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

Böősy Pál;Kis Milán

KUBERNETES OPERATOR-ALAPÚ INFRASTRUKTÚRA MENEDZSMENT IPV6 ALAPÚ MOBILITÁS KEZELÉSRE

KONZULENSEK

Leiter Ákos, Dr. Bokor László

BUDAPEST, 2022. 10. 31.

Tartalomjegyzék

Kivonat

Napjaink távközlési architektúráiban megfigyelhető egyik legnagyobb átalakulást a mikroszolgáltatás alapú (microservice) szemléletmód megjelenése indította. A virtualizáció elterjedésével már nem jelentett problémát az alkalmazások főbb komponenseikre való bontása, azok dinamikus, terhelés függvényében való skálázása. Ezzel hatalmas előnyökre tehetünk szert a monolitikus előd használatához képest mind a fejlesztés, mind a kiszolgálás terén. Éppen ezért jelent meg az igény a paradigma mobil hálózatokban való alkalmazására is, hogy a növekvő elvárásoknak eleget tegyenek. [1] A motiváció egy felhő szemléletű infrastruktúra kialakítása, amiben a mobilhálózati funkciók teljes egészében, vagy akár részben virtualizáltan futnak[2].

Ennek érdekében tekintünk a Kubernetes irányába, amiben már implementálták a konténer orkesztráció legfontosabb elemeit. A Network Function Virtualization (NFV) és a mikroszolgáltatás architektúra együttes megvalósítása azonban jelenleg még nem megoldott, ami a telekommunikációs szolgáltatók szempontjából jelenleg visszaveti a hálózati architektúra fejlődésére vonatkozó törekvéseket. Ennek a legfőbb oka, hogy a Kubernetes alapvetően alkalmazás-centrikus, azaz a központi feladata a konténerizált alkalmazás telepítése és életciklus-menedzsmentje, és habár a hálózatkezelés is egy fontos sarokköve a rendszernek, az főként az alkalmazási rétegre (OSI Layer 7) koncentrál. Azonban ahhoz, hogy az NFV is szerves részévé tudjon válni, fejlett Layer 2 és Layer 3-beli hálózatkezelési képességekre lenne szükség a Kubernetes infrastruktúra részéről, hogy ne az alkalmazás feladata legyen a hálózatkezelés. Ezen probléma megoldására jelenleg is aktívan kutatják a lehetséges alternatívákat [3], [4]. Dolgozatunkban bemutatjuk az általunk javasolt és valós tesztrendszeren megvalósított megoldásokat, amelyekben két irányból, és a Mobil IPv6 protokoll kontextusában közelítjük meg az adott kérdéskört. Azért esett a választás a MIPv6 technológiára, mert úgy gondoljuk, hogy a jövő hálózataiban jelentős szerepet fog játszani, mivel a protokoll által nyújtott újdonságok a mobilitás kezelés komplexitását nagymértékben csökkentik [5, o. 6].

A konténerizált szolgáltatások hálózatmenedzsmentjére az egyik megközelítés egy saját Kubernetes Operátor implementálása, amely a humán hálózati operátorok munkáját idézi meg. Lényegében egy kontrollerről van szó, amely deklaratív kérések

alapján változtatja majd a hálózati infrastruktúrát. Ezzel megvalósíthatóvá válik a két komponens közötti, dinamikus adatsík létesítés is. A Mobil IPv6 esetén ez a vezérlési sík speciális üzeneteinek érkezésekor fog történni. A fentebb taglalt problémák megoldására szintén ígéretesnek mutatkozik a Network Service Mesh [6] nevű, nyílt forráskódú projekt. Ez szolgáltatja a másik megközelítésünk alapjait, melyben a rendszer segítségével olyan hálózati architektúrákat valósíthatunk meg, ahol az egyes munkaegységek klaszterek közötti kommunikációt képesek létesíteni a konténerizált hálózati eszközökkel (CNF). Megoldásainkat különböző funkcionális– és teljesítménytesztnek, elemzésre szolgáló méréseknek vetjük alá, majd a gyűjtött eredmények statisztikai kiértékelésével zárjuk dolgozatunkat, hogy bebizonyítsuk javaslataink működőképességét, megvilágítsuk előnyeit, hátrányait, potenciális gyakorlati alkalmazási lehetőségeit.

Abstract

The emergence of microservice-based systems brought about the most significant change of view in today's telecom architecture. With virtualization becoming increasingly prevalent, decomposing applications into their building components poses no problem anymore, enabling their dynamic, usage-responsive scaling. This setup exceeds its monolithic counterpart by a great deal in terms of both development and user experience. With all that said, it's no surprise that an early demand could be seen in its application in real-world telecom networks. The goal, of course, is to satisfy the ever-growing expectations [1]. Future visions of a cloud-based infrastructure where either all or a part of the mobile network functions run in a virtualized environment motivate us to further improve upon this concept [2]. Thus we look towards Kubernetes in the hopes of accomplishing this, where most of the container orchestration requirements are already met.

Despite all of these promising facts, the joint implementation of Network Function Virtualization (NFV) and the Microservices architecture is currently not yet resolved, which from the point of view of telecommunication service providers, means a major setback for the development of their network architecture. The primary cause is that Kubernetes is mainly application-centric, and thus its primary role is the deployment and the lifecycle management of containerized applications. Even though networking is a significant cornerstone of the architecture, it concentrates more on the application layer (OSI Layer 7). However, for NFV to become an integral part of the system, advanced Layer 2 and Layer 3 networking capabilities are required from Kubernetes so that network management wouldn't be the responsibility of the application. Alternative solutions for solving the problem mentioned above are actively researched [3], [4]. In our TDK work, we shall showcase our solutions to this problem, approaching it from two different perspectives in the context of Mobile IPv6. We examine our proposals through this technology because it is our personal belief that in future networks, IPv6 mobility management will gain more prevalence. MIPv6, as an advanced protocol, provides global reachability for mobile terminals, and thanks to its capabilities, it widens mobility horizons and provides transparent operation for higher layer applications [5, o. 6].

One approach for the network management of containerized services is implementing our own Kubernetes Operator, whose principles of operation are similar to the human network operators. In a nutshell, it's a controller which changes the network infrastructure depending on the incoming declarative requests from its clients. This enables the dynamic creation of data planes between two components. With Mobile IPv6, this will occur when receiving a special control plane message.

The open source project called Network Service Mesh [6] is also a promising tool for resolving the issues discussed above. With the help of this system, we can implement network architectures in which individual workloads can establish inter-cluster communication with Cloud-native Network Functions (CNFs).

Our proposed solutions will be put to the test in specifically designed functional and performance assessments. With a thorough statistical analysis of our collected results, we aim to prove that these approaches are fully functional.

1 Bevezető

A mobil távközlési infrastruktúrákban tapasztalható egyre nagyobb hálózati terheltség és komplexitás újabb kihívások elé állítja a szolgáltatókat. A negyedik generációs mobil celluláris rendszerek a megjelenése idejében bőven kielégítette az akkori elvárásokat, azonban az évek során jelentős változások történtek technológiai és egyéb szempontokból. Átalakultak a mobil előfizetők tartalomfogyasztási szokásai, illetve egyre nagyobb teret hódítanak olyan technológiák, mint az Internet of Things (IoT) vagy a járműkommunikáció. Ezek összessége olyan követelményeket támaszt a mobil telekommunikációs hálózatokkal szemben, melyek kiszolgálására a jelenlegi hálózati architektúra és eljárások mellett vagy nincs, vagy limitáltak a lehetőségek. Ha csak a járműkommunikációt tekintjük, nyilvánvaló, hogy folyamatos, valós-idejű kommunikációra van szükség annak érdekében, hogy a járműkommunikációs rendszerek a közlekedési forgalom hirtelen változásait érzékeljék és azokra megfelelően és gyorsan reagáljanak. Már néhány ezred másodpercnyi késleltetés is drasztikusan befolyásolhatja az alkalmazások működését. A mobil előfizetők esetében tapasztalt változások főként az egyre nagyobb felbontású és egyre jobb minőséget kínáló streaming szolgáltatásoknak tulajdoníthatók. Habár ez az alkalmazási terület kevésbé érzékeny a késleltetésre, könnyen problémát tud jelenteni a hálózatnak a nagy számú felhasználó viszonylag nagy sávszélességet igénylő adatfolyammal történő egyidejű kiszolgálása. Ezen kívül a felfelé áramló forgalom is jelentősen megnőtt az olyan élő sugárzást támogató platformoknak köszönhetően, mint Facebook Live és a Twitch.

A hálózat architektúrájának szervezésén is sok múlhat. Várhatóan nem csak az emberi előfizetők számában lesz tapasztalható exponenciális növekedés, de az IoT folyamatos terjedése és egyre szélesebb körben történő alkalmazása is csak tovább tetézi a problémát. [1]

A fentebb említett esetek mindegyikében nagyságrendekkel több végberendezés mobil hálózati kapcsolatának kezelésére van szükség. Bár már most is tapasztalhatóak az IPv4 hiányosságai, az elkövetkező időszakban egyenesen lehetetlen lenne kizárólag ilyen alapon megoldani a hálózat szerveződését. Így az új mobil és vezeték nélküli hálózatok architektúráját, illetve az azokat igénybe vevő új alkalmazásokat érdemes legalább dual-stack (IPv4 és IPv6 alapú kapcsolat) támogatásra tervezni. Látható tehát, hogy drasztikus

változtatásokra lesz szükség a következő generációs mobilhálózatok architektúráiban, hogy a már fentebb, a teljesség igénye nélkül részletezett alkalmazási területek által támasztott kritériumoknak meg tudjanak felelni.

Mivel a jelenlegi mobil hálózati architektúrák hosszú távon nem tudják fenntartani a mobilitási szolgáltatás elvárt minőségét, így az architektúra adaptálódására is szükség van. A problémára a megoldást a hálózati funciók virtualizálása, a hálózat szoftverizálása és az egyre kiforrottabb felhő-technológiák jelenthetik. A felhő alapú megvalósítás több szempontból kifizetődőnek tűnik. Olyan szituációkban, amikor a szokványos kihasználtsági szinthez képest nagyságrendekkel ugrik meg a felhasználói forgalom (legyen az bármilyen, tömegeket megmozgató esemény, vagy valamilyen katasztrófa bekövetkezésekor, amikor a hálózat stabilitásának biztosítása kritikus feladattá válik), a publikus felhők, mint a hálózatot kiegészítő erőforrások nyújthatják a megoldást. Egy szolgáltató ugyanakkor megteheti, hogy az ilyen esetekre dedikált infrastrukturális tartalékot tart fenn, azonban azt akkor is üzemeltetni kell és karban kell tartani, amikor érdemben nincs használatban, ami nagy mennyiségű felesleges kiadást jelentene a vállalat számára. Azonban, ha a megnövekedett forgalom egy részét a publikus felhőszolgáltatók rendszereiben dolgozná fel, akkor csak a kritikus időintervallumban kellene fizetnie az infrastruktúra használatáért. Azonban nem csak az ilyen hirtelen bekövetkező változások kezelésére lehet felhasználni a felhőt, hanem az ún. Total Cost of Ownership (TCO) is jelentős mértékben csökkenthető, ha a szolgáltatók a szokványos kapacitásaik egy részét is a felhőbe migrálják. Így nem csak technológiai, hanem gazdasági megközelítések mentén is megérheti a szolgáltatóknak az architektúra adoptálása.[7]

Motiváció

Dolgozatunk megírására többek között a felhő-alapú mobilitáskezelésben rejlő lehetőségek adták az inspirációt, melynek egy fontos aspektusát egy kifejezetten új és előremutató architekúraszervezési módszer kontextusában vizsgáljuk meg.

A jelenlegi felhő-technológiák már magukkal vonják a konténerizációt és vele együtt a mikroszolgáltatások architektúrát is. Ezek ipari keretek közötti megvalósításához és a szolgáltatás minőségének biztosításához egy robusztus és az igényekhez könnyen adaptálódó infrastruktúra rétegre van szükség. A Kubernetes konténer-orkesztrációs eszközt közel 10 éves pályafutása során a legváltozatosabb ipari környezetekben vetették be. A széles körű támogatottsága által egy folyamatosan fejlődő és bővülő

funkcionalitással rendelkező eszközzé vált az évek során, melynek következtében de-facto szabvánnyá nőtte ki magát a következő generációs mobilhálózatok infrastruktúráját illetően. Dolgozatunk sarokkövét pedig az általa támogatott Operátor tervezési minta jelenti, mely még tovább mozdíthatja az infrastruktúra automatizálását. Az Operátor minta lényege, hogy a számára meghatározott erőforrás(ok) állapotát a felhasználó által meghatározott, elvárt állapotban tartsa. Dolgozatunk célkitűzése, hogy ezen tervezési minta mentén implementáljunk egy proof-of-concept hálózatkezelési megoldást, mellyel megvizsgáljuk a hálózati funkciók felhősítésének lehetőségeit.

2 IP alapú mobilitás-kezelés

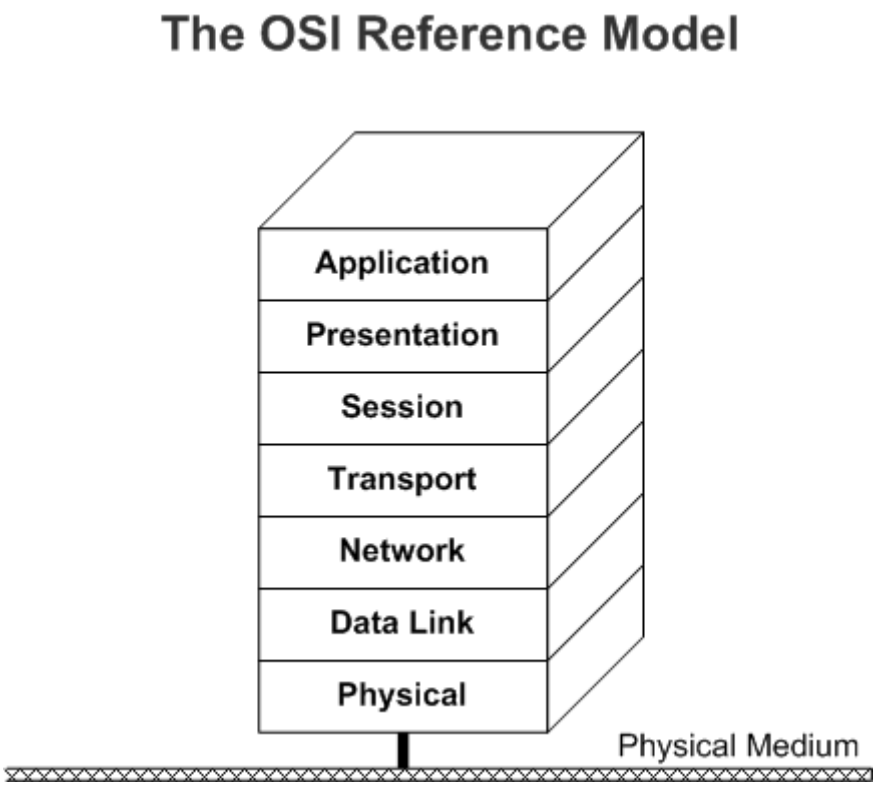
A mobilitás problémája vezetéknélküli hálózatokban

Az adatszolgáltatást használó mobil terminálok folyamatos mozgásban lehetnek. Így a vezetékes megoldásoktól eltérően előnyös, sőt elvárható, hogy ne kelljen egyhelyben maradnunk, hogy használhassuk az internet által nyújtott szolgáltatásokat. Tehát elmondható, hogy a vezetéknélküli kapcsolatok esetén a mobilitás az egyik olyan kihívás, amit mindenképpen meg kell oldani.

Azonban folyamszerű protokollok esetén, mint mondjuk a TCP-t használó SSH, vagy éppen a céges környezetekben széleskörűen használt VPN, nem tehetjük meg, hogy csak úgy elmozdulunk. A probléma gyökere, hogy a szerver nem fog tudni a váltásunkról, de még ha tudathatnánk is vele, ezt muszáj lesz elkerülni. Egyrészt újra ki kell építeni a folyamot a két kommunikáló fél között, másrész lefoglaljuk a hálózat sávszélességét potenciálisan kiküszöbölhető kontroll üzenetekkel. Nem is beszélve arról, hogy a szerver feladata is megkétszereződik, ha minden aktív kapcsolata mozgásban van, ami számottevő terhelést jelent majd számára. Az IPv4 hálózati stack egyébként alapvetően nem képes a másik kommunikáló fél értesítésére a hálózati réteg szinten. Más lesz azonban a helyzet az IPv6 esetén, erre idővel kitérünk.

Előnyösebb lenne, ha ezt a mozgást elfedhetnénk a szerverek elől, és azok különösebb gond nélkül címezhetnék egy helyre a csomagokat, a mozgást pedig lokálisan, a hálózat “szélén” lévő eszközök kezelnék. Szerencsére erre többféle megoldás is létezik, egyik esetben még a mobil terminál elől is elrejtjük a mozgást.

Két kategóriára érdemes ezelet bontani: L2 és L3 mobilitás. Az L a layer angol szót rövidíti és az OSI modell megszokott rétegeire utalnak.[8]



1. ábra: OSI modell [9]

L2 mobilitás

A Wi-Fi esetén egy routerhez esetenként több Wi-Fi Access Point is tartozhat, de az ezekre csatlakozó eszközök mind egy tartományból kapnak IP-t, így az IP-szintű kapcsolat nem szakadhat meg akkor, ha az egyik AP-ről átmegyünk a másikra. Ilyenkor beszélünk L2 handoverről. Erről dolgozatunkban csak érintőlegesen beszélünk, de mindenképpen említésre méltó.

A mozgó terminál mozgás közben folyamatosan méri a jel erősségét és gyengülés esetén elkezd Wi-Fi Probe üzeneteket küldeni, aktívan keresve a jelenlegi Access Point alternatíváját. Ha talál, akkor felcsatlakozik az újra egy Reassociation Request üzenettel, majd lecsatlakozik a régi, gyengülő jelű Access Pointról.

A negyedik generációs hálózatok esetén az X2-es interfésszel összekapcsolt eNodeB-k egy ilyen megoldás, ezzel szemben az ötödik generációs gNodeB-k esetén az Xn interfész adja ezt a funkciót. A handover ezeknél is hasonlóan működik, mint a Wi-Fi-nél, hogy a terminál folyamatosan méri a jelerősséget, de ez az implementáció egy lépéssel tovább megy.

A régi helyre érkező csomagokat is továbbküldik az új gNodeB-nek az előbb említett interfészen keresztül. A L2 handovert befejezve a gNodeB kérést intéz az Access & Mobility Management Function-höz, amit az visszaigazol és ezzel már az új adótorony fogja kapni a mobil terminálnak címzett üzeneteket.

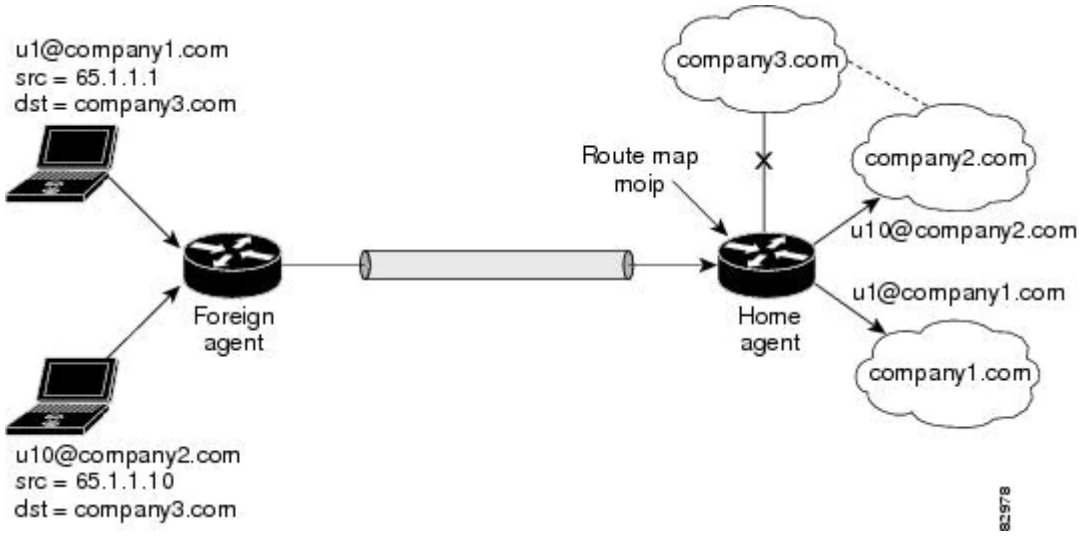
L3 mobilitás

Az előbbi megoldás viszont már nem elégséges abban az esetben, amikor új alhálózatra kerülünk, hiszen itt már új IP címet is kell igényelni. Az itteni megoldás egy kicsit több beavatkozást igényel terminál részéről, ehhez már nem lesz elég a hálózat önmagában. Ezzel elérkeztünk a Mobil IP-hez, vagy rövidítve MIP-hez, ami két ízben is megjelenik a két IP családot támogatva. Az első a MIPv4, ami értelemszerűen az 4-es verziójú internet protokollhoz nyújt mobilitási támogatást, illetve a MIPv6, ami pedig az IPv6 esetén nyújt mobilitást a harmadik rétegben.

Első megközelítés, mobilitás az IPv4 protokollon[10]

A továbbiakban ezt fogjuk taglalni, azonban mindenekelőtt érdemes beszélni a proxy ügynökök létéről. Ugyanis, ha a terminál maga nem is tudja átvenni a csomagot, egy nem mozgó, megbízott ügynök mindenképpen képes rá. Ezt az entitást nevezi a szabvány Home Agent-nek, illetve az új hálózatban Foreign Agent-nek. A Home Agent lesz felelős fogadni az eredeti címre érkező csomagokat és azokat továbbadni a Foreign Agent-nek egy speciális kapcsolaton keresztül, amit a szabvány tunnelnek definiál.

A tunnel egy lenyűgözően hasznos, de rendkívül egyszerű koncepció. Lényegében, ha az érkező IP csomagot egy másik IP fejlécbe újra becsomagolunk, megkapjuk a tunnelt. Ezzel elérjük, hogy ne a belső, hanem a külső fejléc alapján irányítsuk a csomagot. Így indaddig ekképpen utazik, amíg el nem ér az üzenet egy olyan félhez, aki tudja, hogy egy duplán csomagolt üzenetről van szó és ezt kicsomagolva kézbesíthetővé válik.



2. ábra: Mobil IPv4 modellje [11]

Mobil IP architekturális leírása

A Mobil IP szabványban a tunnel egyik végén a Home Agent áll, aki a mobil terminálnak érkező üzeneteket ezen keresztül juttatja át a másik végpontnak. Azonban Mobil IP RFC a végpont esetére kétféle működésmódot definiál. Az első, amikor az előbb említett Foreign Agent-nél terminálódik a tunnel, a másik, amikor pedig a mobil hosztnál, ezzel a Foreign Agent szükségességét kiküszöbölve, de IPv4 esetén az első működésmódot részesítjük előnyben. A preferencia okairól rövidesen beszélünk, de előtte érdemes megközelíteni a problémát a mobil hoszt szemszögéből is.

Ahogy azt fentebb is említettük, a Mobil IP esetén a mozgó terminál aktív félként lép fel. Kapcsolatainak fenntartása érdekében folyamatosan figyel speciális üzenetekre, amikben az ilyen ügynökök hirdetik magukat, hogyha egy ponton észreveszi a mobil hoszt, hogy új hálózatban van, először is igényel egy szabvány által definiált Care-Of-Address-t a Foreign Agent-től, amivel regisztrálja majd magát a Home Agent-jénél. Az eredeti címét természetesen megőrzi, ezt nevezzük Home Address-nek. Viszont a Care-Of-Address egy speciális IP cím, ahol terminálni fog a tunnel. Ez az első szcenáriónál természetesen a Foreign Agent egyik címe lesz, míg a másikban egy plusz cím lesz, amit a mobil hoszt egyik interfészéhez allokálunk.

Az első eset, ahogy azt már említettem, előnyösebb IPv4 esetén, mert a Foreign Agent egy címére akár több hasonló kapcsolat is meghirdethető. Azaz, ha veszünk mondjuk 5 mozgó hosztot, és mindegyik egymástól különböző Home Agent-tel

rendelkezik, akkor mind az 5 hoszt egy Care-of-Address-szel elérhetővé válik. Ez abból adódik, hogy a Foreign Agent már képes lesz a beérkező üzeneteket kicsomagolni és továbbítani a megfelelő Home Address-re. A rohamos ütemben fogyó négyes verziójú internet protokoll címek problémája így enyhíthetővé válik, hiszen nem kell kétszer annyi címet allokálni, ezzel a cella kapacitását pedig fordítottan arányosan a felére csökkentve.

Mobil IPv4 hátrányai

A problémák így nagyrészt megoldódtak, de ahogy azt a szakemberek is állítják, az IPv4 kevés lesz a jövő felhasználói igényeinek kielégítésére.[12] A bevezetőben már szó esett, hogy az aktív IoT eszközök várhatóan erősen növekvő tendenciát fognak mutatni, a V2X applikációkat kiszolgáló road-side unitokról nem is beszélve. Az eleve szűkös, esetenként teljesen lefoglalt, címtartomány nem lesz képes kiszolgálni ezt a mennyiségű IP címre váró eszközt, ezzel teljesen ellehetetlenítve a második Mobil IP szcenáriót, ahol Foreign Agent nélkül kommunikálhatna a terminál Home Agent-jével.

Ennek a következménye, hogy elengedhetetlen entitássá válik a hálózatban a Foreign Agent jelenléte, ami teljesítmény és funkcionális szempontból nagy hátránnyá válik számunkra.

Egyrészt, mivel ki kell csomagolnia egy központi entitásnak a csomagokat és elosztani megfelelően kétszer annyi idő telik el a csomag továbbítással. Hiszen kétszer is döntést kell hozni az üzenet továbbítását illetően a tunnel miatt. Ha sok hasonló csomópont kommunikál így, akkor jelentősen romolhat mondjuk a késleltetés, ami egyes alkalmazásoknál, nagy problémát jelent, mint például videó streaming esetén.

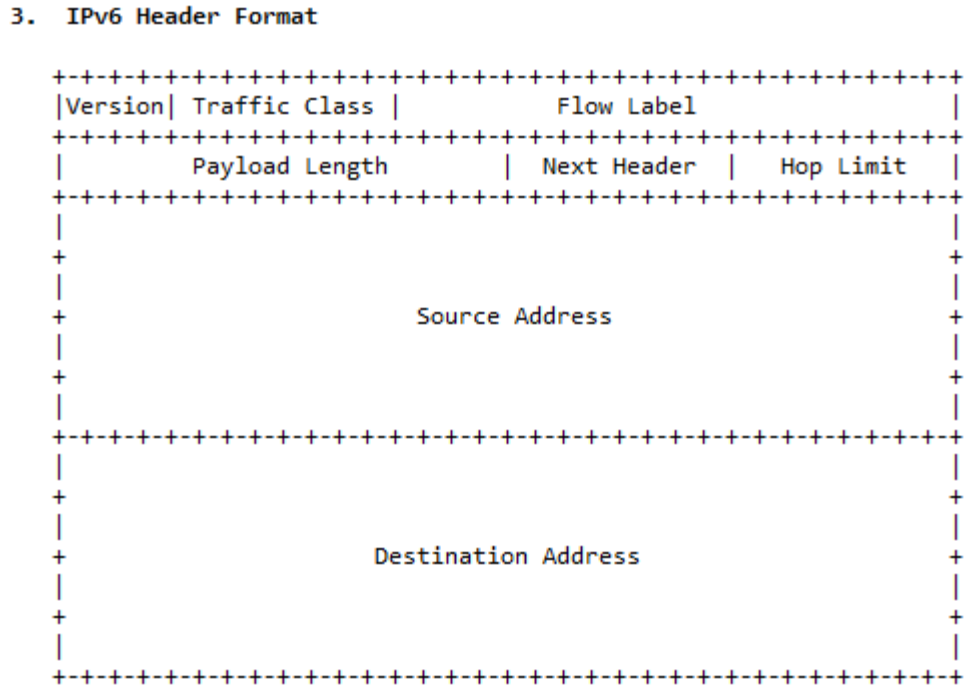
Másrészt pedig funkcionális szempontból a Foreign Agent-ek nagy felelősséggel rendelkező rendszerkomponensek, ha megáll a működésük, az összes mobil a körzetben képtelenné válik futó kapcsolatainak megtartására. Egy teljesen felesleges szűk keresztmetszetet építünk bele a rendszerbe abból a kényszerből, amit az IPv4-es címek rónak ránk.

Mobil IPv6[13], [14, o. 6]

IPv6

Az IPv4 utódjaként specifikált IPv6-os sok újdonságot hoz be az internet világába. Az IP címtartományt megnövelte 296-szorosára, ami körülbelül 79 kvadrilliárdszor

akkora, mint az négyes verzió esetén. Hogy az írásmódot könnyítsék, konvenció szerint hexadecimális számként ábrázoljuk az újfajta címeket, azokat 2 bájtonként egy kettősponttal elválasztva. Maga a protokoll sokat egyszerűsödött, kivették az IPv4-ben ritkán használt mezőket elhagyták, vagy az opcionális mezőkbe rakták azokat, ezzel a könnyebb feldolgozást elősegítve. Továbbá, definiáltak egy új módot az IP fejléc kiterjesztésére.



3. ábra: IPv6 csomag fejléc

Az új kiterjesztések megtestesítője a Next Header mező, ebben az IANA által kijelölt számkódokkal tudathatjuk a feldolgozó egységgel, hogy milyen fejléc következik a sorban. Az IPv4-gyel szemben a Next Header nem csak a következő réteg fejlécét jelezheti, hanem az azt követő kiegészítő fejlécet is. Több ilyet definiáltak, mint a Destination Options, Routing Header, vagy a Mobility Header, ami a dolgozatban kiemelkedő szerepet fog kapni. A felsoroltak mind tartalmaznak Next Header jellegű mezőt, amivel támogatják az úgynevezett “header chaining” technikát, amivel egymás után fűzhetünk tetszőleges számú hasonló fejlécet.

Látható, hogy minden szempontból egy hatékonyabb és robosztusabb megoldást kaptunk az új verzióval, amivel már sokkal több IP cím osztható ki, így pedig már elérhetővé válik számunkra a MIPv6. A következőkben erről fogunk tárgyalni részletesebben, a négyes verzióval összehasonlítva. Alább egy másik tanulmányból átvett

összefoglaló táblázat található, amely az egyes implementációk közötti különbséget emeli ki, itt található a kioszthatő címek száma is.

TABLE I
SUMMARY OF MIPv4, MIPv6, AND PMIPv6 [12]

Category	MIPv4	MIPv6	PMIPv6
Mobility scope	Global	Global	Local
Mobility management	Host-based	Host-based	Network-based
Network architecture	Flat	Flat	Hierarchical
Target network	IP	IP	IP
Operating layer	L3	L3	L3
Required infrastructure	HA & FA	HA	LMA & MAG
MN modification	Yes	Yes	No
Router advertisement	Broadcast	Broadcast	Unicast
Addressing model	Shared-prefix	Shared-prefix	Per-MN-prefix
MN address	HoA	HoA	CoA
Address type	IPv4	IPv6	IPv6
Address length (bits)	32	128	128

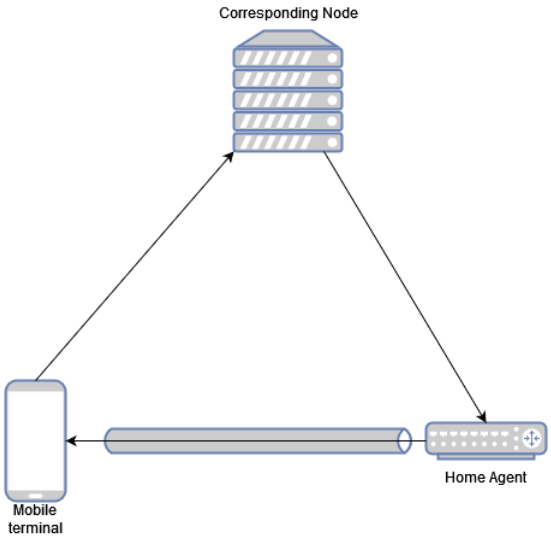
4. ábra: Mobil IP implementációk összehasonlítása [15]

Különbségek

Az új protokoll sok újítást hoz a mobilitás terén, az első legfontosabb, hogy nincs Foreign Agent. Innentől kezdve nincs szükség az idegen hálózat routerének közreműködésére, így megszűnt ez a fajta függőség is. A második fontos különbség a továbbítás terén jelent meg. Itt is kétfajta mód áll rendelkezésre, az IPv6 encapsulation, azaz tunneles megoldás, ami a Home Agentre támaszkodik, illetve a másik, ami pedig a hatos verzió egyik új kiegészítő fejlécén alapszik, a Routing Header-re.

Az utóbbi segítségével kivehető a Home Agent a kommunikációból, mivel szólhatunk a kommunikáló szervernek egy route optimalizációs eljárás során, hogy egy új kötést hozzon létre a Home Address és a Care-Of-Address között. Így a két fél ismét

csak egymással kommunikál. Ennek sok előnye van, de talán a legfontosabb az úgynevezett “triangular routing” probléma megoldása, amit az alábbi képen láthatunk.



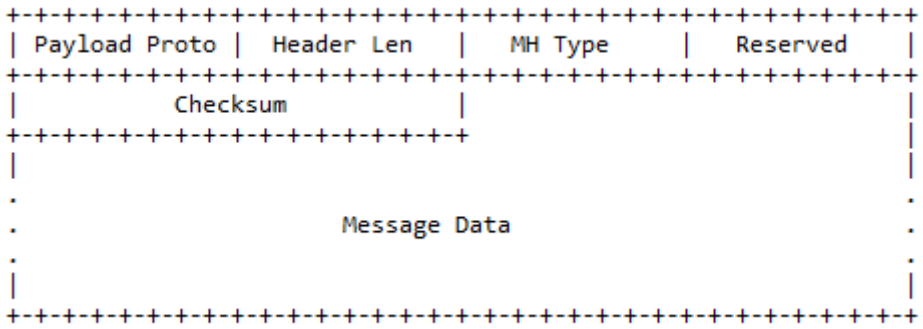
5. ábra: Triangle routing

Az ábrán láthatjuk, hogy míg a mobil terminál direktben kommunikál a szerverrel, addig a szerver csak a Home Agenten keresztül képes kommunikálni. Az így kialakuló háromszögből ered a neve is ennek a helyzetnek.

Dolgozatunkban mindezek ellenére a Home Agent-alapú megközelítéssel fogunk foglalkozni, hiszen ennek az implementációnak cloudosítása nem teljesen egyértelmű, szemben az elsővel, ami felhőagnosztikus.

MIPv6 üzenetek

Az RFC egy új kiegészítő fejlécet definiál, amit Mobility Header-nek nevezünk és IANA 135-ös számmal jelölünk a Next Header mezőben. A formátum az alábbi ábrán látható.



6. ábra: Mobility header csomagfejléc

Itt csak a fontosabb mezőkön megyünk végig, a részletekről az RFC-ben olvashatunk. A kiegészítő fejlécekhez alkalmazkodva az első mező lényegében azonos az IPv6 Next Header-jével, ugyanazokat az értékeket is használja. A fontosabb számunkra azonban az MH Type és a Message Data lesz. Az MH Type jelöli ki, hogy milyen MIPv6 üzenetet hordoz magával a csomag.

Megnevezés	MH Type érték
Home Test Init	1
Home Test	3
Care-of Test Init	2
Care-of Test	4
Binding Update	**5**
Binding Acknowledgement	**6**
Binding Refresh Request	0
Binding Error	7

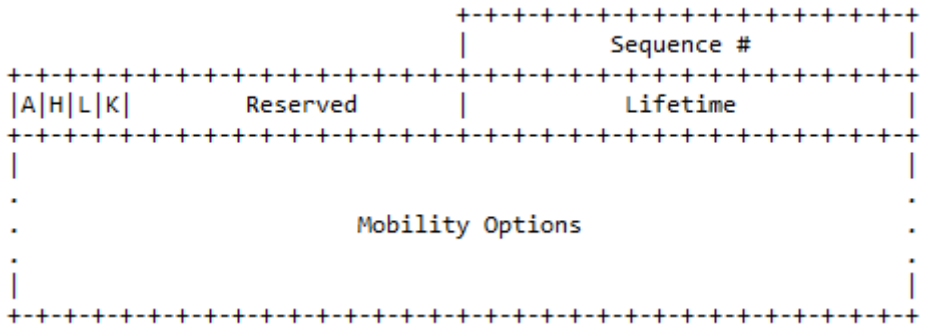
1. táblázat: Mobility csomagtípusok

Mint azt a fenti táblázatban is láthatjuk, a protokoll sokféle üzenetet használ, hogy mindenféle esetet lefedjen és kezelhessen a rendszer. A Home Test, illetve a Care-of Test üzenetek cseréje egy fontos lépés a közvetlen kapcsolatok kiépítésében a szerver és a mozgó terminál között, de ezek olyankor használatosak, ha a terminál szeretné tájékoztatni a vele kommunikáló szervert az új címéről. Ilyenkor életbe lép egy “return routability” procedúra, amivel a szerver verifikálja, hogy a Care-of-Address valóban egy, a Home Address-hez tartozó, kötés.

Éppen emiatt ezekkel nem foglalkozunk, hiszen a tunnel szcenárióban a legfontosabb üzenetek a Binding Update és Binding Acknowledgement, ezeket félkövérrel jelöltük is a fenti táblázatban.

Binding Update

Binding Update csomagokat a terminál küld, a címzett pedig a Home Agent lesz. Így tudja értesíti az otthoni routert az új címéről. A Care-of-Address itt természetesen a forrás címe lesz az üzenetnek.

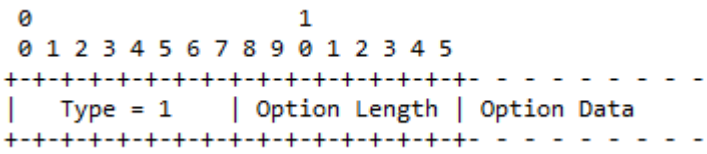


7. ábra: Binding Update csomagfejléc

A Sequence egy 16 bites szám, amit a terminál generál, szabvány szerint az erre érkező válasz üzenet azonos nevű mezőjének a tartalma azonos lesz ezzel az értékkel. Az A, H, L, K státusz jelölő bitek közül fontos az Acknowledge (A), ami arra utasítja a fogadó Home Agent-et, hogy Binding Acknowledgement-tel válaszoljon. A Home Registration (H) bit pedig informálja a fogadót, hogy a mobil terminál Home Agent-jének feladatait kell, hogy ellássa.

A Reserved részt szabvány szerint végig nulla értékre kell beállítani és ignorálni, ezeket a helyeket jövőbeli újításokra tartják fenn. A Lifetime egy 16 bites előjel nélküli szám, ami a kötés (binding) lejártát szabályozza. Ennek a mértékegysége “time unit”, ami lényegében 4 másodpercet jelöl. Tehát, ha mező értéke 2, akkor 2*8=16 másodpercig érvényes a kötés.

A Mobility Options egy érdekes mező minden szempontból, változó hosszúságú adatokat kódolnak benne TLV formátumban. A TLV feloldása Type, Length, Value. Type a típusjelölő 8 bit az elején, a Length a változó hosszúságú adat mérete 8 biten ábrázolva. A végén pedig a Value az adott hosszúságú érték.

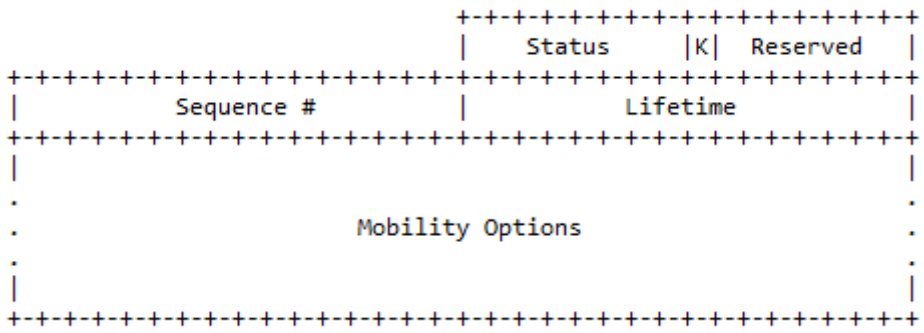


8. ábra: PadN TLV

Ezek közül megkülönböztetünk Pad1, PadN (padding, avagy pányvázó) típusú TLVket, a PadN formátuma a fenti ábrán látszik is. De ezeken kívül Binding Refresh Advice-szal is találkozhatunk és még sok mással.

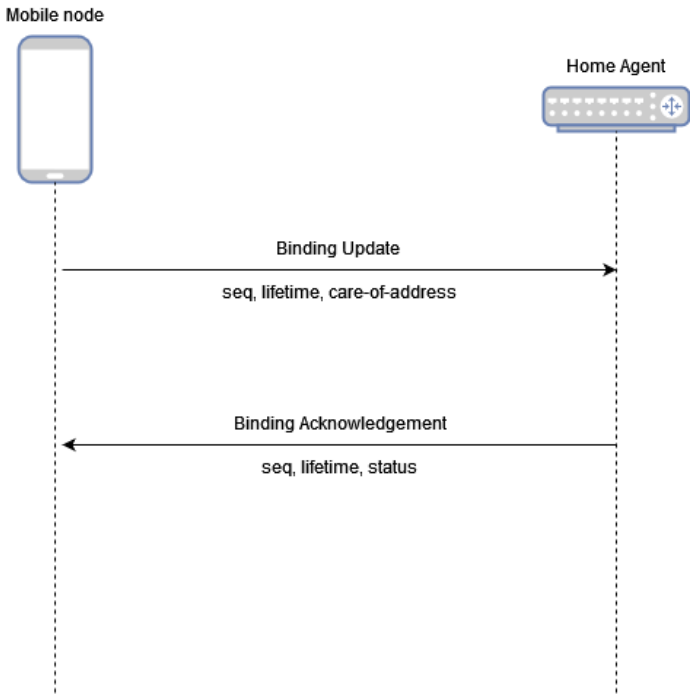
Binding Acknowledgement

Binding Acknowledgement üzenetet a Home Agent küld válaszként a Binding Update-re, a mobil terminál számára ez egy vevény (receipt), hogy a Home Agent átvette a kérését.



9. ábra: Binding Acknowledgement csomagfejléc

Fontos lesz számunkra a Status mező, amiben a Home Agent inidkája a mozgó hoszt számára, hogy kérése teljesíthető-e. Ha nulla, vagy egy a mező értéke, akkor elfogodta a kérést, ha 128, vagy annál több, akkor valamilyen hiba lépett fel. A Reserved ugyanúgy nullára állított mező, mint az előző esetben. A Sequence mindig a BU Sequence mezőjével megegyező értéket vesz fel, ezzel mutatva, hogy arra válasz. A Lifetime szerepe is ugyanaz. Mobility Options között a megszokott paddingek jelenhetnek meg, illetve a Binding Authorization Data és a Binding Refresh Advice opció.



10. ábra: Mobilitás jelzésfolyam

A fenti ábrán láthatjuk az általunk használt, egyszerűsített jelzés folyamot (signaling flow) a két résztvevő között, ahogy érkezik a Binding Update-re válaszként a Binding Acknowledgement. A fontosabb mezőket kiemeltük az vonal alatt.

3 Virtualizációs és felhő alapú technológiák

Ahhoz, hogy a későbbiekben taglaltakat kontextusba tudjuk helyezni, összefoglaló jelleggel bemutatjuk azokat a virtualizáción alapuló, illetve felhő rendszerekben használt technológiákat, melyeket közvetlenül felhasználtunk a dolgozatunk elkészítéséhez. Kitérünk továbbá arra is, hogy az évek során milyen változáson mentek keresztül, illetve milyen mozgatórugói voltak ezeknek.

Virtualizáció és fejlődése

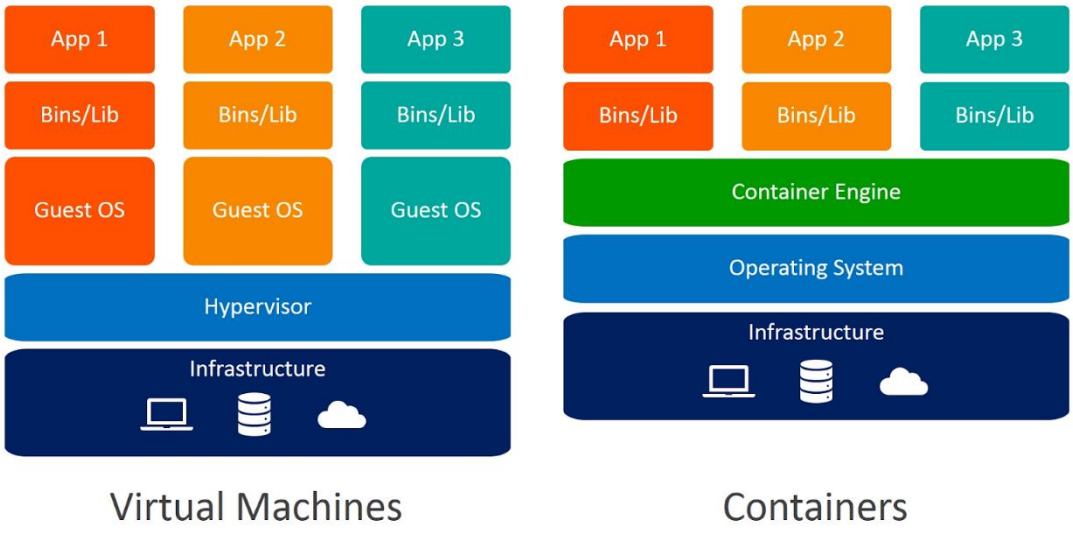
Virtualizáció során valamilyen emulált rendszert hozunk létre, ez jelentheti hardver, tárterület vagy operációs rendszer virtualizálását. Lényegében elmondható, hogy egy absztrakciós rétegen keresztül lehetőséget biztosít erőforrások szeparálására. Maga a koncepció már az 1960-as években megszületett, azonban csak évtizedekkel később kezdett el nagyobb teret hódítani. A technológia előretörését az idézte elő, hogy kezdetben az IT szolgáltatások többségét különálló hardvereken futtatták. Bizonyos esetekben előfordult, hogy egyes szolgáltatások sokkal nagyobb erőforrás igénnyel rendelkeztek, mely negatív hatással volt a többi teljesítményére. Ezt úgy lehetett elkerülni, ha minden programnak saját hardvert allokáltak, ami viszont nem volt gazdaságilag fenntartható: amellett, hogy sokkal több eszközt kellett így üzemeltetni, az egyes eszközök kapacitásai az idő jelentős részében nem voltak teljesen kihasználva.

Ez vezetett el végül a hardver-virtualizáláshoz, azaz a virtuális gépekhez (Virtual Machine, VM). Ebben az esetben egy ún. hypervisor szoftver a gazda gép fizikai erőforrásait felügyeli, illetve felosztja azokat úgy, hogy a virtualizált környezetek felhasználhassák azokat. Ez történhet a gazda operációs rendszeren belül, vagy akár a hypervisor közvetlenül a hardverre való telepítésével. Utóbbi működés az, melyet vállalati szinten alkalmaznak. Az erőforrások ilyen szintű szeparálásának több pozitív mellékhatása is volt. Egyrészt az egyes futtatási környezetek ezáltal izolálva lettek egymástól, illetve a gazda rendszertől, amely még nagyobb biztonságot jelentett a szolgáltatások szempontjából. Másrészt a virtuális gépekről ún. képfájlokat (image) lehetett készíteni, melyek segítségével lehetőség nyílt arra, hogy a teljes futtatókörnyezetet ‘becsomagoljuk’, és azt egy másik gazda gépen elindítva ugyanazt a futtatási környezetet kapjuk vissza a már korábban feltelepített összes programmal együtt. Ezáltal nem kellett minden egyes alkalommal elölről kezdeni a rendszer feltelepítését és

a környezet konfigurálását, hanem elegendő volt kiindulni egy korábban létrehozott képfájlból.

Habár ez a szemléletmód rengeteg problémára nyújtott megoldást, az idő előrehaladtával egyre több olyan elvárást támasztottak a technológiával szemben, melyeknek nem tudott megfelelni. A virtualizált környezetek ugyanis meglehetősen lomhák, hiszen ugyanúgy számolni kell az esetükben akár pár percnyi boot idővel, mint a gazda gép esetében. Másrészt egy ilyen teljesen becsomagolt rendszer-kép az esetek túlnyomó részében több GB nagyságrendű, mely igencsak megnehezíti a rendszer szállíthatóságát. Ezen problémákra a megoldást a konténerizáció megjelenése jelentette.

A konténerek a gazda rendszerrel megosztottan használják ugyanazt a kernelt, egy konténer elindulása ennek következtében szinte azonnal bekövetkezik. Mivel a konténer képfájl nem tartalmazza az operációs rendszert, elegendő csupán a futtatandó alkalmazást az összes dependenciájával együtt tárolni benne, amely már jelentős méret csökkenést jelent a VM képfájlokhoz képest (jellemzően 10 MB nagyságrendű egy konténer képe). Ezen jótékony tulajdonságaik miatt a VM-ekhez képest ’pehelysúlyúnak’ (lightweight) szokás titulálni őket.



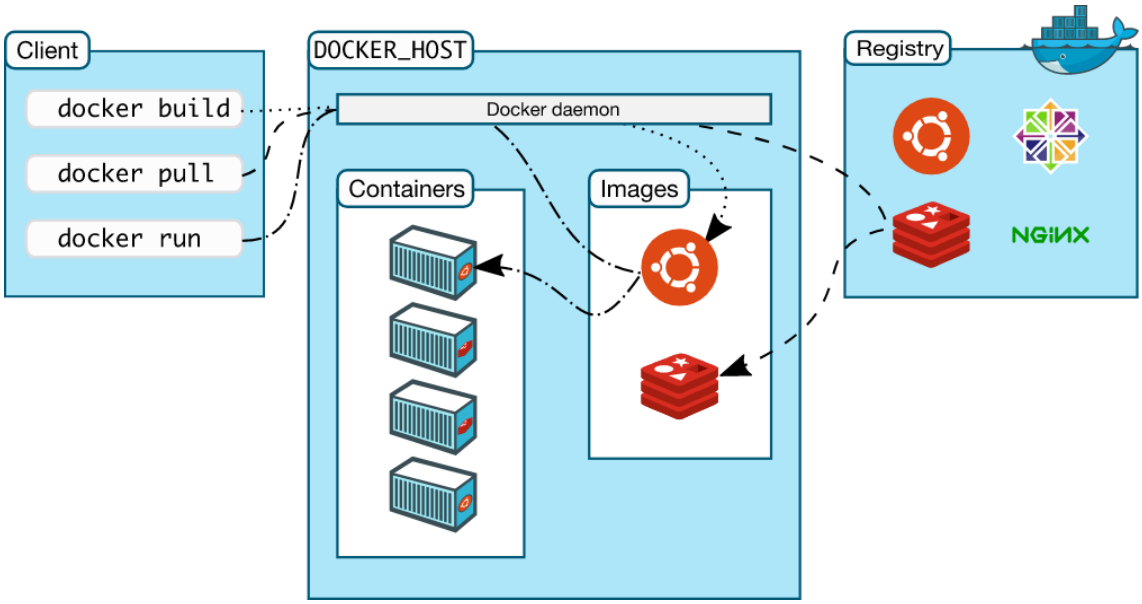
11. ábra: A virtuális gépek és a konténerek architektúrájának összehasonlítása [16]

Docker

A Docker egy nyílt platform, mely lehetőséget nyújt konténerek futtatására, csomagolására és szállítására/terjesztésére. A Docker és a hozzá hasonló konténerizációs keretrendszerek népszerűsége abból ered, hogy kellően széles eszköztárat biztosítanak a

konténerek kezeléséhez, ugyanakkor az egész folyamatot jelentősen leegyszerűsítik az által, hogy mintegy absztrakciós réteget kínál az alacsony szintű műveletek kezelésére. Segítségével könnyedén csomagolhatjuk konténerekbe a futtatni kívánt alkalmazásokat. Ez különösen hasznos lehet fejlesztők számára is, ugyanis rövid időn belül tudják a tesztelni kívánt kódot a szükséges dependenciákkal egy csomagba illeszteni, melyet azonnal futtatni is tudnak.

A rendszer magját a Docker Engine jelenti, mely lényegében a konténerizációs feladatok lebonyolításáért felelős motor, ő közvetít a felhasználó és a Docker daemon között. A Docker esetében is képfájlokat lehet létrehozni, melyekből a kész konténereket indíthatjuk. A képfájl már tartalmaz minden, az alkalmazás futásához szükséges fájlt, melyet aztán egyetlen paranccsal példányosítani is tudunk egy futó konténer képében. A korábban már taglaltak szerint jóval kevesebb erőforrásra van szükség egy konténer futtatásához, így természetesen sokkal több példányt tudunk indítani belőle, mint egy tradícionális VM-ből. A Docker képfájlok készítésének egyik módja az ún. Dockerfile alapján történhet. A Dockerfile egy, a Docker saját leíró szintaxisa szerinti leírófájl, melyben különböző instrukciók találhatók a Docker Engine számára egy konténer képfájl elkészítéséhez. Ebben megadhatjuk például, hogy a saját konténerünk mely másik konténerből induljon ki, pl. (teljesség igénye nélkül) egy alap Ubuntu vagy Alpine Linux képfájlból, milyen fájlokat másoljon bele a konténerbe, illetve a konténer indulásakor milyen processz kezdjen el futni, környezeti változókat definiálhatunk, egyéb, a buildelés során futtatandó shell parancsokat definiálhatunk benne, de lehetőségünk van vele kezelni perzisztens tárterület csatolását vagy különböző hálózati paramétereket is beállítani.



12. ábra: A Docker architektúrája [17]

Az általunk készített konténer képfájlok alapértelmezésben a saját gépen tárolódnak, ugyanakkor szintén használható erre a célra a Docker Hub, ahol különböző alkalmazásokról készült, illetve azok futtatását biztosító környezetekről készült publikus képfájlok érhetőek el. A saját konténereink szintén letölthetőek ide, melyek publikusságát is kezelni tudjuk.

Fontos megemlíteni, hogy a Docker konténerek pehelysúlyú működéséből adódóan, ha a bennük futó alkalmazások működésében hiba lép fel, az a gazda rendszer működését illetően semmilyen következménnyel nem jár, illetve az így kieső szolgáltatás helyére azonnal tudunk egy új konténer példányt indítani. Látható, hogy mivel a futó konténerek jönnek-mennek, ezért bizonyos, adatintenzív alkalmazások esetében kritikus feladat, hogy a konténerizált alkalmazás által létrehozott vagy általa használt adatok perzisztens adattárolása is megoldott legyen. Ugyanis egy konténer törlődésével a benne lévő összes adat is elvész, mely a futása során keletkezett. A Dockerrel lehetőség van ilyen perzisztens tárterületek csatolására is, melyhez egyszerre több konténer is hozzáférhet.

Kubernetes [18]

A konténerizáció előretörésével megjelent egy új architektúra az alkalmazások szervezéséhez: a mikroszolgáltatások-architektúra (microservices architecture). Ezen megközelítés szerint a korábbi, monolitikus szoftverrendszerek tervezésével ellentétben olyan architektúrák elkészítése a végcél, melyben az egyes, jól körülhatárolható

feladatkört gyakorló alkalmazás-komponensek (mikroszolgáltatások) egymással egy lazán csatolt közeget alkotnak. Ez azt jelenti, hogy bár mindegyik különálló, izolált egységként működik, ugyanakkor mégis egy egészként tudnak működni. Ezáltal az alkalmazás karbantarthatósága és fejleszthetősége is jelentősen javul, másrészt nem szorítkozik egyetlen speciális megoldásra az egyes feladatkörökben, hanem egy másik, ugyanolyan feladatot ellátó konténerizált alkalmazással helyettesíthető (pl. webszerver lecserélése Apache-ról Nginx-re). A konténerizáció ennek kiváló alapként szolgálhat, azonban egyedül közel sem elégíti ki az architektúra által támasztott összes kritériumot. Ahhoz, hogy egy ilyen komplex alkalmazás teljesértékűen működhessen, az egyes mikroszolgáltatásokat (konténereket) össze is kell hangolni, ha megnövekedik az igény az alkalmazás elérésére, akkor megfelelő mennyiségben fel kell skálázni a konténerek számát a terhelés egyenletes eloszlatása és a szolgáltatás elérhetőségének biztosítása végett, illetve valahogy azt is biztosítani kell, hogy a szolgáltatást a külső hálózatról is el lehessen érni. A konténerizációs keretrendszerek azonban nem, vagy csak bizonyos elemeket valósítanak meg a fentiek közül. Habár a Docker saját megoldása, a Docker Swarm is kellően komplex konténer-orkesztrációs keretet ad, nem talált olyan széleskörű támogatottságra, mint a riválisnak mondható Kubernetes. Utóbbi szélesebb körű támogatottságát - és ennek következtében nagyobb fokú fejlettségét - annak köszönheti, hogy a kezdetben még Borg névre keresztelt prototípusa a Google saját fejlesztése, melyet 2014-ben tettek nyílt forráskódúvá Kubernetes néven. A feladat elkészítésekor a mi választásunk is erre az eszközre esett, így, hogy a későbbiekben érthető legyen a feladat leírása, ismertetjük a Kubernetes architekturális és a feladat számára releváns tulajdonságait.

A Kubernetes architektúra[19]

A Kubernetes konténerizált alkalmazások telepítésére, felhasználói igényeknek megfelelő skálázására és menedzsmentjére szolgál, illetve mindezt a specifikus feladatot ellátó egységeinek köszönhetően automatizáltan teszi. Jellemzően a Kubernetes feladata, hogy konténerek összehangolását végzi, ezért konténer-orkesztrációs platformnak is szokás hívni. Maga a Kubernetes nem biztosít konténer futtatási környezetet, hanem már meglévőekre építkezik, ennek módját az erőforrások elemzésénél látni fogjuk.

Egy Kubernetes klaszterben kétféle számítási csomópont fordulhat elő: olyan amelyik a kontroll-síkot valósítja meg, illetve amely a számítási kapacitást biztosítja a

telepítendő konténerizált alkalmazásoknak. Előbbit Master node-nak nevezzük, ez valósítja meg a kontroll-síkot, utóbbit pedig Worker node-nak nevezik. Egy szokványos klaszterben általában egy Master és több Worker kap helyet, azonban bizonyos esetekben, ahol különösen fontos a klaszter és szolgáltatásainak magas rendelkezésre állása, ott több Master is alkothatja az architektúrát. A kontroll sík legfontosabb elemei a következők:

Kubernetes API Server

A kontroll-sík magja, amelyen keresztül a Kubernetes API-hoz lehet hozzáférni. Ez az egység a kontroll-sík központja, az összes vezérlő egység vele kommunikál.

ETCD tároló

Konzisztens és magas-elérhetőségű kulcs-érték tároló, mely minden klaszter specifikus adatot nyilván tart.

Kube Scheduler

A konténerek ütemezéséért felel, ha megjelenik egy új konténer, amelyik még nem került valamelyik Workerre ütemezesére, akkor a Scheduler keres számára egy megfelelő csomópontot, amely ki tudja szolgálni a konténer által támasztott erőforrás igényeket, illetve megfelel további kritériumoknak is.

Kube Controller Manager

Vezérlő folyamatok futtatásáért felel, ilyen kontroll folyamatok felelnek például az egyes csomópontok leállásának detektálásért, Job-ok futtatásáért, vagy szolgáltatások elérhetőségének biztosítására.

Cloud Controller Manager

Ha specifikus felhő környezetben fut a Kubernetes (pl. AWS, Google Cloud, OpenStack stb.), akkor ez az egység felhő-specifikus feladatokat, illetve kommunikációt végez a felhő-szolgáltató specifikus API-jával.

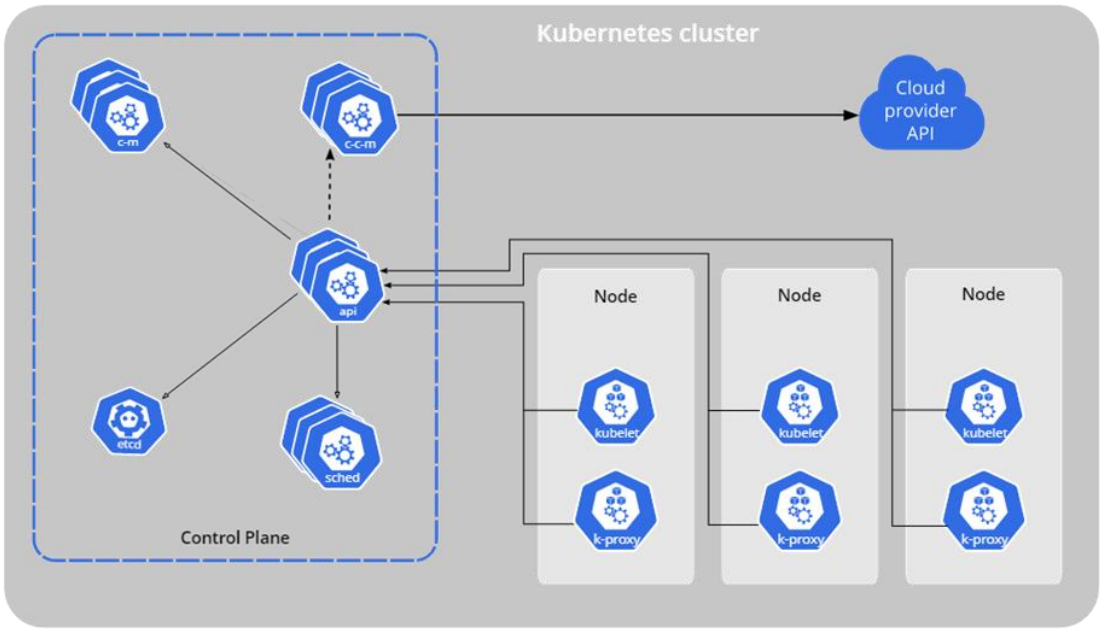
Kubelet

A klaszter minden worker csomópontján futó ágens, mely ellenőrzi, hogy az adott konténerek ténylegesen futnak-e.

Kube Proxy

A kube-proxy egy hálózati proxy, amely a klaszter minden csomópontján fut, és a hálózati szabályokat kezeli a csomópontokon. Ezek a szabályok teszik lehetővé

a hálózati kommunikációt a Pod-okkal a klaszteren belüli vagy kívüli hálózatok számára.



13. ábra: Egy Kubernetes klaszter sematikus vázlata.[20]

Több, a működését alapvetően befolyásoló erőforrás típust definiál, melyeket deklaratív leírók segítségével tudunk példányosítani, vagy meglévő példányokat konfigurálni, módosítani. Néhány erőforrás ezek közül, melyek a későbbiekben előfordulnak:

Pod

Konténerek egy csoportját jelenti. Absztrakciós szint a konténer futtatási környezet és a Kubernetes között. A Kubernetes által ütemezhető legkisebb, legalapvetőbb egység. Egyszerre több konténer is lehet egy Podban, azonban a gyakorlat azt mutatja, hogy általában 1:1 megfeleltetést használnak Pod és konténer között. A több konténeres megközelítést csak olyan esetben szokás használni, ha a konténerek feladata szorosan összekapcsolódik egymással, mégis különálló egységként kell üzemelniük (pl. az alkalmazás konténer mellett egy logokat gyűjtő és azokat egy külső adatbázisba eljuttató konténer). Ezt a lehetséges tervezési mintát sidecar-nak szokás nevezni, a későbbiekben látható implementációink is főként erre a tervezési mintára építenek. A Podnak saját erőforrásai vannak allokálva (CPU, memória), saját hálózati névtérrel és tárhellyel

is rendelkezik, illetve tartalmazhat különböző megkötéseket a konténerek futtatására.

Deployment

A Deploymentben valamilyen kívánt állapotot írunk le, melyet a Deployment Controller igyekszik betartatni kontrollált ütemben. Egy Deploymenttel létrehozhatunk egy ReplicaSetet, (egy ReplicaSet azért felel, hogy meghatározott mennyiségű replika fusson egy azonos Podból) meglévőeket mozgathatunk a Deployment alá, frissíthetjük a Podoktól elvárt állapotot/működést, feljebb vagy lejjebb skálázhatjuk a futó replikák számát.

DaemonSet

Egy DeamonSet feladata biztosítani, hogy a klaszter minden (vagy néhány) node-ján fusson egy adott Pod egy példánya. Ahogy skálázodik időközben a klaszter csomópontjainak száma, a DaemonSet által kezelt Podok száma is növekszik, és ugyanez igaz ellenkező esetben is: ha esnek ki node-ok a klaszterből, a futtatandó replikák száma is csökken.

Service

Absztrakciós réteg, mely a Podok által nyújtott szolgáltatásokhoz való hozzáférést biztosítja, történjen az a klaszteren belülről vagy kívülről. Bár a Pod maga is rendelkezik IP címmel, azaz közvetlenül is elérhetnénk a Pod által nyújtott szolgáltatást, ha ez törlődne és helyette egy új jönne létre, akkor semmi nem garantálja, hogy ugyanazt az IP címet fogja kapni az új példány. A Service ennek megkönnyítésére szolgáltat egy fix IP-t, és az erre beérkező kéréseket pl. egy meghatározott kulcs-érték párral jelölt Podokhoz továbbítja. Ezen felül képes az egyes felügyelt Podok között terhelés elosztást is végezni. Többféle Service típus is létezik, mint ClusterIP, NodePort, LoadBalancer, Headless Service stb.

Ezeken felül a Kubernetes lehetőséget ad arra, hogy a Kubernetes API kiterjesztéseként saját erőforrás típusokat (CustomResources) is definiáljunk. Ezek definiálására a CustomResourceDefinition API nyújt segítséget, melyet az Operátorok esetében fogunk bővebben kifejteni. A Kubernetes esetében a különböző erőforrások kezelését deklaratív úton végezhetjük el. Azaz az adott erőforrás elérni kívánt állapotát

írjuk le, a rendszer pedig ezen adatok alapján mindent megtesz, hogy a kívánt állapot előálljon. Ezt leggyakrabban YAML formátumú leírófájlokkal tesszük meg. Ezt demonstrálandó, egy minta Pod és Service leíróját lehet alább látni.

```
apiVersion: v1
kind: Pod
metadata:
 name: test-pod
 labels:
   app:nginx
spec:
 containers:
 - name: nginx
   image: nginx:1.14.2
   ports:
   - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
 name: test-service
spec:
 selector:
   matchLabels:
     app: nginx
 ports:
   - protocol: TCP
     port: 80
     targetPort: 8081
```

A leírókban az alábbi paramétereket kötelező megjelentetni és értékkel ellátni őket:

apiVersion: a Kubernetes API mely verziójával szeretnénk az erőforrást létrehozni

kind: a létrehozandó objektum típusát jelenti

metadata: adatok, amelyek alapján az erőforrás könnyebben azonosítható, a name megadása kötelező, ezen felül label, namespace stb. is megadható

spec: az objektum elvárt állapotának leírása

Ezek tudatában könnyen dekódolható a fenti leírók tartalma: Létre szeretnénk hozni egy test-pod nevű Pod típusú objektumot, amelynek az elvárt tulajdonságai, hogy tartamazzon egy nginx nevű, nginx webszervert futtató konténert, amelyhez a Pod hálózatán a 80-as porton keresztül férünk hozzá. Illetve szeretnénk egy test-service nevű Service típusú objektumot is, mely azon erőforrásokhoz nyújtson hozzáférést, melyek az app: nginx kulcs-érték párral rendelkeznek. Továbbá a Service a felügyelt Podok konténereinek 80-as portja felé irányítsa azokat TCP alapú kéréseket, melyek a Service 8081-es portjára érkeznek be.

Hálózatkezelés a Kubernetesben

A Kubernetes hálózatkezelését néhány fontosabb pontban össze lehet foglalni:

- Minden Pod saját IP címmel rendelkezik, melyet a klaszteren belülről bárhonnan el lehet érni.
- Két különböző csomóponton lévő Pod kommunikációjához nincs szükség címfordításra (Network Address Translation).
- Az adott node-on futó ágensek (pl. Kubelet) minden, ugyanezen node-on futó Pod-ot elérnek.
- Egy adott Podon belüli konténerek közös hálózati névtérrel rendelkeznek.

Bár a fenti szabályok meglétét elvárja a Kubernetes a hálózatkezelést tekintve, ugyanakkor azt nem rögzíti, hogy ezek milyen módon legyenek implementálva. Emiatt nem alakult ki egyetlen, szabványos, klaszteren belüli hálózatkezelési implementáció.

A Kubernetes networking alapját a Container Networking Interface (CNI) pluginek jelentik. Egy ilyen plugin telepítésére van szükséges ahhoz, hogy a klaszteren belüli hálózat kiépülhessen. A CNI pluginek feladata a hálózati interfészek létrehozása a konténerekben, a számukra való IP cím allokálás, illetve a megfelelő routing szabályok beállítása. A CNI pluginek olyan programok, amelyek adott hálózati konfigurációk alkalmazására képesek egy konténerben. Léteznek általános célú és alacsonyabb funkcionalitással rendelkező pluginek is [21], de komplex funkcionalitással rendelkezőek is, ilyenek például a Calico, Flannel, Weave vagy akár a Cilium nevű, harmadik felektől származó, konténer hálózatkezelő szoftverek. Azonban ezek az eszközök csak a klaszter kontextusában működnek, és a külső hálózatokkal történő konnektivitás nem tartozik a hatáskörükbe. Fejlett hálózati infrastruktúra kiépítésére és menedzselésére a

későbbiekben taglalt eszközök nyújtanak megoldást, legyen az az általunk készített implementáció, vagy harmadik féltől származó beépülő.

Hálózat virtualizálás és felhő-natív funkciók

A hálózat virtualizálásának ötlete (Network Function Virtualization, NFV) bár csak később született meg, mint az előzőleg említett virtualizációs technológiák, legalább ugyanakkora potenciállal rendelkezik. A következő mobilhálózatok szempontjából is szükséges a paradigmaváltás, ugyanis a kitűzött teljesítménybeli célokat egy rugalmas, skálázható és programozható hálózati platform segítségével lehetne csak megvalósítani. [22] Röviden áttekintjük, milyen technológiai előrelépések tapasztalhatók ezen a téren, és hogyan alkalmazhatóak mobilhálózatok tekintetében.

A számítógép hálózatokban a kezdetek óta specializált hardver-szoftver rendszerek teszik lehetővé a hálózatok működését. Napjaink hálózatait is még főként ilyen eszközök üzemeltetik (pl. switch, router), melyek gyártására és tervezésére az évtizedek során több meghatározó cég is specializálódott. Ennek kezdetben az volt az oka, hogy könnyebb, illetve az eszköz teljesítményét tekintve hatékonyabb volt speciális hardveren implementálni a kívánt működést, azonban ahogy egyre nagyobb számítási kapacitást tudhatnak magukénak a különböző általános célú szerverek, ez már szinte nem számít tényezőnek. Ennek hatására sorra jelentek meg a különböző megközelítések a hálózat statikus mivoltának minimalizálására.

Egyik ilyen újszerű megközelítés volt a Software Defined Networking (SDN). Ennek alapötlete a hálózat kontroll- és adatsíkját szeparálása, a kontroll logikájának programozhatósága és egységesítése, majd a két réteg között jól definiált, nyílt interfészek létesítése. Egy ilyen ígéretes próbálkozás az OpenFlow protokoll. A vezérlési síkot itt egy általános célú szerver szolgáltatja melyben a csomagtovábbítási logika implementálásra kerül, az adatsíkot pedig OpenFlow-képes switchek alkotják, amelyek a hagyományos L2 switchekkel ellentétben az OpenFlow flow-bejegyzések felépítéséből adódóan L3, sőt L4 beli paraméterek (pl. forrás IP cím vagy cél TCP port) mentén is képesek a forgalom továbbítására. Azonban az SDN egyik nagy hátránya pont a centralizáltságából adódik, ami a kontroll sík meghibásodásával azt vonná maga után, hogy ezek a switchek képtelenek lennének ellátni a feladatukat. Az ilyen és hasonló problémák megoldására azóta több megoldás is született, például, hogy az adatsík programozhatóságára is már nagyobb hangsúlyt fektetnek.

Az NFV a már korábban említett virtualizációs technológiákra épít. Az alap koncepció szerint a különböző ún. middlebox hálózati eszközök, például terheléselosztók vagy tűzfalak szoftveres implementációi általános célú hardveren kerülnek futtatásra, virtualizált módon. Ez megoldást jelent a skálázhatóság problémájára, ugyanis a hálózati forgalom változásával arányosan lehet újabb virtuális gépeket indítani. Azonban a virtuális gépek esetében tapasztalt hátrányok természetesen itt is felbukkannak. A Virtualized Network Function mellett ezért egyre inkább használják a Cloud-native Network Function kifejezést is. Ez utóbbi is hálózat virtualizálási technológia, azonban a hálózati funkciók virtualizálása már konténerizáltan történik, jellemzően felhő architektúrákban.

Ez a két technológia alapjaiban bár eltér egymástól - azaz míg az SDN a hálózati folyam továbbítására nyújt egy modern megközelítést, addig az NFV a csomagok feldolgozására koncentrál - az együttes használatuk rengeteg előnnyel járhat. Ennek a lehetőségére az 5G hálózatok kapcsán is felfigyeltek, ugyanis bár az SDN megnyitná az utat olyan technológiák, mint a Network Slicing megvalósításához, hiányzik néhány alapvető képesség, melyeket az NFV biztosíthatna.[23]

4 Kitűzött feladat

A fontosabb fogalmak ismertetése, és a problémakör kontextusba helyezése után ismertethetjük a dolgozatunk céljaként kitűzött feladatokat. Feladataink alapját az adja, hogy milyen lehetőségek vannak felhő alapokon üzemeltetett távközlési alkalmazások hálózati infrastruktúrájának dinamikus kiépítésére. Ezt a problémát két irányból fogjuk megközelíteni.

Az egyik irány egy saját, a Kubernetes Operator tervezési mintát megvalósító szoftverimplementáció készítése. Itt egy leegyszerűsített MIPv6 implementációt fogunk megalkotni, amit konténer alapokon tudunk majd futtatni. A konténerként nyújtott szolgáltatást az operátor fogja menedzselni, amihez deklaratív kéréseket intézhetünk, hogy mennyi Home Agent példányt hozzon létre. Ezeknél két felállást is megvizsgálunk, mindkettőnél teljesítménytesztet hajtunk végre és ezek alapján következtetéseket vonunk le.

A másik megközelítés esetében már létező, a Kubernetes képességeit kiegészítő szoftvereket vizsgálunk meg. Segítségükkel dinamikusan építhető ki egyedi adatsík konténerek között, melyre akár klaszterek között is képesek. A feladat, hogy mind funkcionalitás, mind teljesítmény szempontjából megvizsgáljuk ezeket az eszközöket, majd különböző teljesítménybeli szempontok mentén méréseket végezzünk. Ezt követően pedig a kapott eredmények alapján levonjuk a következtetéseket a technológia valós szcenáriókban történő alkalmazhatóságáról.

5 Kubernetes Operátor-alapú megközelítés

A végcél egy olyan szoftverkörnyezet létrehozása a Kubernetes klaszterben, amiben automatikusan létrejön az adatsík egy mozgó terminál és egy Home Agent között. De ennél egy kicsit tovább is megyünk.

Egy telekommunikációs hálózatról beszélünk, amiben napjainkban új trend van kialakulóban, mégpedig a skálázhatóság biztosítása virtualizált környezetben. És a milliszekundumos válaszidő természetesen nem enged meg humán közbeavatkozást az adminisztrátorok részéről.

Ezeket észben tartva, mi lenne, ha a klaszter kérésre a terheléssel arányosan akármennyi Home Agent futtatási egységet létrehozhatna? Mi lenne akkor, ha ezeket automatikusan karbantarthatnánk? Ha ezeket a kérdéseket tesszük fel, akkor eljutottunk a Kubernetes Operátor fogalmához.

A probléma megoldásához vezető egyik utat saját operátorok létrehozása jelenti, ebben a fejezetben ennek a megközelítésnek a részleteit fogjuk megvizsgálni. Először bemutatjuk az elméleti alapokat és a szükséges eszközöket. Majd megvizsgáljuk az architektúra lényegi részeit, kezdésként magasabb szinten, rendszerkomponensekre vetítve, majd az implementációs részletekre is rátérünk. A kidolgozott rendszeren funkcionális és performancia-teszteket hajtunk végre, amik alapján konklúziót vonunk le. Végezetül pedig a továbbfejlesztési lehetőségeket is megvizsgáljuk.

Kubernetes Operátorok

Ezek az entitások nagyon hasonlítanak humán megfelelőjükhöz, bár komplex problémamegoldásra nem képesek, de ahogy azt látni fogjuk, erre nem is lesz szükség. Ha az operátorok munkáját szeretnénk a legabsztraktabb módon leírni, akkor arra jutunk, hogy a hálózat jelenlegi állapotát, a státuszát, alakítják át valamilyen módon az elvárt állapotra, ezt az átmenetet pedig folyamatosan biztosítják, az állapotot egyfolytában ellenőrzik.

A jelenlegi állapot olyan szempontból fontos, hogy megtudjuk, miben van szükség módosításra, milyen jellemzőket kell elmozdítanunk milyen irányba. Az egyetlen elvárás, hogy ezek numerikusan kiértékelhetők legyenek. Elvárt állapot ezzel szemben

lehet mondjuk, hogy a két végpont eléri egymást, vagy akár kettőjük közötti kommunikáció mikéntje, mint például IP tunnel, vagy egyszerű, natív csomagkapcsolat.

A jelenlegi és az elvárt állapot közötti átmenet megtestesítője lesz a Kubernetes Operátor. A Kubernetes ezeket szoftver kiegészítőnek definiálja, amivel bővíthetjük klaszterünk funkcionalitását anélkül, hogy újraírnánk a Kubernetes rendszer részeit. Emiatt akár egy interfészként is tekinthetünk rá, amelyen keresztül érvényesíthetjük akaratunkat a klaszterben történtekkel kapcsolatban. Sok esetben még klaszteren kívüli operációkra is használják, de ezzel dolgozatunkban nem foglalkozunk.

Működésük

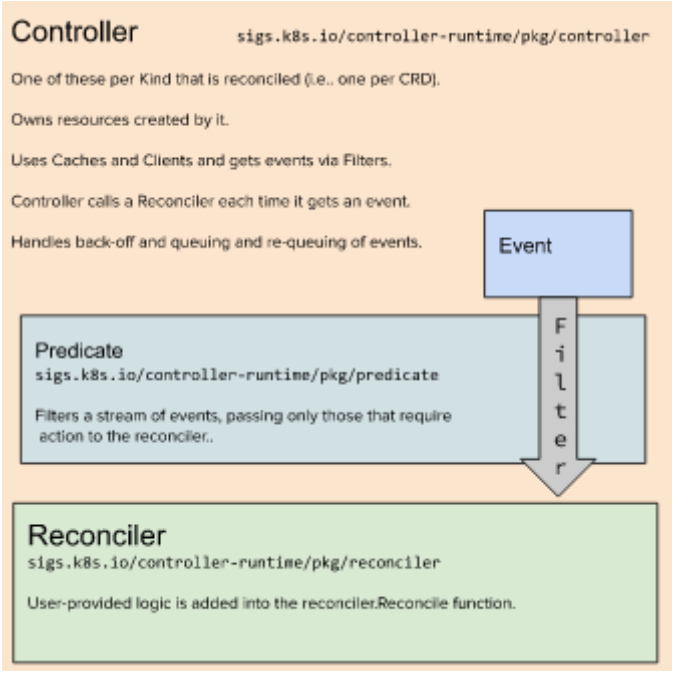
Először próbáljuk meg elképzelni, mire van szükségünk egy ilyesfajta operátor működéséhez. Először is kell egy erőforrás, amit figyelünk. Ennek rendelkeznie kell egy jelenlegi és egy elvárt állapottal, hogy azon operációkat végezhessünk. Végül, de nem utolsó sorban, kell egy logika, ami a megfigyelt objektumot átvezeti egyik állapotból a másikba. Ezt a fajta architektúrát hívja a Kubernetes operator patternnek, és a következőkben ezt fejtjük ki részletesebben.

Ha ránézünk a Kubernetes dokumentációjára, kiderül, hogy valóban az előbb felsorolt elemekből indul ki az operator pattern, csak másképpen nevezi őket. Az erőforrást Custom Resource-nak nevezi, amikor létrehozzuk ezt a Custom Resource-ot, akkor egy Custom Resource Definition-t küldünk el a Kubernetes API-nak, ami értelmezi és engedélyezi az ilyesfajta erőforrások létrehozását a klaszterben. Egy Custom Resource további két fő részre osztható, a spec és a status elemekre. Az előbbi testesíti meg az elvárt állapotot, míg az utóbbi a jelenlegit. Már csak ennyivel és csinálhatunk saját erőforrásokat a klaszterben, de csak egy custom controller segítségével válik elérhetővé az igazi deklaratív API.

A custom controller pedig nem más, mint egy control loop, amivel olyan területeken, mint a robotika és az automatizálás már találkozhattunk. Ezek lényegében végtelen iterációk, amik figyelik és vezénylik a rendszer állapotát.

A lenti ábra egy kicsit bonyolult, de a lényeget jól kiemeli. A Kube API-ból érkező eseményeket a Predicate-nek jelölt elem a Controllerben kapja meg elsőként, amiben feltételek fogalmazódnak meg. Ezeket a feltételeket előzetesen deklarálhatjuk, a Predicate pedig megszűri ezeknek megfelelően az eseményfolyamot. A Reconciler-re nehéz magyar nevet hozni, talán összehangolóként fordítható, de ez csak zavaró lenne,

így a továbbiakban is Reconciler-ként hivatkozunk rá. Ahogy azt láthatjuk, itt írjuk majd meg a fő operátor logikát.



14. ábra: Kontroller felépítése [24]

Idempotencia

Ez a felépítés már megfelelő az erőforrásunkról érkező eseményekre való reagálásra, de érdemes megfelelően átgondolni a logika megvalósítását. Nagyon fontos, hogy ki kell kényszeríteni az idempotenciát. A számítástechnika területén idempotensnek nevezünk egy eljárást akkor, ha többször meghívható ugyanolyan paraméterekkel anélkül, hogy az negatív mellékhatásokat okozna. Ilyen például a HTTP DELETE metódus. Ha kétszer hívjuk meg a törlést egy szerveren tárolt eljárásra, akkor nem változik a szerver állapota a második hívásra, csak az elsőnél törlődött egy elem. A válasznak természetesen nem kell ugyanolyannak lennie, a HTTP DELETE esetén is másodjára 404-es hibaüzenetet fogunk kapni.

Az idempotencia azért lesz nagy jelentőségű kérdés, mert egy felhő környezetben gyorsan változik a komponensek állapota és előfordulhat, hogy többször meghívódik egymás után a kontroller. Emiatt pedig kellemetlen mellékhatásokkal is szembesülhetünk, ha nem idempotens algoritmusokat írunk és nem alkalmazunk erősen defenzívkódolást. Legrosszabb esetben manuális közbeavatkozásra is szükség lehet.

Sidecar konténerek

A bevezetőben megemelítettük a sidecar konténereket, ezt fogjuk megvizsgálni egy kicsit közelebbről. A Kubernetes architektúrában ajánlatos minden független konténert egy külön Podba tenni, hiszen a Podok egy izolált feldolgozási egységet kepviselnek. Azaz egy több konténert tartalmazó Pod esetén a konténerek egy hálózati névtérben lesznek és osztoznak a felcsatolt lemezterületeken is.

Általánosságban nem előnyös ez a fajta osztottság, csak abban az esetben ajánlatos, ha egy fő konténer mellett kisegíti a másik munkáját, mint pl. egy Logger. Különböző Service Mesh implementációk, mint például az Istio, ezt a sémát használják a kontroll síkjaik működtetésére, különböző naplózó és biztonsági funkcióik támogatására.

A mi esetünkben is előnyös lehet ez a fajta elosztás. Ha belegondolunk, akkor a Home Agent szolgáltatás futhatna sidecar konténerként is, ami mondjuk az Access & Mobility Management Function működését, mint fő szolgáltatás kisegíti. Ez a felépítés lesz az egyik irány, amit tesztelünk a dolgozatban. A másik, amikor független Podként hozzuk létre a Home Agent-et. Előbbi esetben nagyobb kényelemre, utóbbiban kisebb késleltetésre számítunk.

Kernel programozás

A Kubernetes rész után érdemes a Linux kernel irányba is elmennünk. Fontos kérdés lesz számunkra az adatsík kihúzásnál, hogy hogyan tudjuk ezt algoritmikus módon, effektíven megtenni. Mindenekelőtt arra van szükségünk, hogy tudjunk kommunikálni a kernellel, közvetíteni tudjuk számára, milyen tunnelt és milyen címre hozzon létre számunkra.

Itt eszünkbe juthat a Linuxban széleskörűen kedvelt két hálózatkezelő CLI (command line interface) csomag, az iproute2 és a ma már elavult net-tools. Mindkettő nagyon kedvelt a hálózat adminisztrátorok körében, mert egyszerű interfészeket nyújtanak a felhasználónak, amivel képesek a legtöbb hálózati beállítást a terminál ablakából megcsinálni. Érdemes megnéznünk ezért közelebbről őket.

Ha megnézzük a kettő forráskódját, akkor azt látjuk, hogy két, egymástól sokban különböző módot használnak a kernellel való interfészelésre. A net-tools a régebbi ioctl könyvtárat veszi igénybe, míg az iproute2 az előbbi utódjaként emlegetett netlink könyvtárat.

Ioctl

Az ioctl-ről érdemes pár szót ejteni. A betűszó feloldása Input/Output Control, és ahogy azt neve is sejteti, lényegében eszköz specifikus operációkra használhatjuk. Érdekesség, hogy Microsoft Windows környezetben a Win32 API is hasonló interfészt nyújt a felhasználóknak az eszköz driverekkel való kapcsolatra, amit DeviceIoControl-nak nevez.[25], [26]

Ioctl hívásoknak három fő aspektusa van: egyrészt egy nyitott fájl leíró (file descriptor), másrészt a híváskód, illetve maga az üzenet azt követően. Esetünkben a netdevice eszközt kellene használnunk, amivel az adott interfész különböző paramétereit manipulálhatjuk. Csak meg kell adnunk a megfelelő híváskódot, amit a netdevice definiál és azután paraméterként átadjuk az üzenetünket.

A probléma ezzel a megközelítéssel, hogy már elavult a Linux kernelben az ioctl, csak régi szoftverek támogatására, mint a net-tools, maradt része a felhasználói könyvtáraknak.

Sok limitációt fedeztek fel vele kapcsolatban az évek során. Egyik ilyen, hogy az ioctl egy szinkron operáció, ezzel effektíve rákényszerítjük a kernelt, hogy azonnal végezze el a kérésünket. Továbbá amint arra is szükség van, hogy a kernel továbbítson számunkra üzenetet, már erőforrást nem kímélő, pollozós technikához kell folyamodnunk periodikus ioctl hívásokkal.

Összességében azt mondhatjuk el, hogy egy nem igazán flexibilis, de robosztus megoldás volt az ioctl. A mai felhasználói igények azonban már egy újfajta gondolkodásmódot igényelnek.

Netlink

A netlink az előzőekhez képest sokkal szabadabb felépítést használ. Míg az ioctl egy függvény hívással analóg, addig a netlink inkább IP csomagok küldésével foglalkozik a hálózaton, ahol a végpontok adott futó programok.

Láthatjuk, hogy a merev ioctl-lel szemben kibővült a lehetőségek tárháza. Képesek vagyunk adott kernel funkciókkal úgy beszélni, mintha egy távoli gépen futnának, de támogat multicast elérést is. A multicast eseménykezelők esetén lesz ez a funkció nagyon hasznos, mert léteznek előre definiált multicast csoportok, mint például az `RTMGRP_LINK`, ami az interfész létrehozás, törlés, lekapcsolás, felkapcsolás

eseményeket adja tovább. Ezekre feliratkozva rögtön megkapjuk az eseményről a kellő információt és megfelelő lépéseket tehetünk. Ez egy sokkal kényelmesebb mód az ioctl-es pollozással szemben.

Mivel lényegében üzeneteket küldünk, ezért egy teljesen aszinkron operációról beszélhetünk. Azaz a maga idejében tud a kernel válaszolni kérésünkre és nem kell rögtön felébreszteni. Ezzel egy kicsit spórolhatunk a költséges kontextusváltásokon.

A legjobb része a netlinknek talán az, hogy ez egy általános inter-process communication-re kialakított megoldás. Tehát a kernel csak egyike a lehetséges célpontoknak és egyszerre több célpontnak is küldhetünk üzeneteket a multicast címzéssel. Lássuk is, hogy hogyan működik.

Fontos elmondani, hogy az aszinkronitás és a nagymértékű flexibilitás mind megnehezítik a netlinkkel való programozást. Ezért érdemes megfelelő körültekintéssel használni ezt a könyvtárat.

Netlink alkalmazása

Először is nyitnunk kell egy új socketet, amit a sys/socket.h-ban definiál a kernel, ennek pedig egy speciális értéket kell megadnunk, hogy netlink üzenetek fogadására alkalmas legyen. Ezt a következőképpen tehetjük meg:

```
nl_sock = socket(AF_NETLINK, socket_type, netlink_family);
```

Az AF address familyt rövidít, tehát netlink címcsaládot továbbító socketet nyitunk meg. Itt fontos elmondani, hogy a socket eredeti definíciója szerint PF_* értékeket várna, ahol a PF feloldása protocol family, de az AF_* és PF_* értékek egyenlőek. Erről megbizonyosodhatunk a sys/socket.h fájlt átböngészve.[27]

Értelemszerűen adódik a kérdés, hogy mi szükség erre a megkülönböztetésre. Amint kiderült, régen arra készültek a szakemberek, hogy egy protokollcsalád több címzésmódot is támogatni fog, de ez sosem történt meg, így ez a fajta szétválasztás szükségtelenné vált.[28]

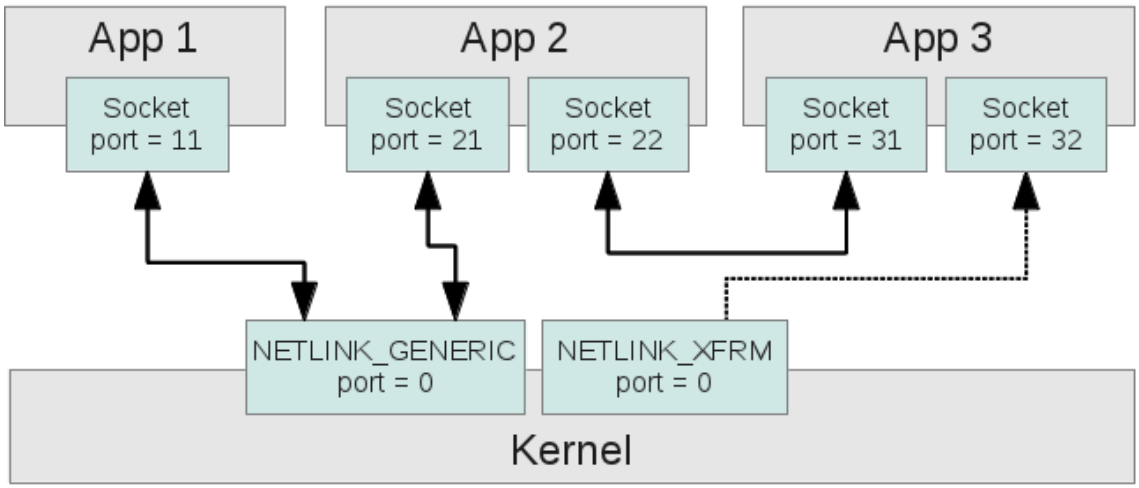
A socket_type argumentum a továbbítás módját definiálja, itt három fontos érték van: `SOCK_STREAM`, `SOCK_DGRAM` és `SOCK_RAW`. Első esetben kapcsolatalapú továbbításmódot támogat a socket, mint a TCP, másodikban kapcsolatmentes, mint az UDP. `SOCK_RAW` esetén pedig nyers hálózati protokoll hozzáférést biztosít számunkra.

A többi típus kivételesebb esetekben használatos, ezekről a belinkelt manpage-en bővebben is olvashatnak.

Esetünkben a netlink datagramalapú protokoll, így legajánlatosabb a `SOCK_DGRAM` érték, de a `SOCK_RAW` is elfogadható, a kettő között nincs megkülönböztetés.

A következő a netlink_family argumentum, aminek a célunknak megfelelő értéket kell adni. Ez lehet `NETLINK_ROUTE`, `NETLINK_USERSOCK`, `NETLINK_FIREWALL` és még sok más, a netlink manpage-e részletesen felsorolja ezeket.[29]

A létrejött socketeknek egyedi portokat állíthatunk be, amiken keresztül elérik egymást. Egy speciális érték a nullás portszám, amit kernel API-ja birtokol. Ezeket a következő ábra szemlélteti is.



15. ábra: Netlink portarchitektúra [30]

A netlink protokoll támogatja multipart üzenetek küldését is, ekkor több fejlécet és rakományt láncolunk egymás után, ekkor minden üzenetre be kell állítani a `NLM_F_MULTI` flaget, kivéve az utolsó üzenetnél, aminek a típusa `NLMSG_DONE`.

A netlink üzenetek formátumát az A függelékben részletesen is átnézzük, illetve a B függelékben egy gyakran használt iproute2-es parancs netlink megvalósítását is átvesszük, amivel mélyebb betekintést nyerünk a netlink könyvtár működésébe.

Raw Socket

Már képesek vagyunk a kernellel kommunikálni, itt az idő, hogy megtanuljunk távoli hosztokkal is. TCP és UDP esetén erre a kernel sok segítséget ad, de ha egy olyan

alkalmazást akarunk használni, ami már túllép ezen a TCP/UDP stacken, akkor magunknak kell implementálni ezek fogadását nyers socketeken.

Egy raw socket sok dologban hasonlít az egyszerű UDP socketekhez datagram szolgáltatásukból adódóan, a különbség annyi, hogy kikerüli a szállítási réteget. Emiatt hozzáférést kapunk ahhoz, ami közvetlenül az IP réteg után jön. A mi esetünkben főleg azért alkalmazzuk ezt a módszert, mert nem mozdulunk ki az internet stack hármas rétegéből, hiszen a MIPv6 Mobility Header-je egy IPv6-os kiegészítő fejléc. Ezt észben tartva a készülő MIPv6 implementációnknak szüksége van raw socketekre.

IPv4 és IPv6-os raw socketek

Az IPv4-es API nem volt teljesen konzisztens, amit főleg a IP_HDRINCL socket opció okozott. Ezt megadva, üzenet fogadásnál megkaptuk az IP fejlécet is, küldésnél pedig annyit jelentett, hogy a fejléc létéről a felhasználó fog gondoskodni. Ez abból a szempontból nem volt előnyös, hogy nehéz volt eldönteni, melyik rétegben is dolgozunk egy raw sockettel. Mindenképpen szükségünk van valamilyen szintű kontrollra a L3-as fejléc felett, de nem ennyire. Ha magunk akarunk IP fejlécet is létrehozni, akkor már egy L2-es socketben kell gondolkodnunk.[31]

Az új IPv6 API már ezt kijavítja, az előbb említett socket opciónak nincs hatása egy IPv6-os socketen. Helyette új socket opciókat hoztak létre, amikkel a fejléc legtöbb mezőjét beállíthatjuk, illetve kisegítő adatokon (ancillary data) keresztül is megtehetjük. Az utóbbi egy haladó koncepció, nem fogjuk használni.

A fejléc manipuláción kívül fontos elmondani, hogy a kernel nem fogja automatikusan kitölteni checksum mezőt, hacsak nem ICMPv6 üzenetről beszélünk, ahol kötelezően megteszi. Ez abból adódik, hogy IPv6-os csomagok nem tartalmaznak checksum mezőt, ezt a felsőbb rétegek fogják adni, éppen ezért változó helyeken lehetnek a checksum mezők.

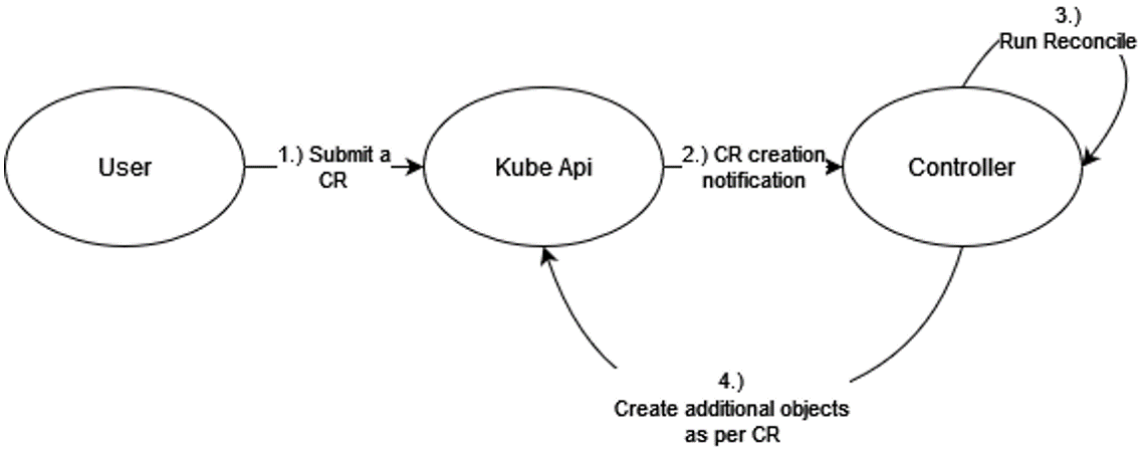
A checksum beállítását és a raw socket sajátosságaival a C függelékben foglalkozunk, itt részletesen is megnézzük, hogyan kell socketet nyitni, azon csomagokat fogadni, illetve csomagokat küldeni.

Megoldás architektúrája

Az elmélet végére értünk, itt az idő elgondolkodni a készülő rendszer architektúráján. Először áttekintjük a Kubernetes oldalbeli teendőket, majd áttérünk a MIPv6 szoftver tervezésére.

Kubernetes Operátor

Nézzük meg először, hogy fog kinézni az útja egy általunk definiált erőforrásnak a Kubernetes klaszterben.



16. ábra: Kubernetes Operátor életciklusa

Ez az ábra csak áttekintésre szolgál, nem teljesen így néz ki a valóságban. De azt könnyen láthatjuk, ki mivel és hogyan kommunikál. A központi szerepet az API veszi fel, minden innen indul ki és itt ér véget. Az ábrán két fontos dolgot kell magunknak implementálni: a Custom Resource-t és magát a controllert.

A Kubernetes oldalról tehát szükségünk lesz egy erőforrásra, annak elvárt és jelenlegi állapotára. Ezt yaml formátumban adjuk majd meg, amivel a bevezetőben már megismerhettünk. Kezdésnek érdemes egy vázlatot írni, hogy mit szeretnénk megadni a Home Agent példányoknak.

```
kind: HomeAgent
metadata:
    name: ha-sample
spec:
    size: 10
```

A fő szempont a méret lesz, így elég az elvárt állapotnak megadni egy egész számot, hogy hány példányt szeretnék látni a hálózatban. A spec segítségével így megadtuk, milyen legyen az elvárt állapot, a jelenlegi állapot pedig fontos, hogy ennek eleget tegyen, ezt azonban az operátor logikájában fogjuk szabályozni.

Ezzel megvan a megfigyelt erőforrás, már csak a logika hiányzik. Ahogy azt említettük, muszáj idempotens algoritmust írnunk, azaz meghívható legyen többször ugyanúgy anélkül, hogy negatív mellékhatásai legyenek.

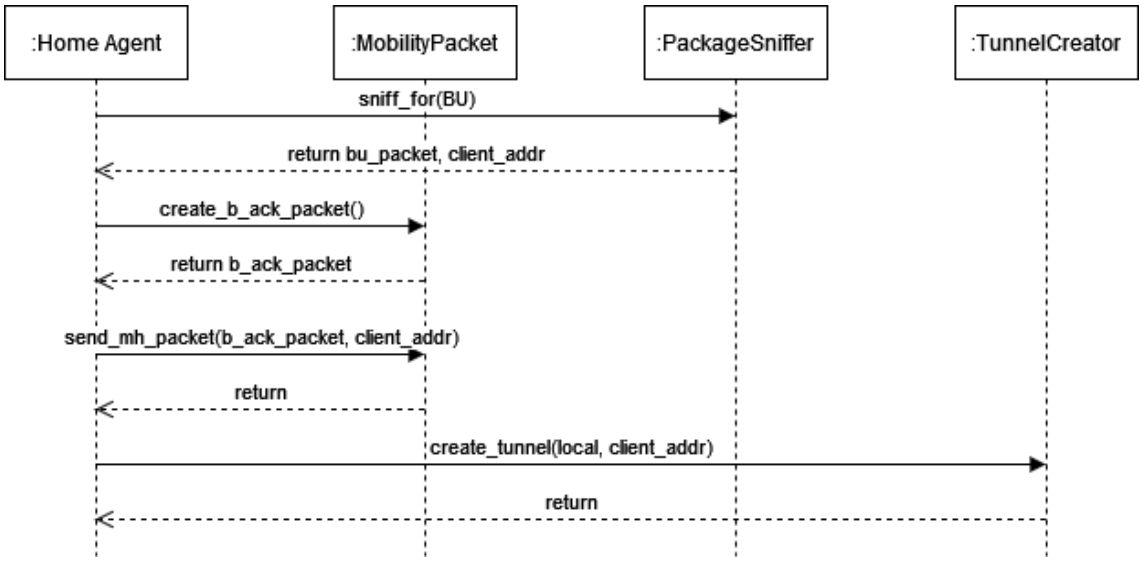
```
Ez a program végigmegy a jelenlegi állapoton és megfelelő
változtatásokat tesz, hogy közelebb hozza az elvárt
állapothoz a klasztert.

Initialise cr
Initialise timeout as 800ms
Initialise pods as Get pods under the name cr.metadata.name
Initialise podIPs as emptyList

If pods is empty do
    Create pods
    Requeue reconcile
end

If not all pods are ready do
    Requeue reconcile after timeout
end

For each pod in pods do
    If pod.PodIP is empty do
        Requeue reconcile after timeout
    end
    Append pod.PodIP to podIPs
end

Update cr.status.podIPs to podIPs
End reconcile
```

A fenti pszeudokódban láthatjuk, hogy először megnézzük, hogy egyáltalán léteznek-e a beküldött Custom Resource-hoz tartozó podok a klaszterben. Ha nem, akkor ezeket létrehozzuk, majd újraindítjuk a control loopot. Ez a lépés talán meglepő, de nem az a cél, hogy egy iteráció rögtön az elvárt állapotra hozza a klasztert. Ha elvégeztünk

egy lépést, mindig újraindítjuk, mert időbe telik, mire egy konténer létrejön és frissül a klaszter állapota. Ezért alkalmazunk erősen defenzív kódolást, mert sose tudhatjuk, mikorra kerül az elvárt állapotba a klaszter. Az sem kizárható, hogy sose éri el azt az állapotot.

MIPv6 implementáció

Első megközelítésre szükségünk lesz egy kliensre és a Home Agent-en futó daemonra, ami figyelni fog a kliensektől érkező üzenetekre. A megoldásban igyekezni fogunk megfelelni a szabvány legtöbb pontjának, a biztonsági részleteken pedig lazítunk, hiszen nem egy telepítésre kész verziót akarunk megalkotni, hanem csak bemutatjuk a javaslatunk működési elvét, proof-of-concept implementációját.



17. ábra: Home Agent szekvencia diagram

Lényegében ilyen modulokra fog oszlani az alkalmazás. Mivel C-ben írjuk, ezért ezek nem ojektumokként értendők, hanem elkülönült modulokként, amikben szeparálva definiáljuk a szükséges struktúrákat, függvényeket.

1. Az ábrán szépen látszik, milyen lépéseket kell tennie egy Home Agent-nek:
2. Várni egy Binding Update üzenetre
3. Létrehozni a Binding Acknowledgement csomagot
4. Elküldeni a Binding Acknowledgement üzenetet

5. A megérkezett üzenetből megkapja a kliens Care Of Address címét, ezzel megalkotható az adatsík, azaz az IP6IP6 tunnel.

A kliens is hasonlóan fog működni, csak más sorrendben hív hasonló függvényeket. Először is létrehoz egy Binding Update csomagot, azt elküldi, majd vár egy Binding Acknowledgement üzenetre, végül kihúzza az adatsíkot. Emiatt csak a Home Agent modulját kell kicserélni a kliens koreográfiájára, a többi modul újrafelhasználható lesz.

Első MIPv6 prototípus

Az első prototípust Python nyelven írtuk és még nem is Mobility Header-t tartalmazó csomagokra figyeltünk, hanem ICMPv6 Echo Request-re. De nagyon szépen látszanak rajta a fent definiált lépések.

```
from gettext import find
from scapy.all import *
from pyroute2 import IPRoute

interface = "eth0"
tunnel_name = "ip6tun"

def main():
    captured = sniff(filter="icmp6[icmp6type]=icmp6-echo", iface=interface, count=1)
    if len(captured) <= 0: return
    signal = captured[0]

    with IPRoute() as ipr:
        ipr.link(
            "add",
            ifname=tunnel_name,
            kind="ip6tnl",
            mode="ip6ip6",
            ip6tnl_local=signal["IPv6"].dst,
            ip6tnl_remote=signal["IPv6"].src
        )
        tunnel = ipr.link_lookup(ifname=tunnel_name)[0]
        ipr.link(
            "set",
            index=tunnel,
            state="up"
        )
        ipr.addr(
            "add",
            index=tunnel,
            address="fd84:c300:ca02:76d2::1",
            mask=64
        )


if __name__ == '__main__':
    main()
            print("Tunnel prepared!")
```

A program a scapy és a *pyroute2* csomagot használja. A *scapy*-vel megvárjuk az echo request csomagot, majd azt követően felállítjuk az adatsíkot, mivel már megkaptuk

a terminál Care Of Address-ét. Az adatsíkot a *pyroute2* segítségével hozzuk létre, aminek a szintaktikája olyan, mint amit az *iproute2*-ben megszokhattunk.

Itt még nincsenek Mobility Header csomagok és nem foglalkozunk válasz üzenettel sem, csak tisztán az automatizálást szerettük volna kipróbálni. Megfelelően működött, így továbbléphettünk.

Végső implementációk

Prairie Operator

A létrehozott operátor neve Prairie Operator lett, aminek a megfigyelt erőforrása a HomeAgent, ennek egy elvárt állapotleírrója van, a méret. A Reconciler algoritmusa nem sokat változott a pszeudokódban leírtakhoz képest, csak az említett Deployment létrehozása hozott újdonságot. Itt be is mutatjuk a Reconciler függvény Go implementációjának az elejét. A teljes kód elérhető a Githubon.[32]

```
func (r *HomeAgentReconciler) Reconcile(ctx context.Context, req ctrl.Request)
(ctrl.Result, error) {
    // …
    log.Log.Info("Reconcile sequence has started.")

    home_agent := &prairiev1.HomeAgent{}
    err := r.Get(ctx, req.NamespacedName, home_agent)
    if err != nil {
        // Resource was most likely deleted before reconcile request,
        // thus we should clean up and return without requeueing.
        if errors.IsNotFound(err) {
            log.Log.Info("HomeAgent CRD not found.")

            r.DeleteDeployment(ctx, req)
            return ctrl.Result{}, nil
        }
        // …
    }
```

Először is lekérjük az eseményhez tartozó erőforrást a Kube API-tól, ha nem találjuk, akkor a felhasználó törölte, emiatt törölhetjük a hozzá tartozó Deployment-et, és visszatérünk. Fontos megjegyeznünk, hogy a Deployment törlése is egy idempotens eljárás. Ellenőrzi, hogy létezik-e, ha nem akkor visszatér, ha igen, akkor törli a Kube API-on keresztül.

```
if deployment.Status.ReadyReplicas < home_agent.Spec.Size {
    log.Log.Info("Not every replica is ready, requeueing...")
    return reconcile.Result{RequeueAfter: wait_duration}, nil
}
```

Ez még egy fontos mozzanat az implementációban. Ha nincs minden Pod kész állapotban, azaz kevesebb kész állapotú Pod van, mint az elvárt, akkor egy kis idő múlva újra próbálkozunk, hiszen időbe telik, mire elindul minden konténer.

Home Agent implementációja C nyelven

Mindenekelőtt definiálnunk kell a csomagstruktúrát, amit a következőképpen tehetjük meg. A mezőket oktettre pontosan felosztjuk a stdint.h-ban definiált típusok segítségével, és a fejlécet követő üzenetre egy pointerrel hivatkozunk, mint az ip6_mh payload pointere, amivel elérhetjük mondjuk a belső Binding Update üzenetet.

```
struct ip6_mh {
    uint8_t ip6mh_proto;
    uint8_t ip6mh_hdrlen;
    uint8_t ip6mh_type;
    uint8_t ip6mh_reserved;
    uint16_t ip6mh_cksum;
    uint8_t payload[];
};
struct mh_back {
    uint8_t status;
    uint8_t reserved;
    uint16_t sequence;
    uint16_t lifetime;
    uint8_t options[];
};
```

Ezeknek az inicializálását láthatjuk a *mobi-packets.c* `create_binding_ack()` függvényében.

```
struct ip6_mh* mh = (struct ip6_mh*) msg;
mh->ip6mh_proto = 59; // No proto
// RFC: in units of 8 octets excluding the first 8 octets -> 1
// hdrlen = 8 bytes + 8 bytes = 16 bytes
mh->ip6mh_hdrlen = 1;
mh->ip6mh_type = 6; // B_ACK type code
mh->ip6mh_reserved = 0; // RFC: Must be initialised to zero and ignored
mh->ip6mh_cksum = 0;

struct mh_back* b_ack = (struct mh_back*) mh->payload;
b_ack->status = 0; // Binding Update accepted
b_ack->reserved = 0; // just like above
b_ack->sequence = htons(sequence); // RFC: same as BU sequence
b_ack->lifetime = htons(16); //in units of 4 seconds->16 lifetime = 16*4 sec = 64 sec

struct mo_padn* padding = (struct mo_padn*) b_ack->options;
padding->type = 1; // Type code for PadN TLV
padding->len = 2; // 2 octets to complete the 16 bytes
memset(padding->pad, 0, 2);
```

A fenti msg egy 16 bájt hosszú változó, amit pointer aritmetikával inicializálunk fel. Látszik, ahogy a payload, illetve options mutatót átalakítjuk egy másik struktúra mutatójává, ezzel kényelmesen szerkeszthetőek a mezők. Érdekesség a `htons()` függvény, ami átalakítja a hoszt byte sorrendjét hálózati sorrendre, azaz big-endian-re.

A sniffer és a küldő modul lényegében úgy lett implementálva, ahogy azt az elméletben megmutattuk, ezért arra nem térünk ki külön, a Githubon található forráskódban részletesen megtekinthetők ezek is. https://github.com/Tenacher/mobility-daemon

Az utolsó dolog, aminek az implementációja fontos még nekünk, az a libnl segítségével létrehozott tunnel interfész. Ennek következik egy részlete a tnl-c.c fájl `create_tunnel()` függvényéből.

```
//Setup socket
struct rtnl_link* tunnel = rtnl_link_ip6_tnl_alloc();

//Set tunnel parameters
rtnl_link_set_name(tunnel, TUN_NAME);
//…
rtnl_link_set_flags(tunnel, IFF_UP);
rtnl_link_ip6_tnl_set_local(tunnel, local);
rtnl_link_ip6_tnl_set_remote(tunnel, remote);

if(rtnl_link_add(socket, tunnel, NLM_F_CREATE) < 0) {
    perror("Could not create link!");
    //Free allocated structures
    return ERROR;
}

//Free allocated structures
```

A könyvtár olyan szinten leegyszerűsíti a dolgunkat, hogy nem is kell netlink üzenetet létrehozni. Az IP cím hozzáadás az egyetlen rész, ahol nincs ehhez hasonló már-már deklaratív interfész a könyvtárban, ott megtekinthető a saját netlink üzenet készítése a *tnl-c.c* forrásfájlban, de az elméleti fejezetben ezt már átvettük, így nem térünk ki rá részletesen.

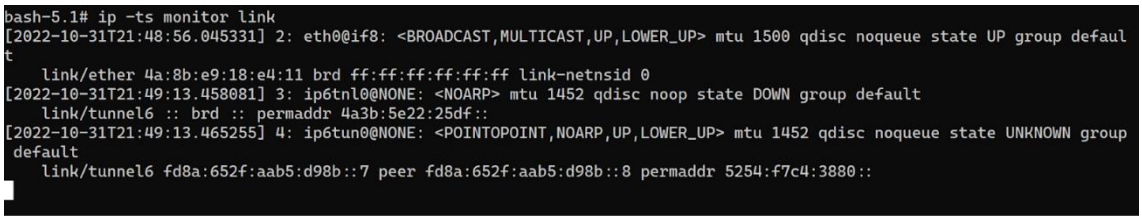
A `create_tunnel()` függvényben tehát egyszerű segédfüggvényekkel, kényelmesen létrehoztuk a tunnelt, majd elküldtük azt a kernelnek.

Tesztelés

A funkcionális teszt rendkívül egyszerű: két podot hozunk létre, az egyiken a kliens, másikon a Home Agent fut majd. A kliens podban elindítunk egy *tcpdump*-ot, ahol láthatjuk majd a Mobility Header forgalmat, egy *ip monitor*-t, ahol láthatjuk az interfész létrehozás időbélyegét, illetve a végén megpingeljük a Home Agentet a kliensről annak a címével, hogy lássuk, valóban működik az adatsík.

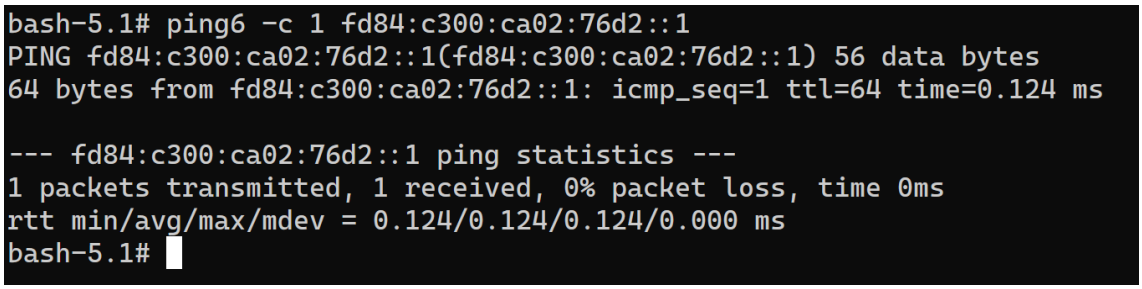
```
bash-5.1# tcpdump -i eth0 -vvv ip6 proto 135
tcpdump: listening on eth0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
21:49:13.416618 IP6 (flowlabel 0xcd057, hlim 64, next-header Mobility (135) payload length: 16) client > fd8a:652f:aab5:
d98b::8: mobility: BU seq#=50 lifetime=64 [|mobility]
21:49:13.461999 IP6 (flowlabel 0x3219b, hlim 63, next-header Mobility (135) payload length: 16) fd8a:652f:aab5:d98b::8 >
 client: mobility: BA status=0 seq#=50 lifetime=64 [|mobility]
```

18. ábra: Home Agent funkcionális teszt - tcpdump



19. ábra: Home Agent funkcionális teszt – ip monitor

Ahogy azt a képeken láthatjuk, a tcpdump sikeresen elfogta a megfelelő csomagokat, látszik a kimenő BU és a bejövő BACK üzenet. Az ip monitoron pedig láthatjuk, ahogy létrejön az ip6tun0 interfész ugyanabban az időben, mint ahogy a tcpdumpban megérkezett a Binding Acknowledgement csomag. A végső teszt echo request/reply üzenetekkel történt: ha megérkezik a request-re a válasz, akkor működik a tunnel és létrejött az adatsík.



20. ábra: Home Agent ping teszt

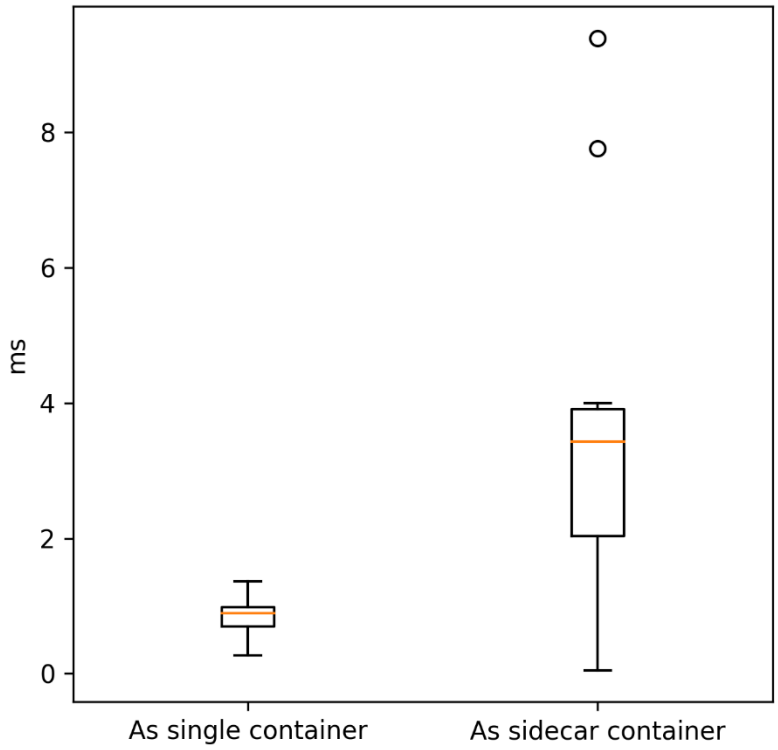
A kliens elérte a Home Agent-et, az adatsík működik, ezzel sikeresen teljesítettük a funkcionális tesztet.

A következőkben a teljesítménytesztet ismertetjük, ahol a round-trip time (RTT) időt vizsgáljuk. Megmérjük, hogy a kliens mennyi idő alatt kap választ a Home Agenttől két különböző felállásban. Az egyikben a Home Agent konténert önállóan hozzuk létre egy podban, a másikban a Home Agent sidecar konténerként szerepel, tehát a fő konténer mellett nyújtja szolgáltatásait szeparálva. Arra számítunk első sorban, hogy a több konténer felállásban megnő az RTT és lassabb szolgáltatást kapunk. Ez azért fontos, mert az ötödik generációs hálózatok specifikációjában törekszenek a pár ms-os válaszidő garantálására, ezért fontos teljesítmény mérce lesz a válaszidő.

Mindkét felállás esetében létrehoztunk 10 Home Agent példányt, és egymás után csatlakoztattuk a klienst hozzájuk, majd elmentettük a mért válaszidőket. A két mérést egy python szkript segítségével hajtottuk végre (lásd D függelék).

Test Nr.	Client instances	Home Agent instances	Home Agent deployment
1.	1	10	Single
2.	1	10	Sidecar

2. táblázat: Teszt forgatókönyvek



21. ábra: Válaszidő teszt boxplot

A kigyűjtött időkből a *matplotlib* segítségével boxplot ábrát csináltunk és a különbségek meglepően jól megmutatkoznak. Sidecar felállásban architekturálisan modernebb módon nyújtja a Home Agent számunkra a szolgáltatásait, hiszen ekkor részben elválasztjuk a hálózatkezelést az alkalmazástól. Ezzel elérjük a funkcionális szeparációt, ami elengedhetetlen hatékonyan működő alkalmazások tervezésénel. Azonban ebben az esetben jelentősen megnövekedett a késleltetés, nem is beszélve a jitterről, ami szintén erőteljesebben mutatkozik meg. A pontos adatokat alul táblázatba foglaltuk.

Deployment mode	MIN	MAX	AVG	MEDIAN	STD
Single	0.270316 ms	1.359743 ms	0.841400 ms	0.890900 ms	0.322647 ms
Sidecar	0.051399 ms	9.394563 ms	3.79855 ms	3.429227 ms	2.681371 ms

3. táblázat: Válaszidő statisztika

Látható, hogy a sidecar esetében erős növekedést tapasztaltunk az átlagos és a maximális válaszidő esetén, a medián is jelentősen eltolódott. A single konténer az átlagos válaszidőben 351%-kal jobban teljesített, mint a sidecar. Ezen kívül a maximális válaszidő 6,91-szeresére emelkedett a sidecar esetében. Ahogy azt írtuk, az RTT szórásában sem elhanyagolható különbséget láthattunk a kettő között. Két egészértékű növekedést tapasztaltunk a sidecaron mért eredmények szórásában, ami ebben az esetben 731%-os növekményt eredményezett. Ezzel kijelenthető, hogy a válaszidő jelentősen destabilizálódott.

6 Deklaratív megközelítések

Jelen fejezetben olyan fejlett hálózati konfigurációkat lehetővé tevő megoldásokat mutatunk be, melyek kvázi nem a Kubernetes Operator keretrendszert valósítják meg. Azonban ezen megoldások esetében is vannak olyan architekturális elemek, melyek hasonlítanak egy Kubernetes Operator működésére. Ezen technológiák közös tulajdonsága, hogy a Kubernetes objektum-leírók annotációs lehetőségeit használják ki. A következőkben ismertetjük az annotálást, mint konfigurációs lehetőséget, illetve néhány, a Kubernetes API-t kiterjesztő, gyakorlati megvalósítás működését, és architekturális tulajdonságaikat, majd az általuk nyújtott lehetőségeket kihasználó, a mobilitáskezelés egyes aspektusait támogató felhasználási szcenáriókat is ismertetünk.

Kubernetes Annotációk [33]

A Kubernetes a már korábban bemutatott metaadat elhelyezési lehetőségeken felül ún. annotációk (annotations) megadását is támogatja. Első látásra nem igazán különbözik a label (címke) típusú mezőktől, azonban sokkal több lehetőséget biztosítanak mint az egyszerű címke párok. Ezek elsősorban nem az adott objektum klaszteren belüli azonosítására szolgálnak, hanem kvázi konfigurációs jelleggel is bírhatnak. Az itt elhelyezett metaadatok lehetnek kisebb vagy nagyobb kiterjedésűek, strukturáltak vagy strukturálatlanok, és olyan karaktereket is tartalmazhatnak, amelyek a címkék esetében nem engedélyezettek.

Egy érvényes annotáció kulcs mezőjét két építőelemre lehet bontani: egy opcionális prefix tagra és a kulcs nevére, ezt a kettőt pedig egy ‘/’ választja el egymástól. A prefix, ha megadásra kerül, DNS név szerű struktúrát kell követnie. A halmazban szereplő kulcsoknak és értékeknek mindenképpen karakterláncoknak (string) kell lenniük és nem használhatóak bennük tisztán numerikus, logikai, listás vagy más adattípusok.

Az így megadott információ a leíró fájlban szerepeltethető többi metaadat megadási formától is megkülönböztethető, ezáltal a lehetséges felhasználási köre is meglehetősen széles. Használhatjuk őket hasznos, a hibakeresést elősegítő adatok megadására: verziószám, Git tároló címe, build információk, használt konténer képfájl tárolójának (registry) címe, de akár az objektum létrehozásáért és karbantartásáért felelős személy elérhetőségei is rögzítésre kerülhetnek. Azonban a legfontosabb felhasználási

lehetőség, hogy ezáltal olyan direktívák adhatók meg annotációk formájában, amelyeket a klaszterben futó, a Kubernetes képességeit kiegészítő beépülő szolgáltatások, az annotált erőforrás működésének befolyásolására tudnak használni. Ezt a tulajdonságot a Kubernetes rendszerösszetevői, mint a kube-scheduler, kube-controller-manager, kube-apiserver vagy a kubectl is kihasználják, a kubernetes.io/ és a k8s.io/ előtagok ezen szolgáltatások számára vannak fenntartva. A továbbiakban bemutatott szolgáltatások is ezt használják ki, jelen esetben azzal a céllal, hogy az erőforrások, specifikusan a Podok hálózati tulajdonságai dinamikusan módosításra kerüljenek.

Szoftverkiegészítők bemutatása

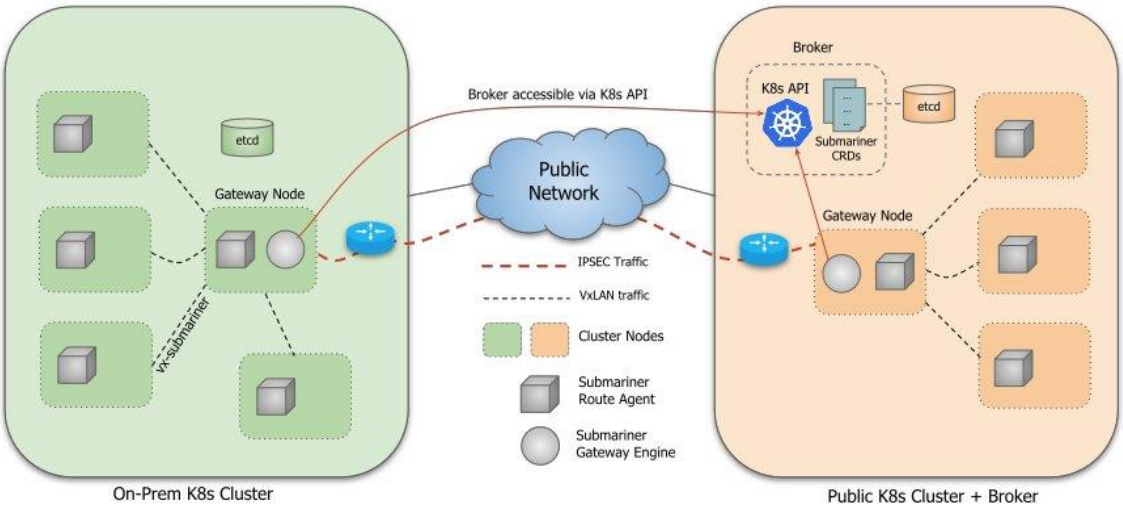
A következőkben ismertetett kiegészítőknek (plugin) elsődleges feladatuk, hogy a különböző felhő-architektúrák közötti kapcsolati gátat áthidalják, legyen szó a publikus felhők vagy akár a szolgáltatók saját üzemeltetésében lévő (on-premises) erőforrásokról. A feladat újdonsága miatt azonban jól bevált architektúrákról és eljárásokról még nem lehet beszélni, így látni fogjuk, hogy mindegyik eszköz más-más oldalról közelíti meg a problémát, és annak megfelelően kínál megoldást. Mindezt jelenleg még csak a Kubernetes kontextusában tudják megtenni, azonban némelyik projekt esetében kitűzött célként szerepel a más futtatási környezetek esetében történő megvalósítás is. Fontos kiemelni, hogy ezen szoftverek nagy része még fejlesztés alatt áll, így még nem lehet végleges és stabil megoldásokat várni tőlük. Azonban az általuk felvetett koncepciók a későbbiekben még hasznos kiindulási pontok lehetnek.

Submariner [34]

Első vizsgálandó pluginként a Submariner nevű projektet vesszük. Mint az elkövetkező pluginek többsége, ez is a Cloud Native Computing Foundation (CNCF) alá tartozó projekt, mely jelenleg még a ‘Sandbox’ fázisban tart, azaz még korai fejlettségi szinten áll a projekt és nem várható el tőle magas szintű stabilitás.

Működését tekintve Kubernetes klaszterek között biztosít biztonságos hálózati átjárókat, és mindezt a klasztereken futó hálózatkezelő runtimeoktól (CNI) függetlenül tudja megtenni. Mindezek mellett biztosítja a Servicek klaszterek közötti felfedezhetőségét, illetve egy parancssori interfészt is biztosít, amelyen keresztül még egyszerűbben konfigurálhatjuk és telepíthetjük az általa nyújtott infrastruktúrát.

Architektúrája négy főbb elemre bontható: a klaszterközi kapcsolatok kezeléséért felelős Broker-re, a klasztereken futó Gateway Engine-re, a DaemonSet-ként futó Route Agent-re és a Service Discovery-ért felelős Lighthouse nevű DNS szerver implementációra.



22. ábra: A Submariner architektúra[35]

A Submariner egy központi Broker komponenst használ a metaadat-információk cseréjének megkönnyítésére a részt vevő klaszterekben telepített Gateway Engine-ek között. A Broker alapvetően Custom Resource Definition-öket (CRD) tárol. A Bróker telepítése nem jár Podok vagy Servicek létrehozásával. A Submariner két fajta CRD-t határoz meg, amelyekhez a Brokeren keresztül lehet hozzáférni: Endpoint és Cluster típusút. Az Endpoint CRD tartalmazza a klaszter aktív Gateway Engine-jével kapcsolatos információkat, például annak IP címét, mely szükséges ahhoz hogy a kapcsolódni kívánó többi klaszter számára is elérhető legyen. A Cluster CRD statikus információkat tartalmaz az adott podról, mint a klaszterben használt Pod és Service CIDR-eket. A Broker egyetlen példányként fut vagy a két klaszter valamelyikén, vagy egy harmadik, mindenki által elérhető klaszteren. [36]

A Gateway Engine komponens minden résztvevő klaszterben telepítve van, és felelős a többi klaszter felé vezető biztonságos kapcsolat létrehozásáért. A Gateway Engine architektúrája független a klaszterek közötti tunnel kiépítését végző implementációtól. Alternatívákat kínál fel a klaszterek közötti kapcsolat formájára, alapértelmezettek a Libreswan vagy Wireguard VPN protokollok, de ha nincs szükség a kapcsolat titkosítására, akkor VXLAN tunnelezést is választhatunk.

A Route Agent a klaszter minden csomópontján futó DaemonSet, az ő feladata, hogy a Brokeren keresztül lekérhető CRD-k mentén beállítsa a klaszter minden csomópontján a routing szabályokat, illetve a szükséges iptables szabályokat.

A Service Discovery-t támogató Lighthouse DNS feladata, hogy azon Service komponensekhez, melyekhez biztosítani szeretnénk a klaszterek közötti elérhetőségüket, egy közös domain zónát definiál, illetve beállítja a Service DNS nevét. A klasztereken futó CoreDNS konfigurációját módosítja úgy, hogy a közös domain-re érkező kéréseket hozzá továbbítsa, tehát ő végezze a névfeloldást.

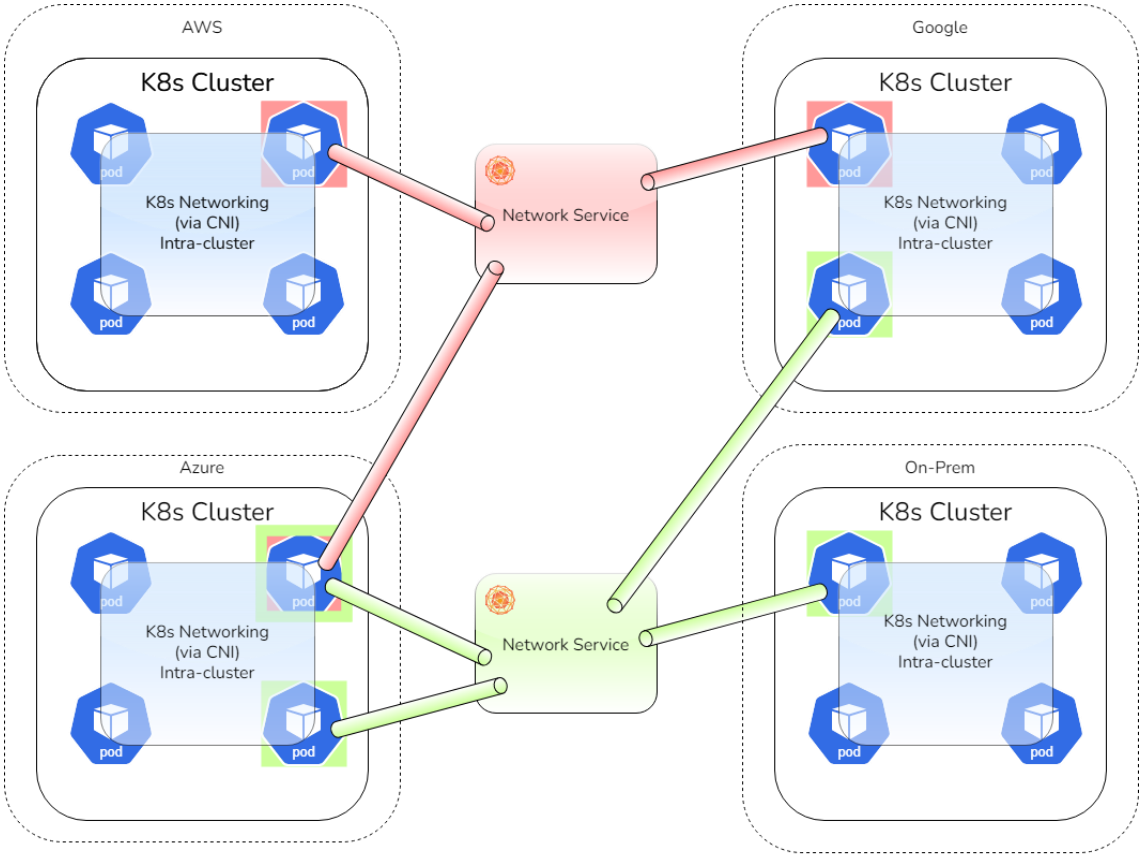
A Submariner tehát egy kifejezetten előremutató projekt. Layer 3 szintű hálózati kapcsolatot nyit klaszterek között, mellyel bármelyik klaszterben futó erőforráshoz hozzáférhetünk az IP címe alapján. Azonban ez magában hordoz egy komoly biztonsági problémát is. Ezáltal az adminisztratív domainek közötti határ teljesen elmosódna, tehát ezzel a módszerrel mindkét fél akadály nélkül hozzáférhet a partnerének teljes felhő infrastruktúrájához. A Submarinerben jelenleg erre nincs támogatás, hogy bizonyos policy-k mentén mely erőforrás melyik másikhoz férhet hozzá. Ettől függetlenül kísérleti céllal és vállalaton belüli klaszterek összekötésére alkalmas lehet.

Network Service Mesh [6]

Láthattuk, hogy a Submariner, mint klaszterek közötti konnektivitást biztosító eszköz nem nyújt teljeskörű megoldást a problémára, így a kitűzött feladat minél teljesebb körű megoldásához más keretrendszert kell keresnünk. A Network Service Mesh (NSM) szintén egy kezdetleges CNCF projekt, viszont az általa nyújtott jelenlegi és a későbbiekben implementálásra kerülő potenciális use-case-eknek köszönhetően szélesebb körű támogatottságra talált.

Bár nevében hasonlít a klasszikus szolgáltatási hálókhoz (Service Mesh) (ilyen implementációk például az Istio vagy a Linkerd), azok csak alkalmazási rétegbeli hálózatkezelést támogatnak. A Network Service Mesh velük ellentétben L2/L3 szintű hálózatkezelést is támogat, első sorban Kubernetes klaszterek között, viszont az API-ja segítségével akár más futtatási környezetekkel (runtime domain) történő összehangolás is lehetséges (pl. virtuális gépek klaszterével, vagy akár fizikai hardverrel is). A projekt által megvalósítható use-casek elég széles körűek. Lehetőség van például közös, klaszterek közötti virtuális Layer 3 kapcsolat létrehozására, mely esetében csak a megadott ‘tulajdonságú’ erőforrások kapcsolódhatnak egymáshoz, de vállalatok közötti

Service Meshek létrehozására vagy különböző klaszterekben üzemelő Service Mesh-ek összehangolására is van lehetőség. Tehát az NSM nem klaszterek között nyit átjárót, hanem csak meghatározott erőforrások számára hoz létre egy közös hálózatot.



23. ábra: A klaszterek közötti együttműködés szemléltetése az NSM esetében[37]

6.1.1.1 Architektúra és működés

Mint az a nevéből is következik, az NSM hálózati szolgáltatások (Network Services) hálóját kezeli, így értelemszerűen az architektúrájának alapját a Network Service (NS) jelenti. Ez egy absztrakt fogalom, melyet a fejlesztők úgy jellemeznek, hogy a "hálózati forgalomra alkalmazott kapcsolódási, biztonsági és megfigyelhetőségi szolgáltatások gyűjteménye". Ez lényegében magát a hálózati szolgáltatást jelenti, amelyhez majd az egyes Podok csatlakozni szeretnének, mely lehet a már korábban említett vL3 VPN kapcsolat két pod között, de jelenthet bármilyen más hálózati funkciót, bridget, útválasztót, tűzfalat stb.

A kliensek ún. Endpoint Podokon keresztül tudnak csatlakozni egy NS-hez. Az NS önmagában nem valósítja meg a szolgáltatást, csak metaadatként kezelendő, így kellenek végpontok, amelyeken keresztül el lehet érni. Ezt a szerepet töltik be az Endpoint

erőforrások. Saját maguk is nyújthatják a szolgáltatást, de alapértelmezésben ők kvázi átjáróként szolgálnak. Az Endpointok által nyújtott szolgáltatásra kapcsolódó Podokat nevezzük Clienteknek. Az NSM működését megvalósító architektúrát több összetevő is alkotja:

Network Service Registry[38]

Minden klaszterben jelenlévő egység, mely nyilvántartja a saját klaszterében elérhető NS-eket, illetve hogy ezeket mely Endpointokon keresztül lehet elérni.

Network Service Manager[38]

Az NSM kontroll síkját alkotják. Egy DaemonSet elemei, azaz a klaszter összes csomópontján jelen vannak, és egymással közösen alkotnak egy elosztott kontroll síkot. Két lényeges feladata van: az egyik hogy a saját nodeján lévő kliensek kapcsolódási kéréseit továbbítsa egy olyan csomópont Manageréhez, ahol az adott NS-t kiszolgáló Endpoint elérhető, a másik pedig hogy a saját nodeján létrejövő Endpointokat a Registrybe bejegyezze.

Forwarder[38]

Az NSM adatsíkját megvalósító komponens. Szintén egy DaemonSet, mely a Client és az Endpoint közötti virtuális vezeték (a dokumentációban vWire-ként hivatkoznak rá) ‘kihúzásáért’, azaz a köztük lévő kapcsolat létrehozásáért felel. Többféle továbbítási mechanizmust is támogat, mint az FD.io VPP, OpenVSwitch vagy akár SR-IOV.

A Network Service a Kubernetes esetében egy Custom Resource Definiton-ként van jelen. Egy ilyen NetworkService objektum látható alább:

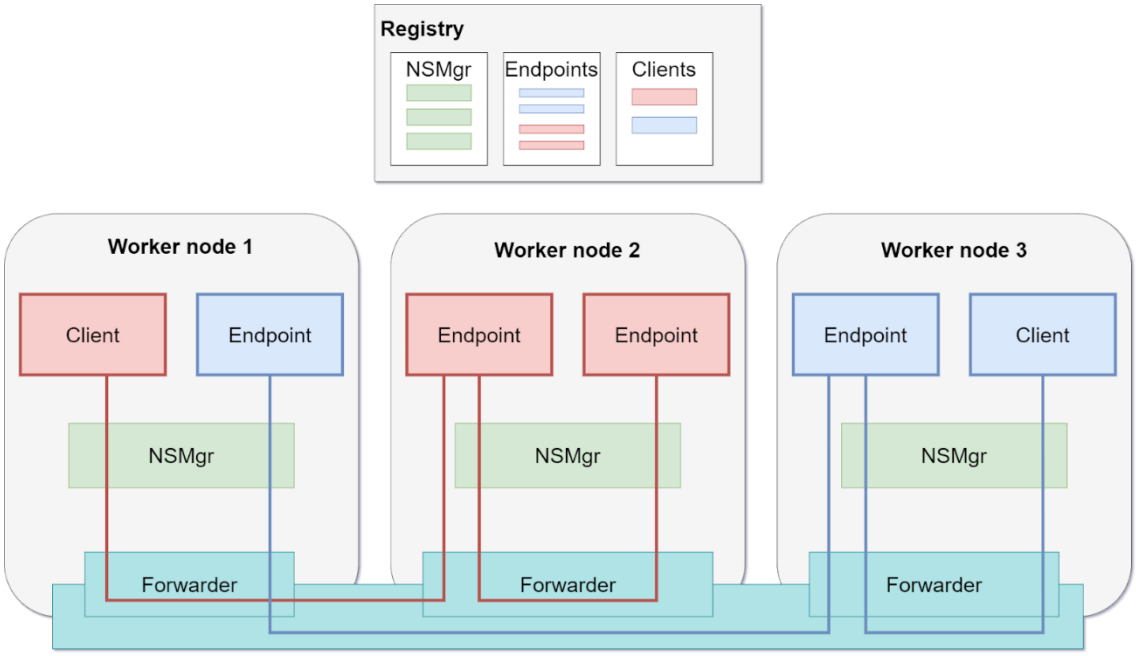
```
apiVersion: networkservicemesh.io/v1
kind: NetworkService
metadata:
  name: my-networkservice-1
spec:
  payload: IP
  matches:
    - source_selector:
        service: envoy-proxy
      routes:
        - destination_selector:
            service: vl3
```

Egy egyszerű leíró, amelyben az NS nevén kívül az általa biztosított konnektivitás típusa kerül megadásra. Ez utóbbi a példában a payload paraméter, melyhez az IP helyett ETHERNET értéket lehet még rendelni. Értelemszerűen ez írja le, hogy a L2 vagy L3 szintű kapcsolatot szeretnénk létrehozni. Továbbá opcionális paraméterekként megadhatjuk, hogy az NS-re felcsatlakozó kliensek mely kulcs-érték párokkal jelölt végpont (endpoint) erőforrásokhoz csatlakozzanak.

Egy Endpoint objektum szempontjából egy NS megvalósítása annyit jelent, hogy az őt létrehozó leíróban környezeti változóként megadásra kerül az általa szolgáltatandó NS neve. A Clientet viszont el kell látni a megfelelő annotációval az NS eléréséhez:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
  annotations:
    networkservicemesh.io:   "kernel://my-networkservice-
1/nsm-1"
spec: ...
```

Ebből kiolvasható, hogy egy kernel interfészen keresztül szeretné elérni a my-networkservice-1 nevű NS-t, illetve a létrehozandó interfész neve legyen nsm-1, ezutóbbi megadása opcionális.



24. ábra: Az Endpointok és Client-ek közötti konnektivitás sematikus ábrája

Egy NSC és egy NSE közötti kapcsolat kiépülése a következő lépések mentén zajlik:

1. Létrejön az NSE Pod a klaszteren belüli nodeon (node1).
2. A node1-en lévő NSMgr regisztrálja az NSE-t a Registrybe.
3. Létrejön a Client Pod, a Podban lévő init-konténer egy kérést küld a saját node-ján (node2) lévő NSMgr-nek, hogy csatlakozni szeretne a leírójában nevezett NS-hez.
4. A NSMgr lekéri a Registryből, hogy létezik-e olyan NSE amelyik ezt az NS-t biztosítja.
5. Ha létezik ilyen Endpoint, akkor továbbítja a node2 NSMgr-e a node1 NSMgr-ének a Client csatlakozási kérelmét.
6. A node1-en lévő NSMgr eljuttatja a kérést az NSE-nek.
7. Az NSE elfogadja a kérést, ha még rendelkezik elegendő erőforrással az új kliens kiszolgálására.
8. A node1 NSMgr létrehoz egy interfészt és beinjektálja azt az NSE Pod névterébe.
9. A node1 NSMgr értesíti a node2 NSMgr-t, hogy a csatlakozási kérelem el lett fogadva.
10. A node2 NSMgr ezt követően létrehoz egy interfészt és beinjektálja azt az NSC Pod névterébe, majd beállítja az útválasztást a Podban az NSE felé.

Az NSM-ben a széleskörű funkcionalitása mellett az erőforrások hitelesítése és azonosítása is implementálva van. Mindezt a SPIRE keretrendzser segítségével végzi, mely a SPIFFE (Secure Production Identity Framework for Everyone) projekten alapszik.[39] A SPIFFE egy nyílt forráskódú szabványkészlet szoftverrendszerek dinamikus és heterogén környezetekben történő biztonságos azonosítására. Funkcionalitását tekintve különböző elosztott és felhő rendszerekben futó alkalmazások azonosítására és autentikálására szolgál. Ennek alapját adja a SPIFFE ID, amely egyedileg azonosítja az erőforrást. Ez egy Uniform Resource Identifier (URI), melyet “spiffe://tartomány/erőforrásazonosító” formátumban adnak meg. Az SVID egy dokumentum, amellyel az erőforrás az azonosságát igazolja. Ez egyetlen SPIFFE ID-t

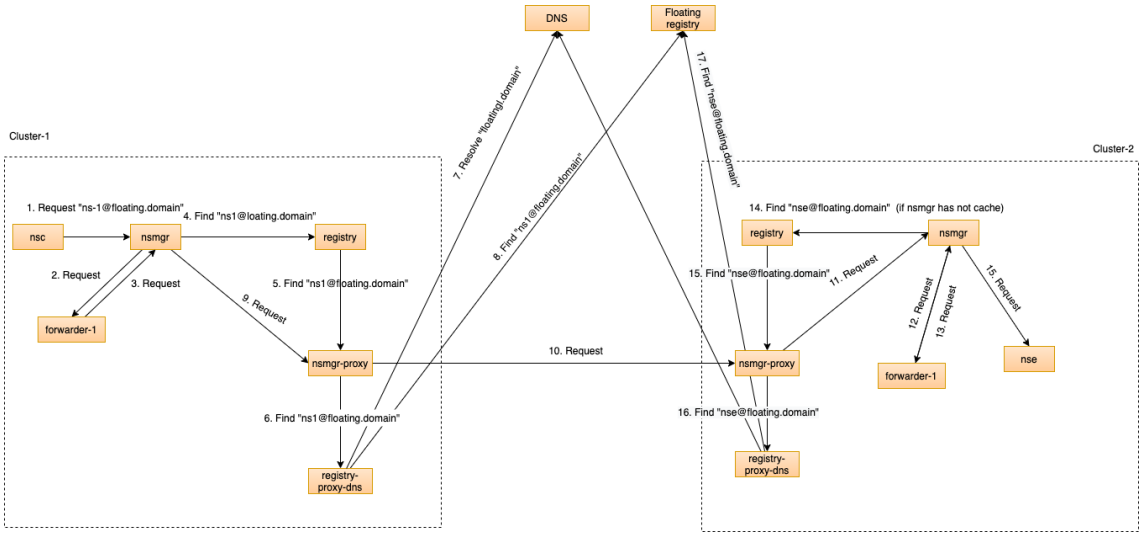
tartalmaz X.509 vagy JSON Web Token tanúsítvány formájában. Az SVID akkor tekinthető érvényesnek, ha a SPIFFE ID kibocsátási tartományán belül lett aláírva.

6.1.1.2 Klaszterek közötti kapcsolat

A következőkben bemutatjuk a kitűzött feladat egy lehetséges megoldását, melyet a Network Service Mesh segítségével hajtottunk végre. A futtatási környezetet 'Kubernetes in Docker' (KIND) klaszterek biztosították, melyek egy közös PC-n futottak, így a köztük való konnektivitást egy Linux Bridge interfészen keresztül oldottuk meg. A KIND egy olyan keretrendszer, melynek segítségével egyetlen Docker konténerben futtathatunk Kubernetes klasztereket. Ez közel sem nevezhető teljeskörű, ipari környezetben is alkalmazható megoldásnak, azonban a bennük futó klaszterek teljes egészében megegyeznek egy másfajta környezetben telepített klaszterrel, így a Kubernetes képességeinek, illetve saját konténerizált alkalmazásaink tesztelésére tökéletesen megfelel.

Az NSM korábban ismertetett architektúrájával ellentétben további szolgáltatásokra és a klaszter bizonyos konfigurációinak módosítására is szükség van a klaszterek közötti konnektivitás létrehozására. Első körben biztosítani kell a klaszterek közötti Servicek elérését. Ehhez az egyik legkézenfekvőbb megoldás, ha a Service-eket LoadBalancer típusúnak hozzuk létre. Azonban a LoadBalancer helyes működéséhez szükség van felhőszolgáltató-specifikus implementációra, ami a mi esetünkben nem áll rendelkezésre. Szerencsére ezen problémára nyújt megoldást a MetalLB nevű implementáció, mellyel lehetőség van saját magunk által hosztolt környezetben futtatott Kubernetes klaszterekben a LoadBalancer implementálására. Ennek telepítése és konfigurálása után biztosítani kell a klaszterek közötti DNS névfeloldást, mellyel egy másik klaszterben futó Service domain nevét fel lehessen oldani. Ehhez a klaszterek kube-dns Service-ét kell kívülről elérhetővé tenni, illetve a klaszterek CoreDns szolgáltatását is el kell látni további konfigurációkkal. Mindhárom klaszternek definiálunk egy saját DNS zónát, melyek segítségével könnyebben hivatkozhatunk a bennük futó Service-ekre, és a DNS szervernek megadjuk, hogy ha ilyen DNS zónákba tartozó domain neveket szeretnének feloldani, akkor a kéréseket továbbítsák a megfelelő klaszter DNS Service-e felé. A következő lépés a Spire telepítése, majd a Spire szerverek összehangolása, mely annyit tesz, hogy a CA tanúsítványaik egymással megosztásra kerülnek, ezáltal azonosíthatóvá válnak egymás számára a nem saját klasztereikben futó erőforrások is.

Miután az előkészítő lépéseket megtettük, nekiláthatunk az NSM érdemi telepítéséhez. Az előzőekben látottakhoz képest annyi változást kell eszközölni, hogy a Registry a harmadik, összekapcsolást segítő klaszteren kerül elhelyezésre, a másik két klaszter pedig egy-egy registry-proxy Service segítségével tud majd a közös Registryhez hozzáférni. Az NSManagerek esetében is elhelyezésre kerülnek NSManager-Proxy Service-ek, melyek a későbbiekben egymással kommunikálva tudják az adatsíkot kiépíteni. Az NSM Floating Interdomain implementációjának telepítése a projekt GitHub oldalán kerül részletezésre, egy lehetséges Client - Endpoint csatlakozási szcenárió pedig az E függelékben található. Az így felépülő rendszert és működését a következő ábra mutatja.



25. ábra: A Floating Interdomain működés[40]

A DNS névfeloldás folyamata bár a klasztereken kívül van ábrázolva, csak egyszerűsítés végett szerepel külön, a DNS feloldás mindhárom klaszteren belül megtörténik. Ezt leszámítva a felfedezési és kapcsolódási folyamatot jól szemlélteti az ábra.

Ezen architektúra saját környezetben történő implementálása néha nem triviális lépéseket is jelentett. A helyzetet tovább nehezítette, hogy új kezdeményezés révén még nem érhető el teljeskörű dokumentáció az NSM-ről, így némelyik komponens, illetve az általuk implementált technológiák működését/konfigurálhatóságát a Go kódok vizsgálatából (vagy saját erőből) kellett kikövetkeztetni. Amit sajnos nem sikerült megoldani az NSM kontextusában, az az IPv6-ot használó klaszterekkel való együttműködés. Bár a fejlesztők úgy tervezték, hogy IPv6-os címekkel is kompatibilisek legyenek a komponensek, a dolgozat írásáig ezt egy ismert, és azonosított bug miatt nem

lehetett megtenni, így a dolgozat egyik fő clkitűzésével szembe kellett mennünk, és a saját implementációk során is IPv4-es címek használatára kellett szorítkoznunk. Ez persze nem jelenti azt, hogy a javasolt megoldásunk ne lenne generikus: ha a fejlesztők javítják a hibát, akkor mi is át tudjuk portolni a javaslatunkat IPv6-ra. [41]

Tesztelés

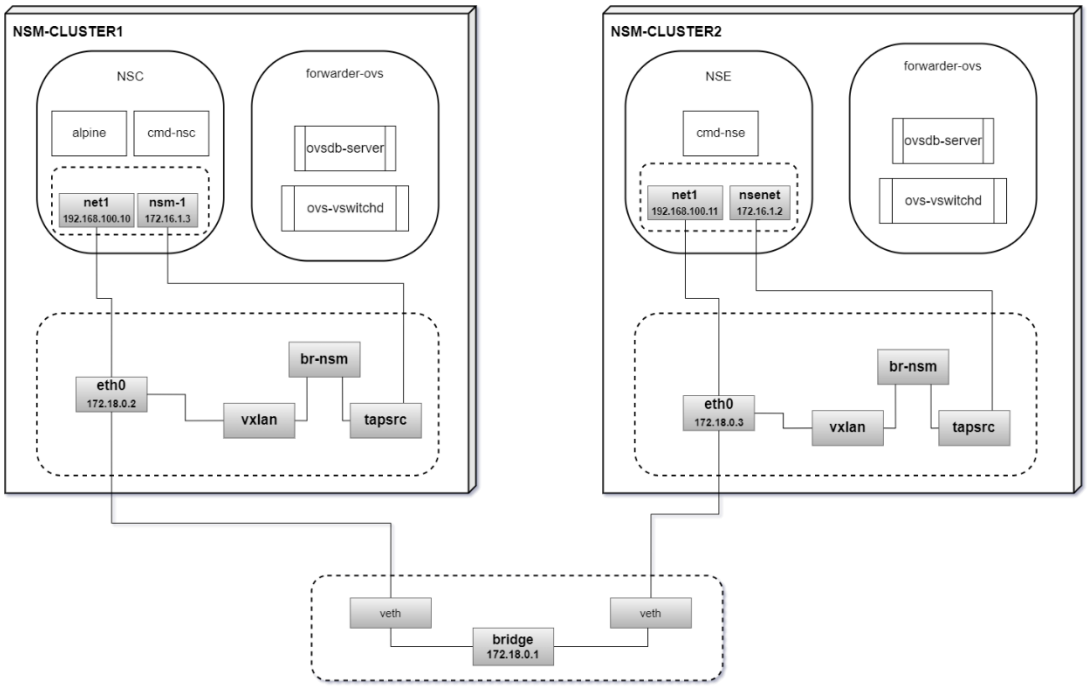
A Network Service Mesh képességeinek vizsgálatára szintén a KIND alapú klaszterekben került sor. Mivel ez nem tekinthető ipari szintű alkalmazási környezetnek, ezért további viszonyítási alapra volt szükség, melyet a Multus nevű, konténer-hálózatkezelést támogató plugin szolgáltatott.

A Multus egy meta CNI plugin, ez azt jelenti, hogy saját maga is CNI pluginnak tekinthető, tehát a konténer futtatási környezet ugyanúgy hívhatja hálózati konfiguráció céljából, azonban ő tovább hív egy megfelelő, alapvető működést biztosító pluginhez. Ez utóbbi pedig egy előre megadott konfigurációt tartalmazó leíró tartalma alapján fog további kernel interfészeket hozzáadni a konténer hálózati névteréhez. A konfiguráció JSON formátumban kerül megadásra, a használni kívánt CNI plugin paramétereinek megfelelően. Ez a JSON formátumú leíró egy Network Attachment Definition nevű CRD-be kerül beágyazásra, majd a telepíteni kívánt Pod leírójának, ezen CR-el történő annotálásával lehet a Podot ellátni ezzel az interfésszel. Használatát tekintve meglehetősen egyszerű, és az általa létrehozott interfészek teljesítmény szempontból is jól teljesítenek. Ez annak köszönhető, hogy nincsenek absztrakciós rétegek közbeiktatva a csomag útját illetően, hiszen a Podot futtató worker csomópont kernel interfészeként kerül létrehozásra, amely a Pod konténereinek hálózati névterében kerül elhelyezésre. Ennek megfelelően jó összehasonlítási alapot nyújthat az elkövetkező mérések számára.

A hálózat teljesítményének méréséhez a végpontok közötti késleltetést mértük ICMP Echo Request folyamok segítségével, illetve a hálózat TCP folyamokra vonatkozó áteresztőképességét vizsgáltuk, melyet az iperf3 nevű, hálózati performancia mérésére alkalmas programmal végeztünk. A késleltetés méréséhez 1000 db ICMP Echo Request üzenetet küldtünk az elérendő végpont felé, az ezekre érkezett ICMP Echo Reply üzenetek beérkezéséig eltelt időt mértük, melyből a hálózat késleltetését tudtuk meghatározni. A TCP folyamokra vonatkozó áteresztőképességet 100 db, egyenként 1 perc hosszú TCP folyam küldésével végeztük. Ezen 1 perces intervallumok, küldő és vevő oldalon mért értékeinek vettük a számtani átlagát, így adódott az egy 1 perc hosszú

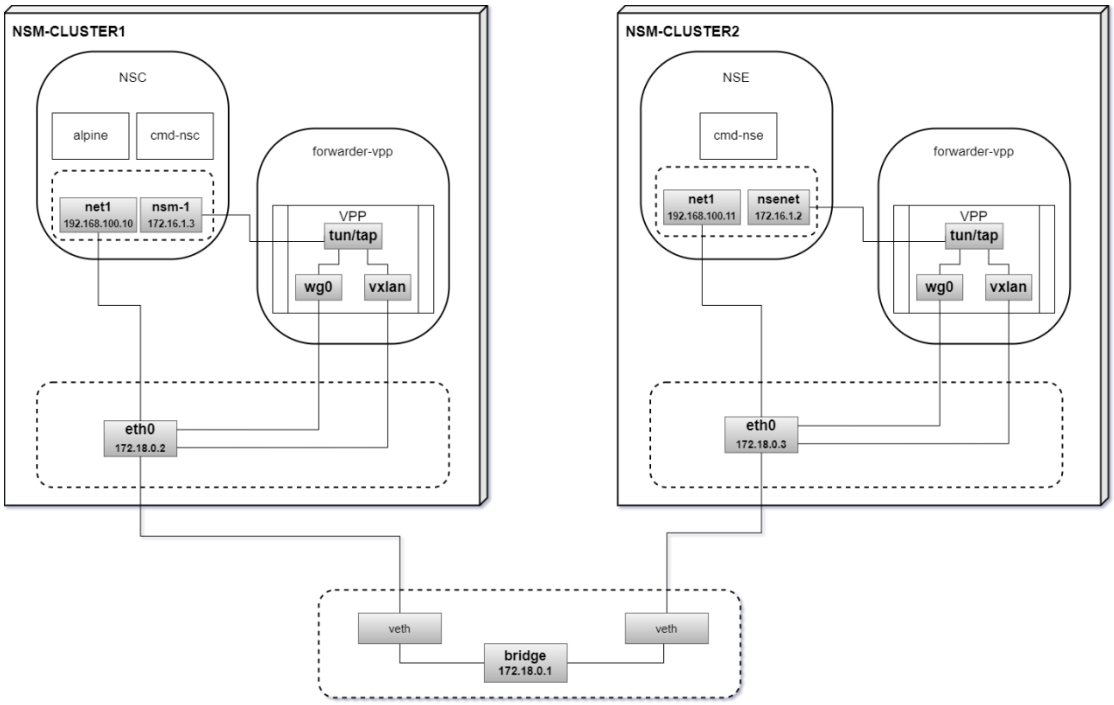
folyamra mért áteresztőképesség. Bár egy hálózat általános teljesítményének precíz meghatározásához további tulajdonságainak vizsgálata is elengedhetetlen, azonban ezek meghatározásához nehezen beszerezhető szoftverekre vagy be nem futtható fejlesztésre lett volna szükség, vagy a jelen mérési szcenárió szempontjából ezek nem számítanak relevánsnak. Az alkalmazott mérések pedig magukban is megfelelő képet tudnak biztosítani a technológiák közötti architekturális különbségek szemléltetésére.

A mérési szcenáriót minden esetben két, külön Kubernetes klaszterben futó Pod közötti hálózat jelentette. A beinjektált interfészek a Multus esetében a ‘macvlan’ CNI plugin segítségével jöttek létre. Az NSM esetében több lehetséges csomagtovábbítási eljárás és enkapszulációs módszer szolgáltatták az egyes eseteket. A továbbítási mechanizmusok az NSM által támogatott, OpenVSwitch és FD.io Vector Packet Processing (VPP) megvalósításai voltak. VPP esetén a csomagok enkapszulációja is változtatható volt, L2 szintű konnektivitás esetén VXLAN tunnelezés, L3 esetében a Wireguard protokol implementációja volt használatban. Az OpenVSwitch esetében L2 szintű konnektivitás létrehozására volt lehetőség, mely szintén VXLAN tunnelinget használt enkapszulációnak. A Multus esetében nem történt további rétegek elhelyezése az adatcsomagokban, az IP fejlécben az újonnan létrehozott interfészek IP címei szerepeltek. A csomag célbajuttatását főként az L2 szintű konnektivitás segítette. Egy macvlan típusú interfész alapértelmezésben egy, a hoszton lévő fizikai interfészt használ bridgeként, melyen keresztül kijut a külső hálózatba. Mivel a KIND klaszterek alapértelmezett interfészei egy bridge-en keresztül vannak összekötve, ezért további útválasztási információk megadására nem volt szükség sem a klaszter, sem az azt összekötő hálózat vonatkozásában. A következő ábrákon a fent említett továbbítási mechanizmusoknak megfelelően kerülnek ábrázolásra a csomagok útjai.



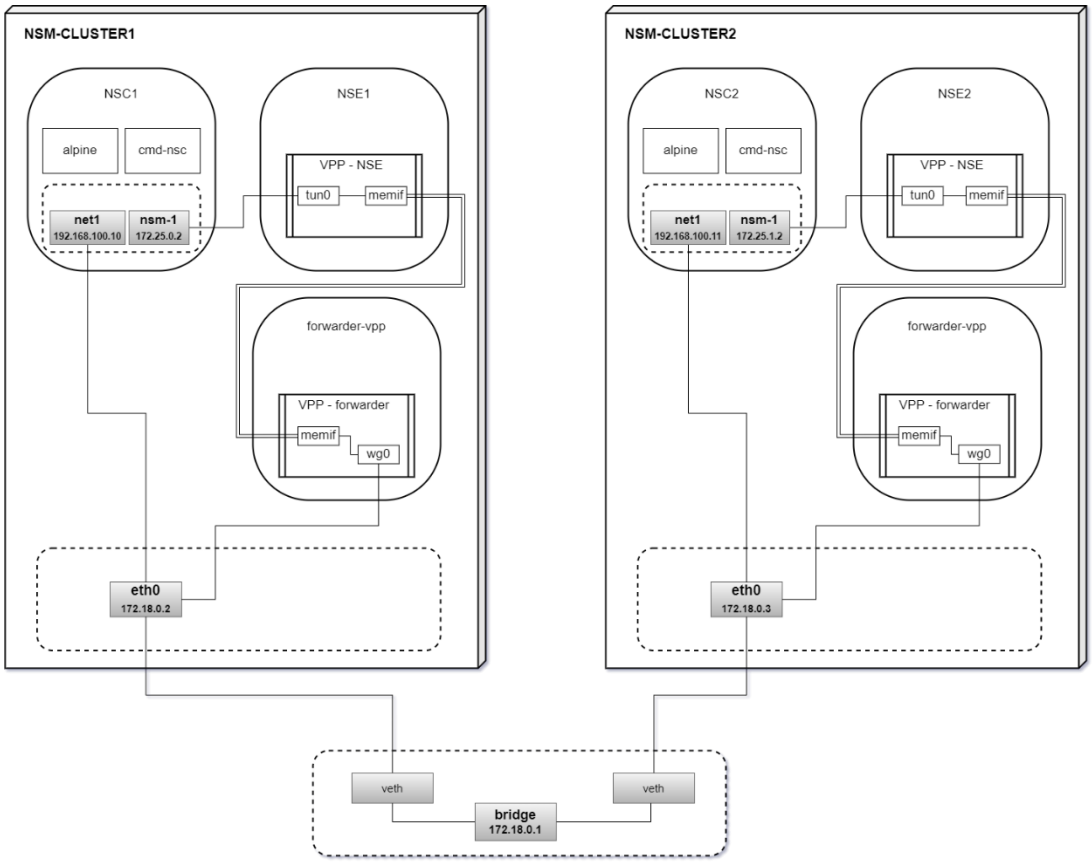
26. ábra: OpenVSwitch továbítás

Ebben az esetben a forwarder futtatja az OpenVSwitch kontroller és adatbázis processzeit. Egy OVS bridge-et hoz létre az érintett worker csomóponton, melyhez egy tap interfészt és egy másik veth interfészt csatol. Utóbbin történik a VXLAN tunnel terminálása, vagy kimenő forgalom esetén a fejléc elhelyezése. A kifelé menő forgalom a Client Pod nsm-1 interfészétől kijut a tap interfészre, itt bejut a br-nsm-be, ahol az érvényes OpenFlow szabályok szerint továbbítódik a vxlan interfészhez, ahol megtörténik a VXLAN enkapszuláció, majd innen egyenes út vezet a worker csomópont alapértelmezett interfészéhez.



27. ábra: VPP továbbítás

A forwarder ilyenkor egy FD.io VPP példányt futtat, melynek esetében kétféle lehetőség van az enkapszulációra. Ha az Endpoint által megvalósított Network Service payload paraméterének értéke IP, azaz L3 szintű kapcsolat biztosítása a feladat, akkor Wireguard protokollt használ, ha az érték ETHERNET, akkor VXLAN enkapszuláció kerül alkalmazásra a csomagokon.



28. ábra: vL3 továbbítás

A vL3 szcenárió esetében több VPP példány is fut, a forwarderen kívül az Endpointban is. Ennek a két példánynak a kommunikációja memif (Shared Memory Packet Interface) interfészek keresztül történik. Ezt leszámítva hasonlóképpen történik a csomagok áramlása, mint az előző esetben, viszont az IP cím allokáció egy központi IPAM közreműködésével történik, az Endpointok tőle kapnak tartományokat, melyekből a rájuk csatlakozó klienseknek tudnak kiosztani.

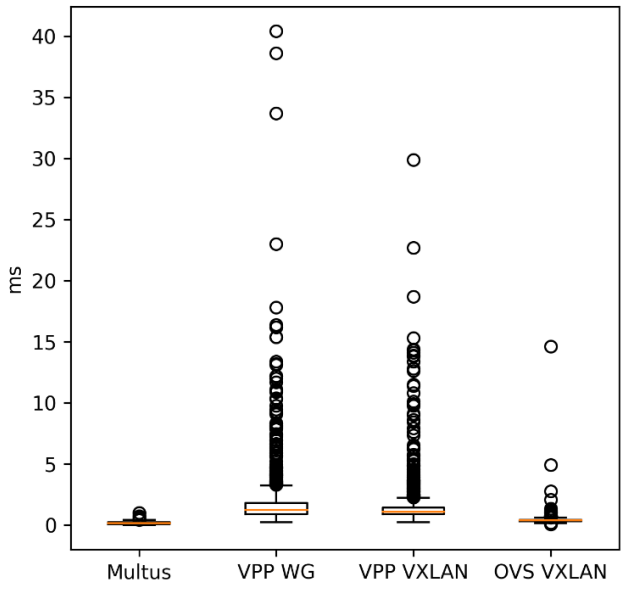
Encap.\FW mech.	VPP	OVS	Multus
VXLAN	1	1	0
Wireguard	1*	0	0
None	0	0	1

4. táblázat: NSM mérési mátrix

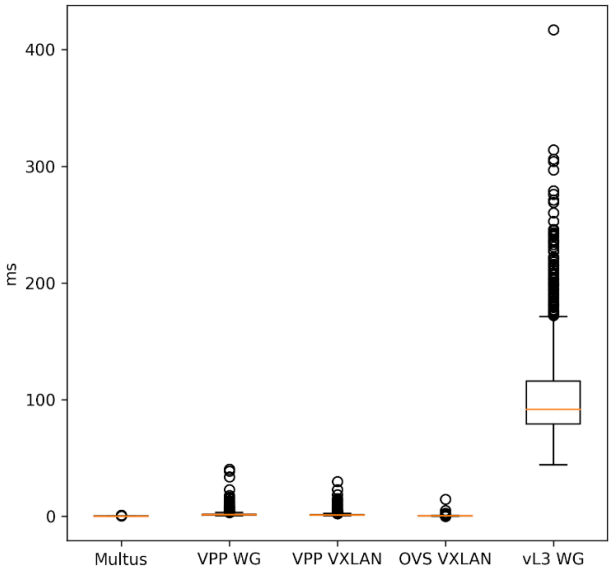
Mérési mátrix a továbbítási mechanizmusok és enkapszulációs eljárások halmazáról

*: Kernel és vL3 is

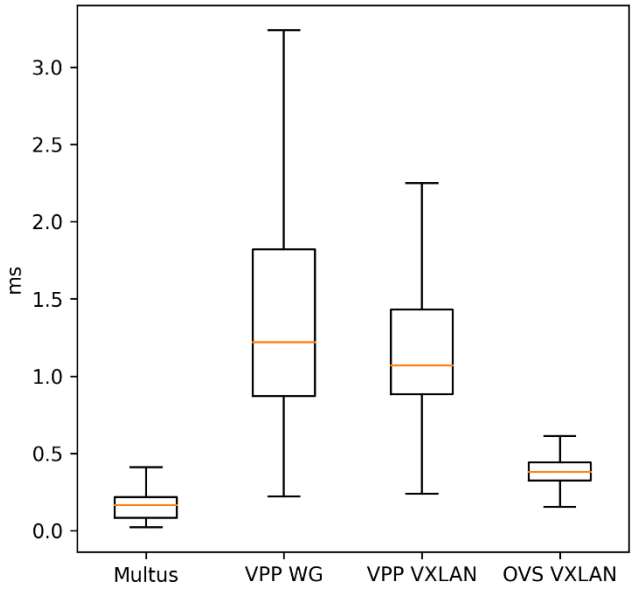
A mérések elvégzése előtt feltételeztük, hogy mivel a Network Service Mesh az architektúrája miatt nagyobb komplexitást visz a csomagok feldolgozásába, továbbá a Multus esetében még enkapszuláció sem növeli a komplexitást, várhatóan kisebb teljesítményt fog mutatni a Multus-szal szemben. Ezt a mérési eredmények be is bizonyították, sőt, némelyik esetben a helyzet még annál is rosszabb, mint vártuk.



29. ábra: NSM továbbítási mechanizmusok késleltetése



30. ábra: NSM továbbítási mechanizmusok késleltetése



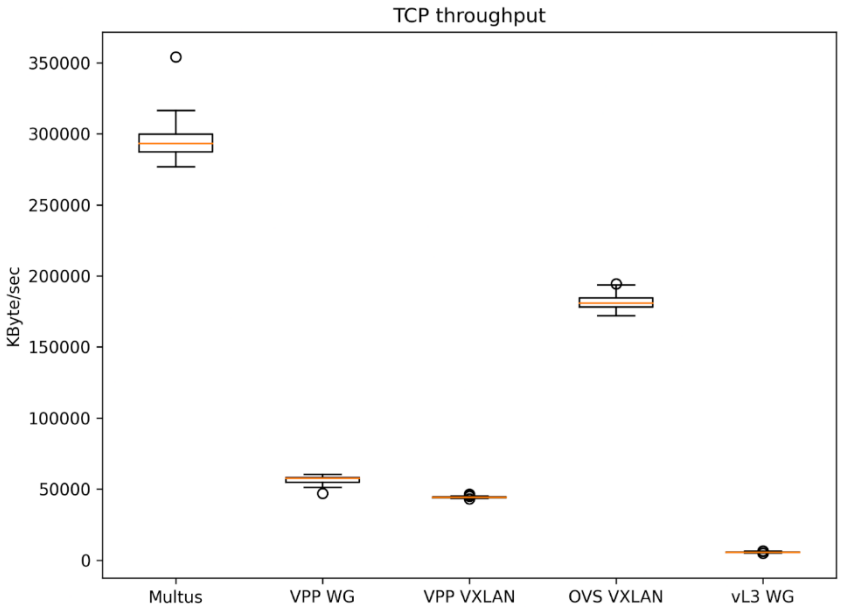
31. ábra: NSM továbbítási mechanizmusok késleltetése

	MIN	MAX	MEDIAN	AVG	STD
Multus	0.023	0.992	0.164	0.159	0.084
VPP WG	0.222	40.4	1.22	1.902	2.96
VPP VXLAN	0.238	29.9	1.07	1.542	2.141
OVS VXLAN	0.089	14.604	0.381	0.409	0.496
vL3	44.1	417.0	106.391	91.6	44.053

5. táblázat: NSM késleltetés statisztika

A előző ábrák mind a késleltetést szemléltetik, azonban a késleltetési viszonyok érzékeltetéséhez célszerűnek gondoltuk, ha több diagramon jelenítjük meg az eredményeket. Az elsőn a vL3 szcenárión kívül mindegyik szerepel, továbbá a szóráson kívüli értékek is meg vannak jelenítve. A másodikon megjelenik a vL3 szcenárió is, ezzel jól lehet érzékeltetni, hogy a többihez képest mekkora ugrást jelent késleltetésben. A harmadik ábra az egymáshoz késleltetésben közelebb eső szcenáriókat szemlélteti. A táblázatban láthatóak a mérési mátrixban jelölt szcenáriókon végzett, a hálózat késleltetését erintő mérések statisztikája, milliszekundumban.

A TCP áteresztőképességre a következő értékeket kaptuk:



32. ábra: NSM TCP áteresztőképesség diagram

	MIN	MAX	MEDIAN	AVG	STD
Multus	276764	354040	293102	293833.84	9826.6
VPP WG	47135	60463	57642	56462.79	2514.2
VPP VXLAN	43133	46489	44291	44308.51	443.05
OVS VXLAN	171992	194453	180828	181732.38	5119.33
vL3	4941	6490	5623	5625.1	272.87

6. táblázat: NSM TCP áteresztőképesség statisztika

A fenti táblázat pedig az egyes adatsíkok TCP áteresztőképességének statisztikáját tartalmazza, Kilobájt/másodpercben. A mérési eredményeket összevetve egyértelműen a Multus kerül ki győztesnek, ahogy ezt vártuk is. Azonban a legmeghökkentőbb eredményt az jelentette, hogy általánosságban a VPP technológiát használó megoldások teljesítettek rosszabbul, a VPP esetében legjobb eredményeket mutató megoldás, a Wireguard protokoll esetében is több, mint 80%-os romlás volt tapasztalható az átlagos TCP sávszélességen, az átlagos késleltetés pedig majdnem

tizenkétszer nagyobb volt. Ugyanakkor a sávszélességben tapasztalható ingadozás a VPP implementációk esetében a legkisebb. Ez azért is furcsa, mert a legtöbb helyen a klasszikus kernel networkingtől sokkal gyorsabb alternatívaként hivatkoznak rá. Elképzelhető, hogy sokkal nagyobb volumenű forgalom esetén a VPP jobban teljesíthet, viszont a fenti mérési eredmények azt bizonyítják, hogy kevésbé nagy szávsélességű alkalmazások esetén rosszabb, mint a kernel networking. Az OpenVSwitch alapú implementáció bár legjobban teljesített az NSM-es implementációk közül, a viszonyítási ponthoz képest ez is közel 40%-al rosszab sávszélességben, késleltetésben 157%-kal lassabb.

Az NSM hátrányát egyértelműen az adattovábbítás komplexitása jelenti. Viszont nem mehetünk el azon tény mellett, hogy a Multus bár teljesítményben kimagasló, nem automatizáltan nyújtja a szolgáltatásait, hanem emberi beavatkozásra is szükség van, főleg olyan helyzetekben, ahol a jelen mérési szcenárióval ellentétben további útválasztási konfigurációkra is szükség van a futtató környezet hálózatát illetően. Funkcionalitás szempontjából tehát nyugodtan kijelenthetjük, hogy az NSM többet kínál, azonban a funkcionalitás ára az általunk kimutatott jelentős teljesítménycsökkenés.

7 Továbbfejlesztési és alkalmazási lehetőségek

A két megoldásunk a MIPv6 felhősítésének lehetőségét vizsgálja meg, de ennél tovább is mehetnénk, akár a Proxy MIPv6 [42] felhősítése felé. Ebben az esetben maga a mobil terminál sem tudja, hogy barangol, aminek köszönhetően nincs szükség arra, hogy módosításokat végezzünk az MN hoszt TCP/IP stackjében. Viszont azt is jelenti, hogy a hálózatnak folyamatosan követnie kell a mobil terminál mozgását, amit teljesen új entitások és funkciók bevezetésével old meg a szabvány. A Local Mobility Anchor és a Mobility Access Gateway együtt koordinálják a routingot. A Local Mobility Anchor elfogja a terminálnak címzett csomagokat és azt egy kétirányú tunnelen keresztül küldi el a Mobility Access Gatewaynek, ami pedig kicsomagolja és továbbítja a hosztnak.

A másik lehetőség mobilitás tekintetében a Segment Routing IPv6, vagy SRv6 [43]. Ez a jövőbe mutató technológia az TCP/IP stack módosításával azt biztosítja számunkra, hogy tetszőleges úton jusson el a csomag az egyik csomópontból a másikba. Egy prezentáció keretében a FOSDEM 2020-as találkozón bemutatták, hogyan lehetne a Kubernetes klaszterben SRv6-alapú hálózati megoldást megvalósítani.[43], [44]

Habár a Network Service Mesh a mi méréseinken nem is szerepelt annyira fényesen, az általa nyújtott funkcionalitás és biztonság nem csak a mi fantáziánkat mozgatta meg. Több publikáció is készült a keretrendszer mobil hálózatokban történő alkalmazására, mint felhő-alapú Service Function Chaininget megvalósító keretrendszer. Ezt a klasszikus NFV infrastruktúrával szembeni mérésekkel támaszották alá, mellyel szemben sokkal jobb eredményeket értek el, mind végpontok közötti késleltetés, mind az felhő-natív alkalmazás telepítési és indítási idejét illetően. [45]

Ezen felül IP Multimedia Subsystem (IMS) NSM-el történő megvalósíthatóságát is vizsgálták, melyről szintén pozitív eredményeket tudtak felmutatni.[4]

8 Összegzés

Dolgozatunk készítése során két irányból közelítettük meg a felhőalapú, IPv6-os mobilitáskezelést. Egyrészt főleg az adatsík kihúzására fókuszáltunk, de egy kezdetleges implementációt is készítettünk, amiben a kliens kommunikál egy Home Agenttel. Az adatsíkot deklaratív szempontból is megvizsgáltuk a Network Service Mesh segítségével, de Kubernetes Operator alapokon erre saját implementációt is készítettünk.

Az előbbi megoldásban két podot kötöttünk össze klasztereken átívelő tunnelekkel, teszteltük az általa nyújtott teljesítményt, és az NSM lehetséges továbbítási módjait összevetettük a Multus által nyújtott szolgáltatásokkal. Itt nem meglepő, hogy a Multus teljesített jobban, ugyanis az NSM esetén komplexebb csomagtovábbítási útvonalak jönnek létre a két Pod között. Ez az eredmények tükrében legjobb esetben is közel 40%-os romlást jelentett az átlagos sávszélesség szempontjából, átlagos késleltetésben pedig 2.5-szer lassabbnak bizonyult. Azonban pont az NSM komplexitása és a benne implementált biztonsági eljárások teszik lehetővé, hogy alkalmazása telco szcenáriók esetén is szóba jöjjön, amikor mondjuk egy másik szolgáltató felhő hálózatával szeretnénk kommunikálni. Ezentúl számos jövőbe mutató, csomagfeldolgozási és összeköttetési technológiát is implementál, mint például a VPP.

A Kubernetes Operator-alapú megoldásban egy deklaratív API-t és saját erőforrást határoztunk meg a Kubernetes bővítésére. Az erőforrások létrehozásakor megvizsgáltunk két potenciális felépítésmódot is. Ezeken teljesítménytesztet is végrehajtottunk. Az egyik architektúra alternatívánkban a Home Agent mint sidecar konténer jelent meg és egy fő konténer mellett nyújtotta szolgáltatásait, míg a másik esetben egyetlen konténerként került futtatásra. Nem meglepő, hogy önálló podként gyorsabb válaszidőt értünk el, de amire nem számítottunk az az, hogy nagyobb késleltetés ingadozást (jittert) is eredményezett, így erősen destabilizálódott a szolgáltatás. 731%-os emelkedést tapasztaltunk a válaszidő szórásában. Ez egyáltalán nem megengedhető számunkra, ha tartani szeretnénk a szigorú elvárásokat, amiket a felhasználók támasztanak a hálózattal szemben. Így valószínűsíthető, hogy ilyen formában nem leszünk képesek virtualizált mobil hálózatokban garantálni a kis mértékű késleltetést. Önálló konténer formában azonban használható lehetne arra, hogy a jövő virtuális hálózataiban dinamikusan skálázhatóvá váljanak a Home Agent és ahhoz hasonló hálózati

funkciók. Ezt a mérési eredmények is alátámasztják, az átlagos válaszidőben 351%-kal hatékonyabbnak találtuk a sidecar-t alkalmazó esethez képest.

A dolgozatunk során letesztelt módszerek útmutatást nyújthatnak a virtualizált hálózati funkciót kutató szakemberek számára. Mivel a dolgozatban minden elméleti és gyakorlati lépést dokumentáltunk, ezért akik hasonló szolgáltatást akarnak készíteni, képesek lesznek ezek alapján elindulni a VNF-ek világába. Megoldásaink relevánsak lehetnek telco cégek számára, akik a felhő infrastruktúrák iránt érdeklődnek és szeretnék a következő szintre emelni saját rendszereiket. Ezen kívül jó kiindulópont lehet abban az esetben is, ha a már meglévő hálózatuk még IPv4 címzést használ, de félnek megtenni az első lépéseket az IPv6 felé. A dolgozatban fontosnak tartottuk az IPv6 használatát, mert bár az újfajta címzéshez hozzá kell szokni, egy igazán intuitív és nagyszerű technológiát ismerhettünk meg az IPv6-ban.

Végül, de nem utolsó sorban szerettük volna mi magunk is megtapasztalni, hogy mit jelent egy teljes körű, valós felhő infrastruktúrában alapvető szolgáltatást tervezni és fejleszteni, a mai mobil hálózatok által felállított speciális követelményrendszer mentén. Végig ez a kíváncsiság vezérelt minket, amikor ezzel dolgoztunk. A tapasztalataink természetesen vegyesek. Látszik, hogy a Kubernetesben rengeteg potenciál rejlik és a hozzá kapcsolódó technológiák, mint az NSM ezt még inkább megsokszorozzák. Azonban sok esetben még fel nem tárt ösvényekre, kutatómunkát igénylő problémákra leltünk, ami elrettentő lehet azok számára, akik csak ki akarják próbálni a konténer orkesztrációt. Reméljük, hogy ezzel a dolgozattal mindezek ellenére megmutattuk, hogy egyáltalán nem lehetetlen hasonló hálózati funkciók futtatása konténerizált környezetben, és hogy ez egy meglepően hatékony megoldás lehet jövő hálózatainak kihívásaira.

Irodalomjegyzék

[1] „Ericsson Mobility Report June 2022”, https://www.ericsson.com/49d3a0/assets/local/reports-papers/mobility-report/documents/2022/ericsson-mobility-report-june-2022.pdf.

[2] D.-H. Luong, H.-T. Thieu, A. Outtagarts, és Y. Ghamri-Doudane, „Cloudification and Autoscaling Orchestration for Container-Based Mobile Networks toward 5G: Experimentation, Challenges and Perspectives”, in 2018 IEEE 87th Vehicular Technology Conference (VTC Spring), 2018, o. 1–7. doi: 10.1109/VTCSpring.2018.8417602.

[3] S. Hirai, T. Tojo, S. Seto, és S. Yasukawa, „Automated Provisioning of Cloud-Native Network Functions in Multi-Cloud Environments”, in 2020 6th IEEE Conference on Network Softwarization (NetSoft), 2020, o. 1–3. doi: 10.1109/NetSoft48620.2020.9165343.

[4] L. Jouin, „Network Service Mesh Solving Cloud Native IMS Networking Needs”, Master’s Thesis, Uppsala University, Department of Information Technology, 2020.

[5] F. Nada, „Performance Analysis of Mobile IPv4 and Mobile IPv6.”, Int. Arab J. Inf. Technol., köt. 4, sz. 2, o. 153–160, 2007.

[6] „Network Service Mesh”, https://networkservicemesh.com/.

[7] B. Martens, M. Walterbusch, és F. Teuteberg, „Costing of Cloud Computing Services: A Total Cost of Ownership Approach”, in 2012 45th Hawaii International Conference on System Sciences, 2012, o. 1563–1572. doi: 10.1109/HICSS.2012.186.

[8] „Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model”, https://www.iso.org/standard/20269.html.

[9] „Windows Network Architecture and the OSI Model”, https://learn.microsoft.com/en-us/windows-hardware/drivers/network/windows-network-architecture-and-the-osi-model.

[10] C. E. Perkins, „IP Mobility Support”, sz. 2002. RFC Editor, 1996. [Online]. Elérhető: https://www.rfc-editor.org/info/rfc2002

[11] „IP Mobility: Mobile IP Configuration Guide, Cisco IOS Release 15M&T”, https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/mob_ip/configuration/15-mt/mob-ip-15-mt-book/imo-hm-agt-pr.html.

[12] „The RIPE NCC has run out of IPv4 Addresses”, https://www.ripe.net/publications/news/about-ripe-ncc-and-ripe/the-ripe-ncc-has-run-out-of-ipv4-addresses.

[13] D. B. Johnson, J. Arkko, és C. E. Perkins, „Mobility Support in IPv6”, sz. 6275. RFC Editor, 2011. [Online]. Elérhető: https://www.rfc-editor.org/info/rfc6275

[14] D. S. E. Deering és B. Hinden, „Internet Protocol, Version 6 (IPv6) Specification”, sz. 8200. RFC Editor, 2017. [Online]. Elérhető: https://www.rfc-editor.org/info/rfc8200

[15] Á. Leiter, M. Saleh Salah, L. Pap, és L. Bokor, „Survey on PMIPv6-based Mobility Management Architectures for Software-Defined Networking”, INFOCOMMUNICATIONS JOURNAL, köt. 14, sz. 2, o. 2–18, 2022.

[16] „Docker vs Virtual Machines (VMs) : A Practical Guide to Docker Containers and VMs”, https://www.weave.works/blog/a-practical-guide-to-choosing-between-docker-containers-and-vms.

[17] „Docker overview”, https://docs.docker.com/get-started/overview/.
[18] „Kubernetes”, https://kubernetes.io/.
[19] „Concepts | Kubernetes”, https://kubernetes.io/docs/concepts/.
[20] „Kubernetes Components | Kubernetes”, https://kubernetes.io/docs/concepts/overview/components/.
[21] „CNI Plugins Overview”, https://www.cni.dev/plugins/current/.
[22] F. Z. Yousaf, M. Bredel, S. Schaller, és F. Schneider, „NFV and SDN—Key Technology Enablers for 5G Networks”, IEEE Journal on Selected Areas in Communications, köt. 35, sz. 11, o. 2468–2478, 2017, doi: 10.1109/JSAC.2017.2760418.
[23] J. Ordonez-Lucena, P. Ameigeiras, D. Lopez, J. J. Ramos-Munoz, J. Lorca, és J. Folgueira, „Network Slicing for 5G with SDN/NFV: Concepts, Architectures, and Challenges”, IEEE Communications Magazine, köt. 55, sz. 5, o. 80–87, 2017, doi: 10.1109/MCOM.2017.1600935.
[24] „Architecture - The Kubebuilder Book”, https://book.kubebuilder.io/architecture.html.
[25] „netdevice(7) — Linux manual page”, https://man7.org/linux/man-pages/man7/netdevice.7.html.
[26] „ioctl(2) — Linux manual page”, https://man7.org/linux/man-pages/man2/ioctl.2.html.
[27] „linux/socket.h at master · torvalds/linux”, https://github.com/torvalds/linux/blob/master/include/linux/socket.h#L241.
[28] „socket(2) — Linux manual page”, https://man7.org/linux/man-pages/man2/socket.2.html.
[29] „netlink(7) — Linux manual page”, https://man7.org/linux/man-pages/man7/netlink.7.html.
[30] „Netlink Library (libnl) - 2.2. Message Format”, https://www.infradead.org/~tgr/libnl/doc/core.html#core_msg_format.
[31] T. Jinmei, W. R. Stevens, és E. Nordmark, „Advanced Sockets Application Program Interface (API) for IPv6”, sz. 3542. RFC Editor, 2003. [Online]. Elérhető: https://www.rfc-editor.org/info/rfc3542
[32] „Tenacher/prairie-operator”, https://github.com/Tenacher/prairie-operator.
[33] „Annotations | Kubernetes”, https://kubernetes.io/docs/concepts/overview/working-with-objects/annotations/.
[34]: : „Submariner k8s project documentation website”, https://submariner.io/.
[35] „Architecture :: Submariner k8s project documentation website”, https://submariner.io/getting-started/architecture/.
[36] „Broker :: Submariner k8s project documentation website”, https://submariner.io/getting-started/architecture/broker/.
[37] „Network Service Mesh | NSM Concepts”, https://networkservicemesh.com/docs/concepts/enterprise_users/.
[38] „Network Service Mesh: A Big Step Toward Cloud-Native NFV-赵化冰的博客 | Zhaohuabing Blog”, https://www.zhaohuabing.com/post/2020-02-21-network-service-mesh-english/.
[39] „SPIFFE – Secure Production Identity Framework for Everyone”, https://spiffe.io/.
[40] „deployments-k8s/floating_interdomain_concept.png at main · networkservicemesh/deployments-k8s”, https://github.com/networkservicemesh/deployments-k8s/blob/main/examples/floating_interdomain/floating_interdomain_concept.png.

[41] „ipv6 urls by acnodal-tc · Pull Request #1369 · networkservicemesh/sdk”, https://github.com/networkservicemesh/sdk/pull/1369.
[42] K. Chowdhury, K. Leung, B. Patil, V. Devarapalli, és S. Gundavelli, „Proxy Mobile IPv6”, sz. 5213. RFC Editor, 2008. [Online]. Elérhető: https://www.rfc-editor.org/info/rfc5213
[43] S. Matsushima, C. Filsfils, M. Kohno, P. Camarillo, D. Voyer, és C. E. Perkins, „Segment Routing IPv6 for Mobile User Plane”, Internet Engineering Task Force, Internet-Draft draft-ietf-dmm-srv6-mobile-uplane-21, 2022. [Online]. Elérhető: https://datatracker.ietf.org/doc/draft-ietf-dmm-srv6-mobile-uplane/21/
[44] Rethinking Kubernetes networking with SRv6 and Contiv-VPP. [Online Video]. Elérhető: http://mirroronet.pl/pub/mirrors/video.fosdem.org/2020/H.1308/rethinking_kubernetes_networking_with_srv6.webm
[45] B. Dab, I. Fajjari, M. Rohon, C. Auboin, és A. Diquélou, „Cloud-native Service Function Chaining for 5G based on Network Service Mesh”, in ICC 2020 - 2020 IEEE International Conference on Communications (ICC), 2020, o. 1–7. doi: 10.1109/ICC40277.2020.9149045.
[46] „rtnetlink(7) — Linux manual page”, https://man7.org/linux/man-pages/man7/rtnetlink.7.html.
[47] „ipv6(7) — Linux manual page”.
[48] „inet_pton(3) — Linux manual page”, https://man7.org/linux/man-pages/man3/inet_pton.3.html.
[49] „Netlink Protocol Library Suite (libnl)”, https://www.infradead.org/~tgr/libnl/.
[50] „recv(2) — Linux manual page”, https://man7.org/linux/man-pages/man2/recv.2.html.
[51] „deployments-k8s/examples/floating_interdomain at main · networkservicemesh/deployments-k8s”, https://github.com/networkservicemesh/deployments-k8s/tree/main/examples/floating_interdomain.

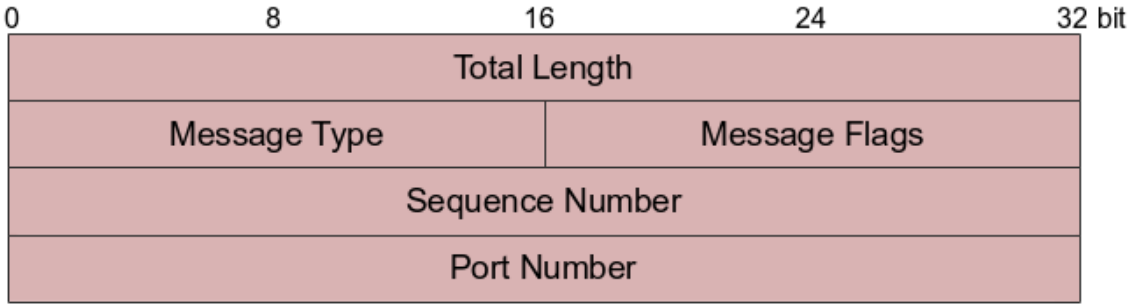
Ábrajegyzék

Táblázatok jegyzéke

9 Függelék

Függelék A: Netlink üzenet fejléc

A beállított paraméterektől függetlenül minden netlink üzenet a következő formátumot veszi fel: netlink message header, payload. Ahol a rakomány természetesen a fejléc által jelzett típusú lesz. Nézzük meg először is a fejlécet. [30]



33. ábra: Netlink csomagfejléc

Láthatjuk, hogy be kell állítanunk az üznenet hosszát, a típusát és egyéb flageket. A szekvencia szám egyszerűen az üzenetek számontartására van. De ami fontos lesz számunkra az a port number, megtévesztően a linux definícióban pid-ra van rövidítve, mint port ID, de ez nem feltétlenül a folyamat PID-ját jelenti. Jelentheti egyébként, az is volt az általános eljárás, hogy a socket portjának a PID-t állítják be. Ez egészen addig működött, amíg meg nem jelentek a több szálon futó alkalmazások.

Függelék B: Netlink üzenet létrehozás egy példán keresztül

A netlink rakományok netlink családtól és header típustól függően változnak, de végigvezethetjük a `NETLINK_ROUTE` példáján, hiszen a kódban is ezt fogjuk használni az interfészek létrehozására, azok manipulására.[46]

Az üzenet netlink header részével már találkoztunk, de érdemes a típus mezőt kiemelni, ahol csak `RTM_*` alakú előre definiált érték szerepelhet. Itt tudatjuk a kernellel, hogy milyen operációt szeretnénk végezni és hol. `RTM_NEWADDR` esetén például egy új címet adunk az adott interfésznek. Vezessük is le a címadás példáján a rakományt. Ennek egyébként az az ekvivalens parancsa az iproute2-es csomagban a következő hívás lenne, ezt fogjuk a következőkben megalkotni.

```
ip -6 addr add 2001:db8::1/32 dev eth0
```

Szabvány szerint egy `RTM_*ADDR` netlink üzenetben mindig egy ifaddrmsg szerepel, amit opcionálisan `rtattr`-nak (routing attribute) nevezett TLV értékek követnek. Az ifaddrmsg azonosítja az operandusként szereplő interfészt. A TLV értékekkel részletesen a Mobil IPv6 szabványt bemutató fejezetben foglalkoztunk, itt nem foglalkozunk ezek működésével, csak használni fogjuk őket.

Kezdésként definiáljuk a struktúránkat, amit a netlink üzenetünk létrehozására fogunk használni. Ezt a következőképpen tehetjük meg.[47, o. 6]

```
struct {
        struct ifaddrmsg ifa;
        struct rtattr rta;
        struct in6_addr n_addr;
} new_addr;
```

Figyeljük meg, hogy az előbb elmondottakkal összhangban megfelelően követik egymást a mezők. A végén azonban van egy `in6_addr` típusú mező, ezt tartalmazza magában a TLV. Ez az új struktúra pedig lényegében az új címünk bináris reprezentációját tárolja. Erről bővebben a kernel dokumentációjában olvashatnak. De mindenekelőtt nézzük meg a ifaddrmsg felépítését.

```
struct ifaddrmsg {
          unsigned char ifa_family;    /* Address type */
          unsigned char ifa_prefixlen; /* Prefixlength of
address */
          unsigned char ifa_flags;     /* Address flags */
          unsigned char ifa_scope;     /* Address scope */
          unsigned int  ifa_index;     /* Interface index
*/
};
```

Az ifa_family értelemszerűen az AF_* értékek egyike lesz. Az ifa_prefixlen az alhálózati maszk hosszát jelenti, az ifa_scope a cím érvényességi zónáját definiálja, ez lehet globális, link-local, vagy unique local IPv6 esetén, az ifa_index pedig a célinterfész számát jelöli. Az ifa_flags-zel egyéb információt adhatunk át a kernellel, de ezek nem fontosak jelen célunk elérése érdekében.

Most hogy az elméletet átvettük, alkossuk meg az ifaddrmsg-et amit át akarunk adni a kernelnek!

```
new_addr.ifa.ifa_family = AF_INET6;
new_addr.ifa.ifa_index = 2;
new_addr.ifa.ifa_prefixlen = 32;
new_addr.ifa.ifa_scope = RT_SCOPE_UNIVERSE;
```

Természetesen az ifa_family-t `AF_INET6`-ra állítjuk, hiszen IPv6-os címről van szó. Az indexet kettesre állítottuk, tegyük fel, hogy a loopback (egyes) interfészt rögtön követi az `eth0`-s, így annak kettes indexe lesz. A prefixlen-t 32-re kell állítani, mert a fenti iproute2 parancsban is ezt adtuk meg alhálónak, az érvényességi zónát pedig beállítjuk globálissá, ezt `RT_SCOPE_UNIVERSE`-ként definiálták a forráskódban. Vegyük észre, hogy még nem látjuk a fentebb példaként felhozott `2001:db8::1` címet. Ez nem véletlen, ugyanis ezt egy routing attribútumként adjuk át a kernelnek. Nézzük meg ennek is mezőit.

```
struct rtattr {
        unsigned short rta_len;    /* Length of option */
        unsigned short rta_type;   /* Type of option */
        /* Data follows */
};
```

Tipikus TLV felépítése van, ezért egyáltalán nem bonyolult. Hossz, típus és változó hosszúságú adatból áll. Többféle típust is beállíthatunk rta_type-nak, a dokumentációban szerepel például `IFA_LOCAL`, `IFA_LABEL`, stb. Az rta_len egyértelműen az adat hosszúságát jelöli plussz az RTA fejléc. Ezek tudatában adjuk át RTA-ként az új címet.

```
new_addr.rta.rta_type = IFA_LOCAL;
new_addr.rta.rta_len      =      RTA_LENGTH(sizeof(struct
in6_addr));
memcpy(&(new_addr.n_addr),     address,     sizeof(struct
in6_addr));
```

Típusnak az `IFA_LOCAL`-t választottuk, hiszen egy új L3 címet adunk át. A hossz kiszámítására egy makrót használunk, amit ha kifejtünk, ezt kapjuk:

```
sizeof(struct rtattr) + sizeof(struct in6_addr)
```

[48]

Már csak annyi maradt, hogy az új címünk bináris reprezentációját belemásoljuk a megfelelő struktúrába, ezt pedig a `memcpy()` függvénnyel meg is tesszük. A bináris

reprezentációt egyébként az `inet_pton()` függvény segítségével kaphatjuk meg, ami átalakítja a szöveges címet binárissá.

Ezzel a végére is értünk példánknak, már csak annyit kell tennünk, hogy elküldjük az üzenetet a korábban létrehozott socketen keresztül. Fontos azonban megemlíteni, hogy a netlinkkel való munka nagyban lecsökkenthető, ha segédkönyvtárakat használunk. A dolgozatunkban például libnl-t használtunk, amit a referenciák között megtalálnak.[49]

Függelék C: Csomagforgalom raw socketeken:

Ebben a függelékben körbejárjuk a raw socket API-t a Linux kernelben, először megnézzük a csomagfogadást, ezt követően pedig a küldés technikáját és sajátosságait is megnézzük. Mindeközben megismerkedünk a socket opciók beállításával is.

Csomagfogadás raw socketeken

Raw socketen üzenetet küldeni nehezebb, mint fogadni, ezért a fogadással átvesszük az alapokat, majd áttérünk a küldésre. Kezdésnek, mint ahogy azt a netlink esetében is tettük, nyitnunk kell egy socketet, amit a következőképpen tehetünk meg.

```
int socket(int domain, int type, int protocol);
```

Az egész szám argumentumok ne tévesszék meg az olvasót, mind előre definiált értékek, amiket a *sys/socket.h* fájlban találunk. A *domain* `AF_INET6` lesz, hiszen IPv6-os socketet szertnénk nyitni, egyébként ez az érték lehetne `AF_UNIX` unix socketekre, vagy az előző fejezetben tárgyalt `AF_NETLINK`, de `AF_PACKET` is, amivel pedig a L2 interfészt nyitunk. A típus érték esetünkben `SOCK_RAW` lesz, mert a fejezet erről szól. Végül pedig a protokoll mező, amiben az IP fejlécet követő csomag típusát adjuk meg. Ez lehet `IPPROTO_UDP`, `IPPROTO_UDPLITE` is, de a mi esetünkben, mivel mobility header fejlécet tartalmazó csomagokat szeretnénk fogadni, `IPPROTO_MH`-t állítunk. Ennek az értéke egyébként megegyezik az IANA által kiállított next header értékkel, ami 135.

Ha megkaptuk a fájlleírót visszatérési értékként, már csak a fogadó strukturákat kell előkészíteni. Egyrészt egy buffer, amibe megkapjuk a csomagot, másrészt egy sockaddr_in6 struktúrát, amiből kiolvashatjuk a küldő címét.

```
const int PKT_LEN = 20000;
uint8_t buf[PKT_LEN];
struct sockaddr_in6 addr;
```

Hogy az üzenetet biztosan átvegyük, lefoglalunk egy 20 kB méretű buffert, és egy sockaddr_in6 struktúrát, ami a következőképpen néz ki.

```
struct sockaddr_in6 {
   sa_family_t  sin6_family;   /* AF_INET6 */
   in_port_t    sin6_port;  /* port number */
   uint32_t     sin6_flowinfo; /* IPv6 flow information */
   struct in6_addr sin6_addr;   /* IPv6 address */
   uint32_t     sin6_scope_id; /* Scope ID (new in 2.4) */
};
```

Ezek mindegyike magától értetődő, az in6_addr struktúrával pedig már találkoztunk, így nem fejtjük ki részletesebben. Annyit érdemes elmondani, hogy a port számot nem vesszük figyelembe, hiszen a mi esetünkben nem létezik a port mint fogalom, tekintve hogy nincs szállítási rétegbeli protokoll. Már csak el kell indítanunk a fogadást, amit a következőképpen tehetjük meg.

```
ssize_t recvfrom(int sockfd, void *restrict buf, size_t
len, int flags,
                    struct sockaddr *restrict src_addr,
              socklen_t *restrict addrlen);
```

[50, o. 2]

Ez a hihetetlenül sok paraméterből álló függvény egy blokkoló hívás, ami addig vár, amíg nem kapunk üzenetet. Ezt érdemes észben tartani kódszervezés szempontjából. Az első paraméter egy fájl leíró, amit régebben a `socket()` hívásból kaptunk. A *buf* az előbb létrehozott bufferre mutató pointer lesz, a *len* pedig a buffer hossza, ezért `PKT_LEN`-re állítjuk. A *flags* paraméterben egyéb paramétereket adhatunk, minthogy ne legyen blokkoló hívás, stb. Az utolsó kettő pedig a fogadó struktúránk, amibe beleírja a küldő címét, illetve annak hossza, ami egy `sizeof()` hívással megadható. Fontos azonban, hogy a függvény egy sockaddr struktúra mutatót vár, nem pedig sockaddr_in6, amit mi létrehoztunk. Hogy a többi address family-vel közös API-t adjon a kernel, át kell konvertálnunk egy ilyen mutatóvá, de mivel megadtuk, hogy IPv6-os socketről van szó, ezért sockaddr_in6-ként fogja kezelni.

```
socklen_t addr_len = sizeof(struct sockaddr_in6);
size_t packet_size = recvfrom(sock, &buf, PKT_LEN,0,
(struct sockaddr*)&addr, &addr_len);
```

Érdekesség, hogy a fogadó sockaddr hosszát pointerként adjuk át, ez azért van, mert ha nem fér bele, akkor a pointeren keresztül átállítja nagyobbra az addr_len értékét,

de csak kezdésként átadott addr_len hosszan másolja át a címet. A dokumentáció erről bővebben is ír.

Ha megérkezett a csomag, a `recvfrom()` továbbengedi a program folyást és a fogadott csomagot a bufferben érhetjük el, azon pointer aritmetikával végezhetünk műveleteket.

Csomagküldés raw socketen

Ha küldeni akarunk csomagokat, akkor hasonló koreográfiával találkozunk, annyi különbséggel, hogy sok opciót magunknak kell beállítani. Először is nyitunk egy socketet, amit az előzőekben már megtettünk, itt nincs változás.

A mi esetünkben az előbb létrehozott fogadó socket ugyan nem ellenőrzi a checksum mezőjét a Mobility Header-nek, de ez nem követendő példa, küldéskor ezt pedánsan beállítjuk. Ehhez egy `setsockopt()` hívást kell végrehajtani.

```
int setsockopt(int sockfd, int level, int optname,
                       const   void   *optval,   socklen_t
optlen);
[31]
```

Az első paraméter nem meglepő, a socket fájl leírója, amin végre akarjuk hajtani az operációt. A szint az OSI rétegekben futó protokollokat jelöli, itt állhat TCP, ha annak a tulajdonságát állítjuk, de a mi esetünkben az IPv6-nak fogjuk, ezért `IPPROTO_IPV6` fog állni. Harmadik paraméterként megadjuk a opció nevét, amit állítunk. Ezekből nagyon sok van és az RFC-ben ezekről részletesen olvashatunk, de felsorolunk párat: `IPV6_MTU`, `IPV6_MULTICAST_IF` és `IPV6_CHECKSUM`. Ezek közül az utolsót fogjuk használni, hogy beállíthassuk a checksumot.

Ez az opció egy offset értéket vár egész szám paraméterként. Az offset értéke megadja, hogy az IP header után hány bájttal található a felhasználó által adott üzenetben a checksum mező. Ha visszagondolunk, a Mobility Header fejlécében az első négy oktett, négy darab nyolc bites mező után következik a checksum mező, ezt kell beállítani. Négy oktett egyenlő négy bájt, tehát az offset értékének négy bájtot kell megadnunk. Ebből adódik az alábbi függvényhívás.

```
int offset = 4;
setsockopt(sock,  IPPROTO_IPV6,  IPV6_CHECKSUM,  &offset,
sizeof(int));
```

Az efféle hívások visszatérhetnek hibával is, itt a tömörség érdekében ezt kihagytuk, de éles kódban mindenképpen ellenőrizni kell e függvény visszatérési értékét.

Ezzel már majdnem kész a küldésre a socket, már csak annyi maradt, hogy beállítjuk a küldési címet, amit használni akarunk. Ezt a `bind()` függvény segítségével érhetjük el.

```
int bind(int sockfd, const struct sockaddr *addr,
                                socklen_t addrlen);
```

Az első paraméter a megszokott socket fájl leíró, a második a fogadásnál megbeszélt sockaddr struktúra, majd annak a hossza. Az átadott sockaddr-ben megadhatjuk a küldő IPv6-os címet, amire rákötjük a socketet. Nézzük meg ennek felparaméterezését.

```
struct sockaddr_in6 host;
host.sin6_family = AF_INET6;
host.sin6_port = 0;
memcpy(&host.sin6_addr, sender, sizeof(struct in6_addr));
```

A *sender* itt egy bináris alakban tárolt IPv6-os cím. Azon kívül fontos a portot kiemelni, amit nullára inicializálunk, hiszen nincs szállítási rétegünk. Ezzel készen is állunk a küldésre.

```
ssize_t sendto(int sockfd, const void *buf, size_t len,
int flags,
                    const struct sockaddr *dest_addr,
socklen_t addrlen);
```

A legtöbb paraméterrel már találkoztunk, így ezeket nem részletezem. Meg kell adni az küldendő üzenetet és annak a hosszát, a *flags* számunkra nem fontos, ezt hagyhatjuk nullán, illetve a címet is meg kell adnunk, ezt ugyanúgy kell, mint az előzőekben. A `sendto()` egy függvénycsalád része a `recvfrom()`-hoz hasonlóan. Létezik `send()` és `recv()` változata is ezeknek a függvényeknek, ahol nem kell megadni a cél, illetve forrás címet, de ezeket stream socketek esetén használjuk, nem pedig datagram.

Függelék D: Home Agent tesztelő script

A Home Agent tesztelés a következő Python szkript segítségével készült. Lényegében előre megírt parancsokat hajtunk végre a subprocess modul segítségével, az időmérést pedig a

process_time segítségével hajtottuk végre. A Home Agent-ek címeinek kigyűjtése a kubectl selector paramétere segítségével történt, ahol a megfelelő label-re kerestünk rá.

```
import time
import subprocess

get_ha_ips = ["kubectl", "get", "pods", "--selector=parent=ha-
test", "-o", "jsonpath='{.items[*].status.podIP}'"]
run_client = ["kubectl", "exec", "-c", "cl-a", "client", "--",
"./cl.out"]
delete_tunnel = ["kubectl", "exec", "-c", "cl-a", "client", "-
-", "ip", "link", "delete", "ip6tun0"]

def main():
    times = []
    proc = subprocess.run(get_ha_ips, capture_output=True,
text=True)
    ips = proc.stdout[1:-1].split() #start and end apostrophe
is removed by [1:-1]
    print(ips)

    for ip in ips:
        run_client_against_ha = run_client.copy()
        run_client_against_ha.append(ip)

        t0 = time.process_time()
        subprocess.run(run_client_against_ha)
        times.append((time.process_time()-t0)* 1000) #time in
ms

        subprocess.run(delete_tunnel)

    print(times)

if __name__ == "__main__":
    main()
```

Függelék E: Network Service Mesh implementálása

A következőkben a Network Service Mesh Floating Interdomain szcenáriójában alkalmazható, Client-Endpoint összeköttetés megvalósítását mutatjuk be. A Floating Interdomain szcenárió telepítési útmutatója a projekt GitHub oldalán van részletezve[51]. Az itt alkalmazott Client és Endpoint leírók megfelelnek a X. fejezetben bemutatott ‘VPP VXLAN’ továbbítási mechanizmust megvalósító mérési környezetnek.

Az NSM végső telepítési fázisa után, mikor már a harmadik, Floating registry-t futtató klaszter is működőképes, nekiláthatunk az Endpoint telepítésének. Az alkalmazott

leíró alapját egy nse-kernel alkalmazás leírója adja, melyben a következő környezeti változók megadásával tesszük a futásra alkalmassá az Endpointot:

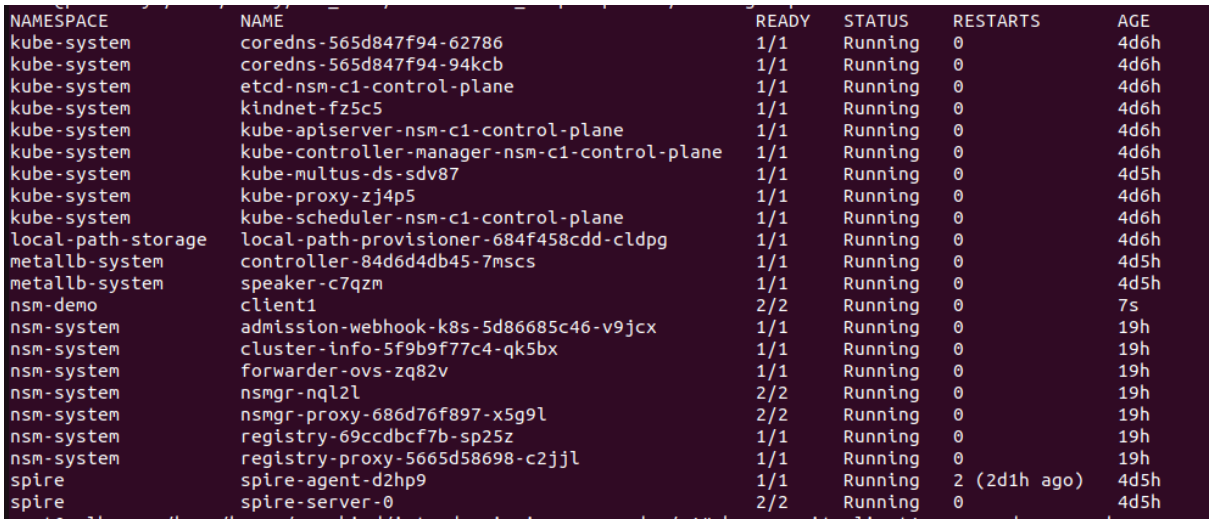
```
…
spec:
      containers:
        - name: nse
          …
          env:
            - name: NSM_NAME
              value: nsendpoint1@my.cluster3
            - name: NSM_CIDR_PREFIX
              value: 172.25.1.4/31
            - name: NSM_SERVICE_NAMES
              value: netsvc-vpp-vxlan@my.cluster3
            - name: NSM_PAYLOAD
              value: ETHERNET
…
```

A kliens Pod leírója a következőképpen néz ki:

```
apiVersion: v1
kind: Pod
metadata:
  name: client1
  labels:
    app: alpine
  annotations:
    networkservicemesh.io: kernel://my-networkservice-
ip@my.cluster3/nsm-1
    k8s.v1.cni.cncf.io/networks: macvlan
spec:
  containers:
    - name: alpine
      image: praqma/network-multitool
      imagePullPolicy: IfNotPresent
      stdin: true
      tty: true
      securityContext:
        privileged: true
        capabilities:
          add:
            - NET_ADMIN
```

Ezt követően telepíthetjük is az erőforrásokat. Ha minden várakozásunknak megfelelően történt, akkor a következőket kellene látnunk:

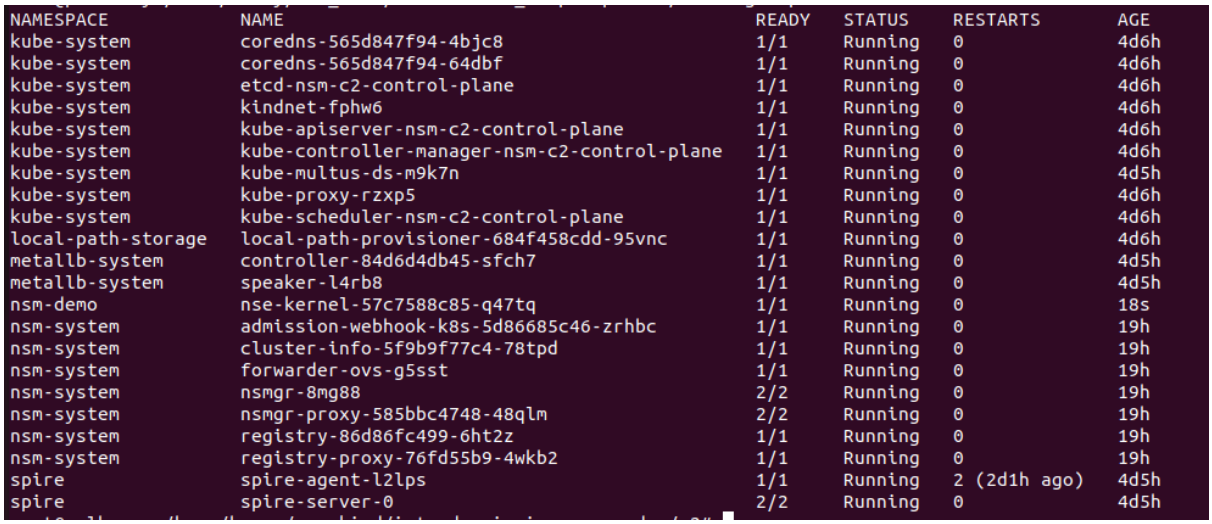
Az egyik klaszteren (client1 nevű pod, az nsm-demo névtérben):



NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
kube-system	coredns-565d847f94-62786	1/1	Running	0	4d6h
kube-system	coredns-565d847f94-94kcb	1/1	Running	0	4d6h
kube-system	etcd-nsm-c1-control-plane	1/1	Running	0	4d6h
kube-system	kindnet-fz5c5	1/1	Running	0	4d6h
kube-system	kube-apiserver-nsm-c1-control-plane	1/1	Running	0	4d6h
kube-system	kube-controller-manager-nsm-c1-control-plane	1/1	Running	0	4d6h
kube-system	kube-multus-ds-sdv87	1/1	Running	0	4d5h
kube-system	kube-proxy-zj4p5	1/1	Running	0	4d6h
kube-system	kube-scheduler-nsm-c1-control-plane	1/1	Running	0	4d6h
local-path-storage	local-path-provisioner-684f458cdd-cldpg	1/1	Running	0	4d6h
metallb-system	controller-84d6d4db45-7mscs	1/1	Running	0	4d5h
metallb-system	speaker-c7qzm	1/1	Running	0	4d5h
nsm-demo	client1	2/2	Running	0	7s
nsm-system	admission-webhook-k8s-5d86685c46-v9jcx	1/1	Running	0	19h
nsm-system	cluster-info-5f9b9f77c4-qk5bx	1/1	Running	0	19h
nsm-system	forwarder-ovs-zq82v	1/1	Running	0	19h
nsm-system	nsmgr-nql2l	2/2	Running	0	19h
nsm-system	nsmgr-proxy-686d76f897-x5g9l	2/2	Running	0	19h
nsm-system	registry-69ccdbcf7b-sp25z	1/1	Running	0	19h
nsm-system	registry-proxy-5665d58698-c2jjl	1/1	Running	0	19h
spire	spire-agent-d2hp9	1/1	Running	2 (2d1h ago)	4d5h
spire	spire-server-0	2/2	Running	0	4d5h

34. ábra: Egyik klaszter Podjai

Illetve a másikon (nse-kernel deploymentből származó Pod, az nsm-demo névtérben):



NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
kube-system	coredns-565d847f94-4bjc8	1/1	Running	0	4d6h
kube-system	coredns-565d847f94-64dbf	1/1	Running	0	4d6h
kube-system	etcd-nsm-c2-control-plane	1/1	Running	0	4d6h
kube-system	kindnet-fphw6	1/1	Running	0	4d6h
kube-system	kube-apiserver-nsm-c2-control-plane	1/1	Running	0	4d6h
kube-system	kube-controller-manager-nsm-c2-control-plane	1/1	Running	0	4d6h
kube-system	kube-multus-ds-m9k7n	1/1	Running	0	4d5h
kube-system	kube-proxy-rzxp5	1/1	Running	0	4d6h
kube-system	kube-scheduler-nsm-c2-control-plane	1/1	Running	0	4d6h
local-path-storage	local-path-provisioner-684f458cdd-95vnc	1/1	Running	0	4d6h
metallb-system	controller-84d6d4db45-sfch7	1/1	Running	0	4d5h
metallb-system	speaker-l4rb8	1/1	Running	0	4d5h
nsm-demo	nse-kernel-57c7588c85-q47tq	1/1	Running	0	18s
nsm-system	admission-webhook-k8s-5d86685c46-zrhbc	1/1	Running	0	19h
nsm-system	cluster-info-5f9b9f77c4-78tpd	1/1	Running	0	19h
nsm-system	forwarder-ovs-g5sst	1/1	Running	0	19h
nsm-system	nsmgr-8mg88	2/2	Running	0	19h
nsm-system	nsmgr-proxy-585bbc4748-48qlm	2/2	Running	0	19h
nsm-system	registry-86d86fc499-6ht2z	1/1	Running	0	19h
nsm-system	registry-proxy-76fd55b9-4wkb2	1/1	Running	0	19h
spire	spire-agent-l2lps	1/1	Running	2 (2d1h ago)	4d5h
spire	spire-server-0	2/2	Running	0	4d5h

35. ábra: másik klaszter Podjai

A Client shelljéhez hozzáférve tesztelni tudjuk az NSM által beinjektált interfészt:

```
Defaulted container "alpine" out of: alpine, cmd-nsc, cmd-nsc-init (init)
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0@if71: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 86:03:f1:65:f4:b1 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.101.50/24 brd 192.168.101.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::8403:f1ff:fe65:f4b1/64 scope link
       valid_lft forever preferred_lft forever
3: net1@if12: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 06:b4:8d:b7:69:bd brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.100.5/24 brd 192.168.100.255 scope global net1
       valid_lft forever preferred_lft forever
    inet6 fe80::4b4:8dff:feb7:69bd/64 scope link
       valid_lft forever preferred_lft forever
73: nsm-1@if72: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 16000 qdisc noqueue state UP group default
    link/ether a2:7c:9c:18:58:d8 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.25.1.5/32 scope global nsm-1
       valid_lft forever preferred_lft forever
    inet6 fe80::a07c:9cff:fe18:58d8/64 scope link
       valid_lft forever preferred_lft forever
/ # ping 172.25.1.4
PING 172.25.1.4 (172.25.1.4) 56(84) bytes of data.
64 bytes from 172.25.1.4: icmp_seq=1 ttl=64 time=0.124 ms
64 bytes from 172.25.1.4: icmp_seq=2 ttl=64 time=0.116 ms
64 bytes from 172.25.1.4: icmp_seq=3 ttl=64 time=0.208 ms
64 bytes from 172.25.1.4: icmp_seq=4 ttl=64 time=0.301 ms
64 bytes from 172.25.1.4: icmp_seq=5 ttl=64 time=0.335 ms
^C
--- 172.25.1.4 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4073ms
rtt min/avg/max/mdev = 0.116/0.216/0.335/0.089 ms
```

36. ábra: A Client Podban létrejött interfész

A Kind által felügyelt bridge interfészen áthaladó forgalmat vizsgálva hasonló csomagokat is kellene látnunk:

```
    27 0.087542     172.25.1.5        172.25.1.4        ICMP      108 Echo (ping) request  id=0xcbc7, seq=2/512, ttl=64 (reply in 28)
    28 0.087774     172.25.1.4        172.25.1.5        ICMP      108 Echo (ping) reply    id=0xcbc7, seq=2/512, ttl=64 (request in 27)
▸ Frame 27: 108 bytes on wire (864 bits), 108 bytes captured (864 bits)
▸ Ethernet II, Src: 02:42:ac:12:00:02 (02:42:ac:12:00:02), Dst: 02:42:ac:12:00:03 (02:42:ac:12:00:03)
▸ Internet Protocol Version 4, Src: 172.18.0.2, Dst: 172.18.0.3
▸ User Datagram Protocol, Src Port: 41890, Dst Port: 4789
▸ Virtual eXtensible Local Area Network
▸ Ethernet II, Src: a2:7c:9c:18:58:d8 (a2:7c:9c:18:58:d8), Dst: 72:07:20:48:9d:04 (72:07:20:48:9d:04)
▸ Internet Protocol Version 4, Src: 172.25.1.5, Dst: 172.25.1.4
▸ Internet Control Message Protocol
```