCl) sandbox [109]. JS is a popular language at the moment, but the restriction of a single language might be a deal-breaker for some usages. If standard virtual machines become too resource-demanding, a solution using containers could provide sandboxing and bring better resource utilization.

In [26] Satyanarayanan et al. represent an interesting view on cloudlets s a "data center in a box.". They give example that cloudleta should support the wide range of users, with minimal constraints on their software. They pun emphesis on transient VM technology. The emphasis on transient VMs is beacuse cloudlet infrastructure is restored to its pristine software state after each use, without manual intervention. In the time they conduct they reserach containers might not be working solution or it might be hard to use them. By today standards, containers may even fit better, and pack more user softvare on same hardware. This may be case for the unikernels, once they reach wider adoption rate and stable products.

Various Kubernetes variants lik [106, 101], give users possibiliety to run different applications like web servers and databses even on smaler devices cretaing green DC [20].

Satyanarayanan et al. [19] propse concept of edge-native applications that will separate space into 3 tiers.Tier 1 represent various mobile and IoT devices. These devices produce a lot of data. Tier 2 represent applications running in cloudlets or other EC models, that will pre-process, fiter data before it goes further. Finaly, tier 3 represnet classic cloud applications that will accept pre-processed and filer data from previous tier. This represent interesting concept, and give wide space for users and application developement.

In [110] Beck et al. argue that applications should use message bus,

because most mobile edge applications are expected to be event driven. Message bus system is an interesting, because virtualized applicatoin can subscribe to message streams, i.e., topics. And if for some reason mobile edge applications can't reach close EC server, it can always send data to cloud. So cloud applications should be changed so so slightly, just to accept this case.

2.5 Thesis position

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

Chapter 3

Micro clouds and edge computing as a service

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

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

3.1 Configurable Model Structure

Satyanarayanan et al. [19] propse architecture eparated into 3 tiers. Combained with MDCs and a zone-based server organization we get a good starting point for building EC as a service and micro cloud system, because we can extend the computing power and storage capacity serving local population. But, we need a more available and resilient system with less latency. To do that we need to extend these concepts and adopt them for different usage scenario.

If we take a look at the CC design, every part contributes to a more resilient and scalable system. Regions (or DCs) are isolated and independent from each other, and also contain resources application needs. They are usually composed of few availability zones [111]. If the one zone fails, there are still more of them to serve user requests. With some adaptations, EC could use a similar proven strategy. That strategy could be than used in applications beyound just EC.

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

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

Table 3.1: Similar concepts between cloud computing and ECC.

We can combine multiple node clusters into a bigger logical concept of *region*, increasing availability and reliability of both system and applications. We are talking about geo-distributed systems, so we have a bit of a different scenario than in the standard CC model. The cloud region is a physical thing [111], while in the ECC a region could be represented in a different way. We now give a forma definition of a *region* in ECC as:

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

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

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

To ensure less latency in the system, vast distance between clusters should be avoided in normal circumstances. In a CC, region extension requires physically connecting modules to the rest of the infrastructure [112].

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

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

When designing a topology, especially if regions need to share information, vast distance between regions should be avoided, if possible. With these simple abstractions, we can cover any geographic region with the ability to shrink or expand clusters, regions, and topology. Separation on *clusters*, *regions*, and *topologies* is a matter of agreement and usages similar to modeling in Big Data systems [113, 114].

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

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

Nodes inside the cluster should run some membership protocol. Gossip style protocols, like SWIM [41], could be used in conjunction with replication mechanisms [115, 116, 117] making the whole system more resilient. In [118] Simić et al. take a look from theoretical point of view on conflict free replicated data types (CRDTs) usage, to achieve SEC in EC and conclude that CRDTs could be natural fit to EC as long as we are aware of potential pitfals of CDRTs.

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

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

This 3 thier acrchitecture with numerous clients in the bottom, micro clouds in the middle, and cloud on the top kinda resembles cache level architecture in CPU [119]. On lower levels we have fastest response time, since data is on the device. But at the same time, we have very limited storage capacity and processing power. As we go on upper tiers, we have more and more storage capacity and processing power, but contrary the response time is higher and higher. Esspecially when we take distance into consideration, and huge volumes of data that need to be moved to the cloud.

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

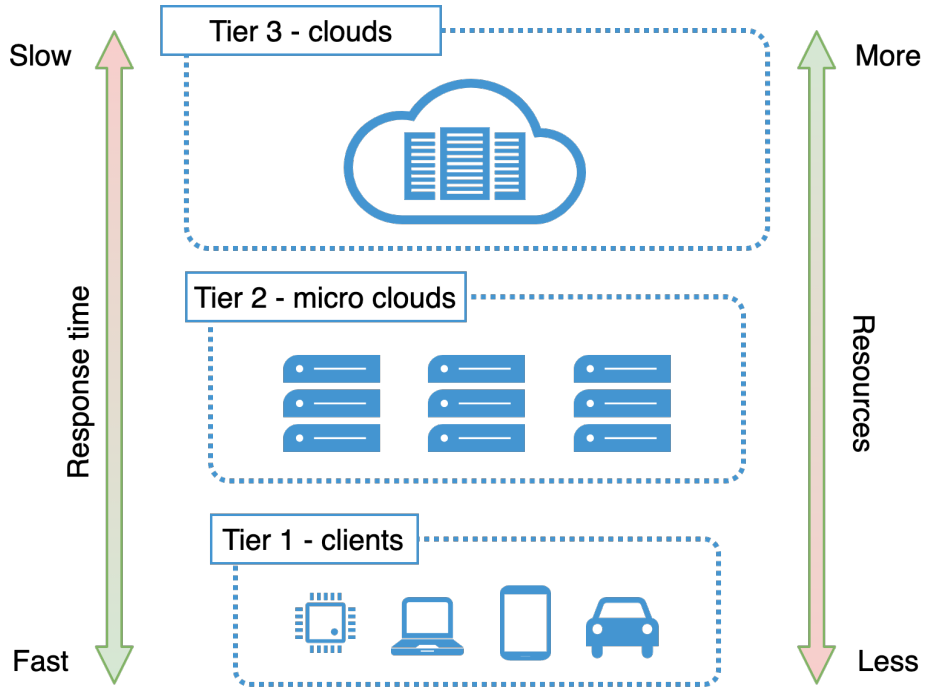

Figure 3.1: 3 tier architecture, with the response time and resource avelability

In everything as a service model [40], EC as a service fits in between CaaS and PaaS, depending on users needs.

3.2 Separation of concers

To describe physical services, Jin et al. [120] proposes three core concepts and specifies their relationships. These concepts are: (1) Devices, (2) Resources, and (3) Services.

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

The first or bottom layer consists of various devices, and these devices represents *data creators* and *services consumers*.

The second layer represents resources. Resources have a spatial features, and they indicate the range of their hosting devices [120]. Developers at any time must know topology, the resource utilization and spread, as well as application's state and health.

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

(**1**) run an operating system with a file system

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

(**3**) have available resources for utilization

(**4**) have stable internet connection

Services expose resources through an interface and make them available on the Internet [120]. These services respond to clients immediately if possible, or cache information [26, 121] for future requests. Services in the cloud should be able to accept pre-processed data, and they are responsible for computation and storage that is beyond the capabilities of ECC nodes. Services in the cloud should be able to take direct requests from client just in case that something catastrific happend to the micro-cloud that is close to the user.

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

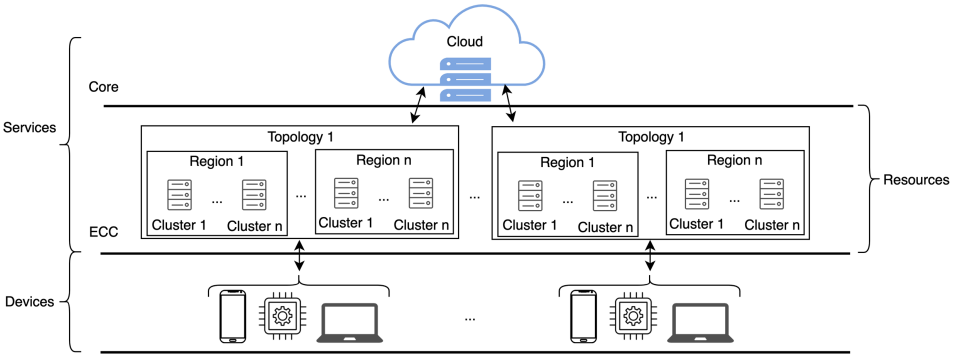

Figure 3.2: ECC as a service architecture with separation of concerns.

3.3 As a service model

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

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

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

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

3.4 Applications Model

Traditional DCs and CC propose specially designed cloud-native applications [122], that are easier to scale, more available, and less error-prone when compared to traditional web applications [122]. Edge-native applications [19] should use the full potential of EC infrastructure, and keep good features of their cloud counterparts.

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

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

3.4.1 Execution models

Frontend services model should be packed in some standard way 1.6 and deployed in the wild, as an event-driven applications, with the subscription policy to message streams using topics [110]. The processing strategy is in the developer's hands, depending on the nature of the use-case.

If an services are subscribed to some stream or topic, it is naturally that events apear in their stream. There are basically 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 event apear for some topic, that data is copied to every service that is subscribed. With this options, deads are fast, but writes are not because the more followers to the topic the longer it takes to persist all updates.

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

to implemente those ideas withoout comlicated synchronization CRDTs could be used 1.2.1, as companies like Soundcloud laready are 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 backgound without explicit user intervention, usually they 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 variantion of that protocol. For example NATS messageing system have request-replay form implemented over topics;

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

These types of applications could be implemented in may ways like those disccused in 1.3.2, or some adapted variant of those models, and should have clear communicatoin interface offered to users or applications and other services.

3.4.2 Packaging options

Because their nature, micro clouds could be most likley composed of ARM devices. These devices in many cases are not able to run full VMs because their hardware restrictions. In recent years there are advances in VMs technology and their ability to run VMs on ARM devices. In [123] Ding et al. show such possibility to run VMs on ARM devices.

But even if VMs are fully compatable with ARM devices, we still inherite VMs large footprint, already descused in 1.6 if we try to use tham *asis*.

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

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

In [124] Simić et al. show benefits of using containres in large sclae edge computing systems from theoretical point of vew, by looking into architecture difference. Authors focus on differences between VMS and containers in cases where services needs to be runned on ARM devices with limited resources.

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

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

3.5 Immutable infrastructure

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

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

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

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

(**2**) **Services deployment**, make sense only if previous action is takane. We must have infrastructure setup already, in order do put any sort of services (applications) into the system. As previous model, this should be done declaratively as well, and all changes dould be done immutably without any in place change. User should specify his new state or "view of the word" declaratively and let system do all the changes he wants. All user services should be packed as described in 3.4.2 because this simplifies way services are put to the nodes. When done properly, this allow operations people to do faster changes with almost zero downtime deployments with all strategies already discused in 1.7. this part of the deployemnt should be done by developers, since they done implementation and testing. They know hos much resources they need for their service, what type of service they had developed. This deployment could be done in colaboration between operations and deveopers, if compay is big or time is properly separated.

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

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

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

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

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

3.6 Formal model

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

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

Infrastructure deployment will not happen until the process is trivial [26], hence the key is to simplify ECC management. The main problem is that going to every node is tedious and time consuming, especially in a geo-distributed environment. In such complex environemnt, formal models are of great help if we can model and prove

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

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

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

(**3**) **idempotency check** protocol for preventing 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 of the geo-distribued Infrastructure deployment.

In Section 3.6.1 we give short introduction what MST are and what properties they guarantee. Section 3.6.2 describe formal model of healh-check protocol. Section 3.6.3 describe cluster formation protocol. Section 3.6.5 describe protocol that return data based on some query.

3.6.1 Multiparty asynchronous session types

We can model communication protocols using [125], an extension of *multiparty asynchronous session types* (MPST) [126] — a class of behavioral types tailored for describing distributed protocols relying on asynchronous communications. The type specifications are not only useful as formal descriptions of the protocols, but we can also rely on a modeling-based approach developed in [125] to validate our protocols satisfies multiparty session types safety (there is no reachable error state) and progress (an action is eventually executed, assuming fairness).

The first step in modeling the communications of a system using MPST theory is to provide a *global type*, that is a high-level description of the overall protocol from the neutral point of view. Following [125], the syntax of global types is constructed by:

$$\mathsf{G} \;::=\; \{\mathsf{p} \dagger \mathsf{q}_i{:}\ell_i(\mathsf{T}_i).\mathsf{G}_i\}_{i\in I} \quad | \quad \mu\mathbf{t}.\mathsf{G} \quad | \quad \mathbf{t} \quad | \quad \mathtt{end} \tag{3.1}$$

where $\dagger \in \{\rightarrow, \twoheadrightarrow\}$ and $I \neq \emptyset$. In the above, $\{\mathsf{p} \dagger \mathsf{q}_i{:}\ell_i(\mathsf{T}_i).\mathsf{G}_i\}_{i\in I}$ denotes that *participant* p can send (resp. connects) to one of the participants q_i, for $\dagger = \rightarrow$ (resp. $\dagger = \twoheadrightarrow$), a *message* ℓ_i with the *payload* of *sort* T_i, and then the protocol continues as prescribed with G_i. $\mu\mathbf{t}.\mathsf{G}_1$ is a recursive type, and $\mathbf{t}$ is a recursive variable, while `end` denotes a terminated protocol. We assume all participants are (implicitly) disconnected at the end of each session (cf. [125]).

The advance of using approach of [125], when compared to standard MPST (e.g., [126]), is in a relaxed form of choice (a participant can choose between sending to different participants), and, $\twoheadrightarrow$, that explicitly connects two participants, hence (possibly) dynamically introducing participants in the session. Both of these features will be significant for modeling our protocols (we will return to this point).

The second step in modeling protocols by MPST is providing a syntactic projection of the protocol onto each participant as a local type, that is then used to type check the endpoint implementations. We use the definition of projection operator given in [125, 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 $\mathsf{G} = \mathsf{q} \rightarrow \mathsf{p}{:}\ell(\mathsf{T})\ldots$ (resp. $\mathsf{G} = \mathsf{q} \twoheadrightarrow \mathsf{p}{:}\ell(\mathsf{T})\ldots$), while the projection operator "skips" the prefix of a global type if participant p is not mentioned neither as sender nor as receiver. Furthermore, a local type must be represented by the following syntax:

$$\mathsf{S} \;::=\; +\{\mathsf{q}_i\alpha\ell_i(\mathsf{T}_i).\mathsf{S}_i\}_{i\in I} \quad | \quad \mu\mathbf{t}.\mathsf{S} \quad | \quad \mathbf{t} \quad | \quad \mathtt{end} \tag{3.2}$$

where $\alpha \in \{!, !!\}$ or $\alpha \in \{?, ??\}$ (in which case $\mathtt{q}_i = \mathtt{q}_j$ must hold for all $i, j \in I$, to ensure consistent external choice subjects, cf. [125, Page 6.]), and $I \neq \emptyset$. Interested reader can find details in [125].

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

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

3.6.2 Health-check protocol

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

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

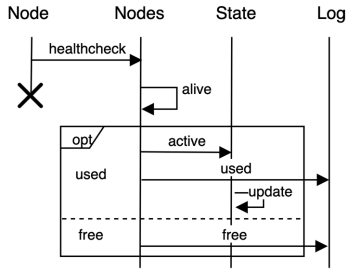

Figure 3.3: Low level health-check protocol diagram.

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

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

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

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

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

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

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

Algorithm 1: Health-check data received

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

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

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

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

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

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

If we have n free nodes in the wild, denoted with s_i, where $i \in \{1, \ldots, n\}$, and they notify the system with a health-check ping that they are free, than we should add them to the server set and thus we have

$$S_{new} = S_{old} \cup \bigcup_{i=1}^{n} \{s_i\} \tag{3.3}$$

The order in which messages arrive is not important.

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

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

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

- L is a set of ordered key-value pairs, i.e., $L = \{(k_1, v_1), \ldots, (k_m, v_m)\}$ where $k_i \neq k_j$, for each $i, j \in \{1, \ldots, m\}$ such that $i \neq j$. L represents node labels or server-specific features. We based labels on Kubernetes [101] 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

$$s_i[L]_{new} = s_i[L]_{old} \cup \bigcup_{j=1}^{m} \{(k_j, v_j) \tag{3.4}$$

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

For the label definition, we can use arbitrary alphanumeric text for both key and value, separated with colon sign (e.g., os:linux, arch:arm,

model:rpi, cpu:2, memory:16GB, disk:300GB, etc.).

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

Following the above, we now present a formal description of the low-level health-check communication protocol (cf. Figure 3.3). The global protocol G_1 (given bellow) conforms the informal description given at page 68: `node` connects `nodes` with *health_check* message and a payload of type T_1 that is required by the system to properly register node.

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

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

Notice that in G_1 we indeed have a choice of `nodes` sending either to `state` or to `log`. Such communication pattern could not be directly modeled using standard MPST approaches. Also, notice that `state` will be introduced into the session only when receiving from `nodes`. Hence, if the session after the first ping from `node` to `nodes` proceeds with the second branch (i.e., connecting `nodes` with `log`) then `state` is not considered as stuck, as it would be in standard MPST, such as, e.g., [126], but rather idle.

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

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

where, for instance, type $\mathsf{S}_{\texttt{nodes}}$ specifies `nodes` can receive the ping message from `node`, after which it will dynamically introduce either `state` or `log` into the session, where in the former case it also connects `log` (but now with message *free*).

3.6.3 Cluster formation protocol

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

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

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

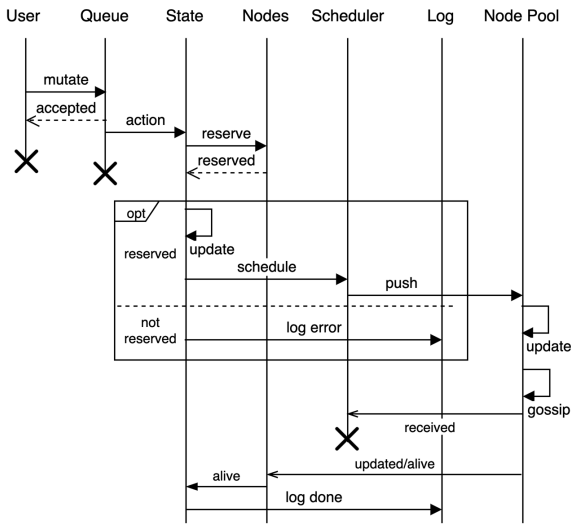

Figure 3.4: Low level cluster formation communication protocol diagram.

The participants follow the protocol that we now describe informally:

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

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

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

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

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

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

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

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

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

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

Algorithm 2: Nodes lookup

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

We start with the empty selector $Q = \emptyset$, in which we append key-value pairs. Hence, when a user submits a set of p key-value pairs we have that

$$Q_{new} = Q_{old} \cup \bigcup_{i=1}^{p} \{(k_i, v_i) \tag{3.5}$$

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

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

$$|s_i[L]| = |Q|\,, \text{ and} \tag{3.6}$$

(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:

$$P(Q, s_i) = \Big(\forall (k,v) \in Q \; \exists (k_j, v_j) \in s_i[L] \text{ such that } k = k_j \wedge v \leq v_j\Big) \tag{3.7}$$

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

$$R = \{s_i \mid |s_i[L]| = |Q| \wedge P(Q, s_i), i \in \{1, \ldots, n\}\} \tag{3.8}$$

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

$$TQ_{new} = TQ_{old} \cup \{(R, md)\}. \tag{3.9}$$

When the task comes to execution, the task queue sends messages to every node.

Algorithm 3 describes the process required for cluster formation.

Algorithm 3: Clustering formation message

input: request, config

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

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

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

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

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

Algorithm 4: Node reaction to clustering message

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

In the following, we formally describe a low-level cluster formation communication protocol (cf. Figure 1.4), using the same extension of multiparty session types [125] as for the health-check protocol.

Global protocol G_2 (given below) conforms the informal description of the cluster formation protocol given at page 74. 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 $\mu\mathtt{t}$, depending on the later choices. Further, `nodes` replies a response *resp* to `state`, that, in turn, forwards the message to `user`. The payload of the message is typed with T_2 that has response data, based on a given query. At this point, `user` sends to `state` one of three possible messages:

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

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

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

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

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

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

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

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

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

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.

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

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

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

where

$$
\begin{aligned}
\mathsf{S}' &= + \\
&\begin{cases} \texttt{nodes}?\mathit{rsrvd}().\texttt{sched}!!\mathit{ids}(\mathsf{T}_5).\texttt{nodes}?\mathit{succ}().\texttt{log}!!\mathit{succ}().\texttt{end} \\ \texttt{nodes}?\mathit{err}(\mathsf{String}).\texttt{log}!!\mathit{err}(\mathsf{String}).\texttt{end} \end{cases} \\
\mathsf{S}_{\texttt{nodes}} &= \texttt{state}??\mathit{query}(\mathsf{T}_1).\mu\mathbf{t}.\texttt{state}!\mathit{resp}(\mathsf{T}_2). \\
&+ \begin{cases} \texttt{state}?\mathit{ids}(\mathsf{T}_4).+ \begin{cases} \texttt{state}!\mathit{rsrvd}().\texttt{end} \\ \texttt{state}!\mathit{err}(\mathsf{String}).\texttt{poll}??\mathit{nids}(\mathsf{T}_4). \\ \texttt{state}!\mathit{succ}().\texttt{end} \end{cases} \\ \texttt{state}?\mathit{query}(\mathsf{T}_1).\mathbf{t} \end{cases} \\
\mathsf{S}_{\texttt{log}} &= + \begin{cases} \texttt{state}??\mathit{succ}().\texttt{end} \\ \texttt{state}??\mathit{err}(\mathsf{String}).\texttt{end} \end{cases} \\
\mathsf{S}_{\texttt{pool}} &= \texttt{sched}??\mathit{update}(\mathsf{T}_6).\texttt{sched}!\mathit{ok}().\texttt{nodes}!!\mathit{nids}(\mathsf{T}_4).\texttt{end} \\
\mathsf{S}_{\texttt{sched}} &= \texttt{state}??\mathit{ids}(\mathsf{T}_5).\texttt{pool}!!\mathit{update}(\mathsf{T}_6).\texttt{pool}?\mathit{ok}().\texttt{end}
\end{aligned}
$$

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

We remark global type G_2 could also be modeled directly using standard MPST models (such as [126]). However, in such models the projection of G_2 onto, for instance, participant `sched` would be undefined (cf. [125]). Since we follow the approach of [125] with explicit connections, projection of G_2 onto `sched` is indeed defined as $\mathsf{S}_{\texttt{sched}}$.

3.6.4 Idempotency check protocol

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

infrastructure is created, or same infrastructure already exists, user can get message that infrastructure is formed. If user change the number of labels per node, nodes per cluster, clusters per region **only** at that case we should start new protocol.

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

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

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

(**2**) **hierarhical keyspace**, with this option we can create nested data-structures of elements for example set of regions, where every region is set of clusters, where every cluster is set of nodes etc. We can go deep as long as we want, but restriction is that every structure **must** be idempotent. So we can use set of sets of sets and so on. With this option, we must test that idempotency is not violated throughout the hierarchy.

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

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

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

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

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

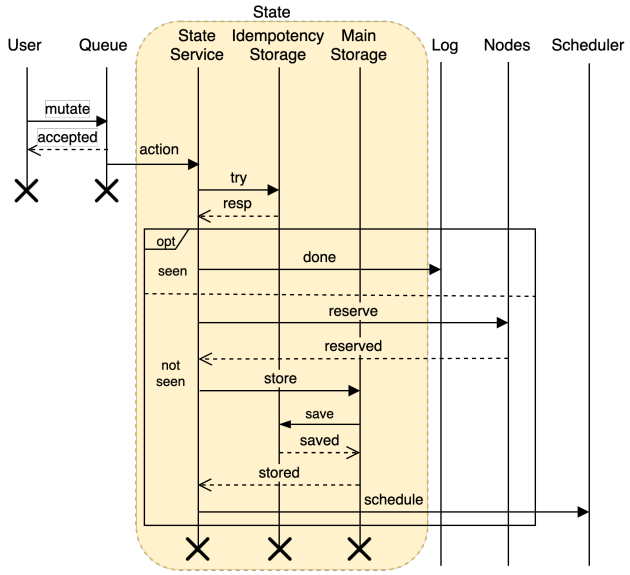

Figure 3.5: Low level view of idempotency check communication.

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 desired state for all formed topologies, regions and clusters.

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

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

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

When testing is request idempotent we must have in mind that stored structure could be (1) **flat keyspace** and (2) **hierarhical keyspace** both options are valid as log as **structure is idempotent** and we can do **idempotent operation** over that structure. For example *set* as a data structure and *intersection* are good candidates. Algorithm that will test structure idempotency must be able to test

both options. Algorithm 5 describe steps requred to test if operation is done before.

Algorithm 5: Mutate idempotency check

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

Next we present a formal description of the idempotency check protocol (cf. Figure 3.5) by using [125]. The global protocol G_3 (given bellow) conforms the informal description given at page 83. 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;

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

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

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

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.

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

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

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

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

where

$$\begin{aligned}\mathsf{S}' =& \texttt{istorage}!\mathit{try}(\mathsf{T}_3).\texttt{istorage}?\mathit{resp}(\mathit{Boolean}).\\ & + \begin{cases}\texttt{nodes}!\mathit{ids}(\mathsf{T}_4).\texttt{nodes}?\mathit{rsrvd}().\mathsf{S}''\\ \texttt{log}!\mathit{done}().\texttt{end}\end{cases}\\ \mathsf{S}'' =& \texttt{mstorage}!\mathit{store}(\mathsf{T}_3).\texttt{mstorage}?\mathit{saved}().\\ & \texttt{sched}!\mathit{ids}(\mathsf{T}_5).\texttt{end}\\ \mathsf{S}_{\texttt{mstorage}} =& \texttt{service}?\mathit{store}(\mathsf{T}_3).\texttt{istorage}!\mathit{store}(\mathsf{T}_3).\\ & .\texttt{istorage}?\mathit{saved}().\texttt{service}!\mathit{stored}().\texttt{end}\\ \mathsf{S}_{\texttt{istorage}} =& \texttt{service}?\mathit{try}(\mathsf{T}_3).\texttt{service}!\mathit{resp}(\mathit{Boolean}).\\ & \texttt{mstorage}!\mathit{store}(\mathsf{T}_3).\texttt{mstorage}!\mathit{saved}()\\ \mathsf{S}_{\texttt{log}} =& \texttt{iservice}?\mathit{done}(\mathsf{T}_3).\texttt{end}\\ \mathsf{S}_{\texttt{nodes}} =& \texttt{service}?\mathit{ids}(\mathsf{T}_4).\texttt{service}!\mathit{rsrvd}().\texttt{end}\end{aligned}$$

Similarly as for G_2 we remark G_3 could also be modeled using standard MPST (e.g., [126]), but again the projection types would be undefined, while following the approach of [125] with explicit connections, we have obtained all valid projections.

3.6.5 List detail protocol

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

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

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

It is important to note, that similar to the query operation, both rules (3.6) and (3.7) must be satisfied in order for information to be present in the response.

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

Figure 3.6. shows a low-level view of the list operation protocol, where users can get details about the formed system. In this setting, the system involves participants: User, State, Nodes, and Log.

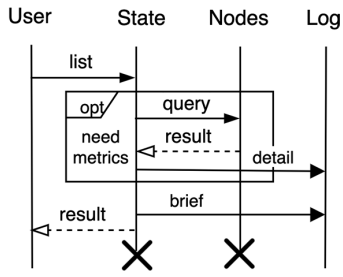

Figure 3.6: Low level view of list operation communication.

We now informally describe the participants roles in the protocol:

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

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

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

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

Algorithm 6 describes steps 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 [125]. 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.

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

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

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

where

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

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

Similarly as for G_2 and G_3 we remark G_4 could also be modeled using standard MPST (e.g., [126]), but again the projection types would be undefined, while following the approach of [125] with explicit connections, we have obtained all valid projections.

3.7 Repercussion

Model presented in this chapter, have two possible repercussions:

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

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

3.8 Limitations

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

(**1**) not all companies and organizations will be able to deploy them, due to the high investments required [12]. Similar to the cloud where massive DCs are built by a few companies and used by many others. These small servers could be deployed by government authorities or large cloud companies for their own needs. The general public can use them, similarly to the cloud — pay as you go, model.

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

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

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

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

Chapter 4

Implementation

In this chapter, we are going to give more details about framework implemented based on formal model and architecture specification given i previous chapter. Section 4.1 framework architecture, 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 microclouds platform. In Section 4.3 we present results of our experiments.

4.1 Framework

In this section we are going to introduce implemented framework based on the model proposed in previous chapter 3. Framework is called *Constellations* or **c12s** for short, because it is strongly influenced by nature and neverending number of galaxies that universe is (not only) composed of. In a similar way, we are trying to create universe of clusters that will server humanity to help them with their day-to-day tasks.

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 request arrive at gateway, if request is valid it will pass request to the rest of the system. It communicate to rest of the services to check if user exists, does he have proper rights for actions that he send, and if not return proper message and do not propagete it to the rest of the system.

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

- **Queues**, purpose of this service is to prevent huge request load to the system, and to accept more user requests. When user submit any **mutation** operation — operation that change state of the system, thse opreations will be put in the queue. User can create their own queues, to prevent long lines for specific tasks. For example, user can crete queues for specific tasks, and use them only for those tasks, while other queues could be general purpos queues. On system start, evey user will start with one queue — **default**. When doing mutations on different parts of the system, user can specify in matadata which queue he wants task to go to. This service implement token bucket rate limiting algorithm [127] to prevent congestion of the system.

- **Nodes**, this service stores and maintain informations about registered nodes in the system. All node hardware and software

details will be stored in this here. This service is also responsabile for storeing metrics data, accept health-check requests from nodes, and inform rest of the system that used node is alive.

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

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

- **Command push**, purpose of this service is to push commands and operations to the nodes user desired. This service implement token bucket rate limiting algorithm [127] to prevent constant data push to the nodes. As State service, this service will store a copy of operation information localy before attempting any push to the nodes. This information will be deleted, once operation reach all desided nodes, and all of them response with ACK message.

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

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

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

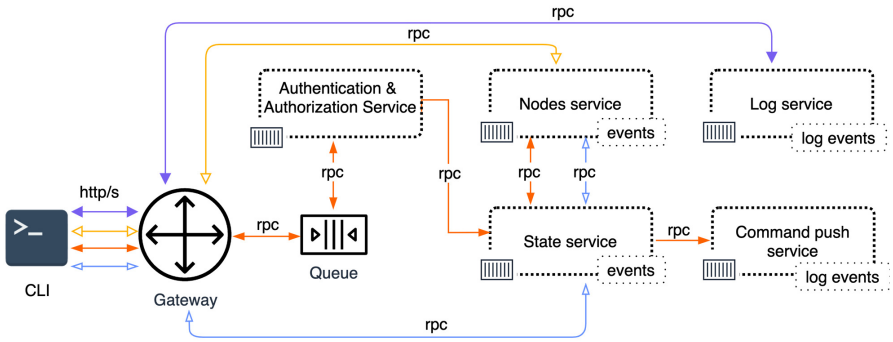

Figure 4.1: Proof of concept implemented system.

4.1.1 Technologies

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

As a main storage layer for our system, we used etcd, a popular open-source key-value database, that show good performance, and it

is mostly used for configiration data. Metrics data are stored in the open-source time-series database InfluxDB.

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

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

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

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

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

Log data will be stored in the two levels:

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

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

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

4.1.2 Node daemon

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

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

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

System messages will be send to the daemon, but he will not react to these messages. Daemon is not able to communicate with any actor dyrectly. All communication goes trough actor system who is responsabile to pass messages to the actors. Actor system at this point have only 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 on messages from other actors in order to update daemon state. For example on new cluster creation, this actor will update daemon id, name, labels etc.

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

(**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 in order that any of them crush for whatever reason, actor system will restart them. Listing 4.1 show actor system hierarchy of existing actors.

```
1 \StarSystem
2   +--\TopologyActor
3   +--\HealthcheckActor
4   +--\GossipActor
5   +--\InternalActor
```

Listing 4.1: Actor system hierarchy.

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

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

(**2**) **name**, represent name of the node. This property also can be changed when 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 are no formal restriction of what they can or can not be. This property can be changed when node is part of some cluster. It is advised, that as a labels

we put some specific features of the node that might be beneficial for user who is looking for nodes to create new cluster/s.

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

(5) **system informations**, represent basic informations for node to know how to contact rest of the system. We should specify system address, so that node knows where to send messages, and where are messages are coamming from. We can also specify 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 of time.

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

```
1  star:
2    version: v1
3    nodeid: node1
4    name: noname
5    flusher: address
6    healthcheck:
7      address: address
8      topic: health
9      interval: 1m
10   labels:
11     os: linux
12     disk: flash
13     arch: arm
14     model: rpi
```

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

Listing 4.2: Daemon configiration file

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

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

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

4.1.3 Separation of concerns

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

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

Upper layer must have set up infrastructure in order to do any processing od storage. This middle layer is binding element between the layers.

4.1.4 Limitations

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

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

The current implementation of 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 managment, but micro-cloud formation, protocols and formal modeling.

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

4.2 Operations

In this secion we are going go describe all implemented operations in the framework, and details about every individual operation.

4.2.1 Query

Query operation is used to show all free nodes or filtered free nodes that are registered into to 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 key an value. 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 3.8 defined in Section 3.6.3 will be present in the result.

This operation is done prior to formation of new topologies, regions or clusters — mutation of the system. User first need to get list of nodes that are free, than he can choose nodes that are best suited for him and try to form topologies, regions or clusters of them.

Querying process is 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 to prevent overflow of requests. This change **does not** affect or valiate formal model presented before. Since the added service do not interfere process of searching nodes or changes to the system.

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

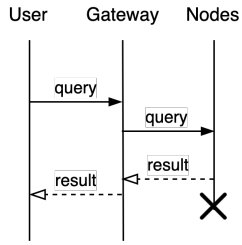

Figure 4.2: Low level communication protocol diagram of query operation.

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, a desired state needs to be specified **declaratively** using a YAML file, and submitted to the system. When state is submitted, the system will do the rest of the job to ensure that user desired state 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 labels to the entire topology. These parameters will be used when the user wants to query all topologies to get full information about regions, clusters, and nodes inside a topology

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

```
1 \topology\region\cluster\node
2   +--\labels
3   +--\informations
4   +--\resources
```

```
5     +--\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 faster querry of elements, when doing labels comparison. To find elements we can query on similar way like doing search in files and folders structure. The etcd in newer version **do not allow** hierarchical keyspace, but what they allow is **ranged** query by some **prefix**. This is useafull as well, because we can still get all regions in topology, clusters in region or nodes in the cluster if we know to which group they belog.

Mutate operaption is not idempotent by the nature, but whole process behind it make mutate an idempotent operation. Whether user try to create already existing infrastructure by changeing order of regions clusters or labes in the nodes, or if he for whatever reason did not get confirmation that infgrastructure is created a existing infrastructure will not be created. This is done in such a way, that *State* service (Figure 4.1) will keep information set about created infrastructure.

This informations is implemented as a **flat keyspace** set, that have information. On every mutation request, system will tets if such 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 unique id which user can use to query steps, logs and traces that are done in process of working towards user desired state. Logs service can give user comlete details about his task state using that unique id.

When creationg topologies, user is free to give whatever name he wants for every region, cluster and node. The only restrictino is that name should be alphanumeric set of characters.

Listing 4.4 show example of user defined state that forms topology of one region with one cluster that contains three nodes, with retention period of 24h. Whole topology will have same set of labels, but *node*3 redefine this rule by specifying his 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 [41], a Gossip style protocol. When every node have 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 sytem that he is part of some cluster. This kind of messages, will be used in *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, sers 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 do 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 views as a "gentleman agreement", that in one team, operations users can proclaim specific queues like *mutatation* queus used maybe for specific topologies, and latter on for other operarations as well.

The queue service starts only $default$ queue exists. Adding new queue to the system is implemented using configuration file, for convenance only. Listing 4.5 show example of queue service with two additional queues with specifications of their parameter needed for token bucket opeation **??**. Also we can see configuratoin informations for instrumentation of single service, and same configuration is implemente 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

List command show 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 maintaing. All other infrastructures, craeted 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 set of labels or list of key-value pairs which will be used by the system to determine what user want to se. This process is similat 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 usefull if operations people need different details level 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, user needs to be registered and logged in to be able to performe this action. With this action, the user can see the state of his operations and actions. The user can choose between two options to querying his logs:

(**1**) **get**, for this option user need to provide unique task id that is given to the uers when he creates a mutate operation. With this option 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 huge amoount of data that require some better visualizaion than CLI.

4.3 Results

Framework described in previous section is put the the test. For ease of testing, we have used few ARM based nodes that easy to move from place to place. Test should be conducted on the heterogenious nodes, to see how the system will react on 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 heterogenious 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 $one - minute$ interval. Logs, resources and other node details are set to $fifteen - seconds$ 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 **??**, and we get message that clusters already exists.

To increase capaticity, we extended clusters by creating new ones using $threeclusters$ 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 $threeclusters$ 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 focus on a platform with geo-distributed edge nodes can be organized dynamically into the micro data-centers and regions based on the cloud model, but adapted for a different environment. This middle layer helps the power-hungry servers reduce traffic by serving the nearby population requests and possibly syncing their data without expensive consensus protocols [118]. These micro data-centers or micro clouds also increase the availability and reliability of the cloud services, and as such could be offered as a service to users.

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

If we imagine a scenario where a new catastrophic event (earthquake, tsunami, war, terrorist attack, pandemic, etc.) is in the human population, an increasing amount of ambulance vehicles must be routed to hospitals fast, in the city area for example. The traffic

control system suddenly needs more resources to continue operating properly. On top of that, if the healthcare system is like the one presented in [128, 129], we can monitor patient health in real-time [130] and transfer data to the healthcare platform.

This gives health workers crucial information about patients on their arrival. Resource vise, this scenario is relatively easy to solve if using the cloud, as we increase resource demand. The main advance of EC, when compared to the cloud only approach, is the acceleration of the communication speed. In our scenario, the cloud could bring huge latency, while EC originates from peer to peer systems [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, cloud-like applications.

One more example where health workers can benefit, is if we imagine that we put sensors on users and monitor them in real-time. We can process these streams of data direcly close to where they are. But also we could be using mobile devices to detect users position, and monitor air pollution polution and make decisions in rela time to suggest users to not walk in some are esspecially if they have some known respiratory problem.

Geo-distributed nodes represent a great idea to do any kind a monitoring and processing esspecially for naturel phenomenons, and do alerting as soon as probes, sensors or other thing detect even 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.

Users are getting a new platform as a blank canvas, and there is no limitation in what applications they can develop. Integrating hardware and/or software even more, connecting sensors and things around us and eventually an operating system that will be capable of

running city/state infrastructure without human intervention.

Chapter 5

Conclusion

In 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

This thesis presents a possible solution for the organization of geo-distributed micro-clouds with EC nodes, with addition of few proven abstractions from cloud computing like zones and regions.

These abstractions allow coverage of any geographic area, and yield a more available and reliable EC system. The organization and reorganization of these abstractions is done descriptively, and their size is determined by the population needs.

We present a cloud to EC mapping and give a formal model of the system, with clear SoC and edge-native application model for future EC as a service development. The thesis also presents a proof of concept implementation and discusses integration into existing solutions with few applications that could be used in.

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

We show what distributed systems are or at least general consensus how we can describe some system as a 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 membershipthat protocol that are important in distributed environment for various reasons.

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

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

In Chapter 2, we show related work done by various other researchers or companies, and again we focuse, 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 komputing. Futher we show implementations of few new platforms to use volounter 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 are into zones, and show how MDCs can help CC to serve requests from local population. There are different offloading techniques that are used today, how to offload tasks from mobile devices closer to fog or edge nodes, but also various applicationas models that could harnes these offloading techniques and nodes organizasions.

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

In Chapter 3, we present heart and soul of this thesis. We deisect all importatnt asspects that we need to have in order to help CC with latency issues, Big Data with huge volumes of data esspecialy in the age of mobile devices and IoT.

Our model is based around MDCs that are zonaly organzied, that will serve only local populations and populations nearby. We present model that is based on CC, but adopted for different scenario and use case.

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

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

At the end, we present repercusions of this model, and how can be used as an integral part of existing systems to server as a nodes storage, or can be used as a new model where we can develop new subsystems and applicatoins on top. We present protocols for creation of such a system, and model them using asynchronous session types. System follow formal model, and it is easy to extend.

At 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 last Chapter 4, we present implemented framework that is based on knowladge compiled from previous chapters. We also presetn in

detail operations that could be done in the framework, where it fits in presetned SoC model.

Further we present results of our experimets in controlled environment, what are the limitations of the framework at the curretn 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 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 into 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 cluster user should be able to do remote configurations, and setting up the data without going from node to node that nodes and/or applicatoins can use.

System should also be extending with namespaces for usage in environments with many users in multiple teams — multi tenant environment. Namespaces provide the separation on virtual clusters, runnin on the same physical hardware. Speaking about multi tenancy, we are

also planning to implement role-based access controll integrate with authentification and authorizatoin services. With the addition og controlling different users with quotas, using reate 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 system. We might also consider rethinking networking and makeing network isolation so that once formed topologies can communicate within herself, and a possibly to spefiсy stretegies of communication with other topologies.

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

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

Another direction for future work, is the implementation which developers could benefit from. First thing that should be implemented here is full applications framework so that users can start developeing services that can actually do something. We should also also implement framework and maybe domain specific langiuage for use case where users just want to pre-processing the data before send it to the cloud, on more convinient than writing 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.

Second would be file system and databses APIs that users can use to store their data. We should also provide interface for extensions, so

that others can create their own databases following different models from which developers can benefits, but also integrating existing ones.

Last part of future woulr would be extending 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 sbould 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.

This thesis in section 3.7, stated that this model could be integrated into existing solustios. 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 a same way or almost the same way as 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] 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

[27] 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.

[28] 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

[29] 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

[30] 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

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

[32] 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

[33] 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

[34] 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

[35] 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

[36] 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

[37] 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

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

[39] 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

[40] 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

[41] 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

[42] 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

[43] 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

[44] 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

[45] 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

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

[47] 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

[48] 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

[49] 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

[50] 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

[51] 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

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

[53] 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

[54] 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

[55] 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

[56] 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

[57] 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

[58] 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

[59] 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

[60] 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

[61] 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

[62] 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

[63] 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

[64] 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

[65] 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.

[66] 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

[67] 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

[68] 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

[69] 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

[70] 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

[71] 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

[72] 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

[73] 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

[74] 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

[75] 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

[76] 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

[77] 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

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

[79] 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

[80] 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

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

[82] 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

[83] 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

[84] 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.

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

[86] 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

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

[88] 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

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

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

[91] 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

[92] 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

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

[94] 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

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

[96] 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

[97] 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

[98] 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

[99] 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

[100] 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

[101] 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

[102] 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

[103] 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

[104] 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

[105] 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

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

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

[108] 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

[109] 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

[110] 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

[111] 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

[112] 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

[113] 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

[114] 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

[115] 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

[116] 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

[117] 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

[118] 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.

[119] 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

[120] 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

[121] 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

[122] 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

[123] 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.

[124] 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.

[125] 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

[126] 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

[127] 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

[128] 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

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

[130] 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