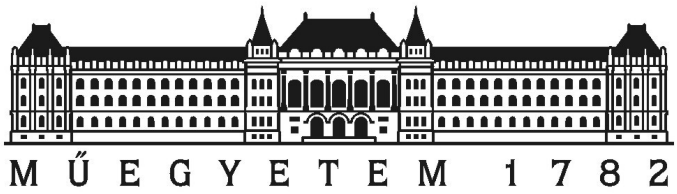


Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Hálózati Rendszerek és Szolgáltatások Tanszék

Felhőalapú mobilitáskezelés szolgáltatás-menedzsmentjének megvalósítása az ONAP orkesztrációs platform zárt láncú szabályozókörének felhasználásával

Lami Edina
TFTXG4

Konzulensek:
Leiter Ákos, Dr. Bokor László

2021. október 28.

Kivonat

A felhasználói igények megváltozása miatt szignifikánsan megnövekedett hálózati forgalom óriási kihívások elé állítja a mobil telekommunikációs szolgáltatókat. A probléma oka nem csak az adatforgalom mértékének ugrásszerű növekedéséből, hanem a felhasználók által elvárt minőség folyamatos garantálásából is fakad. Az újonnan kialakuló felhasználói igények az évek során elvezettek a különböző hálózati funkciók virtualizálásához (Network Function Virtualization) [1] és a hardver erőforrások minél hatékonyabb felhasználásához. A technológia fejlődése és az ezzel együtt egyre növekvő felhasználói mobilitás [2] implikálta a felhőalapú (cloud-native), virtualizálható mobilitáskezelés megvalósítását is. A felhőalapú implementációnak nem csak gazdasági előnyei vannak, hanem az így kapott rendszer egy zárt láncú szabályozókör segítségével hatékonyan menedzselhetővé is válik, amely által a szolgáltatás folyamatosan, az elvárt minőségben biztosítható lesz. A zárt láncú szabályozókör az orkesztráció alapvető eszköze, amely a futó rendszerről információkat gyűjt és továbbítja azokat különféle adatfeldolgozó és adatelemző mechanizmusok irányába, majd képes a rendszer működésébe, annak hatékonyabb működtetése érdekében beavatkozni. A TDK dolgozatom során egy már meglévő, IP szintű mobilitáskezelést támogató Mobile IPv6 Home Agent (MIPv6 HA) [3] szolgáltatásmenedzsmentjét valósítom meg az ONAP (Open Network Automation Platform) platformot [4] felhasználva. Az ONAP egy nyílt-forráskódú rendszer, amely zárt láncú szabályozókörökön keresztül tesz lehetővé különböző automatizációhoz és menedzsmenthez kapcsolódó operációkat. Dolgozatomban bemutatom az IP-szintű mobilitáskezelési protokollokat és ezek hasznosíthatóságát felhőalapú környezetekben. Feltérképezem a zárt láncú orkesztrálás és a MIPv6 HA szolgáltatás együttműködési lehetőségeit, majd egy konkrét szabályozókör megvalósításával bemutatom, hogy mekkora potenciál rejlik a cloud-native alapú megoldásokban. Valós implementációra támaszkodó, szolgáltatói szintű felhőinfrastruktúrát megvalósító tesztrendszeren végzett méréseimmel bizonyítom, hogy ezen innovatív megközelítés alkalmazásával egy hatékonyabban és megbízhatóbban üzemeltethető mobilitáskezelési hálózati architektúra alakítható ki, úgy, hogy közben nincs szükség drága, dedikált céleszközök vásárlására a megoldás működtetéséhez. Az általam javasolt megoldás előnyeire az 5G hálózatok és evolúciós utódjaik egyaránt támaszkodhatnak a felhasználóik IP szintű mobilitáskezelésének skálázható biztosításához.

Abstract

Significantly increased network traffic due to changing user demands is a huge challenge for mobile telecom operators. The problem is caused by the surge in data traffic volumes and the need to guarantee the quality that users expect at all times. Over the years, emerging user demands have led to the virtualization of various network functions (Network Function Virtualisation) [1] and the more efficient usage of hardware resources. The evolution of technology and the increasing mobility of users [2] have also implied the implementation of cloud-native, virtualizable mobility management. Not only does a cloud-based implementation have economic benefits, but the resulting system can also be effectively managed through a closed-loop control system, whereby the service can be continuously provided at the expected quality. The closed-loop control system is a primary orchestration tool that collects information from the running service and transmits it to various data processing and analysis mechanisms. It can intervene in the system to make it run more efficiently. The TDK thesis is about implementing service management for an existing Mobile IPv6 Home Agent (MIPv6 HA) software [3] supporting IP-level mobility management, using the Open Network Automation Platform (ONAP) [4]. ONAP is an open-source platform that enables various automation and management-related operations through closed-loop control circuits. The thesis presents IP-based mobility management protocols and their usability in cloud environments. It also explores the interoperability of closed-loop orchestration and MIPv6 HA service, and demonstrates the potential of cloud-native solutions by implementing a concrete policy framework. Measurements on an actual implementation-based testbed will prove that this innovative approach can be used to build a more efficient and reliable mobility management architecture without the need to purchase expensive dedicated target devices to run the solution. Both 5G networks and their evolutionary successors (i.e., beyond 5G) can rely on the benefits of the proposed solution to provide scalable IP-level mobility management for their users.

Köszönetnyilvánítás

Ezúton szeretném megköszönni témavezetőimnek, Leiter Ákosnak és Dr. Bokor Lászlónak a dolgozatom elkészítése során nyújtott támogatásukat, szakmai tanácsaikat, valamint építő jellegű megállapításaikat.

Köszönettel tartozom Hegyi Attilának és Galambosi Nándornak a dolgozat készítése során alkalmazott technológiák felhasználásában nyújtott segítségükért, tanácsaikért, valamint Salah Mohamed Saleh-nek és Huszti Dánielnek, akik megteremtették a dolgozatom alapjául szolgáló felhőalapú alkalmazást és annak infrastruktúráját.

Külön köszönet illeti meg a Nokia Bell Labs valamennyi vezetőjét és munkatársát, akik egy erős infrastruktúra biztosításával, valamint a dolgozat írásával kapcsolatban felmerülő problémák megoldásában nyújtott segítségükkel lehetőséget teremtettek számomra a jelen dolgozat elkészítéséhez. Segítségükkel jelentős mértékben hozzájárultak a dolgozatom elkészítéséhez szükséges mennyiségű és minőségű tudás- és ismeretanyag megszerzéséhez, elsajátításához.

Végül, de közel sem utolsó sorban szeretném megköszönni valamennyi családtagomnak azt a szavakkal ki sem fejezhető mértékű támogatást, segítséget és szeretetet, amivel lehetővé tették számomra, hogy elérkezhessen ez pillanat is, amikor éveken át tartó megfeszített tanulás és munka után leadhatom az ez idő alatt megszerzett szakmai tudásomról számot adó jelen dolgozatot.

Tartalomjegyzék

Táblázatok jegyzéke

Ábrák jegyzéke

Rövidítések jegyzéke

3GPP 3rd Generation Partnership Project

ADC Application Delivery Controller

API Application Programming Interface

BC Binding Cache

BGP Border Gateway Protocol

BU Binding Update

CapEx Capital Expenditure

CBA Controller Blueprint Archive

CCVPN Cross Domain and Cross Layer VPN

CD Continous Development

CDS Controller Design Studio

CI Continous Integration

CLI Command Line Interface

CN Corresponding Node

CN-MIPv6 Cloud-Native MIPv6

CNF Cloud-Native Network Function

CNI Container Network Interface

CoA Care-of-Address

COTS Commercial off-the-shelf

CP Control Plane

CPU Central Processing Unit

DBaaS Database-as-a-Service

DCAE Data Collection, Analytics and Events

DP Data Plane

DST Destination

ETSI European Telecommunications Standards Institute

GUI Graphical User Interface

HA Home Agent

HA-PP Home Agent-Packet Processor

HoA Home Address

HPA Horizontal Pod Autoscaler

HTTP HyperText Transfer Protocol

IaaS Infrastructure as a Service

IoT Internet of Things

IP Internet Protocol

IPv6 Internet Protocol version 6

ISG Industry Specification Group (ETSI)

JSON JavaScript Object Notation

LB Load Balancer

MANO Management, Automation and Network Orchestartion

MIPv6 Mobile Internet Protocol version 6

MN Mobile Node

MoIP Mobile IP

NFV Network Function Virtualization

NFVi Network Functions Virtaulization infrastructure

noSQL not only SQL

O-RAN Open Radio Access Network

ONAP Open Network Automation Platform

OpenECOMP Open Enhanced Control, Orchestration, Management & Policy

OpEx Operational Expenditure

OS Operation System

OSPF Open Shortest Path First

P4 Programming Protocol-independent Packet Processors

PNF Physical Network Function

PP Packet Processor

R Router

RAM Random Access Memory

REST Representational State Transfer

RFC Request for Comments

RIP Routing Information Protocol

SDC Service Design and Creation

SDN Software Defined Networking

SLA Service-level Agreement

SO Service Orchestartor

SQL Structed Query Language

SRC Source

SSH Secure Shell

TCO Total Cost of Ownership

TCP Transmission Control Protocol

TMForum TeleManagement Forum

vCPE Virtual Customer Premises Equipment

vDNS Virtual DNS

VES VNF Event Streaming

vFW Virtual FireWall

VM Virtual Machine

VNF Virtual Network Function

VoLTE Voice over LTE

1. fejezet

Bevezetés

A mobil telekommunikációs szolgáltatókat a felhasználói szokások szignifikáns változásai és a hálózati forgalom exponenciális növekedése az utóbbi években — a szolgáltatás minőségének fenntartása érdekében — óriási kihívások elé állította. A felhasználók megnövekedett mobilitás iránti igénye jelentősen megnöveli a hálózat komplexitását, mely növekedés következtében a hálózati funkciók virtualizációja (VNF) egyre nagyobb teret hódít. Skálázhatóságának és hatékonyságának köszönhetően a megvalósításra a legideálisabbnak a cloud-nativ környezet ígérkezik.

Ez a mai modern megközelítés vezetett a mobilitást támogató Mobile IPv6 (MIPv6) Home Agent felhő alapú implementációjához. Azonban ahhoz, hogy a megfelelő funkcionalitás biztosítható legyen, szükség van a rendszer menedzsmentjére. Erre a célre kiválóan használható, nyílt forráskódú platform az Open Network Automation Platform (ONAP), amely zárt láncú szabályozókörökön keresztül számos lehetőséget kínálva támogatja a menedzsmenthez kapcsolódó operációkat. Az orkesztráció alapvető eszköze a zárt láncú szabályozókör. Ennek lényege, hogy a futó rendszerről információkat gyűjt és továbbítja azokat a különféle olyan adatfeldolgozó és adatelemző mechanizmusok irányába, melyek az adatok alapján képesek a rendszer működésébe — annak hatékonyabb működtetése érdekében — beavatkozni.

Manapság jelentős népszerűségnek örvend ez a kutatási terület a hálózati szolgáltatást nyújtó nagyvállalatok körében, hiszen az emberek hálózat használati szokásai az utóbbi időben a mobiltelefonok elterjedésével és az általuk nyújtott szolgáltatások körének bővülésével, valamint az IoT egyre nagyobb népszerűségének köszönhetően oly mértékben megváltozott, hogy erre a szolgáltatók is kénytelenek már reagálni. Tovább nehezíti a problémát az is, hogy az emberek a különböző napszakok folyamán jelentősen eltérő mennyiségű forgalmat generálnak, ez által nem homogén az általuk generált adatforgalom. Természetesen a szolgáltatóknak nem éri meg jelentős erőforrás túlskálázással a legnagyobb terhelésre felkészíteniük a rendszert, ugyanis ekkora mértékű kihasználtsága csak ritkán van a hálózatnak, a dedikált hálózati berendezések pedig nagyon drágák. Ebből következően ez a fejlesztés akkora többletköltséget jelentene számukra, amely már messze nem nevezhető megtérülő beruházásnak. Ezt a célt szolgálnák a hálózati virtualizációs technológiák, hiszen segítségükkel megoldhatóvá válna egy adott eszközből,

például routerből annyit példányosítani olcsóbb általános célú hardvereken, amennyire az adott helyzetben szükség van. Ezt nevezik a hálózat orkesztrációjának. Ez a megoldás egyrészt jó a felhasználóknak, hiszen ők nem vesznek észre változást a szolgáltatás minőségében — sőt, a minőség inkább csak javulhat — másrészt előnyös a szolgáltatók számára is, hiszen mindezt anélkül tudják biztosítani, hogy be kellene ruházniuk az idő nagy részében kihasználatlan, ám rendkívül drága céleszközökre.

1.1. Motiváció

A dolgozat témájának ötletét is ezen felhasználói szükségletek inspirtálták, hiszen a problémakör jelentős kutatási potenciált rejt magában. A dolgozat keretén belül megismerkedtem a különböző virtualizációval kapcsolatos témakörökkel, valamint jelentősen elmélyedtem az Open Network Automation Plaftorm-ban (ONAP) és ezen plaftorm fejlődésének mobil piacot érintő jelentőségével. Az ONAP-ról már számos hálózati fejlesztéssel foglalkozó vállalat megállapította, hogy ideális alapját képezheti egy mobil hálózati rendszer elemei teljes életciklusának megvalósításához, így a rendszer megtervezésétől és létrehozásától kezdve, a kabarntartáson és felügyeleten át minden életszakaszt magába foglalva. Az ONAP összes eleme és folyamata ügyféltől függően konfigurálható, ez által biztosítva lehetőséget egy általános modell létrehozására, amely a későbbiekben — a felhasználás függvényében — a megrendelő által egyedileg is testre szabható lehet.

1.2. Dolgozat szerkezete

A dolgozatom a következőképpen épül fel. Az 2. fejezetben bemutatom a különböző felhő alapú technológiákat, amelyek szükségesek a dolgozat során elkészített hálózati architektúra teljeskörű megtervezéséhez, hiszen a számos különböző forrásából származó informatikai megoldás együttes használata teszi a megoldást átfogóvá. Majd ezekre a technológiákra építkezve, a 3. fejezetben mutatom be a hálózat automatizálási rendszereket, ahol ismertetem a legnagyobb jelentőséggel bíró platformot, az ONAP-ot is, amire a későbbiek folyamán bemutatott megvalósítás is épül. A 4. fejezetben ismertetem a dolgozat kitűzött célját, amelyre a 5. fejeztben megvalósítást kínálok. Az elméleti hátteret biztosító technológiák és a kitűzőtt feladat bemutatása után a 5. fejezetben a tényleges felhőalapú mobilitáskezelés szolgáltatás-menedzsment megvalósítása olvasható. A rendszer működését ezután a 6. fejezetben értékélem ki, majd levonom az ezek alapján megállapítható következményeket, melyek akár az ilyen irányban végzett későbbi kutatások számára is relevánsak lehetnek. Ezen későbbi kutatások támogatásának érdekében a 7. fejezetben bemutatom milyen továbbfejlesztési lehetőségek mentén lehet a megoldást javítani egy jobb rendszer elérése érdekében. A dolgozat lezárására a 8. fejezetben kerül sor, mely fejezetben egyben bemutatom a jövőbeni kutatásaim tervezett irányát is.

2. fejezet

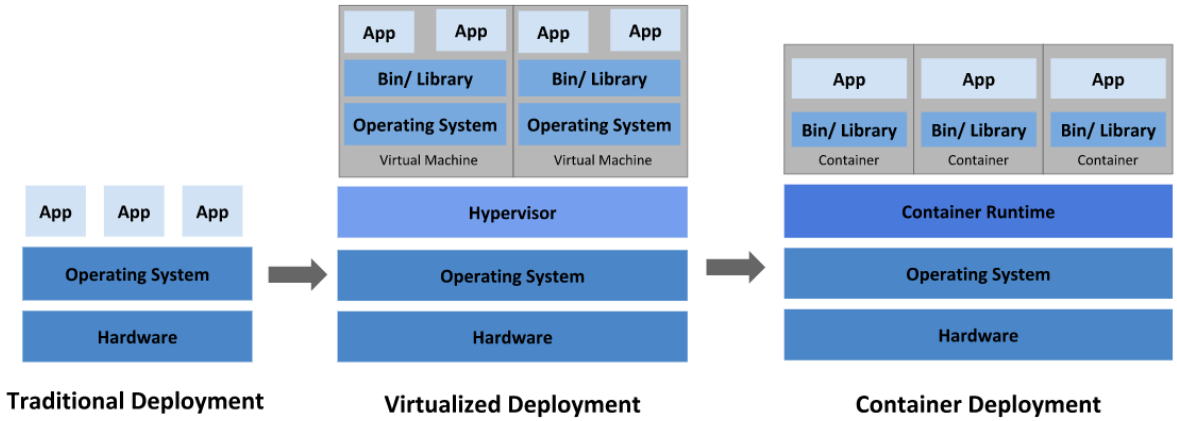
Felhő alapú technológiák

A következő fejezet során bemutatom a különböző felhő alapú technológiákat, melyek relevánsak a dolgozat szempontjából. A felhő alapú paradigmák megértéséhez elengedhetetlen ezen technológiák kifejlődésének bemutatása, hogy mi inspirálta ezeket és miként változtatták meg az ipart. Ezekre a technológiákra alapozva fogom a későbbiek során bemutatni a felhőalapú mobilitáskezelés szolgáltatás-menedzsmentjének lehetséges megvalósítását. A bemutatott technológiák dolgozaton belüli körüljárása kizárólag a technológiák azon bizonyos aspektusából történik, melyekre a dolgozat a későbbiek során épül. Ugyanis ezen technológiáknak számos más olyan alternatív felhasználási lehetőségeik is vannak, illetve további olyan funkcionalitással is rendelkeznek, melyek a dolgozatban nem kerülnek bemutatásra.

2.1. Virtualizációs technológiák fejlődése

A virtualizáció lehetőséget ad arra, hogy több szimulált környezetet hozzunk létre, vagy dedikált erőforrást biztosítsunk egyazon fizikai rendszeren. A virtualizáció olyan hasznos IT szolgálatásokat tesz elérhetővé, amelyek hagyományosan hardverhez kötöttek. A program, amely a fizikai gép hardvereit hozzákapcsolja a virtuális gépekhez a hypervisor. A virtuális gépek egymástól függetlenül futó biztonságos környezetek. Virtualizációval a fizikai gép teljes kapacitása kihasználható úgy, hogy az erőforrások megoszlanak a különböző felhasználók, vagy környezetek között.

A virtualizáció gondolata valamikor az 1960-as években merült fel először az informatika világában, népszerűsége azonban egészen a 2000-es évekig meglehetősen csekély volt [5], [6]. Ekkor jöttek rá a nagyvállalatok arra, hogy az abban az időben népszerű trend, miszerint minden szoftver csak a hozzá készített hardveren képes futni, a jövőben nem lesz fenntartható. Ez ugyanis azt vonta maga után, hogy a megvásárolt berendezések nagy része kihasználatlanul futott. Emiatt, valamint a felhasználók számának exponenciális növekedése miatt a vállalatok óriási, sok esetben felesleges beruházásokra kényszerültek. Emiatt kezdett a virtualizáció egyre szélesebb körben elterjedni. A virtualizáció fejlődése több különböző időszakra osztható. Ezek jellemzőit foglalom össze a következőkben. A megértésben segít a 2.1 ábra.



2.1. ábra. Virtualizáció evolúciója [7]

Tradícionális fejlesztés [8]

Eleinte a szervezetek az alkalmazásokat fizikai szervereken futtatták. Nem volt mód az alkalmazások erőforrás-használatának korlátozására, ami komoly problémákat okozott. Ha egy alkalmazás a rendelkezésre álló erőforrások nagy részét foglalta, a többinek nem maradt elegendő hely a megfelelő működéshez, ez által ezek teljesítménye jelentős mértékben romlott. Megoldást az jelenthetett, hogy minden egyes alkalmazást külön szerveren futtattak. Ez azonban nem bizonyult gazdaságosnak, hiszen a szerverek sokszor kihasználatlanul működtek, viszont az üzemeltetésük és karbantartásuk nagyon költséges volt.

Virtualizált fejlesztés [8]

Az előző korszak problémájának megoldására vezették be a virtualizációt. Ez lehetőséget adott több különböző virtuális gép egyazon fizikai szerveren történő futtatására. A virtuális gépek saját, virtualizált erőforrásokkal rendelkeztek, amelyen saját operációs rendszert futtattak. A virtuális gépek memóriaterülete a fizikai szerveren elkülönült egymástól, így ez a megoldás biztonságot is nyújtott az alkalmazói számára. A virtualizációval jobb erőforrás kihasználtságot lehetett elérni a szervereken, melynek következtében a hardverekre fordított költségek jelentősen csökkentek. A virtualizáció továbbá jobb skálázhatóságot is biztosított, hiszen egy alkalmazás telepítése és frissítése sokkal könnyebbé vált.

Konténerizált fejlesztés [8]

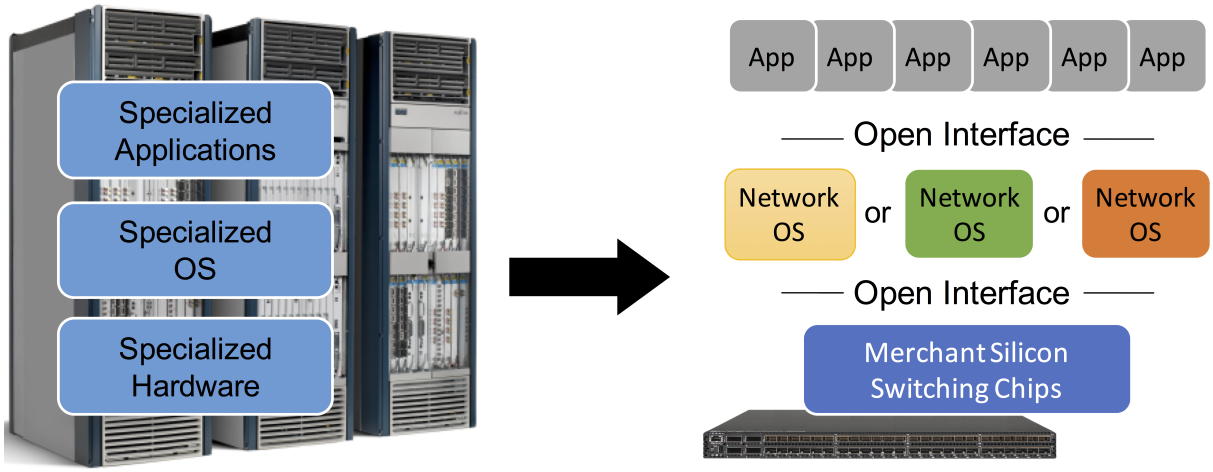
A konténerek hasonlóak a virtuális gépekhez, de ezek csak különféle izolációs beállításokon keresztül osszák meg az alkalmazások között az operációs rendszert. A konténerek emiatt "könnyűsúlyúnak" tekinthetők a virtuális gépekhez képest. A konténerek is rendelkeznek saját fájlrendszerrel, CPU-val és memóriával. Az erőforrások használata ebben az esetben a leghatékonyabb, az alkalmazás a lehető legnagyobb erőforrás kihasználtsággal működik. Mivel a konténerek az adott fizikai gép infrastruktúrájától függetlenül működnek, hordozhatóak különböző rendszerek között.

2.2. SDN és NFV

Ebben a fejezetben az Software Defined Networking (SDN) és az Network Function Virtualization (NFV) technológiákat, architektúrájukat és az általuk nyújtott számtalan és sokszínű lehetőséget mutatom be.

2.2.1. Software Defined Networking (SDN)

Az SDN [9] a hálózati rendszerek implementálásának egy újszerű megközelítése, amely technikai és üzleti szempontból is előnyös megvalósításra nyújt lehetőséget. Az SDN-re való igény a 2000-es évek végén került bele a köztudatba a virtualizáció elterjedésével. Az SDN és a hagyományos architektúra közti különbséget a 2.2 ábra mutatja. Az átalakulás hatásásra a "gyártó függő" hálózati termékeket felváltják a hétköznapi célokra is használható eszközök, ezáltal sor kerülhet a hálózati forgalomirányító (router) és hálózati kapcsoló (switch) eszközök cseréjére.

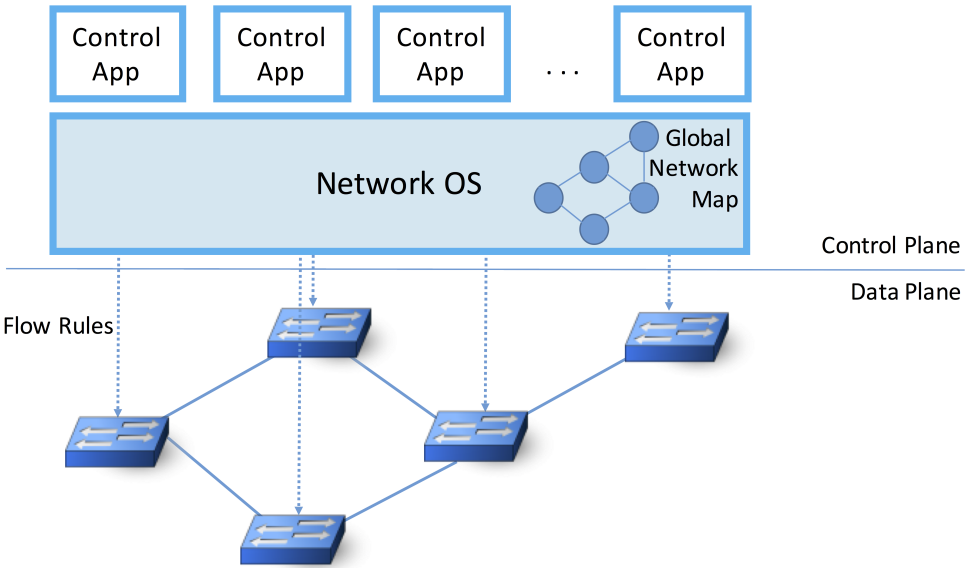


2.2. ábra. Hagyományos és SDN alapú megközelítés[9]

Az SDN koncepció lényege a vezérlési- és adatsík egymástól való elválasztása és a szétválasztott rétegek egy nyílt, jól definiált interfészen keresztül történő kommunikációjának megvalósítása. A vezérlési sík (control plane) meghatározza a hálózatban közlekedő csomagok útját, vagyis egy tetszőleges routing protokollt valósít meg, mint például BGP (Border Gateway Protocol)[10], OSPF (Open Shortest Path First)[11] vagy RIP (Routing Information Protocol)[12]. Az adatsíkot (data plane) a hálózatban elhelyezkedő, egymással összekötött eszközök, a switchek alkotják. A switchek a csomagok továbbításáról döntenek, vagyis arról, hogy az egyes portokon beérkező csomagokat melyik másik porton kell továbbítani, hogy az célba érjen. Alapvetően a control plane a routing tábla karbantartásáért felelős, de feladatköre módosulhat attól függően, hogy a control plane-t más-más alkalmazások esetén milyen egyéb logikával látják el. A switch-ek a forwarding tábla segítségével határozzák meg az adatcsomagok útját. A control plane az adatcsomagok fejlécében utazó mezők lehetséges értékeihez valamilyen szabályrendszert állít fel és ezt leküldi a switch-eknek, amelyek ezt eltárolják a forwarding tábláikban.

Abban az esetben, ha az adatcsomag megfelel egy szabálynak, akkor az adatcsomagot annak megfelelően továbbítja a switch. Az SDN-hez kapcsolódó általános információkat leíró RFC szabvány a 7149-es [13], az SDN felépítésére vonatkozó RFC szabvány a 7426-os [14]. Az SDN kontroller teljesítményére vonatkozó RFC szabvány az 8456-os [15].

Az SDN egyik fontos alapelve, hogy a control plane-nek a data plane-től teljesen függetlenül, centralizáltan kell működnie. Ezzel elérhető, hogy a controller kiszervezhető egy logikailag különálló egységbe, így akár egy felhő szolgáltatásként is futtatható lesz. A logikai különállóság azt jelenti, hogy a controller a külvilág felé egy egységként látszik, de a belső megvalósítása lehet elosztott is, amely azt teszi lehetővé, hogy a kontroller az igények függvényében horizontálisan is skálázhatóvá válik. Egy ilyen rendszert mutat be a 2.3 ábra. A centralizáltság hátránya, hogy a control plane sajnálatos módon még elosztott esetben is single point-of-failure a rendszer egészét tekintve, viszont modern technikák alkalmazásával az ebből adódó veszélyek megfelelő módon kezelhetők és kiküszöbölhetők. Erről szóló tudományos cikkek [16]–[19].



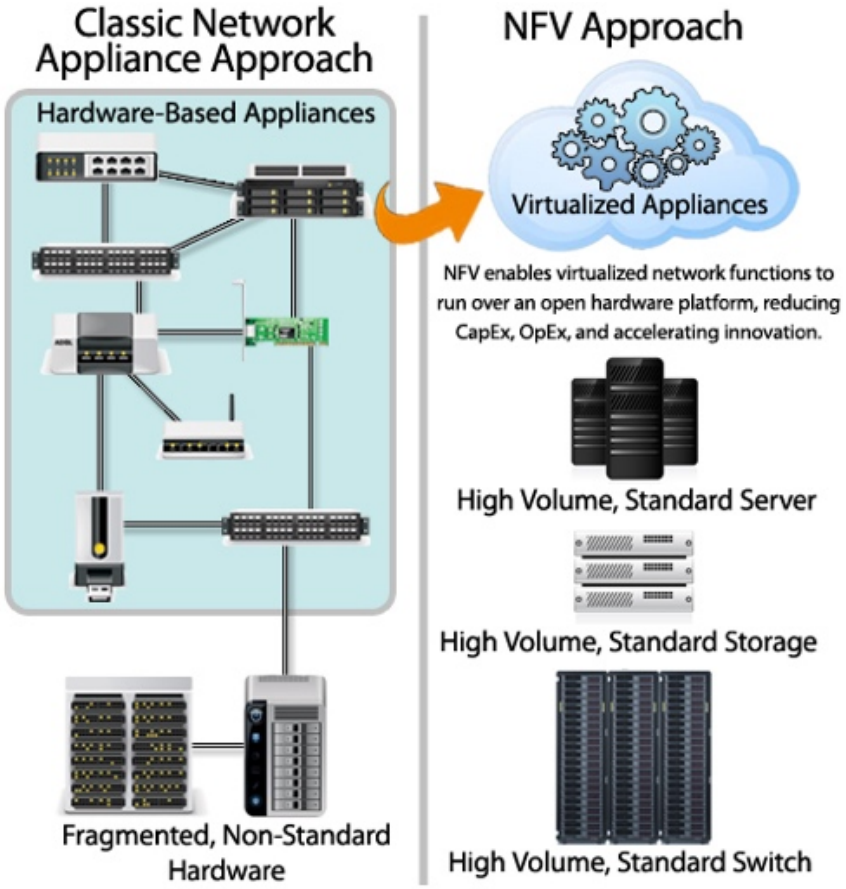
2.3. ábra. Centralizált és elosztott vezérlői sík[9]

Az SDN másik fontos alapelve, hogy a control plane-nek programozhatónak kell lennie. Arról azonban, hogy a data plane-nek programozhatónak vagy fix megvalósításúnak kell lennie, kezdetben nem voltak konkrét elképzelések. Az ajánlást az elmúlt évek tapasztalatai hozták meg. Az SDN előtt használatos ipari switchek esetén nem volt jellemző a hardverek képességeinek korlátoltsága, azonban ezek hétköznapi eszközökre történő lecserélése azt az igényt vonta maga után, hogy a controllernek az eszköz típusától függően kell a Flow Rule-okat meghatároznia, amelyeket a switchek-re telepíthet. A másik érv a programozhatóság mellett, hogy a protocol stack-en történő előre nem látható változtatásokat leghatékonyabban programozható pipeline-ok segítségével lehet lekezelni. A P4 [20] nyelv erre a problémára kínál megoldást úgy, hogy lehetővé teszi a pipeline-ok bemeneti és kimeneti match-action tábláinak programozhatóságát. A tapasztalatok alapján az SDN mára már nem csak a control plane programozhatóságát jelenti, hanem a data plane programozhatóságának is ugyanolyan jelentőséget tulajdonít. A

Data Plane programozhatóságának problémáját tárgyaló cikkek [21]–[23].

2.2.2. Network Function Virtualization (NFV)

Az NFV [24] hálózati eszközök virtualizációjára ad lehetőséget. Az eszközök lehetnek például olyan routerek vagy tűzfalak, amelyek hagyományosan ilyen feladatok ellátásra létrehozott hardveren futottak. A NFV ezeket az eszközöket virtuális gépeken/konténereken valósítja meg, amelyek általános célú hardvereken futnak. Az NFV megteremti a skálázhatóság és agilitás lehetőségét, hiszen a szolgáltató az olcsóbb COTS (Commercial off-the-shelf) hardwereken költséget takaríthat meg, miközben ez a technológia a rendszer jobb skálázhatóságát eredményezi. A kisebb forgalmat kevesebb virtuális gép, a nagyobb forgalmat több virtuális gép segítségével szolgálja ki. Az NFV újítását a hagyományos megoldásokhoz képest a 2.4 ábra mutatja be. Az NFV az új technológiák bevezetését egyszerűbbé teszi, mert a tesztelési folyamat csupán csak egy virtuális gép létrehozását követeli meg, nem kell egy futtatáshoz alkalmas hardvert is legyártani, amelyen a szoftvert tesztelni lehet. A VNF-ek teljesítményének mérése a 8172 RFC [25] szabványban olvasható. A hálózati virtualizáció kihívásairól a 8568 RFC [26] szabvány ír.



2.4. ábra. NFV újítások a hagyományos architektúrához képest[27]

Az NFV [28] szabványosítását — a minél nagyobb interoperabilitás elérése érdekében — az ETSI kezdte meg. Ez azt jelenti, hogy a korábban fennálló kompatibilitási problémák megelőzé-

se érdekében olyan szabványok megtervezésével kezdtek el foglalkozni, amelyek megteremtik a különböző gyártók termékei közti zavartalan együttműködés lehetőségét. Az NFV architektúra a következő elemekből épül fel:

- **Virtual Network Functions (VNFs)** olyan szoftverek, amelyek valamilyen hálózati feladatot látnak el, mint például fájlmegosztás, IP konfiguráció.
- **Network Functions Virtualization infrastructure (NFVi)** azon infrastrukturális komponenseket tartalmazza, amelyek támogatják a hypervisor-t a számításban (computing), tárolásban (storage) vagy hálózati feladatok ellátásában (networking).
- **Management, Automation and Network Orchestration (MANO)** a keretrendszert biztosítja az NFV számára.

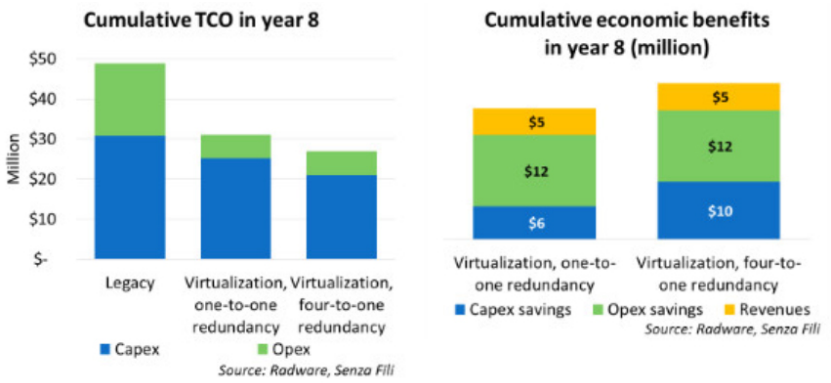
Az ETSI ISG NFV az NFV alapjainak megalkotása után ma már annak részletes kidolgozásával foglalkozik, sorra adják ki publikációikat az NFV témakörében. Ezek közül a legfrissebbek [29]–[31]. A csoport jelenleg az NFV Release 4-en [32] dolgozik. Ebben főként az NFV architektúra elemeinek optimalizációja, a korábbi kiadások óta megjelent új technológiák, mint például az 5G integrációja az NFV-ben, valamint felmerülő biztonsági kérdések és a korábbi Release-ekből kimaradt elemek kapnak helyet.

2.2.3. SDN és NFV kapcsolata

Az SDN [9] és NFV [24] fogalmak a köztudatban gyakran keverednek össze, azonban a két technológia egymástól függetlenül és egymás kiegészítéseként is használható. Mindkét technológia virtualizációra és hálózati absztrakcióra épít, de ezeket eltérő módon alkalmazza. Az SDN elválasztja a vezérlési réteget az adattovábbítási rétegtől és egy olyan hálózat kiépítésre törekszik, amely teljesen centralizált és programozható. Az NFV elválasztja a hálózati feladatokat ellátó szoftvert a hardvertől. Az NFV megteremti az infrastruktúrát az SDN-nek, ezzel is kibővítve a rugalmasságban, skálázhatóságában és erőforrás felhasználásban rejlő lehetőségeit.

Az SDN és az NFV technológiák bevezetésével a szolgáltatók hatalmas költségeket takaríthatnak meg a működésük során. A Senza Fili [33] azt vizsgálta, hogy az ADC (Application Delivery Controller) virtualizálása mekkora költséget takaríthat meg a szolgáltatók számára. A vizsgálat alapján a megtakarítás akár a 45%-ot is elérheti. A felmérés eredménye a 2.5 ábrán látható. A költségeket a felmérésben két csoportra osztották. A CapEx a beruházási költségeket, az OpEx a működési költségeket azonosítja. A hagyományos eszközpark használatát kétszintű virtualizáció bevezetésével hasonlították össze. Az SDN és NFV bevezetésével a beruházási költségek akár 10 millió dollárral is csökkenhetnek, aminek oka, hogy a virtualizációra használt eszközök olcsóbbak, mint a hálózati feladatok ellátásra dedikált eszközök. A két vizsgált eset beruházási költségében megnyilvánuló különbséget a vásárlandó eszközök száma okozza. Az első esetben négyszer több erőforrásra van szükség, ami látszik abból is, hogy ekkor kevesebb a beruházási megtakarítás. Az a körülmény, hogy a megtakarítás nem négyszeres a második

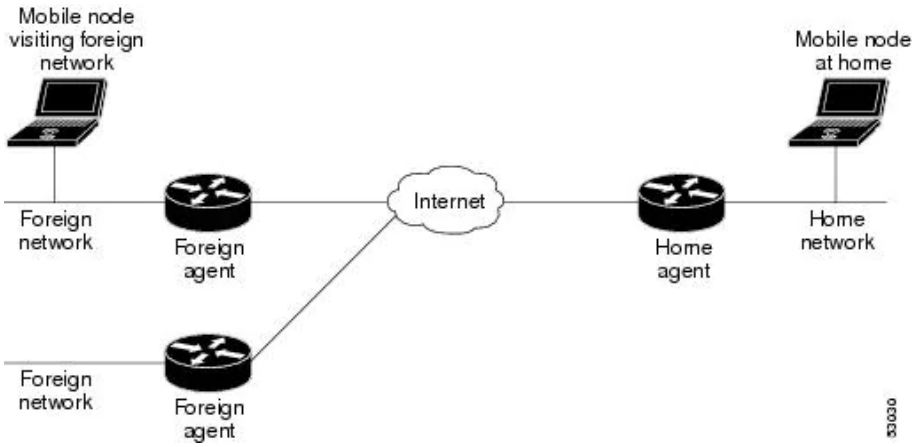
esetben az elsőhöz képest azért lehetséges, mert a vállalatnak van egy alap költsége, amely független a virtualizáció bevezetésével járó anyagi megtakarításoktól. A működési költségek a vállalat mindennapos költségeit jelentik. Ez magában foglalja az üzemeltetés, valamint a emberi foglalkoztatás költségeit. A megtakarítás a működési költségek esetében 12 millió dollár is lehet. A felmérés alapján belátható, hogy a virtualizációs technológiák alkalmazása sok más előnyük mellett, a vállalatok költségeinek jelentős csökkentéséhez is hozzájárul.



2.5. ábra. Total Cost of Ownership SDN és NFV használata esetén[33]

2.3. IP szintű mobilitáskezelés

A felhasználók általában mozognak a mobilkészülékeikkel és mozgásuk során cellákat vagy bázisállomást váltanak. Ilyenkor a rendszer felé való elvárás, hogy a készülékek két végén lévő felhasználók zavartalanul tudják használni a nekik nyújtott szolgáltatásokat. Ezért a hivásátadásokat (handover) úgy kell a hálózaton belül lebonyolítani, hogy azt a felhasználó ne érzékelje, zavartalanul tudja folytatni a már meglévő kommunikációt. A 2.6 ábrán látható egy általános Mobile IP (MoIP) hálózat architektúrája, ahol az egyes komponensek a következő feladatot töltik be a rendszerben.



2.6. ábra. MoIP komponensek és kapcsolataik [34]

Router	Forgalomirányításért felelős eszköz.
Node	Azok az eszközök, amelyek IPv6 protolkollt futtatnak és csatlakoznak a hálózathoz.
Mobile Node	Az az eszköz, ami képes csatlakozni az alhálózatokon keresztül a hálózatra és közben mindig elérhető az otthoni IP címén keresztül.
Home Address	Otthoni alhálózatban használt statikus IP cím.
Care-of-Address	Idegen alhálózathoz csatlakozva kapott ideiglenes IP cím.
Binding	Kötés az otthoni cím és az idegen cím között.
Binding Update	Fejléc kiterjesztése, a küldő mobil aktuális kötését és érvényességi idejét tartalmazza.
Binding Acknowledgement	Fejléc kiterjesztése. A kommunikációs partner válasza a Binding Update üzenetre.
Binding Cache	A bindingek tárolására szolgál a megadott ideig.
Binding List	A mobil készülék ebben tárolja, hogy kiknek küldött Binding Updatet.
Flow Bindings	Lehetővé teszi, hogy egynél több címet bindeljen Care-of-Addresshez.
Home Agent	Otthoni alhálózat egy routere, ami az egyes mobil eszközök idegen IP címét beregisztrálja, valamint a neki címzett csomagokat erre az IP címre továbbítja.

A rendszernek kettős feladata van, felelős a felhasználó helyzetének nyomonkövetésért, valamint az egyes cellák és frekvencia sávok közötti váltások esetén a hívásátadásért. E kettő szorosan egybekapcsolódik, elválaszthatatlan fogalmak, hiszen együttes létükkel oldható meg a felhasználói mobilitás kezelése. Location Management-nek hívjuk a helyzet nyílvántartást és a Handover Management-nek a hívásátadás kezelését.

A Location Management keretén belül a mobil csomópont helyzetének változását rögzítjük, illetve frissitjük helyzetváltozás esetén. Nagyon fontos ismerni az eszköz helyzetét a hálózati topológián belül, főleg ha egy adott mobil csomópont számára szeretnénk üzenetet továbbítani. Ebben az esetben problémát jelenthet, hogy ha nem teljesen pontos a helymeghatározás, vagy nem frissült időben. Ekkor szokták a Paging-el megkeresni a mobil csomópont elhelyezkedését a topológiában.

A Handover Managment esetében a felhasználói hívásátadást értjük. Két fő típusát különböztetjük meg. Egyik, amikor a felhasználói csomópont nem hagyja el az adott cella által

lefedett területet, csakis a rádiós frekvencia sávokon belül történik váltás (ez történhet az interferencia csökkentése érdekében, vagy pedig a mozgás sebessége miatt is), akkor cellán belüli hívásátadásról beszélünk. Amikor viszont a mobil csomópont elhagyja a cella által lefedett területet és egy másik cellába lép át, cellák közötti hívásátadásról beszélünk. Ez utóbbi eset lekezelése nehezebb feladat, mivel a frekvencia sávok közötti átadás a link rétegen belül elvégezhető, de itt magasabb szintű hálózati rétegekben kell a jelzésátvitelt lefolytatni. A különböző fejlécek felépülésével minnél magasabbra megyünk a hálózati szintekben, annál nagyobb késleltetés kerül a csomagra.

Felhőalapú mobilitáskezelés

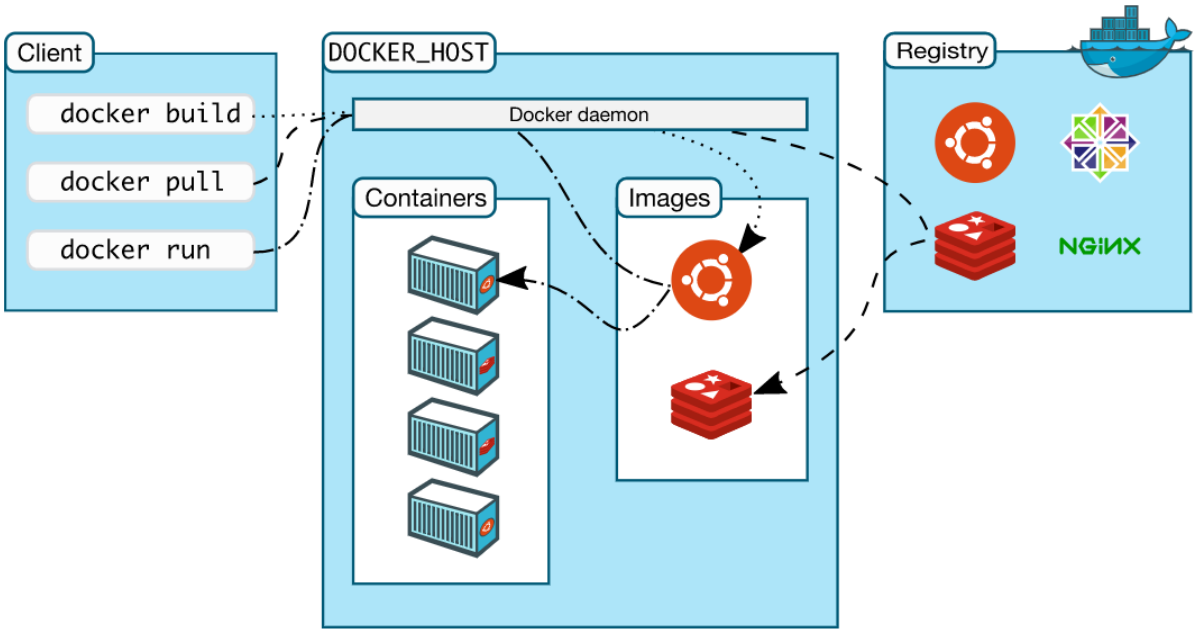
A fizikai közeget (antenna) leszámítva a mobil hálózat minden komponense átültethető felhőalapú technológiákra, amelynek célja a cloud-native megközelítés előnyeinek kihasználása. Felhő alapú környezet esetén a mobilitáskezeléshez tartozó üzenetek ugyan azon metódus szerint épülnek fel, mint a hagyományos megközelítés esetében. Jelentős előnye a cloud-native megközelítésnek, hogy az egyes hálózati építőelemek dinamikusan skálázható alkalmazásokként operálnak. Ebből következően a felhasználói szükségletek függvényében csökkenthető, vagy pedig növelhető az egyes szolgáltatáshoz tartozó példányok száma. Ezen megoldások még nem tökéletesek, viszont rengeteg potenciált rejtenek magukban a hatékony és rugalmas hálózatüzemeltetési nézőpontok miatt. Ezért a lehető legjobb megoldás megtalálása érdekében számos kutatás zajlik a témában.

2.4. Docker

A Docker [35] egy nyílt platform konténerizált alkalmazások fejlesztésére és futtatására. A konténerizálásból kifolyólag a Docker használatával az alkalmazások elkülöníthetők az infrastruktúrától, aminek köszönhetően egyre elterjedtebb ez a platform. A Docker lehetőséget nyújt arra, hogy az alkalmazásokat becsomagoljuk egy lazán elszigetelt környezetbe és azon belül futtasuk őket. Az izolált környezeteket konténereknek nevezzük. Az izoláció és az abból eredő biztonság lehetővé teszi, hogy több konténert futtassunk egyszerre ugyanazon hoszton. Mivel konténerek futtatásához nincs szükség külön hypervisorra, mint a virtuális gépek esetén, hanem a hoszt gép kernelén futnak, ez az architektúra könnyűsúlyúnak mondható a hagyományos virtualizációhoz képest. Következményként ugyanazon hardveren általában több konténer futtatható, mint ahány virtuális gép. A Docker által biztosított eszközök segítségével a konténerek életciklusát is kezelhetjük. A Docker használatával készült alkalmazás könnyen integrálható meglévő rendszerekbe, mint konténer vagy orkesztrált szolgáltatás. Az alkalmazás független az ipari környezettől, legyen az helyi adatközpont, felhőszolgáltató vagy a kettő keveréke (hybrid). A konténerek különösen jól használhatók a folyamatos integráció és rendszeres alkalmazás verziók kiadásának támogatására (Continuous Integration/Continuous Development (CI/CD)).

A Docker Engine egy kliens-szerver alkalmazás, amelynek a komponenseit a 2.7 ábra szem-

lélteti. Látható, hogy a kliens a Docker REST API-t használja arra, hogy a Docker Daemon-t — szkripteken vagy közvetlen CLI parancsokon keresztül — irányítsa, vagy kapcsolatba lépjen vele. A daemon hozza létre és menedzseli a Docker objektumokat, mint például a képfájlokat, konténereket, hálózatokat és tárolókat. A kliens és daemon futhatnak egyazon rendszeren, de a kliens akár távolról is rácsatlakozhat a daemonra.



2.7. ábra. Docker architektúra [35]

A Docker registry tárolja a képfájlokat. A Docker Hub egy publikus registry, amelyet bárki használhat. Alapértelmezetten a Docker a Docker Hub-on keresi a képfájlokat, azonban úgy is beállítható, hogy privát registry-t használjon. A Docker Datacenter tartalmazza a Docker Trusted Registry-t, ahol privát képfájlokat tárolhatunk.

Az image fájlok olyan, csak olvasható sablonok, amelyek tartalmazzák a konténerek létrehozásához szükséges instrukciókat. A képfájlok gyakran más képfájlok alapján készülnek el, azokat extra funkciókkal egészítik ki. Az image fájlokat a Dockerfile-ok határozzák meg, a daemon az ebben leírtak alapján generálja az image-t.

A konténerek a képfájlok futtatható példányai. A Docker API-n vagy a konzolos felületen keresztül lehet őket létrehozni, indítani, leállítani, áthelyezni vagy törölni. A konténerek akár több hálózathaz is csatlakozhatnak egyszerre, valamint tárhely is csatolható hozzájuk. A konténerek izolációja a környezetüktől a fejlesztők által szabályozható. A konténerek törlése után azon adatok, amelyek nem voltak benne a perzisztens tárolóban elvesznek, később már nem állíthatók vissza.

A Docker hordozhatóságából és "könnyűsúlyúságából" eredően lehetővé teszi az elvégzendő munka dinamikus kezelését. Ez azt jelenti, hogy mindig az adott pillanatban az alkalmazással szemben támasztott igényeknek megfelelően, több vagy kevesebb példányban futtatva az alkalmazásokat, az igény minden esetben kiszolgálható úgy, hogy nincsenek kihasználatlan erőforrások a rendszerben. A Docker használata rendkívül előnyös olyan esetekben, amikor ke-

vesebb erőforrás áll rendelkezésre, hiszen a konténerekből sokkal több futtatható, mint virtuális gépekből ugyanazon hardveren. Ebből kifolyólag a gépek számítási kapcitása sokkal jobban kihasználható.

2.5. Kubernetes

A Kubernetes [8] egy nyílt forráskódú konténer-orkesztrációt megvalósító rendszer, amely képes alkalmazások automatizált fejlesztésére, skálázására és menedzsmentjére. A rendszer megvalósításának ötlete a Google-től származik, azonban a karbantartásáról ma már a Cloud Native Computing Foundation gondoskodik. A Kubernetes hatalmas, gyorsan bővülő ökoszisztémával rendelkezik, melynek szolgáltatásai, támogatása és eszközei széles körben elérhetőek. A Kubernetes név a görög kapitány, pilóta szóból ered.

Az utóbbi években a konténer alapú alkalmazások egyre népszerűbbek a szolgáltatók körében. Ipari környezetben az alkalmazásokat futtató kontéreknek úgy kell működniük, hogy ne legyen szolgáltatáskiesés. Például, ha egy konténer tönkremegy, azonnal egy másikat kell indítani helyette. A Kubernetes egy olyan keretrendszert biztosít, amely ezt lehetővé teszi. A Kubernetes számos szolgáltatást biztosít, mint például terheléselosztás, tárhely-orkesztráció, "öngyógyító" képesség, biztonsági és konfigurációs menedzsment.

2.5.1. Kubernetes architektúra

A Kubernetesben való fejlesztés során egy úgynevezett klasztert (cluster) kapunk. A Kubernetes cluster worker gépekből áll, amelyeket node-nak hívunk. Ezek futtatják a konténerizált alkalmazásokat. Minden cluster-nek van legalább egy node-ja. A worker node-okon belül találhatóak a pod-ok, ezeken belül futnak a konténerek. A vezérlő sík menedzseli a node-okat és a pod-okat egy cluster-en belül. Ipari környezetben a control plane általában több gépen fut és egy cluster általában több node-ot futtat a rendszer hibatűrő képességének fenntartása és a magas rendelkezésre állás biztosítása céljából.

A Kubernetes számos komponensből épül fel, ezek közül a legfontosabbak:

Pod A pod [36] egy konténerizált alkalmazás alapvető végrehajtó egysége. A Kubernetesben létrehozható objektumok között a legegyszerűbb és legkisebb egység. A pod tartalmazza a konténer(eke)t, rendelkezik saját tárhellyel és IP címmel, valamint tartalmaz olyan opciókat, amelyek meghatározzák egy konténer futtatásának módját. A pod egy fejlesztési egységet képvisel, egy példánya egy alkalmazásnak a Kubernetesben. A pod általában egy konténert tartalmaz, de ha a megvalósítandó feladat úgy igényli, akár több szoros összefüggésben lévő konténert is tartalmazhat. Ebben az esetben a konténerek osztoznak a pod erőforrásain. A Kubernetes leggyakrabban használt konténer típusa a Docker konténer. Minden pod egy adott alkalmazás egy példányát futtatja. Ahhoz, hogy az alkalmazást horizontálisan skálázhassuk, vagyis több példányban futtassuk, több pod-ra van

szükség. Ez az úgynevezett replikáció. A replikációval létrehozott pod-okat a Controller hozza létre és menedzseli.

Deployment A Deployment-ek [37] alapján a podok, illetve ReplicaSet-ek könnyedén létrehozhatók vagy frissíthetők. A Deployment-ben leírható a kívánt állapot, amely alapján a Deployment Controller megváltoztathatja a jelenlegi állapotot annak céljából, hogy elérje a kívánt állapotot. A Deployment-ek alapján létrehozhatók új ReplicaSet-ek, valamint a már meglévő Deployment-ek erőforrásai hozzárendelhetők az újonnan létrehozott Deployment-ekhez. A Deployment-nek fontos tulajdonsága a skálázás. Skálázás segítségével nagyobb terhelés kiszolgálására van lehetőség.

Service A service-k [38] segítségével az alkalmazást — amely akár több pod-on is fut egyszerre — hálózati szolgáltatásként futtathatjuk. Minden egyes futó pod saját IP címmel rendelkezik, azonban közös DNS címet használnak, így a Kubernetes a terhelést egyenlően tudja megosztani köztük. A service-ek azon túl, hogy megkönnyítik az alkalmazások nyilvános hálózaton keresztüli elérését, segítséget nyújtanak a pod-ok életciklusából adódó számos probléma kezelésében is. A gondot az okozza, hogy a pod-ok halandóak és a haláluk után nem éleszthetőek fel újra. Ez azért lehet baj, mert a deployment-en belül futó podok menet közben cserélődhetnek (ha egy meghal a helyére új pod-ot kell állítani) és az aktuálisan futó pod-ok száma is változhat az igényektől függően, ami pedig negatívan befolyásolhatja az alkalmazást, ha a probléma nincs megfelelően lekezelve. Ahhoz, hogy az alkalmazás futásában ne látszódjon meg ezen változásoknak a hatása, a pod-okat folyamatosan értesíteni kell az őket érintő változásokról. Ebben segítenek a service-ek, akik folyamatosan monitorozzák a futó rendszert és intézkednek, ha erre szükség van. A service összességében egy olyan absztrakció, amely podok egy csoportját és az ő elérésüket lehetővé tevő irányelvet (policy-t) kapcsolja össze.

Node A node [39] egy worker gép a Kubernetesben, korábban minion-nak hívták. Egy node lehet virtuális gép, vagy egy fizikai gép. Minden node tartalmazza a szükséges service-eket pod-ok futtatásához. Ilyen service-ek például a kubelet, kube-proxy és a container runtime.

Controller A Kubernetesben a controller-ek [40], olyan vezérlő egységek, amelyek figyelik a hozzájuk tartozó cluster állapotát és módosításokat kérvényeznek, vagy hajtanak végre annak érdekében, hogy a cluster elérje az elvárt állapotát.

2.5.2. Kubernetes closed-loop támogatás

A Closed Loop valamilyen szintű támogatására a legtöbb modern rendszer, így a Kubernetes is lehetőséget ad [41]–[44]. A Kubernetes-ben lehetőség van a pod-ok és az azokon belül futó konténerek folyamatos monitorozására. A konténerek egyes életeseményeikről jelzést adnak. Ezeket a jelzéseket lifecycle hook-nak nevezték el. A hook-okra reagálva a fejlesztők kódot

írhatnak, amely akkor fut le, amikor a hook végrehajtódik. A konténerek két típusú hook-ot tudnak küldeni. A PostStart hook közvetlenül a konténer létrejötte után hajtódik végre, de ez nem garantálja azt, hogy az itt megírt kód még a Dockerfile-ban meghatározott ENTRYPOINT, vagyis a konténer belépési pontja előtt fut le. A PreStop hook közvetlenül a konténer leállása előtt hajtódik végre. A leállást okozhatja egy API kérés vagy más menedzsment funkció által kiváltott esemény is kérvényezheti a konténer leállítását. A PreStop hook-ot abban az esetben, ha a konténer már leállt (terminated) vagy elkészült (completed) állapotban van nem lehet meghívni. A PreStop-ban megírt kód blokkoló (szinkron) módon fut le, ami azt jelenti, hogy a konténer leállítását jelző üzenet nem küldhető el addig, amíg a PreStop hook nem fejeződik be. Paraméter sem a PostStart sem a PreStop eseménykezelőnek nem adható át. Hook-ok implementációjára két féle lehetőség van. Lifecycle hook egy pod-hoz a pod-ot leíró yaml fájlban definiálható.

- **Exec** Ebben az esetben egy meghatározott parancs vagy egy szkriptpen leírt parancsok sorozata hajtódik végre a konténer névterén belül.

- **HTTP** Ebben az esetben a konténer egy meghatározott endpoint-ját célzó HTTP kérés fut le.

A hook-ok kezeléséről összességében a következő mondható el. Amikor egy hook meghívásra kerül, akkor az eseménykezelő annak típusától függően végrehajtódik a konténeren belül. A PostStart a konténer belépési pontjától (ENTRYPOINT) függetlenül aszinkron módon fut le, azonban abban az esetben, ha a PostStart eseménykezelő túl hosszan fut vagy valamiért megakad, a konténer soha nem érheti el a running (futó) állapotot. PreStop esetben ahhoz, hogy a konténer leállítása megkezdődhessen, az eseménykezelőnek teljesen le kell futnia. Ebből következően, ha ez valamiért megreked, akkor a konténer terminating (leállítás alatt) fázisban marad egészen addig, amíg a leállásra szánt türelmi idő (terminationGracePeriodSeconds) le nem jár. Éppen ezért ezt az időintervallumot úgy kell meghatározni, hogy alatta a PreStop hook és a konténer leállítása is befejeződhessen. Ha a PostStart vagy a PreStop eseménykezelő végrehajtása alatt valamilyen hiba lép fel, akkor az azonnal megöli a konténert.

A Kubernetes egy másik diagnosztikára alkalmas szolgáltatása az úgynevezett probe-ok használata. A kubelet a probe-okat a konténereken periodikusan hívja meg, segítségükkel megállapítható, hogy a konténer egészségesen működik-e. A Kubernetes-ben három féle probe létezik:

- **Liveness Probe** Ez arról ad visszajelzést, hogy a konténer fut-e. Ha sikertelen a probe végrehajtása a konténer törlésre kerül és vele kapcsolatban a rendszer az újraindítási irányelvekben (restart policy) meghatározott módon jár el. Az alapértelmezett állapot, ha a konténerhez nem volt specifikálva liveliness probe a Success (sikeres). Segítségével detektálhatók a holtpontok, amikor az alkalmazás fut, de nem csinál semmit sem.

- **Readiness Probe** Ez azt jelzi, hogy a konténer készen áll-e kérések kiszolgálására. Ha a probe sikertelen, a kontroller eltávolítja a pod IP címét az összes olyan szolgáltatásból

(service), amelynél az adott pod fel van jegyezve. A probe kezdeti állapota Failure viszont, ha nincs definiálva az alapértelmezett állapota a Success. A pod akkor áll készen, ha az összes benne futó konténer készen áll.

- **Startup Probe** Segítségével megállapítható, hogy a konténeren belül futó alkalmazás elindult-e már. Ha egy konténer startup probe-ot definiál, az összes többi probe le van tiltva mindaddig, amíg ez nem sikeres. Ha sikertelen a probe végrehajtása a konténer törlésre kerül és vele kapcsolatban a rendszer a restart policy-nak megfelelően jár el. A probe alapértelmezett állapota a Success. Használatával megelőzhető olyan pod-ok idő előtti megölése, például a liveliness probe által, amelyek indítása hosszú folyamat.

A probe-okat a hook-oknál látottakhoz hasonlóan három féle képpen lehet kivitelezni. A probe-ok eredménye szintén három féle lehet: Success, Failure és Unknown. A probe-okat a pod-ot leíró yaml fájlban lehet specifikálni.

- **ExecAction** Ez egy parancsot hajt végre a konténerben. A probe abban az esetben sikeres, ha a visszatérési érték 0.

- **TCPSocketAction** Ez egy TCP ellenőrzést hajt végre a pod IP címének egy előre meghatározott portján. A probe akkor sikeres, ha a port nyitva van.

- **HTTPGetAction** Ez egy HTTP GET kérést küld a pod IP címének egy előre meghatározott porton és endpoint-on. A probe abban az esetben sikeres, ha a válasz kódja 200 vagy annál nagyobb és 400-nál kisebb.

A Kubernetes-ben a Metrics API[45] segítségével lehetőség van a node-okról és pod-okról különféle metrikákat gyűjteni. A mért adatok elérhetők közvetlen a felhasználó által a kubectl top parancs segítségével vagy egy kontroller által is, például Horizontal Pod Autoscaler, hogy döntsön a megszerzett információk alapján. A Metrics API használatával olyan információkhoz juthatunk, mint CPU vagy memória használat. A Metrics Server a Metrics API továbbfejlesztése, aminek bevezetésével az a cél, hogy a monitorozó komponens egy deployment-ként legyen telepíthető a Kubernetes klaszterbe. A Metrics Server[46] segítségével az összes klaszterben futó node-ról információ gyűjthető a Summary API-n keresztül. A Metrics Server bevezetését a Kubernetes 1.7-es verziójára tervezik. (Jelenleg a Kubernetes 1.21 a legfrissebb verzió.)

A Horizontal Pod Autoscaler automatikusan képes skálázni a deployment-en belül futó pod-ok számát a Metrics API-tól gyűjtött információk alapján. Annak érdekében, hogy megközelítse a felhasználó által meghatározott erőforrás felhasználást, a Horizontal Pod Autoscaler periodikusan állítja a deployment replikáinak számát.

2.6. Helm

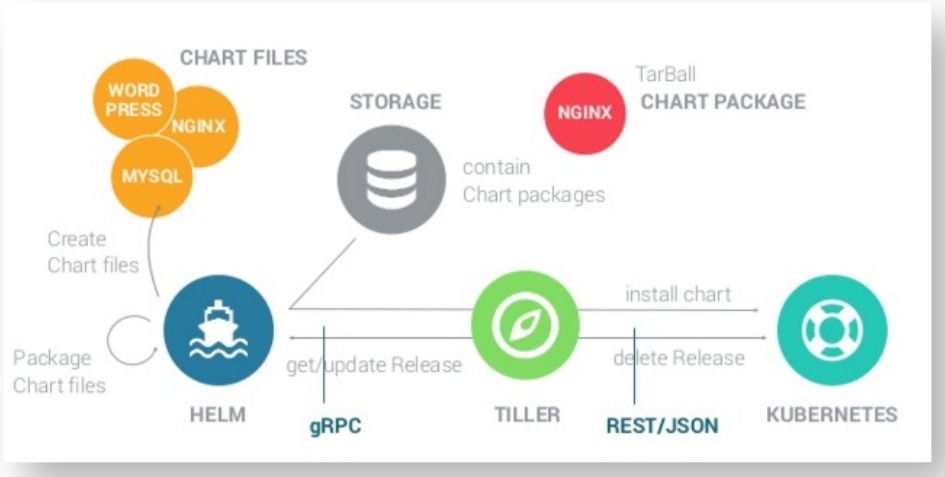
A Helm a Kubernetes csomagkezelőjének tekinthető önálló alkalamzás. Használata során Helm Chart-okat készíthetünk, amelyeket a Kubernetes erőforrások klaszterekbe való kitelepí-

tése során használhatunk fel. Ezen technológia alkalmazásakor egyetlen helmfile-ba lehet azon változókat és paramétereket kiszervezni, amelyek több különböző felhasználási hellyel rendelkeznek. Ennek köszönhetően egy esetleges változtatás során, csakis a konfigurációs helmfileban szükséges a módosítást elvégezni, amely által a rendszer koherens működése garantáltá válik.

A Helm javítja a fejlesztők termelékenységét, hiszen a tesztelésre fordítandó időt más, hasznosabb tevékenységekre lehet allokálni. Ugyanis a Helm Chart-oknak köszönhetően a tesztkörnyezetekbe történő kitelepítés automatizálható lesz. Ehhez hasonlóan, a Helm Chart-ok felhasználásával adatbázisokat telepíthetünk anélkül, hogy a fejlesztői csapatnak erre plusz időt kellene áldoznia. A Helm architektúra a 2.8 ábrán látható, amelynek két fő komponense létezik.

- **Helm Client** Lehetőséget nyújt a fejlesztők számára, hogy Helm Chart-okat hozzanak létre és a Tiller Server-rel interkacióba tudjanak lépni.

- **Tiller Server** Kubernetes klaszteren belül fut. A Helm Client-tel lép kapcsolatba abból a célból, hogy a Helm Chart-ok alapján elvégezze a szükséges konfigurációkat a Kubernetes API-n keresztül. A Tiller Server továbbá felelős a Helm Chart-ok frissítéséért és az adott Kubernetes klaszterből való törlésükért.



2.8. ábra. Kubernetes Helm architektúra [47]

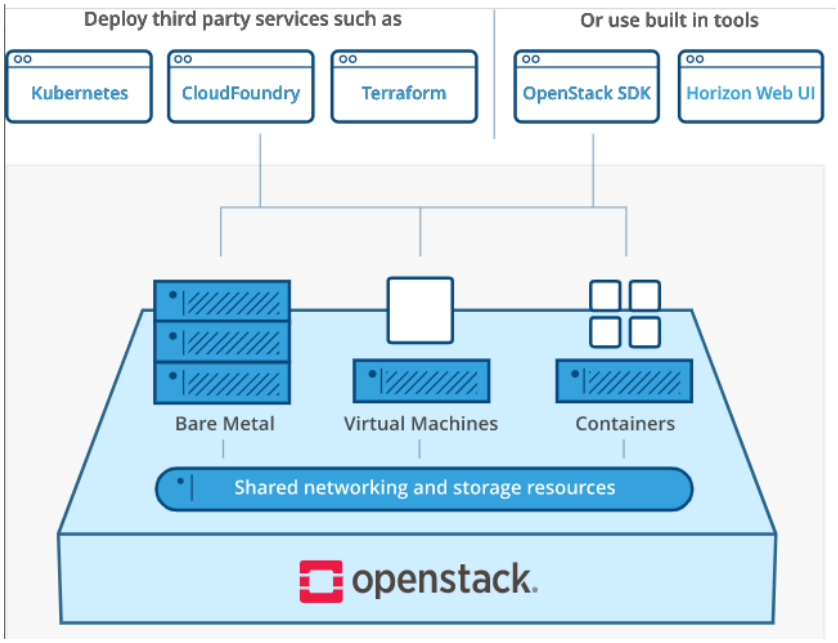
2.7. Infrastructure as a Service

Az Infrastructure as a Service (IaaS) alatt olyan felhőalapú számítási szolgáltatásokat értünk, ahol a számlázás használatba vétel alapon történik, ezért kizárólag akkor kell az erőforrásért fizetni, amikor azt igénybe vesszük. IaaS lehet egyaránt számítási, tárolásra alkalmas,

vagy hálózati erőforrás is. A megközelítés célja, hogy a fizikai hardverhez kapcsolódó karbantartási, üzemeltetési és járulékos hardverköltségek csökkenjenek. Új alkalmazások kitelepítéséhez nem szükséges a vállalatoknak a szükséget hardverkörülményeket megteremteni, ezért gyorsan lehet a megbízható és könnyen skálázható rendszerek igénybevételével új értéket teremteni. Egy ilyen kitelepített alkalmazás a felhasználói igények függvényében tudja skálázni az erőforrásait, hogy a lehető legjobban tudjon alkalmazkodni az előre nem tervezhető, hirtelen megnövekedett/lecsökkent keresletre.

OpenStack

Az OpenStack egy olyan felhő alapú rendszer, amely képes szabályozni hatalmas mennyiségű tárhely, valamint számítási és hálózati feladatokat ellátó erőforrásokat egy adatközponton belül. Kezelése és karbantartása különféle API-kon keresztül lehetséges. Az OpenStack további komponensei képesek orkesztrációra, hibakezelésre, vagy szolgálatás kezelésre, miközben biztosítják az alkalmazások magas szintű rendelkezésre állását. Az OpenStack elhelyezkedését a felhőalapú architektúrában a 2.9 ábra mutatja.



2.9. ábra. OpenStack szerepe ar architektúrában [48]

3. fejezet

Hálózat automatizálási rendszerek

Ebben a fejezetben bemutatom a hálózat automatizálási rendszerek témakörében elsajátított ismereteimet. Kifejtem ezek fejlődését, illetve bevezetem a technológia megértéséhez szükséges releváns fogalmakat. Itt kerül bemutatásra az ONAP Platform és a zárt láncú szabályozókör fogalma, amelyek meghatározó szerepet töltenek be a dolgozat szempontjából, hiszen a megoldás az ONAP platformra épít a technológiai megközelítés és kivitelezés értelmében. Ezen automatizálási rendszerek segítségével a fenntartás teljes életciklusát meg lehet valósítani, kezdve az alkalmazás kitelepítésétől a karbantartásig az üzemeltetésén át.

3.1. Hálózat automatizálási platformok fejlődése

2015-ben lezárult az NFV szabványosításának első szakasza és ezzel egy időben az NFV MANO első implementációi is napvilágot láttak. A piac egyre nagyobb érdeklődése miatt az ETSI az NFV MANO kidolgozásába rengeteg időt és energiát fektetett, aminek hamarosan meg is lett a haszna. Egyre nagyobb vállalkozások kezdték el saját NFV MANO-juk megvalósítását. Az open-source termékek közül a legjelentősebbek az OpenMANO, az Open Source MANO, az Open-O és az OpenECOMP voltak. 2017-ben a Linux Foundation által kiadott Open-O és az AT&T által kiadott OpenECOMP egybeolvadásából jött létre az ONAP. Az ONAP-ról részletesen egy későbbi fejeztben lesz szó.

3.2. Zárt láncú szabályozókör fogalma

Ebben a fejezetben az óriási hálózatok üzemeltetésének automatizálására való igényéből kialakult zárt láncú szabályozásról lesz szó. A zárt lánc, angol nevén closed loop, megvalósításra több eszköz is rendelkezésre áll, amelyek mind másképp segítik a szolgáltatások életciklusának menedzselését. Ezek ismertetésére is ebben a fejezetben kerül sor.

Az egyre nagyobb körben elterjedő IoT és az 5G az utóbbi években a hálózati eszközök számának robbanás szerű növekedését okozta. A szolgáltatók felfedezték, hogy hálózataik komplexitása már csak a virtualizáció bevezetésével kezelhető. Ez az átállás azonban újabb kapukat

nyitott meg számukra. Szoftverek segítségével ugyanis olyan szolgáltatásokkal tudják bővíteni hálózataikat, mint például a szolgáltatás biztosítás (service assurance), orkesztráció, analitika vagy adat központú feldolgozás. Ezen szolgáltatások bevezetésével és automatizálásával a hálózatok komplexitása csökkenthető, a hálózat jobban karban tartható és végeredményként egy megbízható hálózat alakítható ki.

A closed loop kifejezés magában foglalja a hálózatok automatizációs és menedzsment képességeit. A hálózati eszközöktől begyűjtött adatok alapján a closed loop meg tudja figyelni a hálózatot és képes a fellépő hibák és torlódások detektálására. Ennek következtében képes a detektált anomáliák emberi beavatkozás nélküli javítására. A loop, magyarul hurok, a closed loop egyes elemei közti állandó kommunikációra és visszacsatolásra utal. A closed loop lehetőséget ad olyan hálózatok kialakítására, amelyek képesek teljesen automatizáltan mindenféle külső beavatkozás nélküli működésre.

A closed loop segítségével minimálisra csökkenthetők az emberi beavatkozást igénylő, időigényes folyamatok. Alkalmazása a valós idejű automatizációs képességének köszönhetően gyorsabb fejlesztést, nagyobb biztonságot és magasabb szintű agilitást eredményez. A closed loop amellett, hogy jobb szolgáltatás minőséghez vezet, sok terhet vesz le a szolgáltatók válláról, akik a felszabadult kapacitásaikat új területek felfedezésére, új szolgáltatások biztosítására tudják fordítani, így új bevételi forrásokra tehetnek szert.

A closed loop rendszerek több fázisra oszthatók. A fázisokat a 3.1 ábra szemlélteti. A closed loop az adatgyűjtéssel kezdődik. A gyűjtött adatok ezután különböző módokon kerülhetnek feldolgozásra. Elsőként általában a gyűjtött adatok előzetes feldolgozása, majd a közöttük lévő lineáris kapcsolatok meghatározása, vagyis korrelálása történik, amit az így előállt adatok elemzése követ. Ehhez sok esetben valamilyen mesterséges intelligenciás megoldás használható, például egy döntési fa vagy egy neurális háló. A tervezés során szükség van olyan irányelvek (policy) meghatározására is, melyek alapján eldönthető, hogy a szolgáltatás megfelelően működik-e, vagy sem. A rendszerben rendkívül sokféle hiba fordulhat elő, ezért az policy-k meghatározása rendkívül alapos tervezést követel meg a mérnököktől. Az adatok elemzését végző folyamatokat a policy-kkal egy analitikus alkalmazás köti össze. Az analitikus alkalmazás ezután dönthet arról, hogy be kell-e valamilyen módon avatkozni a rendszer működésébe annak érdekében, hogy az a tőle elvárt módon működjön, így biztosítva az állandó minőséget. A closed loop segítségével nem csak a hagyományos értelemben vett hibák detektálhatók a rendszerben, hanem olyan jelenségek is, mint az erőforrások túlterheltsége, vagy éppen kihasználatlansága, de használható jövőbeli anomáliák felderítésére is.

A zárt láncú szabályozásról összességében elmondható, hogy használatával egy teljes értékű szoftver ökoszisztéma alakítható ki, amely az állandó megfigyelésnek és visszacsatolásnak köszönhetően dinamikusan képes detektálni a hibákat, ezeket akár már jó előre jelezni, így a szolgáltatás teljes életciklusa alatt stabilan tudja nyújtani a tőle elvárt működést. Képes alkalmazkodni a folyamatosan változó igényekhez is, ezért a rendszer könnyen skálázhatóvá válik. A closed loop a rendszert önkonfiguráló, öngyógyító és önoptimalizáló képességekkel ruházza fel. Closed loop használatával a rendszer teljesen önállóan, automatizáltan működik, azonban

3.1. ábra. Zárt láncú szabályozókör[49]

fontos megemlíteni, hogy ez nem azt jelenti, hogy emberek egyáltalán nem szükségesek a működtetéséhez. Ha magát a rendszert nem is, a closed loop-ot emberek tartják karban, hiszen az idő során fény derülhet olyan hibákra, amelyek a closed loop helytelen működését eredményezhetik. Ebből kifolyólag a policy-k vagy az analitikus alkalmazás felülvizsgálatára, illetve frissítésére időnként szükség van.

3.3. Open Network Automation Platform (ONAP)

Ebben a fejezetben az ONAP [50] platform bemutatására kerül sor. Szó lesz a platform kialakulásáról, legfontosabb feladatairól és komponenseiről. Annak ellenére, hogy az ONAP ennél sokkal több funkcionalitást lát el, a dolgozatban csak az annak témája szempontjából jelentőséggel bíró komponensek kerülnek részletes kifejtésre.

3.3.1. Platform fejlődése

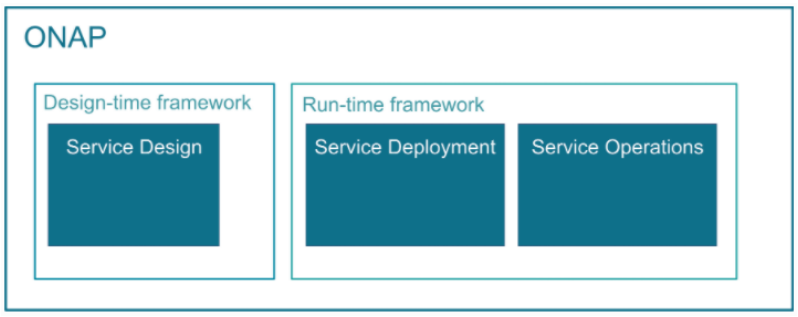
Az ONAP a megalakulása óta már a nyolcadik (Honolulu) kiadásnál jár. A Guilin kiadás megjelenésének dátuma 2021. április 15. Az ONAP az NFV MANO funkcionalitását kibővítve az általa nyújtott különféle hálózati szolgáltatások biztosításáért is felel, ezek a Service Fulfillment és Service Assurance.

Az ONAP betűszó az Open Network Automation Platform rövidítése. A névből a nyílt forráskódúságra és az ONAP által nyújtott szolgáltatásokra is lehet következtetni. A nyílt forráskódú szoftvereknek számtalan előnye van. Ezek közül ez együttműködés lehetősége bír a legnagyobb jelentőséggel, hiszen így a legnagyobb vállalatok együtt alakíthatnak ki egy olyan rendszert, amelyben minden fél igénye a rendszer szempontjából a lehető legoptimálisabb módon teljesül. Másik fontos szempont az open-source világban, hogy a szoftvert a jellegéből

adódóan nem csak az tesztelheti aki készítette, hanem azok is, akik aktívan használják. Ezáltal a szoftver minősége esetenként magasabb lehet, mint egyéb nem open-source termékek esetében. Annak, hogy ha egyszerre akár több vállalat is használja ugyanazt az open-source projektet az az előnye, hogy az ilyen szoftvereket a vevő a saját rendszerébe könnyen tudja integrálni. A platform meghatározó szerepe maga után vonja azt is, hogy a szabványosítással foglalkozó szervezetek, mint például az ETSI NFV MANO és a 3GPP az ONAP esetében a szoftverhez tudják igazítani szabványaikat, ha az ott látott megvalósítás megfelel a követelményeiknek. Erre már voltak példák az ONAP-nál, hiszen az ETSI GS NFV-IFA 011 [51] és az ETSI GS NFV-SOL 001 [52] szabványok az ONAP alapján kerültek frissítésre [53].

Az ONAP az ETSI NFV architektúráját tekintve egyfajta megvalósítása a MANO-nak, amelyet az általa nyújtott szolgáltatás biztosításával (Service Assurance) egészít ki. Az ONAP egy olyan rendszer, amely képes hálózati funkciók orkesztrációjára, legyenek ezek felhő alapú (CNF), virtuális (VNF) vagy valódi (PNF) hálózati funkciók. A Service Assurance-t a zárt láncú szabályozás (closed loop) segítségével valósítja meg, amely lehetőséget ad a begyűjtött és elemzett adatok alapján -a rendszer javítása érdekében- szükség esetén automatikusan beavatkozni. Az ONAP fő gazdasági célja, hogy csökkentse a szoftverek leszállításának idejét, legyen alkalmazkodóképes és biztosítsa azt a rugalmasságot, amelyet a mai modern rendszerek, mint például az Amazon vagy a Google biztosít a felhasználói számára. Továbbá az is fontos, hogy az ONAP-ot használó vállalatok elkerülhetik az egy gyártóhoz való kötöttséget (vendor lock-in), hiszen a rendszer akármilyen eszközön képes futni és a különféle hardverek könnyen együtt tudnak működni a nyílt interfészeken keresztül.

Az ONAP legfőbb tevékenységei a tervezés, a fejlesztés és az üzemeltetés, melyet a két legfontosabb keretrendszere biztosít. Ez a két keretrendszer a Design-time Framework és a Run-time Framework. A tevékenységek alapján a szolgáltatások leszállításának folyamata három fő részre osztható. A keretrendszerek és a részfolyamatok kapcsolata a 3.2 ábrán látható.



3.2. ábra. Szolgáltatás tervezéstől az üzemeltetésig[54]

Az erőforrások modellezése és a köztük lévő kapcsolatok meghatározása a szolgáltatás megtervezésének (Service Design) fázisában történik. Ekkor történik azon irányelvek (Policy) megtervezése is, melyek alapján az alkalmazásról és egyes elemeiről eldönthető az üzemeltetés időszakában, hogy helyesen működik-e, vagy sem. Ezt a fázist más néven "Day 0"-nak hívják.

A szolgáltatás fejlesztésének (Service Deploy) fázisában történik a szolgáltatás automatizált indítása. Az automatizáltság azt jelenti, hogy a szolgáltatás akkor lesz példányosítva, amikor

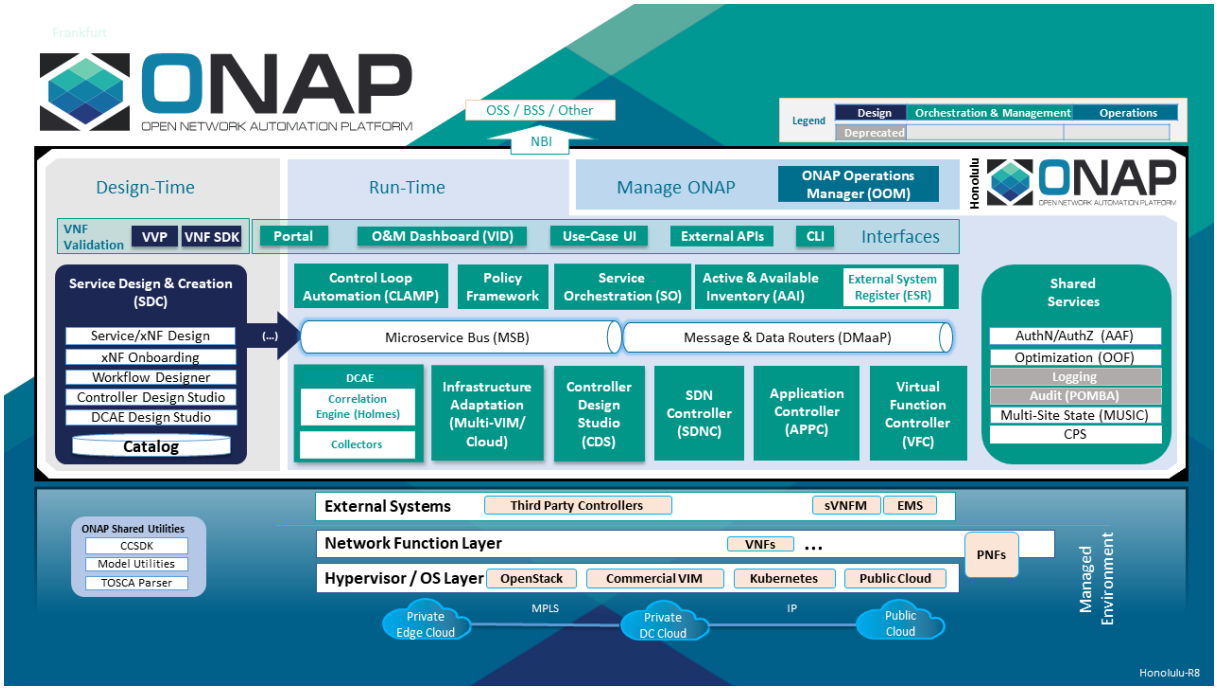
arra szükség lesz, valamint azt, hogy a szolgáltatás annyi példányban kerül elindításra, amennyiben szükség van rá. A szolgáltatások fejlesztése a tervezési fázisban megállapított irányelvek alapján egy orkesztrációs és vezérlési keretrendszer (Service Orchestrator and Controllers) segítségével történik. Ez a fázis a "Day 1".

A szolgáltatás üzemeltetésének (Service operations) fázisában egy analitikus keretrendszer felügyeli a szolgáltatás viselkedését. Az analitikák és irányelvek alapján a vezérlési keretrendszer a meghibásodott szolgáltatásokat vagy javítja, vagy az igények változásával le, illetve felskálázza a rendszert. Az itt felsorolt tevékenységeket más néven "Day 2" beli tevékenységeknek nevezik.

Az ONAP egy olyan platformot nyújt, amely terméktől és szolgáltatástól függetlenül mindenki számára egyformán biztosítja a tervezési, fejlesztési és üzemeltetési lehetőségeket.

3.3.2. Platform felépítése

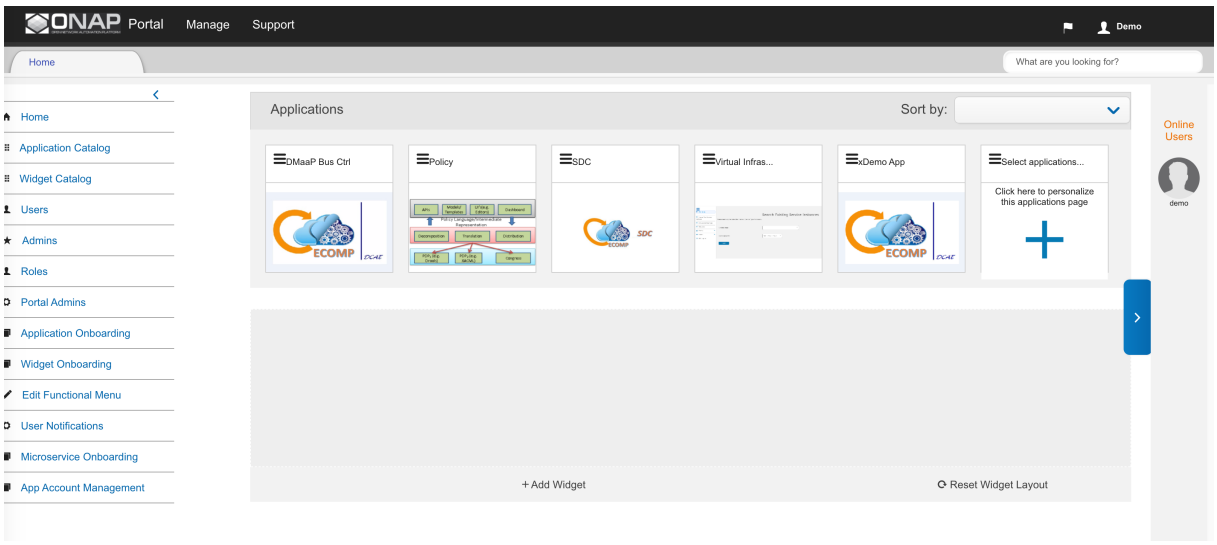
Az ONAP egy rendkívül összetett rendszer.Használata során nem csak a komponensek száma, hanem a felhasznált technológiák változatossága is szembetűnő. A Honolulu ONAP verzió architektúrája a 3.3 ábrán látható.



3.3. ábra. ONAP architektúra[55]

Az ONAP telepítését követően elérhetővé válik a grafikus interfész, az ONAP Portal, amely segítségével a tervező mérnökök és operátorok sokkal könnyebben tudják az egyes szolgáltatásokat megtervezni és később a rendszert üzemeltetni. Az ONAP Portal felhasználói felülete a 3.4 ábrán látható. Innen az ONAP számos más komponense is elérhető, mint például a Service Design and Creation (SDC) vagy a Policy Framework.

Az ONAP számtalan felhasználási lehetőség támogatására képes. Ennek alátámasztására



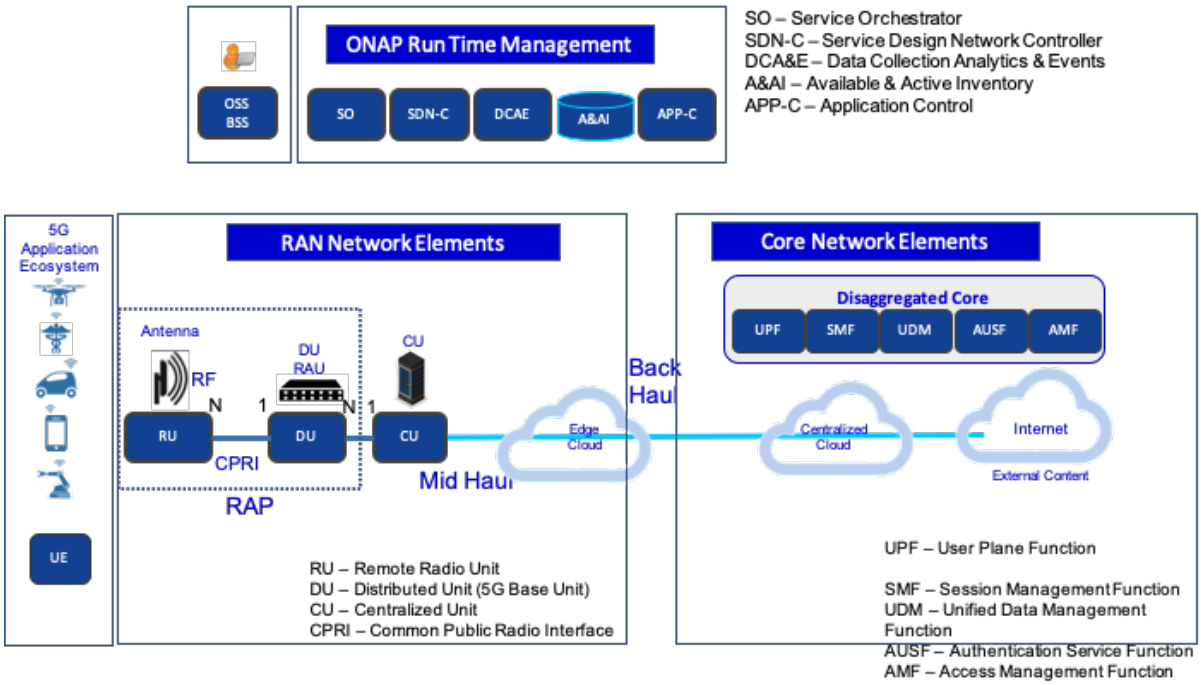
3.4. ábra. ONAP Portal[56]

az ONAP fejlesztői közössége folyamatosan olyan való életből vett problémák megvalósításán dolgozik, amellyel minél inkább megmutatható az ONAP sokszínűsége és a benne rejlő hatalmas potenciál. Az így elkészült tervezetek (blueprint) mind a vevőknek, mind a gyártóknak nagy segítséget nyújtanak egyrészt a lehetőségek ismertetésével, másrészt egy komplett megvalósítás megmutatásával. Ilyen blueprint-ek például, a Virtual Customer Premises Equipment (vCPE), a Voice over LTE (VoLTE), a virtual FireWall (vFW)/ virtual DNS (vDNS), az 5G és a Cross Domain and Cross Layer VPN (CCVPN). Ezek közül az 5G és a CCVPN a hatodik (Frankfurt) kiadás során jelentős továbbfejlesztésen mentek keresztül.

Az ONAP 5G szolgáltatás a 3.5 ábrán látható. Az 5G blueprint öt fő hajtómotorja a végponttól végpontig terjedő szolgáltatás orkesztráció terén a hálózat szeletelés (network slicing), PNF/VNF életciklus menedzsment, PNF integráció és hálózati optimalizáció. A hálózat szeletelése egyedi követelményeket támasztott a rendszer felé. Az 5G blueprint-et az ONAP fejlesztői más szabványosító és open source szervezetekkel szoros együttműködésben készítették el. Ilyen szervezetek voltak a 3GPP, a TM Forum, az ETSI és az O-RAN Software Community.

3.3.3. Zárt láncú szabályozási lehetőségek

Az ONAP teljes körű támogatást kínál a closed loop megvalósítására. Használatával a closed loop folyamatok automatizált módon hajthatók végre. A closed loop megjelenik az ONAP Day0, Day1 és Day2 tevékenységei közt is. Az ONAP célja a closed loop megvalósításával a rendszer által detektált hibák kijavítása és az erőforrások menedzsmentje abban az esetben, ha a rendszer erőforrásainak kihasználtsága az előre meghatározott küszöbértékeket átlépi. Azok a fő ONAP komponensek, amelyek részt vesznek a closed loop kivitelezésében a Service Design and Creation (SDC), a VNF Event Streaming (VES), a Data Collection, Analytics and Events (DCAE), valamint a Policy Framework és a Service Orchestrator (SO). Egy closed loop megtervezése a closed loop sablon (template) elkészítésével kezdődik, amelyet ezután egy hálózati



3.5. ábra. ONAP 5G Blueprint[57]

szolgáltatáshoz kell rendelni. A sablon elkészítéséhez szükség van egy artifact-ra, ami egy "cloudify" blueprint. Ezt követően a control loop-ot közzé kell tenni az ONAP-ban. A closed-loop az indítást követően mikro service-ként fut a rendszerben. A vizsgált hálózati szolgáltatás a VES-en keresztül küldi az adatot a DCAE-nek. Ezeket az adatokat a DCAE-ből olvassa ki a Policy Framework, amely megvizsgálja őket és ha problémát észlel a rendszer működésében -zárva a láncot-, beavatkozik.

4. fejezet

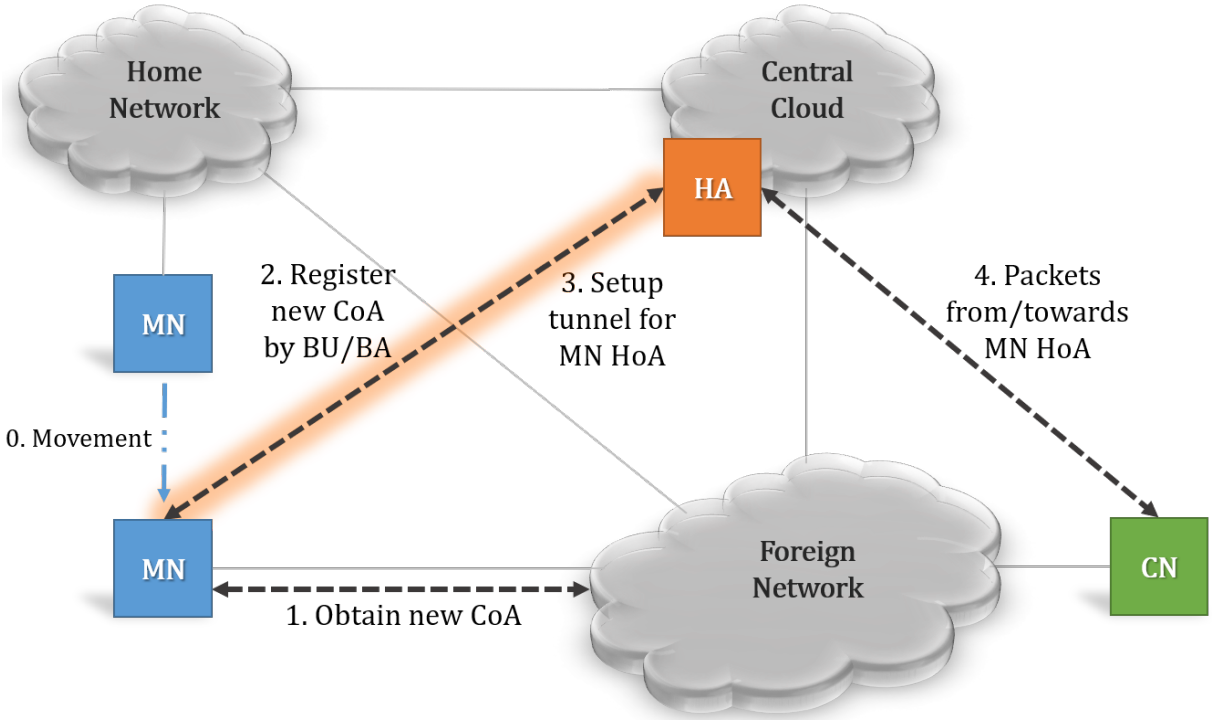
Kitűzött feladat

Ebben a fejezetben bemutatom a dolgozat céljaként kitűzött feladatomat. Az általam megvalósítandó problémakör egy hagyományos fizikai hardveren futó hálózati szolgáltatás felhőalapúvá tétele. A szolgáltatás valós hálózati környezetbe történő telepítése után gondoskodom kell annak orkesztrálásáról is annak érdekében, hogy a szolgáltatás megbízható módon működjön és megfeleljen a hagyományos fizikai hardver-alapú megfelelőjétől elvárt követelményeknek. Az a hálózati szolgáltatás, amelyet a kutatásaim során felhasználok, az IPv6 alapú mobilitáskezelés egy felhőalapú megvalósítása. A végső megoldásom egyik fő célja, hogy az mindenféle infrastruktúrától és hálózati szolgáltatótól függetlenül működjön és bármilyen felhőinfrastruktúrán telepíthető legyen. A dolgozatom elkészítése során figyelembe veszem az összes erre a célra alkalmas virtualizációs és hálózati automatizációs technológiát és a végső javaslatban a számomra legmegfelelőbb technológiák megjelelölésével egy teljeskörű megoldást dolgozok ki a kijelölt feladatra. A dolgozatomban részletesen ismertetem a megoldás elméleti hátterét és a gyakorlati megvalósítását. A megoldás ismertetése során fontosnak tartom felhívni a figyelmet a hardver alapú megoldástól való eltérésekre, melyek egyaránt magukban foglalják az új technológia alkalmazásával járó előnyöket és hátrányokat. A komponensek egymástól független kezelhetősége és skálázhatósága érdekében a megvalósítás során a szolgáltatás egyes komponensei által megvalósított feladatkörök lehető legnagyobb mértékű szeparációjára törekszem. A dolgozatban ismertetem a különböző komponensek szerepét a megvalósításban. Ahhoz, hogy a megoldás tesztelhető és különféle vizsgálatok során teljesítménye is mérhető legyen, megvalósítom azt a valós hálózati környezetet, amelyben a tesztelés elvégezhető és a mérések a rendszer egészének teljesítményéről objektív képet mutatnak. A dolgozat zárásaként a mért eredményeket ismertetem és értékelem. A felhasznált technológiák által nyújtott teljesítményt összehasonlítom majd más, felhő- és fizikai hardver-alapú megoldásokkal.

5. fejezet

Felhőalapú mobilitáskezelés szolgáltatás-menedzsment megvalósítása

Ebben a fejezetben bemutatom a felhőalapú mobilitáskezeléshez elkészített szolgáltatás-menedzsment megvalósításomat. A Mobile IPv6 (MIPv6) protokollnak megfelelően implementált és a protokoll szerint definiált Home Agent felhőalapú változatához az ONAP zárt láncú szabályozókörének felhasználásával készítettem el a megoldást. A felhőalapú mobilitáskezelés magas-szintű architektúra rajzát a 5.1 ábra mutatja. A mobilitáskezelés három fő szereplője a mobil eszköz (Mobile Node), a honos ügynök (Home Agent) és a távoli eszköz (Corresponding Node). Az eltérés a nem felhőalapú mobilitáskezeléshez képest az, hogy ebben az esetben a Home Agent egy felhőben futó hálózati szolgáltatásként (Cloud-native Network Function) szolgálja ki a hozzá csatlakozó Mobile Node-okat.



5.1. ábra. Felhőalapú mobilitáskezelés magas-szintű architektúra rajza

A Mobile Node, amikor kilép az otthoni hálózatból és egy idegen hálózatba lép, akkor a meglévő otthoni címe (Home Address - HoA) mellé az idegen hálózat-beli azonosításhoz Care-of-Address (CoA) címet igényel. A CoA címét ezután egy Binding Update formájában felküldi a felhőben futó Home Agent-nek, ami a saját Binding Cache-ében eltárolja a CoA címet. A Binding Cache tartalmazza a megfeleltetéseket a HoA és a CoA címek között. A Home Agent így tudja, hogy ha a Mobile Node otthoni címére egy csomag érkezik, merre kell továbbítania azt. A Home Agent a Binding Update nyugtázására Binding Acknowledgement üzenetet küld a Mobile Node-nak. Ezután a Mobile Node és a Home Agent között tunnel felépítés történik, ami után közöttük a továbbiakban ezen keresztül történik a kommunikáció. Ha a Mobile Node üzenetet küld egy távoli eszköznek, vagy ha a távoli eszköz csomagot küld a mobil eszköznek a kommunikáció a honos ügynöktől/-ig a tunnel-en keresztül folyik. A fejezet későbbi részében bemutatom a felhőalapú MIPv6 Home Agent szolgáltatás implementációját, majd az ehhez elkészített szolgáltatás-menedzsment megvalósítást.

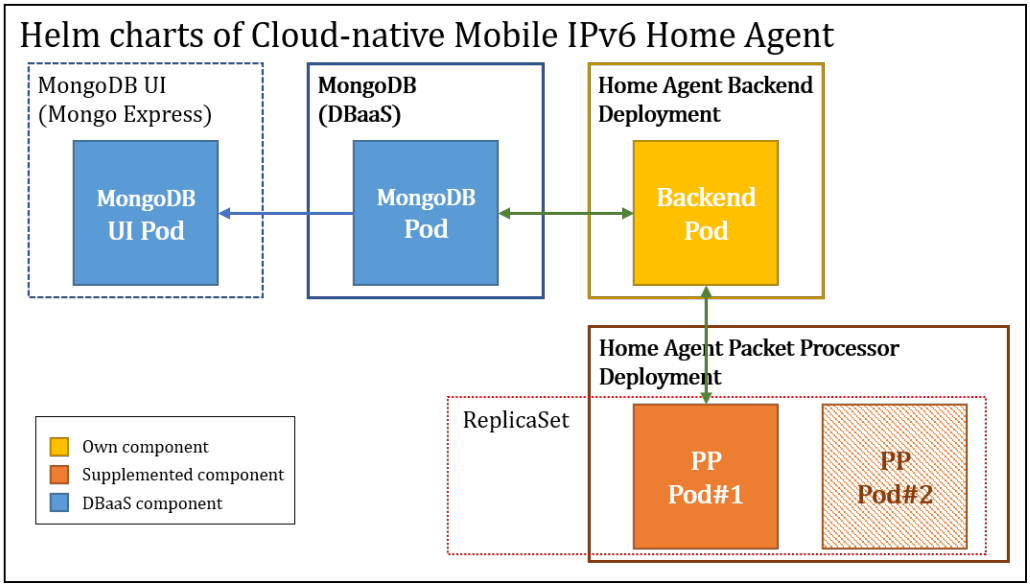
5.1. Felhőalapú MIPv6 Home Agent

A felhőalapú MIPv6 Home Agent szolgáltatás, amelyhez a zárt láncú szabályozókört készítettem, megvalósítja a MIPv6 protokoll Home Agent-re vonatkozó számos funkcióját, azonban az általa ellátott feladatok köre nem nyújt átfogó megoldást a protokollban definiált összes funkcionalitásra. A megoldás a protokollban leírtaknak megfelelően, szabványosan készült el, ezáltal jövőben -a hiányzó funciók implementálását követően- alkalmas lehet nem felhőalapú társai leváltására. A felhőalapú MIPv6 Home Agent architektúrájának ismertetése után a szabványban meghatározott és az általa megvalósított feladatköröket mutatom be. Ezt követően demonstrálom, hogy egy hálózati automatizációs platform és a zárt láncú szabályozókör segítségével hogyan lehet a Home Agent szolgáltatást biztosítani.

5.1.1. Architektúra

A felhőalapú MIPv6 Home Agent szolgáltatást három fő alkotóelem építi fel. Ezeket a 5.2 ábra szemlélteti. A mobilitáskezelés szempontjából az első és legfontosabb a Home Agent Packet Processor, amely képes a Mobile Node-ok felől érkező Binding Update-eket fogadni, az érkező üzeneteket validáni, majd számukra a megfelelő Binding Acknowledgement-et visszaküldeni. A Home Agent Packet Processor-on keresztül folyik a kommunikáció a hozzá beregisztrált Mobile Node-ok és a velük kapcsolatba lépő Corresponding Node-ok között. A Home Agent Packet Processor korábbi megvalósításon alapul (ezt az ábrán narancssárga színnel jelöltem), amelyről a "Cloud-native MIPv6 mobilitás-menedzsment protokoll implementációja és vizsgálata" [58] című diplomatervben olvasható további információ. Az IPv6 alapú működés miatt egy speciális hálózati infrastruktúra kialakítására is szükség volt, amelynek részletei a "Cloud-native MIPv6 mobilitás-menezsment szolgáltatás tervezése" [59] című diplomatervben olvashatók. Ahogy említettem, a Home Agent Packet Processor egy korábbi megvalósításon

alapul, amely nem tartalmazta a szolgáltatás-menedzsmenthez és a zárt láncú szabályozáshoz szükséges logikát, kiegészítésére emiatt volt szükség.



5.2. ábra. Felhőalapú MIPv6 Home Agent architektúra

A felhőalapú MIPv6 Home Agent következő fontos alkotóeleme a Home Agent Database. A Home Agent Database egy adatbázis, amely tartalmazza a szabványban definiált Binding Cache-t, így tehát az összerendeléseket az egyes Mobile Node-ok Home Address címe és Care-of-Address címe között. A Home Agent Database a Binding Cache-en felül további adatokat tartalmaz, amelyek főként a szolgáltatás-menedzsmenthez kapcsolódó feladatok ellátásában vannak segítségre. Ilyenek például a futó Home Agent Packet Processor(ok) azonosítására szolgáló adatok, valamint az ezek állapotáról gyűjtött információk. Az Home Agent Database-hez tartozik egy grafikus felület, amely az adatbázis aktuális állapotát tükrözi. A grafikus felületen keresztül különféle adatbázis műveletek is egyszerűen végrehajthatók, de ezt ebben az esetben kizárólag az adatok megtekintésére használom. A Home Agent Database és a hozzá tartozó grafikus felület (Home Agent Database GUI) Database-as-a-Service (DBaaS) rendszerintegrációs szinten van megvalósítva, ezáltal az adatbázis az architechtúrában tetszőlegesen kicserélhető alternatív SQL/noSQL alapú adatbázis megoldásokra. A DBaaS előnye, hogy direkt hozzáférést biztosít az adatbázishoz úgy, hogy a fizikai hardver fenntartása és üzemeltetésének feladatai nem terhelik a rendszer fejlesztését. A MongoDB adatbázist és a hozzá tartozó MongoUI grafikus adminisztrátori felületet az ábrán kékkel jelöltem. A felhőalapú MIPv6 Home Agent harmadik komponense a Home Agent Backend, amely szerepe szolgáltatás-mendezsment szempontból a legnagyobb jelentőséggel bíró. A Home Agent Backend egyben kiszolgálja a Home Agent Packet Processor-t, hiszen a Home Agent Database rajta keresztül érhető el a Home Agent Packet Processor számára és ezen felül a szolgáltatás-menedzsmenthez kapcsolódó feladatokat is ő látja el. Ilyen feladat például a Home Agent Packet Processor(ok) aktuális állapotának jelentése a hálózati automatizációs platform felé. A Home Agent Backend bevezetésére abból az okból következően volt szükség, hogy a Home Agent Packet Processor szabványban definiált felada-

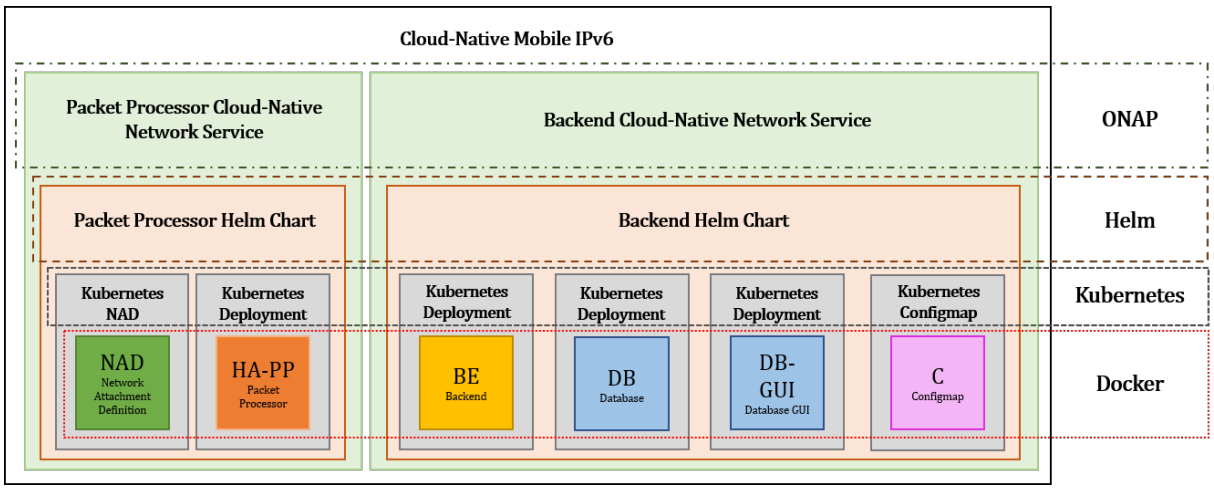
tai szeparáltan működjenek a hálózati orkesztrációt segítő egyéb feladatoktól. Ezáltal a Home Agent Packet Processor felelősségei közé teljes egészében a mobilitáskezeléshez kapcsolódó operációk tartoznak és teljesítményére is csak ezen feladatok ellátása gyakorol hatást. Ezt az általam megvalósított komponenst az ábrán sárga színnel jelöltem.

5.1.2. MIPv6 szabvány megfelelés

A Home Agent Packet Processor jelen verziójában egy leszűkített körét tartalmazza azoknak a feladatoknak, melyeket a MIPv6 szabvány [60] definiál. A megvalósított feladatok közé tartozik a Binding Update fogadása, illetve validálása , továbbá a Binding Acknowledgement küldése, valamint a Binding Cache karbantartása. A Binding Acknowledgement kiküldése után a Home Agent Packet Processor tunnelt épít ki a Mobile Node felé, amin keresztül fogadja a Mobile Node-tól érkező csomagokat és amin keresztül a Mobile Node-nak címzett csomagokat továbbítja. A felsorolt feladatokat a Home Agent Packet Processor szabvány szerint látja el, így a jövőben kiegészíthető a teljes funkcionalitás elérése érdekében. Mind a Binding Update üzenet, mind a Binding Acknowledgement üzenet formátuma megfelel a szabványban leírtaknak. A Home Agent Database tárolja a Binding Cache-t, amelyben egy bejegyzés (Binding Cache Entry) az összes kötelező mezőt tartalmazza. A Binding Cache érvénytelen rekordjainak törléséről a Home Agent Backend gondoskodik egy általam készített, periodikusan futtatott szemétgyűjtő algoritmus felhasználásával.

5.1.3. Alkalmazott technológiák

A felhőalapú MIPv6 Home Agent megvalósításához számos technológia alkalmazásásra volt szükség, ezt mutatja a 5.3 ábra. A technológiák bemutatását az alacsonyabb szintektől felfelé haladva teszem meg. Az ábrán pirossal bekeretezve a Docker konténerekben futó alkalmazás komponensek láthatóak, mint például a Packet Processor a Backend vagy a Database. A Docker felett a hierarchiában a Kubernetes helyezkedik el. A konténereket közvetlenül a pod-ok futtatják. A pod-ok futásuk során nem képesek az öngyógyításra, így ha meghibásodnak akkor a Kuberenetes egyszerűen eltávolítja őket a klaszterből. Ezért kell egy olyan vezérlési egység, amely gondoskodik arról, hogy mindig legyen egy, vagy több, az alkalmazáshoz tartozó pod, amely egészségesen működik. Ezt a célt szolgálják a Kubernetes deployment-ek. A deployment-eken felül az ábrán feltüntettem egyéb Kubernetes erőforrásokat is, mint például a Network Attachment Definiton vagy a Configmap. A Network Attachment Definition a Home Agent Packet Processor esetén arra szolgál, hogy a Packet Processor pod-okhoz az indulásuk után IPv6 címet rendel. Ezt a címet fogja használni a Packet Processor a későbbiekben a Mobile Node-al való kommunikációhoz. A Configmap segítségével különféle konfigurációs beállításokat adhatunk meg, amelyeket a deployment-ek figyelembe vesznek a pod-ok vezérlése során. A CN-MIPv6 Home Agent két Helm Chart-ból áll össze.
Egy-egy Helm Chart meghatároz egy Kubernetes-ben futó alkalmazást annak minden erőforrá-



5.3. ábra. A felhőalapú mobilitáskezelés fejlesztése során alkalmazott technológiák

sával együtt. A CN-MIPv6 Home Agent szolgáltatáshoz az egymástól független kezelhetőség és skálázhatóság miatt, két Helm Chart tartozik. Az egyikben a Packet Processor, a másikban a Backend található az adatbázis komponenssel. Az ONAP egy Helm Chart-hoz egy felhőalapú hálózati szolgáltatást rendel, ezért a CN-MIPv6 Home Agent a Home Agent Packet Processor és a Home Agent Backend szolgáltatások egysége.

5.1.4. Szolgáltatás-menedzsment lehetőségek

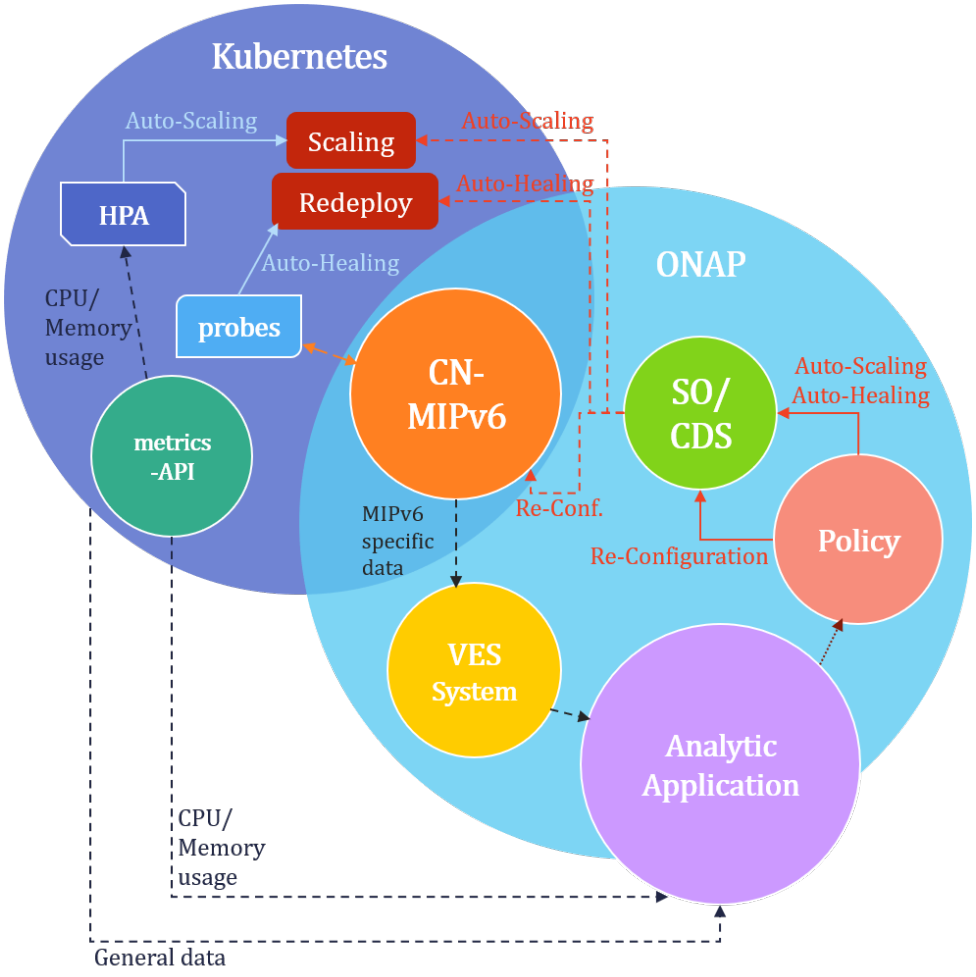
A felhőalapú MIPv6 Home Agent felhőalapú hálózati szolgáltatásként fut (Cloud-native Network Function - CNF), így könnyen orkesztrálható hálózati automatizácós platformok modern technológiáinak segítségével. A Home Agent szolgáltatás folyamatos elérhetőségének biztosítása a felhasználók (Mobile Node) aspektusából kritikus szempont, hiszen sok esetben a távoli eszközzel (Corresponding Node) való kommunikáció a Home Agent-en keresztül történik. Ezáltal, ha megszűnik a kapcsolat a Home Agent-el, akkor megszűnik a kapcsolat a távoli eszközzel is, ami felhasználói elégedetlenséget vált ki. Annak érdekében, hogy ilyen szolgáltatáskiesés ne, vagy csak a lehető legrövidebb időintervallumban fordulhasson elő, szükség van a Home Agent szolgáltatás állandó biztosítására. Manuális eszközökkel egy Home Agent felkonfigurálása a meghibásodást követően órákba is telhet [61], de hálózati automatizációs technológiák segítségével és az életciklus-menedzsment automatizálásával ez az időtartam a töredékére csökkenthető. Ezen állítás igazolására az 6. fejezetben kerül sor. Az automatizált Day0 (design), Day1 (telepítés és konfigurálás) és Day2 (üzemeltetés) tevékenységek következménye, hogy a felhőalapú szolgáltatások egy állapot-visszacsatolásos rendszerben zárt láncú szabályozókörökön keresztül menedzselhetők. A zárt láncú szabályozás lényege, hogy a hálózati szolgáltatást permanens felügyelet alatt tartja és abban az esetben, ha a működésben valamilyen anomália lép fel, automatikusan beavatkozik. Egy ilyen zárt láncú szabályozókört alkalmazva a felhőalapú MIPv6 Home Agent szolgáltatáson a Home Agent kiesése minimalizálható lenne, ezzel maximalizálva a felhasználói megelégedést.Ez a felismerés a dolgozat fő motivációs tényezője.

5.2. Javasolt szolgáltatás-menedzsment megoldás

A felhőalapú mobilitáskezelés szolgáltatás-mendezsmentjének megvalósítását az ONAP zárt láncú szabályzókörének felhasználásával oldottam meg. A megoldás részletes kidolgozása mellett bemutatom a tervezett és a teszteléshez használt valós hálózati környezetet és a tervek alapján elkészített implementációt. A szolgáltatás biztosításához először a meglévő Home Agent Packet Processor-t egészítettem ki és létrehoztam a Binding Cache-t tartlamazó Home Agent Database-t, valamint implementáltam a kettő komponens között található Home Agent Backend-et. Ezután megvalósítottam a Home Agent Packet Processor életciklus mendezsmentjét a Kubernetes által biztosított Konténer Probe-ok segítségével. A szolgáltatás ONAP-ba való kitelepítése után a zárt láncot az ONAP Policy Framework-ben készítettem el. A konkrét megvalósítás részleteit a következő szekciókban írom le.

5.2.1. Felhasznált technológiák

A megoldásomban a szolgáltatás-menedzsmenthez használt egyes technológiák felelősségköreit és összefüggéseit a 5.4 ábra mutatja.



5.4. ábra. A fejlesztés során alkalmazott fontosabb szolgáltatás-menedzsment technológiák

A Kubernetes számtalan lehetőséget nyújt a felhőalapú szolgáltatások felügyeletére és menedzsmentjére. Önmagában már a deployment-ek alkalmazása is jelent egyfajta biztonságot az alkalmazások számára, hiszen a deployment automatikusan kitörli a meghibásodott pod-okat és újakat indít, de ennél sokkal nagyobb jelentőséggel bírnak az életciklus-mendezsmentben aktívan részt vevő probe-ok és a Horizontal Pod Autoscaler. A Horizontal Pod Autoscaler segítségével egyes metrikák alapján az alkalmazás komponensek száma az igények függvényében skálázható felfelé és lefelé. A Horizontal Pod Autoscaler a metrics-API-n keresztül a komponensek processzor és memória felhasználásáról információt gyűjt és ha a kapott értékek elérnek egy tetszőlegesen meghatározott alsó vagy felső értéket, akkor a skálázás automatikusan megtörténik. A probe-ok a deployment-en belül futó pod-ok állapotának folyamatos monitorozásával vizsgálni tudják, hogy azok az elvárásoknak megfelelően működnek-e és abban a pillanatban, amikor a pod elérhetetlénné válik, tájékoztatják erről a deployment-et, ami kitörli a meghibásodott pod-ot és a helyére egy újat indít. Az ONAP a zárt láncú szabályozáshoz biztosítja a keretrendszert. A zárt láncú szabályozásban részt vevő ONAP komponenseket feltüntettem az ábrán is. Az ONAP előnye a Kubernetes-hez képest, hogy az általa menedzselt alkalmazás komponenseknek nem kell ugyanazon felhő lokációban futnia, hiszen képes a különböző felhőkben futó komponenseket összefogni és ezek működését felhőkön átívelve szabályozni. Azt fontos hangsúlyozni, hogy az ONAP nem helyettesíti a Kubernetes-t minden funkcionalitásában, ezért a kettő technológia együttes alkalmazásásával érhető el a legátfogóbb és leguniverzálisabb megoldás. Hatalmas előnye mindegyik felsorolt technológiának, hogy a saját implementáció függvényében egyedileg módosíthatók, ezáltal felhasználásukkal minden típusú alkalmazás egyedileg felügyelhető és szabályozható. Az alábbi felsorolásban összefoglalom a felhőalapú MIPv6 Home Agent szolgáltatás-menedzsmentjének megvalósításához felhasznált technológiákat.

A CN-MIPv6 Home Agent implementációban felhasznált technológiák:

- A CN-MIPv6 Home Agent szolgáltatás elemeinek konténerben való futtatásáról a **Docker** platform gondoskodik.
- A Docker konténerek menedzsmentjét a **Kubernetes** különböző logikai egységei végzik.
- A Kubernetes komponensek által meghatározott alkalmazást a **Helm Chart**-ok fogják össze egy egységgé.
- A CN-MIPv6 Home Agent felhőalapú hálózati szolgáltatás orkesztrációját és menedzsmentjét az **ONAP** végzi.

Ahhoz, hogy a CN-MIPv6 Home Agent szolgáltatás a Kubernetes és az ONAP alapú zárt láncú szabályozókörök segítségével menedzselhető legyen, az alább felsorolt követelményeknek kell eleget tennie.

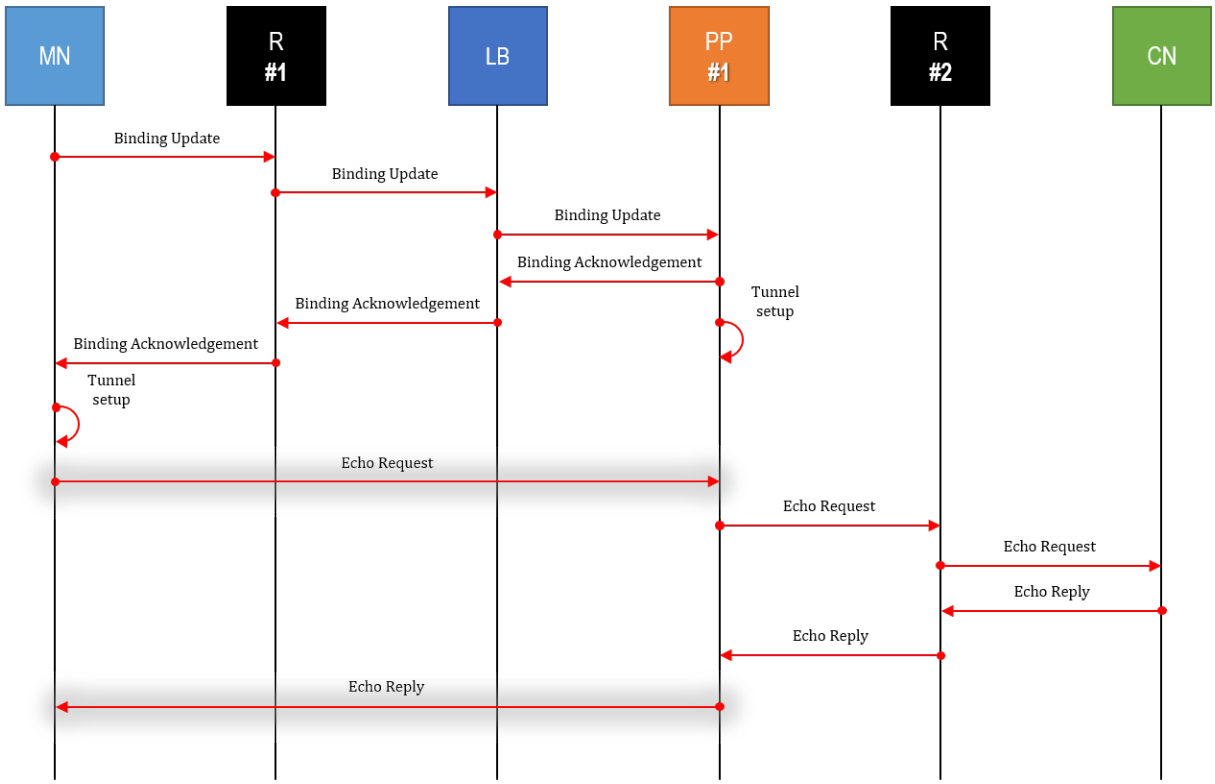
Követelmények a CN-MIPv6 Home Agent szolgáltatással szemben:

- A szolgáltatás egyes komponensei egy hozzájuk tartozó Docker image alapján létrehozott Docker konténerben futtathatóak.
- A komponensek megfeleltethetők egy-egy Kubernetes Deployment-nek, amelyben a Pod-ok a komponensekhez tartozó Docker konténert futtatják.
- A Home Agent Packet Processor Deployment definálja a komponenshez tartozó Readiness és Liveness Kubernetes Probe-okat.
- A Home Agent Backend a Packet Processor-tól érkező analitikus adatokat a Home Agent Database adatbázisban tárolja el egy erre a célra a létrehozott adatbázis táblában.
- A Home Agent Backend implementálja az ONAP által meghatározott szabványos VES üzenetek különböző típusait.
- A Home Agent Backend meghatározott események bekövetkezésekor VES üzenetet küld az ONAP VES Collector komponensének.
- A CN-MIPv6 Home Agent elemeihez tartozó Kubernetes Deployment-eket egy Helm chart fogja össze, amelyben a változók értékeit a values.yaml fájl tartalmazza.
- A CN-MIPv6 Home Agent szolgáltatáshoz tartozik egy ONAP által feldolgozható CBA csomag, amely alapján a felhőalapú hálózati szolgáltatás létrejön.
- A CBA csomag tartalmazza a szolgáltatást leíró Helm chart-ot, az alkalmazáshoz elkészített controller blueprint-et és a zárt láncú szabályozáshoz szükséges elemeket, mint például az ONAP Policy-t TOSCA nyelven leírva, és a Policy-hez tartozó Kotlin vagy Python szkripteket.

5.2.2. A javasolt architektúra és jelzési folyamatai

Az általam javasolt szolgáltatás-menedzsment megoldás a felhasznált technológiákat illetően egy általános ajánlást fogalmaz meg felhőalapú hálózati szolgáltatások zárt láncú szabályozásához. A megvalósítást illetően viszont egy, a felhőalapú mobilitáskezelésre jellemző szcenárió köré épül fel. Ez a szcenárió, amelyre Failover néven hivatkozom a továbbiakban, azt az esetet dolgozza fel, hogy mi történik akkor, amikor a Home Agent szolgáltatás elérhetetlenné válik és a Mobile Node-nak megszűnik a kapcsolata a külvilággal. Failover-re minden szolgáltatás esetén szükség van, azonban a megvalósítás ezen része már teljesen specifikus az egyes alkalmazásoknál.
A Home Agent meghibásodása előtti üzentváltásokat a részt vevő komponensek között a 5.5 ábra mutatja. A üzentváltást ábrázoló folyamat résztvevői a Mobile Node; az egyes számú router, amely a Home Agent és a Mobile Node közti hálózati infrastruktúrát helyettesíti; a Load

Balancer, amely terhelés megosztást végez a Home Agent Packet Processor-ok között; a kettes számú router, amely a Corresponding Node és a Home Agent közti hálózati infrastruktúrát szimbolizálja, valamint a Corresponding Node. Az első üzenet elküldése előtt feltételezzük, hogy a Mobile Node elhagyta azt a hálózatot, amelyben korábban tartózkodott és az új hálózatban kapott Care-of-Address címéről szeretné a Home Agent-et tájékoztatni, hogy a Corresponding Node-dal kommunikálni tudjon. Ezért a Home Agent-nek Binding Update (BU) üzenetet küld.

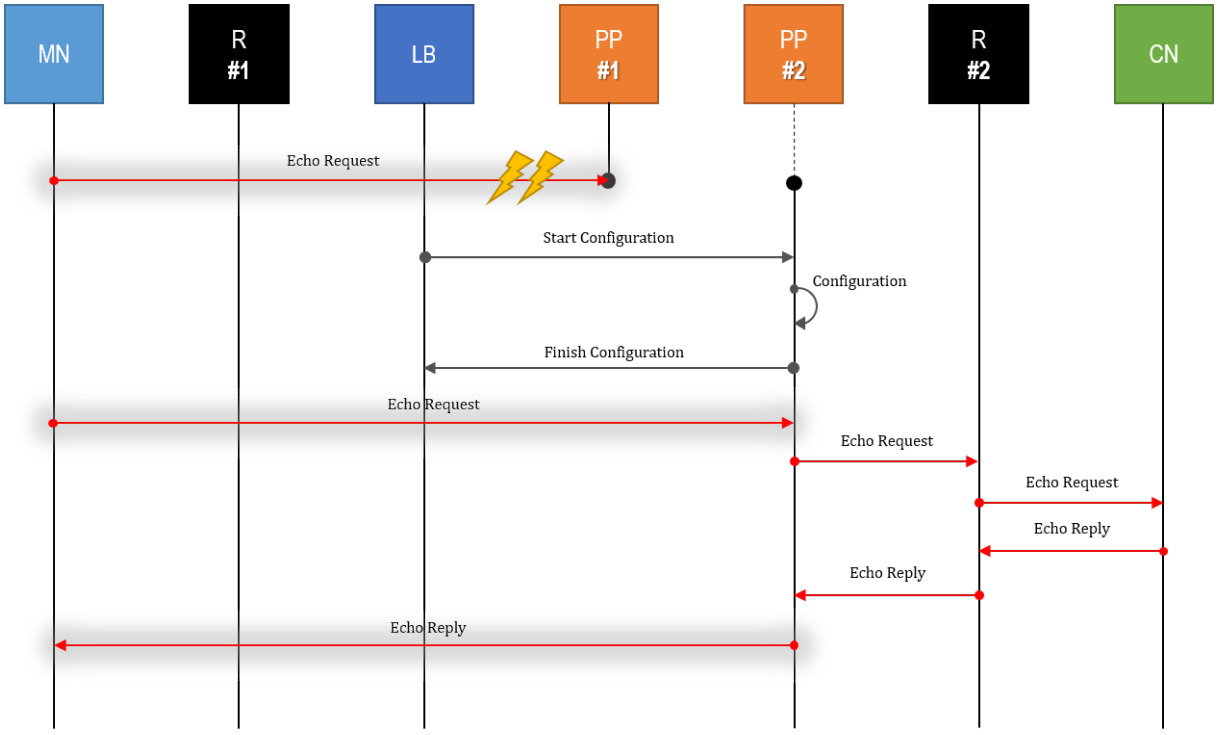


5.5. ábra. A javasolt architektúra jelzési folyamatai

A Binding Update megérkezik az egyes számú routerhez, amely a csomagot a Load Balancer felé továbbítja. A Load Balancer terheléselosztó egység a külvilág felé elrejti a Home Agent belső működését és a Packet Processor-ok között a terhelést egyenletesen ossza meg. A hálózatban, ha valamelyik Mobile Node igénybe akarja venni a Home Agent szolgáltatást, a csomagot a Load Balancer-nek kell címeznie, hiszen a Load Balancer gondoskodik arról, hogy az üzenetet megérkezzen a megvalósított kiválasztási algoritmus által kisorsolt Packet Processor-hoz. A Mobile Node későbbi üzeneteit a Load Balancer ezután mindig ugyanahhoz a Packet Processor-hoz irányítja tovább, egészen addig amíg valami miatt az a Packet Processor elérhetetlenné válik. A Load Balancer által megvalósított terheléselosztási algoritmusra számtalan ajánlás létezik, mint például (Súlyozott) Round Robin, First Alive, (Súlyozott) Least Connections vagy Hash alapú megoldások [62]. A Load Balancer tehát a Binding Update üzenetet egy általa választott Packet Processor-nak címzi, amely feldolgozza a Binding Update-et és összeállítja a Binding Acknowledgement (BA) üzenetet, amelyet visszaküld a Load Balancer-nek. Ezután a Binding Acknowledgement az egyes számú router-en keresztül visszaérkezik a Mobile Node-hoz. A BU/BA üzenetváltás hatására a a Mobile Node és a Packet Processor között megtörténik a tun-

nel felhúzása. A Mobile Node Corresponding Node-nak küldött adatcsomagja a csatornában halad a Packet Processor-ig. A Packet Processor a csomagot a Corresponding Node felé irányítja a közbe eső kettes számú router-en keresztül. A Corresponding Node válaszként küldött csomagja a Packet Proccesor-hoz érkezése után újra bekerül a csatornába, amelyen keresztül megérkezik a Mobile Node-hoz.

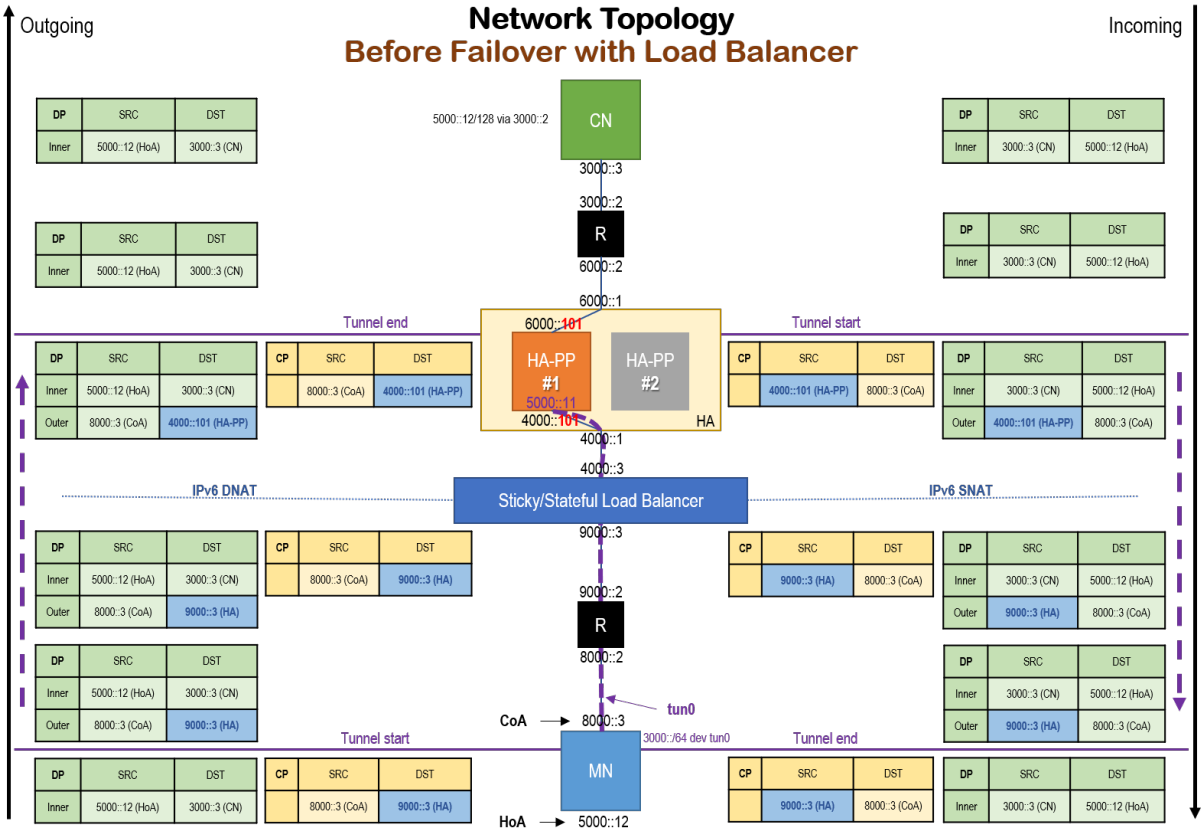
A Load Balancer-rel kiegészített megoldásban az az innováció a hagyományos felhő alapú megoldásokhoz képest, hogy ilyenkor a Mobile Node nem észlel semmit a Load Balancer által elfedett szolgáltatás architektúrájának megváltozásából, így ha a Packet Proccesor elérhetetlenné válik, a Load Balancer a Mobile Node-ot automatikusan egy másik Packet Processor-hoz rendeli. A meghibásodott helyébe lépő új Packet Processor az adatbázisban eltárolt konfigurációs beállítások alkalmazásával elődjét szinte teljesen észrevétlenül váltja fel. Ez a Failover, amelynek szekvencia diagramját a 5.6 ábra mutatja. Az ábra szemlélteti, hogy amikor megszűnik a szolgáltatás az első Packet Processor-on keresztül, a Load Balancer automatikusan elindítja a második Packet Processor konfigurációját. Miután ez megtörtént, a Mobile Node és a Corresponding Node között újra létrejön a kapcsolat. Ideális esetben a Mobile Node-nak nem kell várnia arra, hogy a Load Balancer egy újabb Packet Processor-t instanciáljon, mivel a tartalék, de aktív státuszban lévő Packet Processor ilyenkor egyből át tudja venni a meghibásodott erőforrás szerepét. A Failover időtartama ebben az esetben a legrövidebb. Ha az új Packet Processor még nem létezik a meghibásodás bekövetkezésekor a Failover tovább tart, hiszen ekkor meg kell várni, hogy a Packet Processor elinduljon, megtörténjen a felkonfigurálása és képessé váljon feladata ellátásra. Az általam javasolt megoldásban ezért a rendszerben mindig kell lennie egy tartalék Packet Processor-nak is.



5.6. ábra. A javasolt architektúra jelzési folyamatai Failover során

5.2.3. A javasolt megoldás hálózati integrációja

A javasolt megoldás hálózati integrációja után előálló architektúrát a 5.7 ábra mutatja. A Mobile Node a Home Agent-től különböző hálózaton helyezkedik el, ezzel utalva arra a szituációra, amikor a Mobile Node egy idegen hálózatban tartózkodik. A hálózati integráció javasolt módjának és így a hálózati architektúrának az általános részleteit az érthetőség kedvéért minta IP címekkel mutatom be. Az idegen hálózat IPv6 tartománya a 8000:8000:8000:8000::/64-es. A Mobile Node idegen hálózatbeli IP címe (Care-of-Address) a 8000:8000:8000:8000::3. A Mobile Node azonosítója, amely örökké tartó egyediséget biztosít számára, az otthoni címe (Home Address). Ez az 5000::12 az 5000::/124 IP tartományból.



5.7. ábra. A javasolt megoldás hálózati architektúrája és a User/Control Plane jellemzői

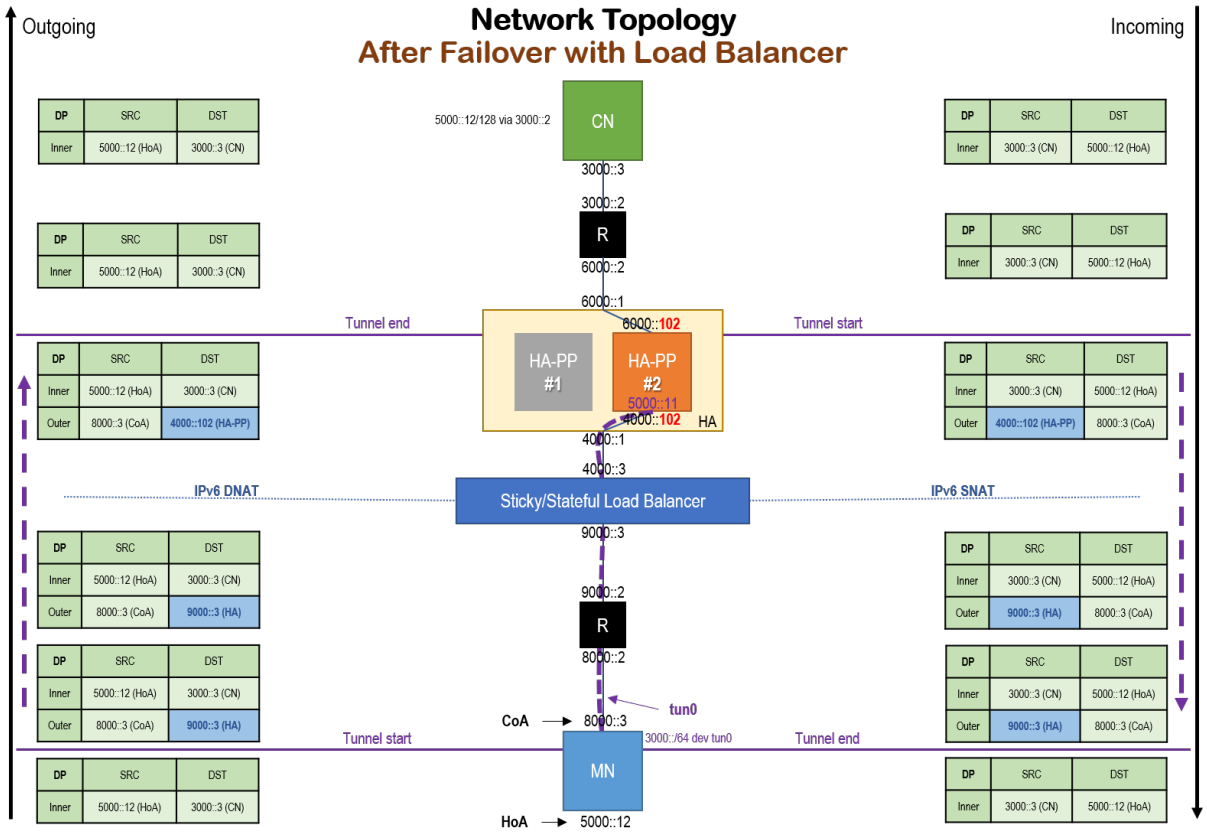
A topológiában a Mobile Node felett elhelyezkedő hálózati elem egy router. Ez a router biztosítja a Home Agent és a mobil eszköz közti összeköttést. A Router két hálózati interfésszel rendelkezik a két különböző hálózat irányába. A Mobile Node felé eső interfész IP címe a 8000:8000:8000:8000::2, a Home Agent felé eső a 9000::2, a 9000::/64 tartományból. A Home Agent a Mobile Node számára egy load balanceren keresztül érhető el. A Load Balancer szerepe a javasolt hálózati architektúrában, hogy a Home Agent belső működését elrejtse a külvilág felé, valamint, hogy a Packet Processor-ok közt a terhelést egyenlően ossza el. Ezáltal a Mobile Node soha nem fogja megtudni, hogy valójában melyik Packet Processor-al van közvetlen összeköttetésben. A Load Balancer példaként használt IP címe a 9000::3. A belső kommunikáció lebonyolításhoz a Load Balancer egy további interfésszel is rendelkezik a Home Agent

felé. Az ehhez tartozó IP cím a 4000:4000:4000:4000::3. A Home Agent szolgáltatás egy felhőben található virtuális gépre telepített Kubernetes klaszteren fut. A Home Agent szolgáltatást futtató virtuális gépnek és a Home Agent részét képező Packet Processor-(ok)nak szintén két IPv6 címe van. A 4000:4000:4000:4000::/64-es tartományba tartozó interfészükön keresztül a Load Balancer-rel, míg a 6000::/64 tartományba tartozó interfészükön keresztül a külvilággal kommunikálnak. A virtuális géphez a 4000:4000:4000:4000::1 és a 6000::1 címek vannak rendelve. A Home Agent szolgáltatáshoz tartozó komponensek közül, csak a Packet Processor rendelkezik a felsorolt tartományokba eső IPv6 címekkel, hiszen csak a hozzá tartozó pod(ok)-on keresztül van összeköttetésben a külvilággal és a Load Balancer-el. A Packet Processor és a többi komponens egymás közti kommunikációjáról a Kubernetes belső hálózata gondoskodik. Az ábrán ezért csak a Packet Processor van feltüntetve, mint a megoldás szempontjából lényeges hálózati elem. A Packet Processor pod(ok) IP címei minden egyes instanciálás során dinamikusan változnak, mivel a Packet Processor Kubernetes Deployment-jéhez rendelt Multus Container Network Interface (CNI) mindig egy másik szabad IP címet oszt ki a Packet Processor pod-(oknak) a meghatározott tartományokból. Az ábrán feltüntetett IP címek a konkrét IP tartományok egy-egy önkényesen választott címei. Ezek a 4000:4000:4000:4000::101 és 6000::101. A topológián felfelé haladva a következő elem egy a virtuális géphez csatlakoztatott router. Ez a router mindig a Kubernetes klasztert futtató virtuális géppel összeköttetésben lévő első router-t szemlélteti (Border Router). A Border Router össze tudja rendelni a kívülről érkező csomagokat a megfelelő Packet Processor pod-dal és képes a csomagokat közvetlenül az illetékes pod felé továbbítani. Ezt az általa használt routing protokoll miatt tudja megtenni. A Border Router IP címei a 6000::2 és a 3000::2. A topológiában legfelül a Corresponding Node helyezkedik el. A Corresponding Node a 3000::/64-es IP tartományban foglal helyet, IPv6 címe a 3000::3. Az ábrán az is látszik, hogy a Mobile Node és a Home Agent Packet Processor pod között egy IPv6/IPv6 tunnel került felhúzásra a feltételezett Binding Update-et követően, amely az idegen hálózat elfedésével biztosítja a MIPv6 szabvány szerinti állandó összeköttetést a Mobile Node otthoni címe (HoA) és a Home Agent között. A tunnel két végpontja a Mobile Node otthoni címe és egy vele megegyező tartományból választott IPv6 cím. A tunnel Home Agent Packet Processor felőli oldalán található IPv6 címe az 5000::11.

Hálózati topológia megváltozása Failover esetén

A 5.8 ábra mutatja, hogy a javasolt architektúrában hogyan változik a hálózati topológia Failover esetén. A Failover az a szcenárió, amely során a felhőalapú hálózati szolgáltatás meghibásodik és az orkesztrációs platform a helyére új példányt indít. A Failover történthet azonos felhőre, de történhet egy, a korábbitól eltérő felhőre is. A Failover lényege, hogy a Mobile Node egy rövid szolgáltatáskiesésen túl, semmit se tapasztoljon a meghibásodásból, a Home Agent elérése a Failover után zavartalan legyen. Ilyenkor a Mobile Node-nak nem kell új Binding Update-et küldenie a Home Agent-hez való csatlakozáshoz. A Mobile Node Corresponding Node-al való kommunikációja során az IP címek szintjén semmi változást nem

tapasztal, hiszen a Load Balancer elrejti előle a Home Agent belsejében zajló folyamatokat. Az újonnan elinduló Home Agent Packet Processor Pod új IPv6 címet kap a Multus CNI-től. Ez a tapasztalatok alapján az eggyel nagyobb IP címet jelenti, így ebben az esetben a címe a 4000:4000:4000:4000::102 és a 6000::102. A Corresponding Node sem érez meg semmit a Failover-ből, mert az ő irányából is el van rejtve a szolgáltatás belső működése a Border Router által. A tunnel felhúzása a Failover után az újonnan létrejött Packet Processor-on automatikusan (a Failover részeként) a korábban használt paraméterek megtartása mellett történik meg.



5.8. ábra. A javasolt megoldásban a hálózati topológia változása Failover esetén

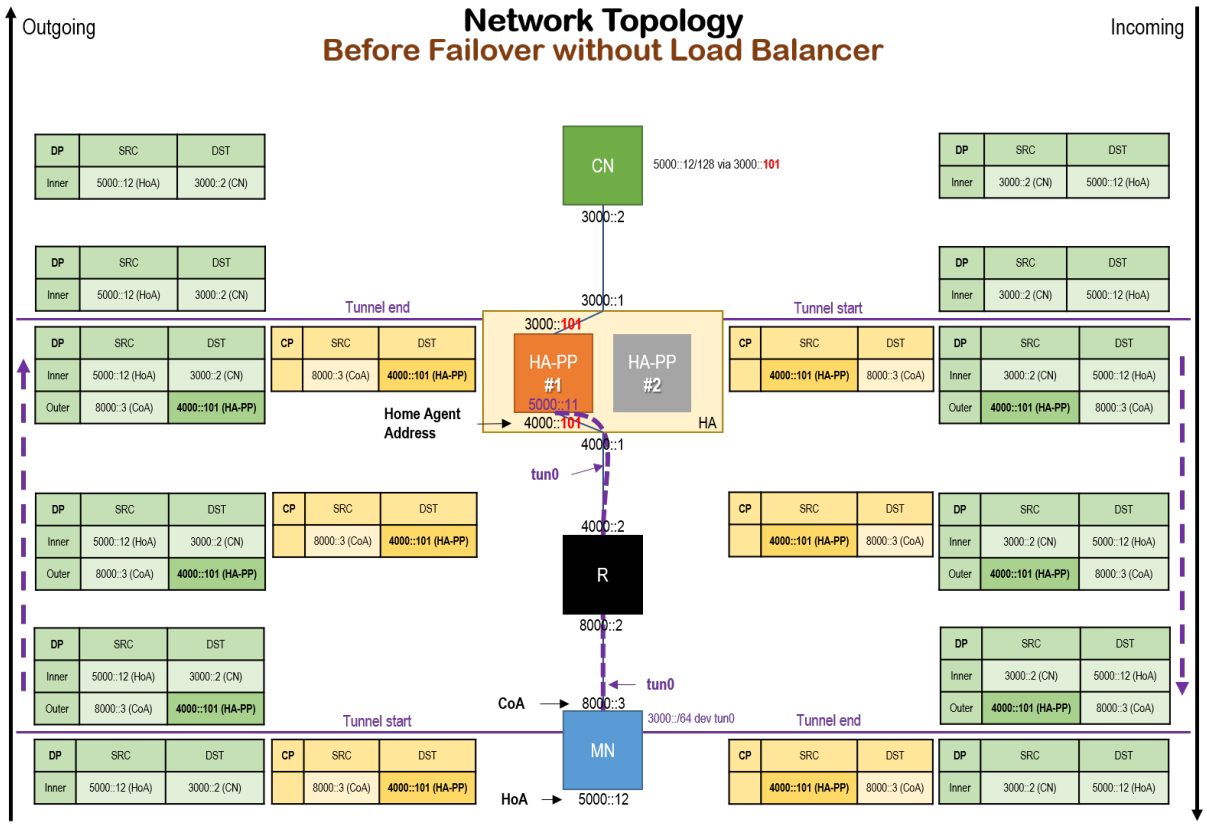
5.3. Megvalósított szolgáltatás-menedzsment megoldás

A felhőalapú mobilitáskezelés szolgáltatás-menedzsmentje céljából megvalósított megoldásom teljes egészében nem egyezik meg az általam készített kimerítő javaslattal. Ebben a fejezetben a javasolt és az implementált architektúra eltéréseit írom le. A szolgáltatás biztosításához szükséges zárt láncú szabályozókört az ONAP által nyújtott összes lehetőség kihasználásával valósítottam meg. A VES üzenetek implementálásával és a policy létrehozásával egy egyedi megoldást készítettem a Packet Processor hálózati szolgáltatáshoz. A felhőalapú mobilitáskezeléshez kidolgozott szolgáltatás-menedzsment javaslatomban a probe-okat, az életciklus eseményeket és a Horizontal Pod Autoscaler-t javasoltam a Kubernetes technológiák közül. A Horizontal Pod Autoscaler-t végül nem alkalmaztam, így ezt továbbfejlesztési lehetőségként

viszem tovább kutatásaim során. A megvalósított hálózati architektúrában is vannak eltérések a tervekhez képest. Az architektúra a Load Balancer kihagyásával kevésbé komplex, ugyanakkor a Failover mérésére ugyanolyan mértékben alkalmas maradt. A mérések során nem alkalmaztam duplikátumot a Packet Processor komponensből, mert a tesztelés során elsősorban azt szerettem volna bemutatni, hogy a teljes rendszer felépítése mennyi időt vesz igénybe az instanciálástól kezdve az új csatorna kiépítéséig. A megvalósított hálózati topológiát alábbiakban mutatom be.

Az implementált hálózati topológia

A tesztkörnyezet hálózati topológiáját a 5.9 ábra mutatja. A Mobile Node egy saját hálózaton helyezkedik el, amely IP tartománya a 8000:8000:8000:8000::/64-es. Ez szimbolizálja a mobil eszközt, amikor idegen hálózatban tartózkodik.



5.9. ábra. Az implementáció hálózati architektúrája és a User/Control Plane jellemzői

A Mobile Node IPv6-os idegen-hálózatbeli IP címe, a 8000:8000:8000:8000::3-as. A Mobile Node rendelkezik egy egyedi azonosítóval is, amely az otthoni címe. Ez a cím egy tetszőlegesen választott IPv6 cím, az 5000::12 az 5000::/124 tartományból. A mobil eszköz felett a Router helyezkedik el, amely összeköti a Mobile Node hálózatát a Home Agent hálózatával, amely a 4000:4000:4000:4000::/64-es IP tartományba esik. Ezáltal a Router két IPv6 címmel rendelkezik, melyek a következők: A Mobile Node felé eső interfészen a 8000:8000:8000:8000::2, a Home Agent felőli interfészen a 4000:4000:4000:4000::2 cím tartozik hozzá. A Home Agent

hálózati szolgáltatás egy felhőben található virtuális gépre telepített Kubernetes klaszteren fut, amelyhez tartozó címek a 4000:4000:4000:4000::3, illetve a 3000::3. A két IP címre azért van szükség, mert míg a 4000:4000:4000:4000::/64-es tartományba eső cím a Mobile Node-dal való kommunikációt teszi lehetővé, a 3000::/64-es tartományba eső cím a Corresponding Node-dal való kommunikációt biztosítja. A Home Agent szolgáltatáshoz tartozó komponensek közül, csak a Packet Processor rendelkezik a megfelelő tartományokba eső IPv6 címekkel, hiszen csak ez a pod van összeköttetésben a külvilággal és a többi komponenssel való kommunikációról a Kubernetes gondoskodik. Az ábrán ezért csak a Packet Processor van feltüntetve, mint a megoldás szempontjából releváns hálózati elem. A Packet Processor IP címei minden egyes instanciálás során dinamikusan változnak. Jelen esetben a feltüntetett IP címek a meghatározott IP tartományok egy-egy önkényesen választott címei. Ezek a 4000:4000:4000:4000::101 és 3000::101. A topológián tovább haladva a virtuális gép fölött, a tesztkörnyezet utolsó eleme, a Corresponding Node helyezkedik el. A Corresponding Node a 3000::/64-es IP tartományban foglal helyet, IPv6 címe a 3000::2. A Corresponding Node az implementált megoldásban a Home Agent-tel közvetlen összeköttetésben van. Az ábrán az is látszik, hogy a Mobile Node és a Home Agent Packet Processor pod között egy IPv6/IPv6 tunnel került felhúzásra a feltételezett Binding Update-et követően, amely az idegen hálózat transzparenssé tételével biztosítja a MIPv6 szabvány szerinti állandó összeköttetést a Mobile Node otthoni címe (HoA) és a Home Agent között. A tunnel két végpontja a Mobile Node otthoni címe és egy vele megegyező tartományból választott IPv6 cím. A tunnel Home Agent Packet Processor felőli oldalán található IPv6 címe az 5000::11. A Failover hatásásra bekövetkező topológiát érintő változásokat a 6. fejezetben mutatom be.

5.4. Meglévő Home Agent Packet Processor kiegészítése

A Home Agent Packet Processor egy hálózati szolgáltatás komponense, amely az ONAP rendszerben fut felhő alapon. Ahhoz, hogy ez a szolgáltatás automatikusan konfigurálható és folyamatosan felügyelhető legyen, kiegészítettem a hálózati szolgáltatást tartalmazó CBA csomagot és a Packet Processor kódján is módosításokat hajtottam végre. Elsőként a Packet Processor-t futtató pod-hoz életciklus követő eseményeket rendeltem Kubernetes Probe-ok formájában. Ezután megvalósítottam a gyűjtött életciklus adatok szinkronizációját a Home Agent Backend-del és a Home Agent Database-el. Erre azért van szükség, hogy a Home Agent folyamatosan monitorozhassa a Home Agent szolgáltatáshoz tartozó Packet Processor-okat és azok állapotát, amelyet az ONAP felé VES üzenetek által továbbít.

5.4.1. Kommunikáció megvalósítása a Home Agent Backend-el

A Home Agent Backend egy webszervert futtat, amely REST interfészen keresztül elérhető a kliensek számára. Kliensnek tekintem ebben az esetben a Home Agent Packet Processor-t, amely a Backend-el való kommunikációhoz hálózati hívások lebonyolítására képes. A Backend-

el való kapcsolatra szükség van mind a Binding Update validálása, mind az analitikus adatok gyűjtése során. A Packet Processor-t futtató main.c fájl mellé elkészítettem a requests.c fájlt és a requests.h fájlt. A requests.c fájlban a requests.h fájlban deklarált függvények definíciói találhatóak. Ilyenek a GET és POST hívások küldése[63]. A hálózati hívások során küldött és fogadott adatcsomagok tartalma JSON alapú, így egy olyan csomag telepítésére is szükségem volt, amely a JSON fájlok kezeléséért felel. Erre a célra a json-c [64] csomagot használtam.

5.4.2. Binding Update validálása

A beérkező Binding Update-ek validálásához a Home Agent Packet Processornak ismernie kell, hogy mely Mobile Node-ok milyen CoA címekkel szerepelnek a Binding Cache-ben. A Binding Cache tartalmát a Home Agent Packet Processor a Backend-en keresztül éri el. A Packet Processor egy Binding Update érkezésekor feldolgozza a kapott csomag tartalmát, majd különféle validációs lépéseket hajt végre a Binding Update érvényességének ellenőrzése során. Első lépésben a Home Agent Backend megfelelő végpontjára üzenetet küld, amelyben elkéri az adatbázisból azt a bejegyzést, amely HoA címe megegyezik a Binding Update-ben kapottal. A HoA címet a véponthoz tartozó útvonalba ágyazva küldi el. Amennyiben van az adatbázisban hozzá tartozó bejegyzés, válaszul egy JSON fájl érkezik hozzá. Abban az esetben, ha még korábban nem tartozott bejegyzés az adott HoA címhez, a Packet Processor kérvényezi a megfelelő bejegyzés létrehozását az adatbázisban. Ha korábban volt már a HoA-hoz tartozó bejegyzés, akkor a sorozatszám ellenőrzését követően vagy frissítésre kerül a Binding Cache bejegyzés, szintén a Backend közrejátszásával vagy elutasításra kerül a Binding Update. A Packet Processor minden esetben a megfelelő Binding Acknowledgement üzenettel tájékoztatja a Mobile Node-ot kérése sikerességéről.

5.4.3. Életciklus-menedzsment Kubernetes segítségével

Egy felhőalapú hálózati szolgáltatás állapotának folyamatos monitorozása alapvető jelentőségű, hiszen csak ilyen módon biztosítható a szolgáltatás állandó minősége. Erre számtalan lehetőséget biztosítanak a különféle technológiák, de a feladatra egyedi mechanizmusokat is alkalmazhatunk. A Home Agent Packet Processor esetében a Kubernetes által nyújtott megoldásokat használtam a diagnosztizációra. A futó Home Agent Packet Processor pod-ok állapotát probe-okkal és a pod életciklus eseményeire feliratkozott eseménykezelőkkel ellenőriztem.

Probe-ok

A Kubernetes lehetőséget nyújt a futó pod-ok állapotának periodikus ellenőrzésére probe-okon keresztül. A 2.5.2 fejezetben említett három féle probe közül a Home Agent Packet Processor-nál két félét alkalmaztam. A readiness probe állítja a Packet Processor pod-ot "Kész" állapotúra, amikor az készen áll a Binding Update-ek fogadására. A liveness probe működés

közben felügyeli a Packet Processor pod-ot, minden fél percben ellenőrzi, hogy a pod még mindig "él"-e. Mind a két esetben egy fájl meglétét ellenőrzöm az ExecAction típusú probe-ok segítségével. A fájlokat a Packet Processor indulása alatt hozza létre és futása során, indokolt esetben törölheti is őket. Ha a fájlok léteznek, a probe-ok kimenete sikeres lesz. A liveness probe esetén négyben limitáltam a sikertelen probe-ok számát, ezért két perc után, ha a Packet Processor nem elérhető, a Kubernetes automatikus törli a pod-ot és egy másik példányt hoz létre a helyére. Amikor a liveness probe sikeresen lefut, egy heartbeat üzenetet küld a Home Agent Backend-nek, amely ezt az információt analitikus adatok létrehozásásra alkalmazza.

Életciklus események

A 2.5.2 fejezetben leírt életciklus események közül a Packet Processor esetén a PreStop eseményhez hoztam létre ExecAction eseménykezelőt. A pod megszűnése előtt meghívja a PreStop eseménykezelőt, amely a Packet Processor-nál egy szkript futtatasát eredményezi. Ez a szkript üzenetet küld a Backend-nek, amelyben arra kéri őt, hogy az elérhető kliensek közül szedje ki a megszűnő Packet Processor-t. Erre az aktív kliensek nyilvántartásának naprakészen tartása miatt van szükség.

5.4.4. Adatgyűjtés

A Home Agent Packet Processor saját állapotáról és a hozzá beérkező Binding Update-ek számáról előre meghatározott időközönként küld üzenetet a Backend felé. A periodikus jelentéseket a Backend megfelelő végpontjaira küldi. A Packet Processor Docker konténer indulását követően végrehajtja dockerentrypoint.sh szkriptben meghatározott lépéseket. Ekkor kerül sor arra, hogy a Packet Processor beregisztrálja magát a Backend-nél, mint futó Packet Processor kliens. Ilyenkor a konténer az őt futtató pod nevét és egyedi azonosítóját küldi el JSON formátumban. Ezután a pod minden egyes sikeres liveness probe-ot követően heartbeat üzenetet küld a Backend-nek. Mielőtt a Packet Processor pod megszűnne, lefut az életciklus megváltozására feliratkozott PreStop hook eseménykezelő. Ebben a pod de-regisztrálja magát a Backend-nél. Ezekre az információkra a Backend-nek azért van szüksége, hogy számon tarthassa az aktuálisan futó klienseket és ezek állapotáról az ONAP-ot folyamatosan tájékoztathassa.

5.4.5. Zárt láncú szabályozókör megvalósítása

A Packet Processor zárt láncú szabályozókörét az ONAP Policy segítségével valósítottam meg. Egy policy megalkotásához az interneten számos példa elérhető el, amelyeket felhasználtam a feladat megoldásához [65]. Az ONAP-ban az egyes policy-khoz TOSCA [66] nyelvű leíró fájlok tartoznak. A policy-k eseménysorozatokat írnak le, amelyeket végre kell hajtani meghatározott események bekövetkezése után. A Packet Processor-nál a Fault event után lezajló eseménysorozatot "failover-workflow"-nak neveztem el, amelynek egyes lépéseivel kiegészítettem a szolgáltatást leíró CBA csomagot is, amelyet utána feltöltöttem az ONAP-ba.

6. fejezet

Tesztelés

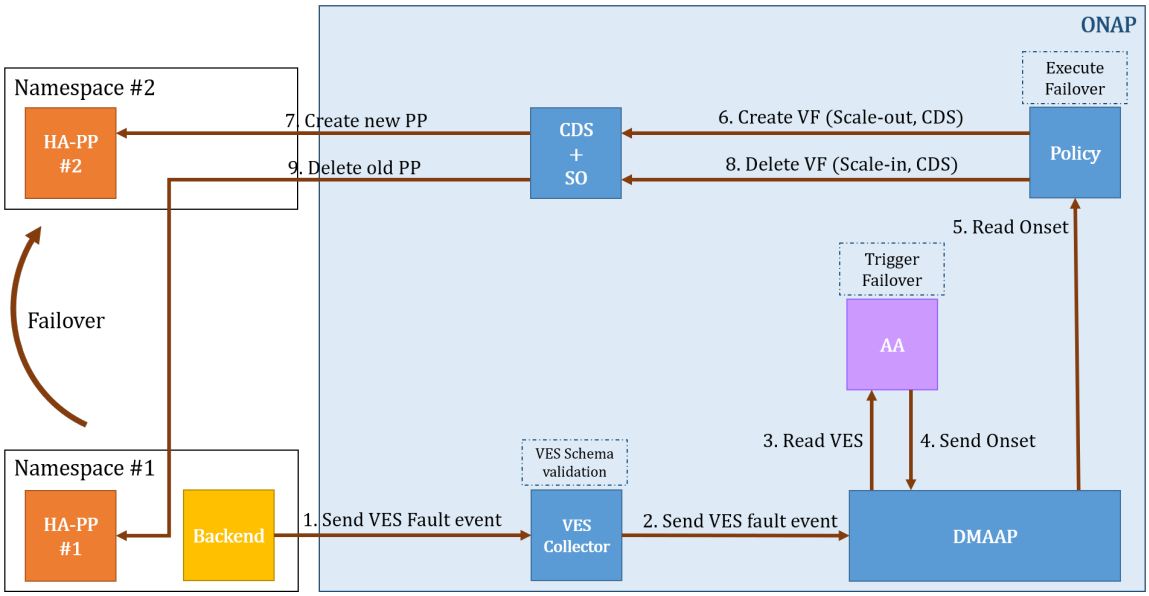
Ebben a fejezetben mutatom be a valós hálózati környezeten végzett teszteket és azok eredményeit. A tesztelés során a 5.3 fejezetben bemutatott hálózati topológiát használtam. A tesztek futtatásakor azt vizsgáltam meg, hogy egy olyan hiba bekövetkezését követően, amely a CN-MIPv6 Home Agent szolgáltatás kiesését eredményezi, mennyi időt vesz igénybe a korábban felkonfigurált ONAP rendszerben a szolgáltatás egy másik példányának működőképes állapotra juttatása. Ezt a szcenáriót Failover-nek neveztem el.

6.1. Failover szcenárió

Ahogy a fejezet beveztésében is említettem a Failover szcenárió tesztelése annak mérését jelenti, hogy CN-MIPv6 Home Agent felhő alapú hálózati szolgáltatás egy példányának meghibásodását követően az ONAP keretrendszer mennyi idő alatt képes reagálni az eseményre és ezután mennyi időbe telik, mire a sérült példány feladatait egy újonnan létrehozott példány teljes mértékben el tudja látni. A Failover tesztekben csak a Home Agent Packet Processor szerepel, mint meghibásodásra képes elem. A döntés oka, hogy felhasználói szempontokból ennek a hiánya a legmérvadóbb, és komplexitását tekintve ez az a komponens, amely a legkönnyebben valamilyen hibára futhat. Ipari környezetben szükség lehet felhők közti failover-re, azonban a teszteknél egyik névtérből másik névtérbe történő failover-t valósítottam meg, ugyanis a rendelkezésre álló infrastruktúra ezt tette lehetővé. A Failover előfeltétele, hogy a Home Agent Packet Processor CBA csomagja tartalmazza a failover végrehajtásának módját, valamint hogy az ONAP-ban létezzen az a policy, amely a failover-t kezdeményezi. A policy létrehozását és az alap CBA csomag kiegészítését a 5.4.5 fejezetben írtam le részletesen.

Az általam tervezett Failover szcenárió egy kilenc lépéses folyamat. Folyamatábráját a 6.1 ábra mutatja. A teszt elején az első névtérben már létezik egy működő Home Agent Packet Processor és a hozzá csatlakozó Mobile Node-dal már megtörtént a sikeres Binding Update/Binding Acknowledgement üzenetváltás. A Mobile Node tehát a külvilág felé a Packet Processor-on keresztül, a már felépített tunnel-en kommunikál. A Corresponding Node elérhetőségét a Mobile Node-ról egy ping üzenetváltással ellenőrzöm. A Failover első lépéseként a Home Agent

Backend erre alkalmas végpontjának meghívásával egy ONAP által értelmezhető VES Fault eseményt küldök. A Mobile Node számára ekkor szűnik meg a kommunikáció a Corresponding Node-al. A VES üzenet először a VES Collectorhoz érkezik meg. A VES Collector az ONAP része, ahogy ezt az ábra is jelöli. Feladata, hogy a beérkező üzeneteket validálja egy meghatározott séma alapján és az érvényes üzeneteket továbbítsa a DMaaP felé. Ez a Failover második lépése. A DMAAP az ONAP-ban az üzenetek terjesztését végző busz. Az ide érkező VES Fault eseményt az általam készített Analitikus Alkalmazás olvassa ki. Az Analitikus Alkalmazás egy Kubernetes Pod, ami az ONAP-pal azonos Kubernetes klaszteren fut. Ez folyamatosan monitorozza DMAAP-ra érkező VES üzeneteket és abban a pillanatban, amikor Fault esemény érkezik a Packet Processorhoz tartozó VNF-től, Onset üzenetet küld a DMaaP felé.

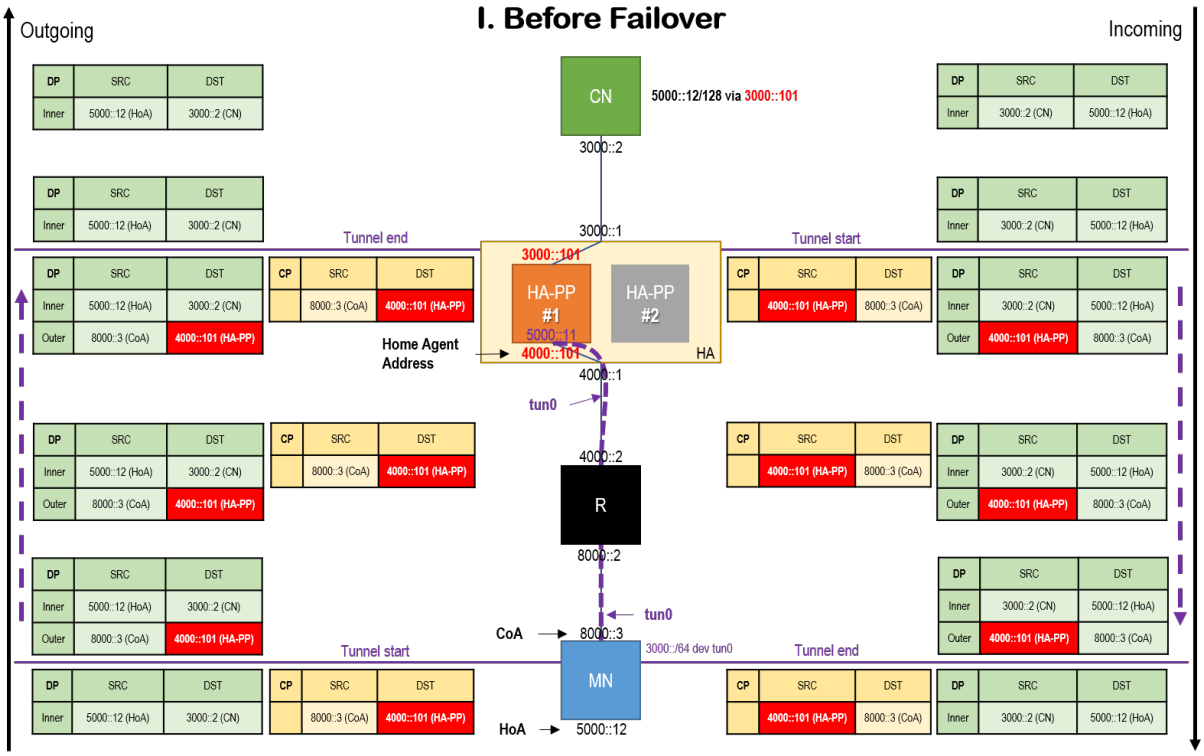


6.1. ábra. Failover lépései

Az Onset üzenet alkalmas arra, hogy valamilyen policy érvénybe lépését kezdeményezze. Ez az üzenet tartalmazza a forrás VNF-et és a végrehajtandó workflow nevét, ami ez esetben a "failover-workflow". A Failover szcenárióban az Onset üzenet küldése a negyedik lépés. Következő lépésként az ONAP Policy komponense kiolvassa a DMaaP-ról a beküldött Onset üzenetet. A Policy ezután kezdeményezi a "failover-workflow" végrehajtását a Packet Processor-on. A "failover-workflow" parancsok egy szekvenciális sorrendjét adja meg, ahol a soron következő parancs csak abban az esetben fut le, ha az előző sikeresen véget ért. A "failover-workflow" először kifelé történő skálázást hajt végre, amely során a Packet Processor-hoz egy új VF Module-t rendel, amelyet a második névtérbe telepít. A telepítésért az ONAP két komponense, a CDS és az SO felel. A CDS futtatja a failover egyes lépéseihez tartozó szkripteket, amelyek az új VF Module instanciáláshoz/törléséhez az SO-t hívják. A sikeres kifelé skálázást követően néhány konfigurációs lépés következik. Ekkor küldi el a Mobile Node a Binding Update-et az újonnan létrehozott Packet Processor-nak és ekkor kerül felhúzásra az új tunnel is, valamint a Corresponding Node-on egy új útvonal kialakítása történik meg az új Packet Processor-on keresztül. A "failover-workflow" második lépése a befelé történő skálázás végrehajtása. Ekkor az

első névtérben futó hibás VF Module törlésre kerül az SO által. Miután ez a lépés is sikeresen megtörtént a Failover végetér. A Failover verifikációját újbóli pingeléssel teszem meg.

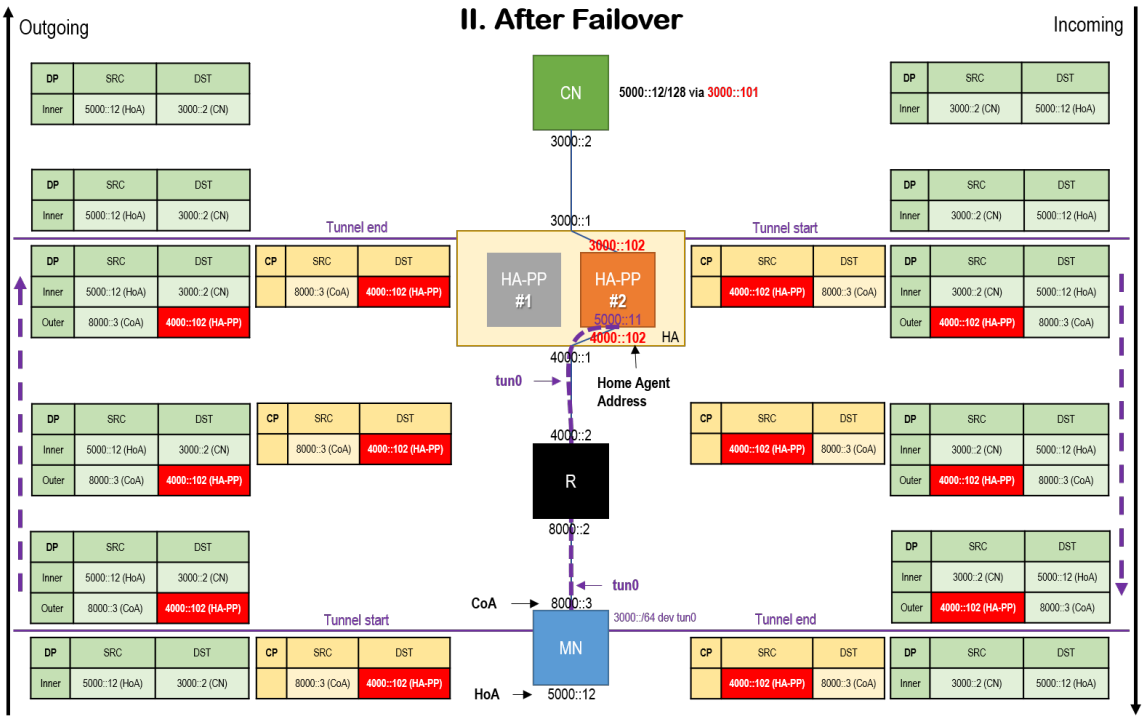
A hálózati topológia alakulását a Failover szcenárió futtatása előtt a 6.2 ábra mutatja. Ez az ábra megegyezik a 5.3 fejezetetben bemutatott hálózati topológiával, ugyanakkor jobban kihangsúlyozza azokat a paramétereket, amelyek a futtatás során megváltoznak. A Failover előfeltétele, hogy a Mobile Node és a Home Agent között megtörténjen a tunnel felhúzása, valamint hogy a Mobile Node és a Corresponding Node közötti kommunikáció sikeres legyen. Az ábrát most az ehhez szükséges üzenetváltásokra fókuszálva mutatom be. Az ábrán a valódi címeket hosszúságukból kifolyólag rövidített verzióban tüntettem fel, így például a 4000:4000:4000:4000::1 cím megfelelője a 4000::1. Az ábra két oldalán található táblázatok azt mutatják meg, hogy az egyes üzenetváltásokban mely eszközök érintetettek, vagyis, hogy ki a küldött csomagok feladója és címzettje. Az első üzenetcsere a vezérlési síkot (Control Plane) érinti, hiszen a Binding Update és Binding Acknowledgement üzenetek küldése az ő felelősségkörébe tartozik. Ezeket az üzenetek az ábrán a sárga táblázatok mutatják. A Mobile Node-tól balra a Mobile Node által a hálózati topológiában felfelé küldött csomagok részletei, míg tőle jobbra a Mobile Node-hoz fentebbi csomópontoktól érkező csomagok részletei olvashatók.



6.2. ábra. Üzenetváltások alakulása a Control/Data Plane-ben Failover előtt

A Mobile Node által küldött Control Plane-beli üzenet a Binding Update, amelyben a Home Agent-et tájékoztatja CoA címenének megváltozásáról. A Binding Acknowledgement üzenetet a Mobile Node a CoA címéről küldi a Home Agent Packet Processor-nak, hiszen ő fogja ezt az üzenetet feldolgozni. Implementációmban a Mobile Node tudja a Home Agent Packet Processor IP címét. A forrás címe tehát a 8000:8000:8000:8000::3 és a cél cím a

4000:4000:4000:4000::101. A Binding Update-et a Mobile Node még nem a tunnel-ben küldi, hiszen azt csak a Binding Acknowledgement megérkezését követően építi ki a Packet Processor felé, ezért a csomag eredeti formájában érkezik meg a Packet Processor-hoz. A Packet Processor a Binding Acknowledgement üzenetet a Binding Update-hez hasonlóan a saját 4000:4000:4000:4000::101-es címéről a Mobile Node 8000:8000:8000:8000::3-as CoA címére küldi. Ez a Control Plane-hez tartozó második üzenet, amelynek útját a topológiában a jobb oldalon található sárga táblázatok jelölnek. A tunnel felhúzására csak ezt követően kerül sor a Mobile Node és a Packet Processor között. A Mobile Node és a Corresponding Node közötti kommunikáció ellenőrzésére pingelést használtam. Ez az üzenetváltás már a Data Plane felelőssége, amelyet a zöld táblázatok mutatnak a Data Plane-nél látott, azzal ekvivalens logika alapján. Az Echo Request üzenetet a Mobile Node az 5000::12 otthoni címéről (HoA) a Corresponding Node 3000::2 címére küldi, azonban erre a tunnel miatt egyből egy külső fejléc kerül, amely elfedi ezeket a címeket. Helyette a csomag forrása a Mobile Node CoA címe és célja a Packet Processor lesz. Ezt a külső fejlécet a Packet Processor fogja leszedni a csomagról és a Corresponding Node felé már eredeti formájában kerül továbbításra. A Corresponding Node az Echo Reply üzenetet a Mobile Node HoA címére küldi a hozzá kapcsolódó Packet Processor-on keresztül. Azt, hogy melyik Mobile Node-nak melyik Packet Processor-on keresztül kell küldeni a csomagot, a routing táblájában eltárolt megfelelő útvonal határozza meg. Amikor megérkezik az Echo Reply üzenet a Packet Processor-hoz, az a tunnel miatt egy külső fejlécet tesz a csomagra, amely elfedi az eredeti forrás és cél címeket, helyette forrásként a Packet Processor-t célként a Mobile Node CoA címét tünteti fel. Az eredeti küldőt a Mobile Node a fejléc eltávolítása után ismeri meg.

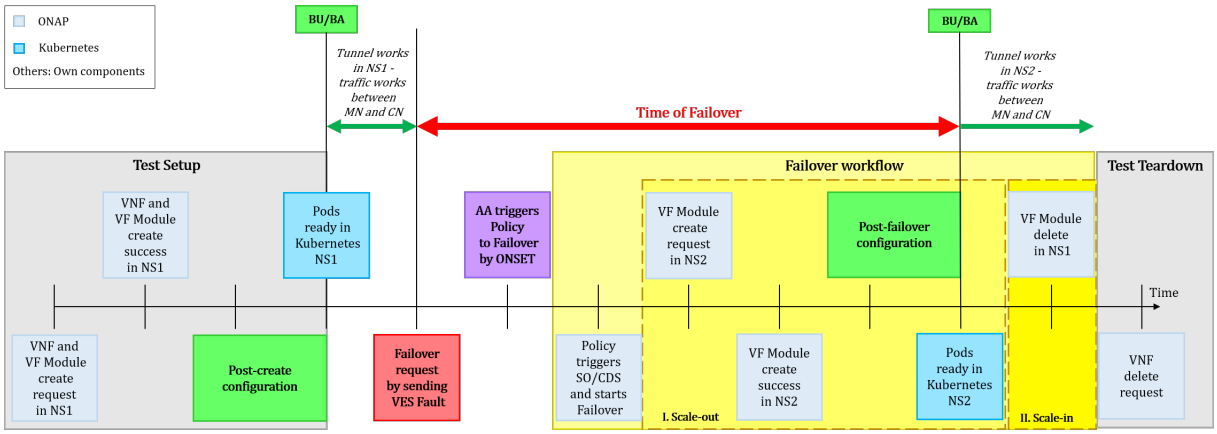


6.3. ábra. Üzenetváltások alakulása a Control/Data Plane-ben Failover után

Miután az infrastruktúra alkalmassá vált a teszteléshez, a Failover szcenárió indítása megkezdődhet. A Failover célja az, hogy miután egy ismeretlen mighibásodás hatására a Mobile Node elvesztette a kommunikációt a távoli eszközzel, az újonnan létrehozott Packet Processoron keresztül új, működőképes kapcsolat alakulhasson ki közte és a Corresponding Node között. Ez azonban a hálózati topológiában bizonyos paraméterek megváltozását eredményezi. A Failover utáni hálózati topológiát a 6.3 ábra mutatja. Az ábrán pirossal jelöltem azokat a mezőket, ahol változás történt a kiindulási állapothoz képest. A szcenárióból adódik, hogy ezek a változások csak az infrastruktúra azon részét érintik, ahol a Packet Processor előfordul, hiszen a Failover során csak az ő IP címe változott meg, mivel a Multus CNI az új példányhoz egy új IP címet rendelt, miközben más címek változatlanok maradtak. Ahhoz, hogy a tunnel újra felépülhessen a Mobile Node és a Packet Processor között, a Mobile Node-nak új Binding Update-et kell küldenie. Erről a "failover-workflow" konfigurációs lépése gondoskodik. Ennek hatására a Failover után a Mobile Node az új tunnel-en keresztül újra elérhetővé válik a külvilág felé, így a Corresponding Node-al való kommunikációja újra biztosított lesz. A következő fejezetben bemutatom a Failover szcenárió méréséhez kialakított automatizált tesztrendszert.

6.2. Megoldás tesztelése valós környezetben

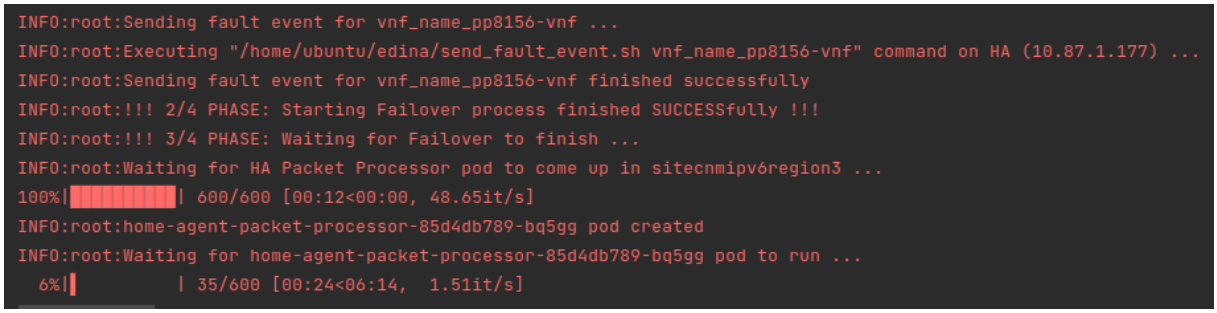
A Failover szcenárió méréséhez egy Python szkriptet írtam. Ez a szkript manuális beavatkozás igénye nélkül futtatja le az egész tesztesetet. A szkript kiterjed az infrastruktúra felállításra és a Failover-t követő tesztkörnyezet lebontására is. Ezáltal az egyes tesztek futtatása egymástól teljesen elszeparalható és egy korábbi tesztnek sincs befolyásoló hatása egy későbbi tesztre, így a mérési eredmények egymástól függetlenek és a mérésekből levont következtetések valós képet adnak a rendszer teljesítményéről. Az automatizált tesztrendszer három elkülönülő fázisra bontható, amelyek egymást szekvenciális sorrendben követik. Ezt és az egyes fázisok részleteit a 6.4 ábra mutatja.



6.4. ábra. Automatizált Failover tesztrendszer

Az első fázisban történik meg a hálózati szolgáltatás egy példányának instanciálása és a Failover megkezdéséhez szükséges előfeltételek megteremtése. Ennek során létrejön a hálózati

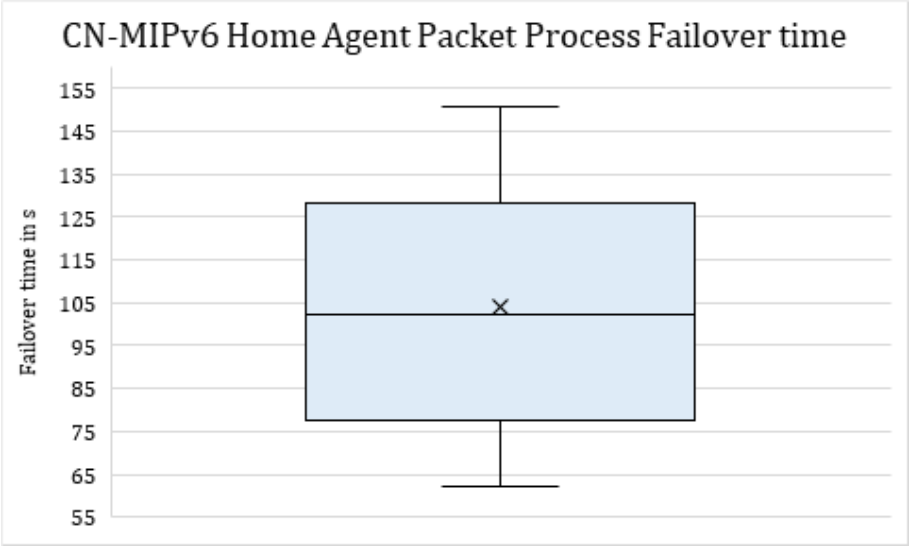
funkció (VNF) és a hozzá tatozó VF Module, amely a hálózati szolgáltatás egy konkrét példányát jelenti. Miután a Packet Processor elérhetővé válik az első névtérben, néhány konfigurációs lépés következik. Ekkor kerül sor a Binding Update elküldésére, a tunnel felépítésére és a Corresponding Node felkonfigurálására. Mivel minden egyes node -beleértve a Home Agent-t ismásik virtuális gépen fut, a beállítások elvégzéséhez szükség van az egyes virtuális gépekhez SSH-n keresztül történő csatlakozásra. A Mobile Node és a Corresponding Node közötti kapcsolat ellenőrzésére használt Echo Request/Echo Reply üzenetek ezután kerülnek elküldésre.
Miután megtörtént a felépült rendszer rendelkezésre állásának validálása, elkezdődik a Failover szcenárió futtatása. Ez a szkript második fázisa. Előtte azonban még a szkript letárolja az aktuális időbélyeget, hogy a Failover időtartamát mérni lehessen. A Failover kiváltásához felküldöm a VNF adatait a Home Agent Backend-nek, amely a megadott információk alapján előállítja a VES Fault eseményt és azt elküldi az ONAP VES Collector komponensének. Az Analitikus Alkalmazás ez idő alatt folyamatosan monitorozza a DMaaP-ra érkező üzeneteket és amikor beérkezik hozzá a Fault esemény, ellőállítja a megfelelő Onset üzenetet a Failover megkezdéséről. Az ONAP Policy detektálja az Onset üzenet megérkezését és elkezdi a Failover végrehajtását. Ehhez az ONAP két komponensét, a CDS-t és az SO-t hívja. A Failover a "failover-workflow" policy és a Home Agent Packet Processor CBA csomagjában meghatározott lépéseket követi. A szkript megvárja, amíg a "failover-workflow" első lépése, vagyis a kifelé történő skálázás -a konfigurációs lépéseket is beleértve- sikeresen lezajlik. Ekkor egy pingeléssel ellenőrzi, hogy kiépült-e a kapcsolat a Mobile Node és a Corresponding Node között. Ha a ping sikeres volt, akkor egy újabb időbélyegben eltárolja a Failover "végét". A Failover-nek hivatalosan ilyenkor még nincs vége, hiszen ekkor még hátra van a meghibásodott Packet Processor törlése.
A tesztrendszer az utolsó fázisban kitörli az összes Home Agent Packet Processor szolgáltatáshoz tartozó erőforrást és ezzel visszaállítja a rendszert a kiindulási állapotra. Miután ez megtörtént a két időbélyegből kiszámításra kerül a Failover időtartama és ezt az értéket a szkript eltárolja egy szöveges fájlban. A mérések eredményeinek összevetéséhez ezt a fájlt haszáltam fel. A szkript a futása alatt az aktuálisan végrehajtandó folyamatokat naplózza. Ez látható a 6.5 ábrán.



6.5. ábra. Failover Python szkript futása

6.3. Mérések és eredményeik

A teszt szkript számottevő futtatása utána az eredmények boxplot diagrammon a 6.6 ábrán látható. Az ábra alapján megállapítható, hogy az esetek 50%-ban a $[75, 125]$ zárt intervallumon belüli futtatási időt mértünk, amely másodpercben értendő. Ez az az időintervallum, amíg a felhasználó szolgáltatás-kiesést észlelt. Az estek 25%-ban sikerült 60 másodperc körüli értékeket mérni, amely azért lehetett, mert ekkor az ONAP rendszere kevés konkurens műveletvégzést hajtott végre, ezáltal az alkalmazás nagyobb erőforrás szeletet kapott. Rosszabb szituációkban az esetek 25%-ban ennek ellenkezője történt. Mivel az ONAP túlterhelt volt, ez az érték felugrott 150-155 másodpercre, mely érték már jelentősen magasabb, mint a mért adatok mediánja.



6.6. ábra. Failover mérési eredmények diagramja

Az adatok mediánja, minimuma, maximuma, átlaga és standard szórása pontosan látható a 6.1 táblázatban. A sztandard szórás alapján elmondható, hogy a rendszer újraindítása és felépítése stabil, megbízható módon működik, hiszen nincs az átlagértéktől jelentős mértékű kiugrás.

Min	Avg	Median	Max	Stdev
62,34	103,93	102,17	150,67	27,3493

6.1. táblázat. Mérési eredmények statisztikája

A 6.1 táblázatban látható statisztikák alapján kijelethető, hogy a rendszer jelen formátumában még nem adaptálható egy az egyben ipari körülmények között, hiszen egy ekkora mértékű szolgáltatás-kiesés felhasználói elégedetlenséghez vezetne. Továbbá a felhasználói elégedetlenségen túl, a hálózat sem bírná biztosítani a Service Level Agreement-et (SLA), amit a hálózat felé támasztunk. A rendszer, további optimalizálását és megbízhatóvá tételét követően már alkalmazható lenne ipari körülmények között is, hiszen az automatizált helyreállítás és vizsgálat a manuális feladatvégzéshez képest már most is jelentősen effektívebb tud lenni.

7. fejezet

Továbbfejlesztési lehetőségek

Ebben a fejezetben a dolgozatben leírt megvalósítás továbbfejlesztési lehetőségeit mutatom be. Ezek olyan lehetőségek, melyeknek köszönhetően gyorsabb és megbízhatóbb megoldást kapnánk. A későbbiek során a fejezetben leírtaknak megfelelően fogom megvizsgálni a különböző lehetőségeket, illetve megvalósítani azokat valós felhőinfrastuktúrán. Mivel ezek a továbbfejlesztési lehetőségek még csak elvi síkon kerültek feltérképezésre, addig nem validálhatók a velük kapcsolatban kijelentett feltételezések, mert ezek alátámasztásához valós mérési eredmények és komperációs statisztikák szükségesek.

Ahhoz, hogy a lehető legjobban megállapítható legyen, hogy melyik állapotában tölti a rendszer az idő legnagyobb hányadát indokolatlan várakozással egy failover esetén történő újraindítás során, új mérési pontokat kell a rendszerbe felvenni. Ezeknek az új mérési pontoknak a segítségével már alkalmas lenne a rendszer arra, hogy meghatározható legyen az egyes állapotokban eltöltött idő.

Miután sikerült ily módon megállapítani az egyes állapotokban eltöltött időt, meg kell vizsgálni, hogyan lehetne azt a legjobb mértékben kioptimalizálni az várakozással eltöltött időhányad csökkentése érdekében. Az optimalizáció nem csak kód szinten értendő, hanem az egyes komponensekre is vonatkozik. Érdemes lehet körüljárni, hogy az adott komponens, amely jelentős mértékű várakozást okoz rendszer szinten, helyettesíthető-e valamilyen alternatív, saját vagy létező eszközzel. A kód szintű optimalizáció esetén a kód refaktorálásán túl meg kell nézni a különböző adatszerkezeteken végzett műveletek, valamint a memória igényes ciklusok optimalizálását is. Ezen kívül fel kell térképzni az asszinkron műveleteket abból a célból, hogy egy több szállal rendelkező hardver eszköz miként tudná ezeket párhuzamosan futtatni.

A felsorolt optimalizációkon túl érdemes lehet a már meglévő technológiai stack-et bővíteni plusz komponensekkel. Ilyen lehet például a Horizontal Pod Autoscaler (HPA) bevezetése, melynek segítségével dinamikusan lehet skálázni a Kubernetes-ben futó rendszer erőforrásait. Az erőforrás skálázás ekkor a CPU és RAM kihasználtságokon alapul. Alkalmazásának hátulütője, hogy a HPA kizárólag egy klaszterre lát rá, ezért ilyen formájában valós környezetben nem adaptálható. Ugyanis ezek a rendszerek általában több klaszteren üzemelnek. Erre lehet megoldás, ha minden klaszterbe kitelepítésre kerül a HPA, viszont ilyenkor mindet külön-külön

kell karbantartani, ami többletmunkát generál az üzmeltetési részlegnek. Azonban, ha erre egy megfelelő centralizált felügyeleti alkalmazást lehet készíteni, akkor egyetlen helyről vezérelhetőek lennének a különböző klaszterekhez tartozó HPA-k, aminek köszönhetően már nem lenne akadálya a technológia alkalmazásának.

Továbbfejlesztési lehetőség a tervezés során megjelölt Load Balancer implementálása, amely a Packet Processor-ok belső szintű menedzsment feladatát látná el. Alkalmazásának köszönhetően egy biztonságosabb működést érhetnék el, hiszen így a belső rendszer működése a külvilág felé egy zárt "fekete doboz" maradna. A Load Balancer töltené be az egyes IP címek menedzsmentjét is, ezáltal dinamikusan továbbítaná az erőforrások felé a csomagokat úgy, hogy figyelembe venné azok terheltségi szintjét.

További lehetőségeket nyújt a szekudner Packet Processor bevezetése is, mely egy másodlagos Packet Processor, amire a Load Balancer által automatikus átirányítás történne abban az esetben, ha az elsődleges Packet Proceessor meghibásodik. Ebben a kompozícóban megtakarításra kerülne az az időhányad, melyet az új Packet Processor teljeskörű inicializálására allokálna a rendszer. A szekunder Packet Processor és a HPA által nyújtott dinamikus skálázás bevezetésével megoldható lenne a rendszer legmegbízhatóbb és legoptimálisabb működtetése, az aktuális igényeknek megfelelő hálózati erőforrások bevonásával.

Látok lehetőséget ezeken felül a Data Plane és Control Plane felelőségköreinek szeparációjában is. A rendszerben több erőforrást lehetne asszociálni a Data Plane-t megvalósító pod-ok számára, hiszen ezek a nagyobb adatcsomagokért felelős entitások, míg a jelentősen kisebb forgalmat bonyolító Control Plane-t megvalósító pod-okhoz kevesebb erőforrás is elegendő lenne az általuk képviselt funkcionalitás betöltésére.

A fejezetben megvizsgált továbbfejlesztési lehetőségeket a későbbi kutatásaim során tervezem részletesebben kielemezni, illetve ezekből implementált megoldást készíteni, hogy validálni tudjam az általam definiált elvárt működésüket. Mivel ez a kutatási terület még nagyon friss és kiaknázatlan, úgy gondolom érdemes ilyen irányban vizsgálódni, hogy a szolgáltatók egy fejlettebb és stabilabb mobilkommunikációs hálózatot építhessenek ki.

8. fejezet

Összegzés

A dolgozat készítése során körbejártam a felhőalapú mobilitáskezelés szolgáltatás-menedzsmentjének megvalósítását az ONAP orkesztrációs platformon. Ezzel az volt a célom, hogy bemutassam annak elméleti és megvalósítási lehetőségeit, és azt, hogy egy zárt láncú szabályozókörök által vezérelt megoldás, milyen módon tudná a telco operátorok számára automatizálni a jelenleg manuálisan végzett karbantartási munkákat. A dolgozatomban a cloud-native megoldás egy valós szolgáltatói szintű felhőinfrastruktúrában, létező implementációra támaszkodik. Az ezen dolgozat keretén belül bemutatott technológiák és eszközök felhasználásával egy olyan rendszert lehet kiépíteni, amelyben a megoldás működtetése nem igényli dedikált céleszközök beszerzését. Ezáltal az egyes funkciók nem függnek a kibocsájtó gyártó által adott fizikai eszközök hardver-karakterisztikájától.

A mérési eredmények alapján megállapítható, hogy az IP szintű mobilitáskezelés cloud-native alapokon történő megvalósítása igenis esszenciális a modern hálózatszervezési paradigmák szempontjából. Az általam javasolt megoldás előnyeire az 5G hálózatok és evolúciós utódjaik egyaránt támaszkodhatnak a felhasználók IP szintű mobilitáskezelésének skálázható megvalósításához. A mérési eredmények alapján levonható, hogy a rendszer további optimalizálása még szükséges annak érdekében, hogy azt élesben is, azaz, valós ipari alkalmazásban is fel lehessen használni. Viszont a már jelenleg meglévő eredmények alapján is elmondható, hogy a dolgozat által körüljárt téma kétségtelenül egy jó megközelítési irány lehet, hiszen annak ellenére, hogy az eredmények még nem ideálisak a rendszer valódi kitelepítésére, alkalmazása már jelen állapotában is olyan mértékű időbeli és teljesítménybeli javulást eredményezett, mely alapján feltétlenül érdemes e témában további kutatásokat folytatni. Ez a teljesítmény-javulás konkrét számokkal is mérhető, hiszen korábban egy hasonló jelentőségű hálózati komponens cseréje, mint amelyet a dolgozat keretén belül szimuláltam — feltétlezve, hogy az ekkor elkészült végtermék már nem tartalmaz emberi hibázásból eredő rendellenességet —, átlagosan 3-4 hónapnyi időtartamot emésztett fel egy átlagos vállalkozás munkaidejéből [61]. Természetesen ezen automatizáció nem alkalmazható közvetlenül az összes jelenleg használatos hálózati funkcióra. Alkalmazása előtt ugyanis alaposan meg kell vizsgálni, hogy az adott funkcionalitás megfelelően kiszervezhető-e cloud-native alapú megoldásba. A hatékonyságra tett korábbi

megállapításom érvényes egy-egy új szolgáltatás bevezetésére vonatkozóan is, mint például már meglévő hálózati rendszerek skálázása vagy egy teljesen új funkcionalitás alkalmazására. Ezek telepítésekor ugyanis jelentős mértékű ráfordított idő takarítható meg, hiszen a dolgozat keretén belül bemutatott Day0, Day1, Day2 lépések egy ilyen ONAP alapú megoldás segítségével percekben/másodpercekben mérhető időn belül végrehajthatók.

A dolgozatomban levont következtetések és eredmények segíthetik a hálózati elemzők munkáját és egyaránt hasznosak lehetnek a telco vállalkozások és a telekommunikációs témában tevékenykedő kutatók számára is. A dolgozat segítséget nyújthat jobb hálózattervezési architektúrák kidolgozásához azzal, hogy olyan architektúrális döntéseket befolyásolhat, amelyek alapjaiban változtathatják meg a jelenlegi hálózatépítés és tervezés folyamatát. A megoldás telco cégek számára is releváns lehet, hiszen a szolgáltatásaik egy ilyen, felhőalapú megoldás esetén dinamikusan skálázhatóvá válnak. Nem utolsó sorban a minőségi felhasználói élmény is garantálható lenne azáltal, hogy a felhasználót nem érné nagy mértékű szolgáltatáskiesés, ugyanis számára a lehető legrosszabb, melyet megtapasztalhat, a szolgáltatás kiesése. Ez az esemény pedig olyan mértékben meghatározója lehet későbbi döntéseinek, hogy szélsőséges esetben emiatt az adott szolgáltató akár még el is veszítheti az előfizetőinek sorából.

Jövőbeli terveim között szerepel a dolgozat keretén belül bemutatott megoldás további fejlesztésének, valamint optimalizálásának szándéka. Az ebben az irányban megvalósuló kutatások során, a rendszer optimalizálását követően, esedékessé válhat a különböző SLA-kban definiált metrikák kimérése. Ez azért fontos, mert ha ezeket a paramétereket a rendszer megfelelően tudja teljesíteni, akkor az akár még élesben, azaz ipari felhasználásban is alkalmazható lenne. Jövőbeli terveim között szerepel továbbá annak vizsgálata, hogy milyen módon lehet prioritáskezeléssel előtérbe helyezni egy-egy hibakezelést, ami szintén a teljesítmény javulását eredményezheti. A későbbiekben szükségesnek tartom elvégezni a dinamikus skálázás implementálását, valamint dedikált felhőinfrastruktúrában — valós körülményeket szimulálva — a rendszer ilyen módon bővített működésének ellenőrzését. Mindezektől azt várom, hogy rámutassanak a még lehetséges, kiaknázatlan felhasználási területekre. Vizsgálni fogom a rendszer lehető legnagyobb mértékben történő univerzálissá tételének lehetőségét azáltal, hogy feltérképezem azon műszaki megvalósulási formákat, melyekben a különböző alkalmazási igények esetén lehetséges a rendszer egyes alkotóelemeinek újrafelhasználása. Mindezekkel azt kívánom szemléltetni, hogy a már meglévő megoldásokon — a virtualizációs technológiáknak köszönhetően — egy-egy új funkció bevezetését milyen egyszerűen lehet adaptálni. A fenti előnyökön túlmenően ez a megoldás nagymértékben járul hozzá a vállalkozások fejlesztés esetén jelentkező kiadási költségek csökkentéséhez.

Irodalomjegyzék

[1] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck és R. Boutaba, „Network Function Virtualization: State-of-the-Art and Research Challenges”, *IEEE Communications Surveys Tutorials*, 18. évf., 1. sz., 236–262. old., 2016.

[2] *Ericsson Mobility Report*, https://www.ericsson.com/4a03c2/assets/local/mobility-report/documents/2021/june-2021-ericsson-mobility-report.pdf, Felkeresve: 2021-10-10, 2021.

[3] C. E. Perkins, J. Arkko és D. B. Johnson, *Mobility Support in IPv6*, RFC 3775, 2004. jún. cím: https://rfc-editor.org/rfc/rfc3775.txt.

[4] *Open Network Automation Platform*, https://www.onap.org/, Felkeresve: 2021-10-10, 2021.

[5] M. E. Bob Young, *Red Hat; Understanding virtualization*, https://www.redhat.com/en/topics/virtualization, Felkeresve: 2020-05-22, 1993.

[6] M. E. Bob Young, *Red Hat; What is virtualization?*, https://www.redhat.com/en/topics/virtualization/what-is-virtualization, Felkeresve: 2020-05-22, 1993.

[7] *Kubernetes Virtualization*, https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/, Felkeresve: 2020-05-17, 2014.

[8] *What is Kubernetes?*, https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/, Felkeresve: 2020-05-17, 2014.

[9] *SDN; Chapter 1: Introduction*, https://sdn.systemsapproach.org/intro.html, Felkeresve: 2020-12-10.

[10] Y. Rekhter, S. Hares és T. Li, *A Border Gateway Protocol 4 (BGP-4)*, RFC 4271, 2006. jan. cím: https://rfc-editor.org/rfc/rfc4271.txt.

[11] J. Moy, *OSPF Version 2*, RFC 2328, 1998. ápr. cím: https://rfc-editor.org/rfc/rfc2328.txt.

[12] G. S. Malkin, *RIP Version 2*, RFC 2453, 1998. nov. cím: https://rfc-editor.org/rfc/rfc2453.txt.

[13] M. Boucadair és C. Jacquenet, *Software-Defined Networking: A Perspective from within a Service Provider Environment*, RFC 7149, 2014. márc. cím: https://rfc-editor.org/rfc/rfc7149.txt.

[14] E. Haleplidis, K. Pentikousis, S. Denazis, J. H. Salim, D. Meyer és O. Koufopavlou, *Software-Defined Networking (SDN): Layers and Architecture Terminology*, RFC 7426, 2015. jan. cím: https://rfc-editor.org/rfc/rfc7426.txt.

[15] B. Vengainathan, A. Basil, M. Tassinari, V. Manral és S. Banks, *Benchmarking Methodology for Software-Defined Networking (SDN) Controller Performance*, RFC 8456, 2018. okt. cím: https://rfc-editor.org/rfc/rfc8456.txt.

[16] J. Kempf, E. Bellagamba, A. Kern, D. Jocha, A. Takacs és P. Sköldström, „Scalable fault management for OpenFlow", *2012 IEEE International Conference on Communications (ICC)*, 2012, 6606–6610. old.

[17] M. Kuundefinedniar, P. Perešıni, N. Vasić, M. Canini és D. Kostić, „Automatic Failure Recovery for Software-Defined Networks", *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13 sor., Hong Kong, China: Association for Computing Machinery, 2013, 159–160. old. cím: https://doi.org/10.1145/2491185.2491218.

[18] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman és R. Kompella, „Towards an Elastic Distributed SDN Controller", *SIGCOMM Comput. Commun. Rev.*, 43. évf., 4. sz., 7–12. old., 2013. aug. cím: https://doi.org/10.1145/2534169.2491193.

[19] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman és R. Kompella, „Towards an Elastic Distributed SDN Controller", *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13 sor., Hong Kong, China: Association for Computing Machinery, 2013, 7–12. old. cím: https://doi.org/10.1145/2491185.2491193.

[20] *P4 Open Source Programming Language*, https://p4.org/, Felkeresve: 2020-12-10, 2013.

[21] G. Antichi, T. Benson, N. Foster, F. M. V. Ramos és J. Sherry, „Programmable Network Data Planes (Dagstuhl Seminar 19141)", *Dagstuhl Reports*, 9. évf., 3. sz., G. Antichi, T. Benson, N. Foster, F. M. V. Ramos és J. Sherry, szerk., 178–201. old., 2019. cím: http://drops.dagstuhl.de/opus/volltexte/2019/11295.

[22] R. Bifulco és G. Retvari, „A Survey on the Programmable Data Plane: Abstractions, Architectures, and Open Problems", *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, 2018, 1–7. old.

[23] E. Kaljic, A. Maric, P. Njemcevic és M. Hadzialic, „A Survey on Data Plane Flexibility and Programmability in Software-Defined Networking", *CoRR*, abs/1903.04678 évf., 2019. cím: http://arxiv.org/abs/1903.04678.

[24] M. E. Bob Young, *Red Hat; What is NFV?*, https://www.redhat.com/en/topics/virtualization/what-is-nfv, Felkeresve: 2020-05-22, 1993.

[25] A. Morton, *Considerations for Benchmarking Virtual Network Functions and Their Infrastructure*, RFC 8172, 2017. júl. cím: https://rfc-editor.org/rfc/rfc8172.txt.

[26] C. J. Bernardos, A. Rahman, J.-C. Zúñiga, L. M. Contreras, P. A. Aranda és P. Lynch, *Network Virtualization Research Challenges*, RFC 8568, 2019. ápr. cím: https://rfc-editor.org/rfc/rfc8568.txt.

[27] *Datacomm; Network Function Virtualization*, https://www.datacomm.co.id/en/telco/nfv/, Felkeresve: 2020-12-10.

[28] *ETSI Network Functions Virtualisation (NFV)*, https://www.etsi.org/technologies/nfv, Felkeresve: 2020-12-10, 1988.

[29] *Network Functions Virtualisation (NFV); Protocols and Data Models; Specification of Patterns and Conventions for RESTful NFV-MANO APIs*, https://www.etsi.org/deliver/etsi_gs/NFV-SOL/001_099/015/01.02.01_60/gs_NFV-SOL015v010201p.pdf, Felkeresve: 2020-12-10, 2020.

[30] *Network Functions Virtualisation (NFV) Release 3; Protocols and Data Models; VNF Snapshot Package specification*, https://www.etsi.org/deliver/etsi_gs/NFV-SOL/001_099/010/03.03.01_60/gs_NFV-SOL010v030301p.pdf, Felkeresve: 2020-12-10, 2020.

[31] *Network Functions Virtualisation (NFV) Release 3; Testing; Specification of Networking Benchmarks and Measurement Methods for NFVI*, https://www.etsi.org/deliver/etsi_gs/NFV-TST/001_099/009/03.04.01_60/gs_NFV-TST009v030401p.pdf, Felkeresve: 2020-12-10, 2020.

[32] *NFV Release 4 Definition*, https://docbox.etsi.org/ISG/NFV/Open/Other/ReleaseDocumentation/NFV(20)000160_NFV_Release_4_Definition_v0_2_0.pdf, Felkeresve: 2020-12-10, 2020.

[33] M. Paolini, *Lowering TCO with NFV and SDN*, https://www.linkedin.com/pulse/lowering-tco-nfv-sdn-monica-paolini, Felkeresve: 2020-12-10, 2016. márc.

[34] R. T. Len Bosack Sandy Lerner, *CISCO; Introduction to Mobile IP*, https://www.cisco.com/c/en/us/td/docs/ios/solutions_docs/mobile_ip/mobil_ip.html, Felkeresve: 2020-05-17, 1984.

[35] *Docker overview*, https://docs.docker.com/get-started/overview, Felkeresve: 2020-05-17, 2013.

[36] *Kubernetes Pods*, https://kubernetes.io/docs/concepts/workloads/pods/pod/, Felkeresve: 2020-05-17, 2014.

[37] *Kubernetes Deployment*, https://kubernetes.io/docs/concepts/workloads/controllers/deployment/, Felkeresve: 2020-05-17, 2014.

[38] *Kubernetes Services*, https://kubernetes.io/docs/concepts/services-networking/service/, Felkeresve: 2020-05-17, 2014.

[39] *Kubernetes Nodes*, https://kubernetes.io/docs/concepts/architecture/nodes/, Felkeresve: 2020-05-17, 2014.

[40] *Kubernetes Controller*, https://kubernetes.io/docs/concepts/architecture/controller/, Felkeresve: 2020-05-17, 2014.

[41] *Kubernetes; Container Lifecycle Hooks*, https://kubernetes.io/docs/concepts/containers/container-lifecycle-hooks/, Felkeresve: 2020-12-10.

[42] *Kubernetes; Attach Handlers to Container Lifecycle Events*, https://kubernetes.io/docs/tasks/configure-pod-container/attach-handler-lifecycle-event/, Felkeresve: 2020-12-10.

[43] *Kuernetes; Container probes*, https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#container-probes, Felkeresve: 2020-12-10.

[44] *Kubernetes; Configure Liveness, Readiness and Startup Probes*, https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/, Felkeresve: 2020-12-10.

[45] *Kubernetes; Resource metrics pipeline*, https://kubernetes.io/docs/tasks/debug-application-cluster/resource-metrics-pipeline/, Felkeresve: 2020-12-12, 2020. okt.

[46] L. Jiawei, *Metrics Server*, https://github.com/kubernetes/community/blob/master/contributors/design-proposals/instrumentation/metrics-server.md, Felkeresve: 2020-12-12, 2018. aug.

[47] *Kubernetes Helm Architecture*, https://www.slideshare.net/alexLM/helm-application-deployment-management-for-kubernetes, Felkeresve: 2021-10-17.

[48] *OpenStack*, https://www.openstack.org/software/, Felkeresve: 2020-05-17, 2010.

[49] *Closed Loop Security and Compliance Helps You Safely Migrate to and Expand AWS Usage*, https://aws.amazon.com/blogs/apn/closed-loop-security-and-compliance-helps-you-safely-migrate-to-and-expand-aws-usage/, Felkeresve: 2021-05-19, 2019. jún.

[50] *Open Network Automation Platform Overview*, https://docs.onap.org/en/latest/guides/overview/overview.html, Felkeresve: 2020-12-10, 2017.

[51] *Network Functions Virtualisation (NFV); Management and Orchestration; VNF Packaging Specification*, https://www.etsi.org/deliver/etsi_gs/NFV-IFA/001_099/011/02.01.01_60/gs_nfv-ifa011v020101p.pdf, Felkeresve: 2020-12-10, 2017.

[52] *Network Functions Virtualisation (NFV) Release 2; Protocols and Data Models; NFV descriptors based on TOSCA specification*, https://www.etsi.org/deliver/etsi_gs/NFV-SOL/001_099/001/02.07.01_60/gs_NFV-SOL001v020701p.pdf, Felkeresve: 2020-12-10, 2017.

[53] S. T. Lingli Deng Hui Deng, *The Progress of ONAP*, https://www.onap.org/wp-content/uploads/sites/20/2019/04/ONAP_HarmonizingOpenSourceStandards_032719.pdf, Felkeresve: 2020-12-10, 2017.

[54] *Open Network Automation Platform Overview*, https://docs.onap.org/en/latest/guides/overview/overview.html, Felkeresve: 2020-12-10, 2017.

[55] *ONAP Architecture*, https://www.onap.org/architecture, Felkeresve: 2020-12-10, 2017.

[56] *ONAP PORTAL*, https://wiki.onap.org/display/DW/Tutorial%3A+Accessing+the+ONAP+Portal, Felkeresve: 2020-12-10, 2017.

[57] *ONAP Introduction*, https://docs.onap.org/en/frankfurt/guides/onap-developer/architecture/onap-architecture.html#onap-architecture, Felkeresve: 2020-12-10, 2017.

[58] H. Dániel, *Cloud-native MIPv6 mobilitás-menedzsment protokoll implementációja és vizsgálata*, https://diplomaterv.vik.bme.hu/hu/Theses/Cloudnative-MIPv6-mobilitasmenedzsment, Felkeresve: 2021-10-17, 2020.

[59] S. M. Saleh, *Cloud-native MIPv6 mobilitás-menezsment szolgáltatás tervezése*, https://diplomaterv.vik.bme.hu/hu/Theses/Cloudnative-MIPv6-mobilitasmenezsment, Felkeresve: 2021-10-17, 2020.

[60] D. B. Johnson, J. Arkko és C. E. Perkins, *Mobility Support in IPv6*, RFC 6275, 2011. júl. cím: https://rfc-editor.org/rfc/rfc6275.txt.

[61] *Migration from Physical to Virtual Network Functions: Best Practices and Lessons Learned*, https://www.gsma.com/futurenetworks/5g/migration-from-physical-to-virtual-network-functions-best-practices-and-lessons-learned/, Felkeresve: 2021-10-17, 2018.

[62] *Algorithms for making load-balancing decisions*, https://www.ibm.com/docs/en/datapower-gateways/10.0.1?topic=groups-algorithms-making-load-balancing-decisions#lbg_algorithms__fa, Felkeresve: 2021-10-17.

[63] *libcurl - the multiprotocol file transfer library*, https://curl.se/libcurl/, Felkeresve: 2021-10-17.

[64] *JSON-C - A JSON implementation in C*, https://github.com/json-c/json-c, Felkeresve: 2021-10-17.

[65] *vFirewall CNF Use Case*, https://docs.onap.org/projects/onap-integration/en/latest/docs_vFW_CNF_CDS.html, Felkeresve: 2021-10-17.

[66] *What Is TOSCA?*, https://cloudify.co/what-is-tosca/, Felkeresve: 2021-10-17.