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

Matusovits Balázs Döme

eBPF ALAPÚ ADATKÖZPONTI HÁLÓZATI MEGOLDÁSOK FUNKCIONÁLIS ÉS PERFORMANCIA VIZSGÁLATA

KONZULENSEK

Leiter Ákos, Dr. Bokor László

BUDAPEST, 2023

Tartalomjegyzék

Összefoglaló 3
Abstract 4
1. Bevezetés 5
2. Network Slicing 8
3. Multi Protocol Label Switching (MPLS) 11
4. Segment Routing over IPv6 (SRv6) 16
5. Kubernetes 21
6. Iptables 24
7. eBPF 27
8. Cilium 31
9. Adatforgalom útvonala a kernelben kube-proxy vs. eBPF esetén 33
10. SRv6 csomagfeldolgozás implementálása Ciliumban 36
11. Performanciaelemzések áttekintése 38
11.1. Performanciaméréshez használt környezet összeállítása 39
11.2. Átviteli sebesség mérés IPv6-os csomagok esetén 40
11.3. Mérési eredmények kiértékelése 41
11.4. SRv6 alapú hálózat létrehozása Openstack-en 46
11.5. További fejlesztési irányok, összegzés 49
Ábra- és táblázatjegyzék 50
Irodalomjegyzék 52
Rövidítések jegyzéke 59
Függelék 62

Összefoglaló

Az utóbbi években a felhő alapú technológiák új kihívást jelentenek a távközlési szolgáltatóknak. A valós idejű adattovábbítás kulcsfontosságú szerepet tölt be számos területen, többek között az intelligens közlekedés során felmerülő azonnali beavatkozásoknál, videókonferenciánál, élő közvetítéseknél vagy bármilyen kliens-oldali alkalmazásoknál ahol megbízható, valós idejű kiszolgálást kell biztosítanunk a felhasználó számára. Az ötödik generációs mobilhálózatok megjelenésével ezt az igényt ki lehet szolgálni, köszönhetően egy újszerű technológiának a Hálózat szeletelésnek (Network Slicing). Ennek a segítségével képesek vagyunk hálózati szegmenseket elkülöníteni abból a célból, hogy a kívánt szolgáltatási igényeket folyamatosan kielégítsük. Az iparban jelenleg elterjedtek a Multi Protocol Label Switching (MPLS) alapon működő transzport- és maghálózatok, melyekben az MPLS lehetőséget nyújt a hálózatszeletelés támogatására, és szolgáltatási garanciát is képes biztosítani. A hátránya azonban az, hogy nagyon sok féle protokollt használ egy ilyen teljes vertikumú hálózat. Célszerű lenne olyan forgalomtovábbítási technológiára támaszkodni, amely az MPLS szolgáltatásait megtartva képes lenne egy egyszerűbb architektúrális működést lehetővé tenni, és támogatja a ráépülő maghálózati és adatközpont-hálózati igényeket is, egyben egységesebb protokollhierarchiát honosít meg. Az egyik lehetséges megoldás a Segment Routing over IPv6 (SRv6). Ennek a segítségével a hálózatszeletelés megvalósítható, azonban az MPLS-hez hasonlóan itt is szolgáltatási garanciát kell biztosítanunk. Az ehhez szükséges SRv6 alapú kiegészítő funkciók implementálásához jól használható a szintén újszerű extended Berkeley Package Filter (eBPF) technológia. Munkám során ezen technológiai együttes megoldásokat fogom vizsgálni és javaslatokat tenni a csomagfeldolgozás módjairól felhő alapú környezetben, különös tekintettel a Kubernetes, mint konténer orkesztrációs keretrendszer hálózat kezelésében betöltött szerepére.

Abstract

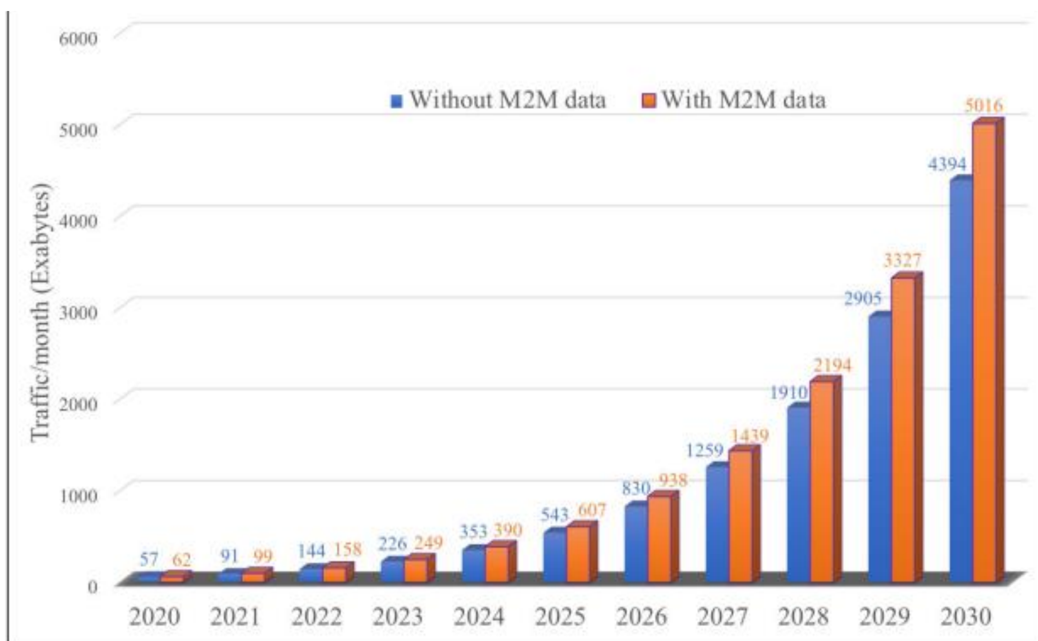
In recent years, cloud-native technologies have presented new challenges for telecommunications service providers. Real-time data transmission is crucial in various areas, such as immediate interventions in intelligent transportation, video conferencing, or client-side applications where reliable real-time service must be provided. With the advent of fifth-generation mobile cellular networks, this demand can also be met thanks to an innovative technology called Network Slicing. In terms of its operation, we can separate network segments to continuously satisfy the desired service requirements. In the industry, Multi-Protocol Label Switching (MPLS)-based transport and core networks are currently prevalent, offering the capability to support network slicing and provide traffic engineering. However, the drawback is that it uses a wide range of protocols within a complete vertical network. It would be advisable to rely on a traffic forwarding technology that, while retaining the services of MPLS, can facilitate a simpler architecture and support the overlay core and data center network requirements while also establishing a more unified protocol hierarchy. One possible solution is Segment Routing over IPv6 (SRv6). With this, network slicing can be achieved, but similar to MPLS, we must also provide traffic engineering here. That is where the eBPF technology comes in, which is also innovative and can be well utilized to implement the necessary SRv6-based additional features. During my work, I will examine these technological solutions and make recommendations regarding packet processing methods in a cloud-native environment regarding the Kubernetes container orchestration software's role in network management.

1. Bevezetés

A felhő alapú technológiák megjelenésével több probléma is felvetődött. Nagyon sok multimédiás alkalmazásnál, mint például videókonferencia, élőközvetítés, vagy a V2X alapú kommunikációra gondolva, alapvető követelmény, hogy az adattovábbítás valós időben történjen meg (1. ábra). Emellett a tartalomfogyasztási igény és általában véve a mobil adatforgalom rohamos növekedése is kihívás a mobilszolgáltatóknak (2. ábra).

Applications		Use cases	KPIs [2,7-8,10-12,19]	Class
Narrowband tactile Internet with haptic feedback	Others	Wearables/exoskeletons for healthcare, heavy-labour, and mission-critical	BLER 10^{-5} - 10^{-6}	
Future ITS	Fusion	Autonomous driving	E2E 5-10 ms	
	V2X	Smart highway, Smart road, Smart intersection	BLER 10^{-5}	
Future factory	Factory automation	Motion control	Cycle time < 0.5 ms Payload <50 bytes	Extreme URLLC
		Automated guided Vehicles	E2E 10-50 ms Payload < 300 bytes BLER 10^{-6}-10^{-9}	
		Manufacturing process	E2E < 10 ms Payload < 50 bytes BLER 10^{-9}	

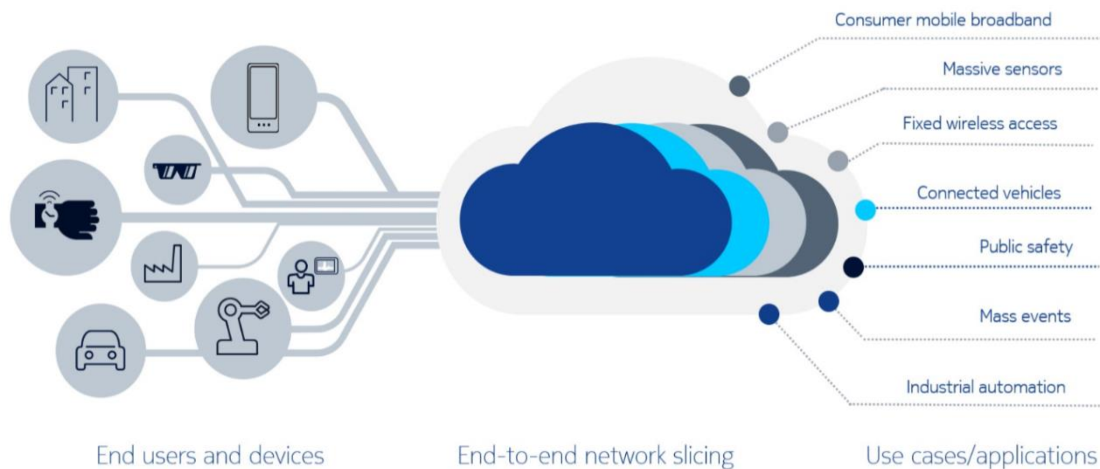
1. ábra: Végpontok közötti kritikus késleltetés követelmények [1]



2. ábra: A mobil adatforgalom exponenciális növekedése [2]

Ahhoz hogy megfelelő szolgáltatási garanciát biztosítsunk, különböző hálózati technológiák adottak, amit előszeretettel használnak az ötödik generációs

mobilhálózatokban. Ilyen például a Network Slicing [3], amellyel hálózati szegmenseket különítünk el abból a célból, hogy a kívánt szolgáltatási igényeket folyamatosan kielégíthessük. Ezek lehetnek a már korábban taglalt kritikus késleltetési követelményeket igénylő járműkommunikáció, távsebészet, nagy mennyiségű adatforgalom kiszolgálása (multimédiás tartalmak, virtuális valóság) vagy például mobilfelhasználók kiszolgálása nagy cellasűrűségű környezetben (3. ábra).



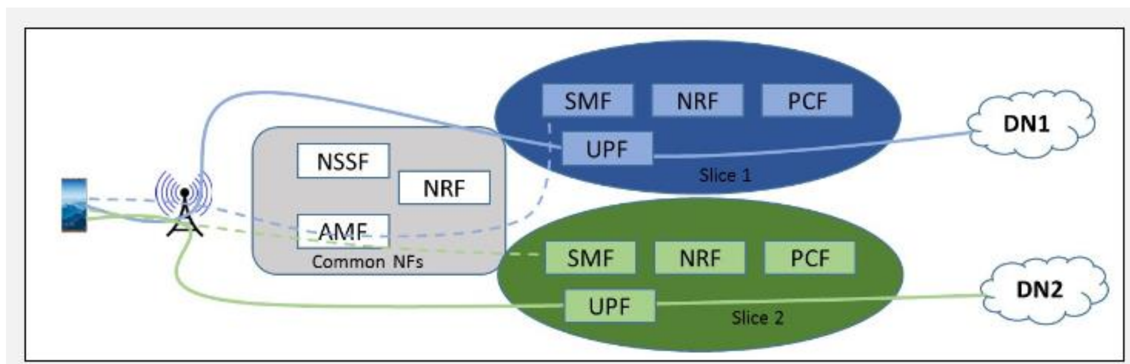
3. ábra: Network Slicing alkalmazási területei [4]

A jelenleg iparban használt transzport hálózatban az MPLS [5] van implementálva az útvonalválasztás megvalósítására. Ezzel a technológiával szolgáltatási garanciát (Traffic Engineering) tudunk biztosítani a hálózatban. Tulajdonképpen ezzel a forgalom útvonalának kiválasztása az erőforrások hatékony kihasználása mellett tehető meg. Az éppen aktuális hibás linkek is kikerülhetőek és a forgalom útvonala gyorsan újra számítható ennek megfelelően. Virtuális bérelt vonalat is megvalósíthatunk vele, hiszen end-to-end garancia biztosítható az átviteli sebességre. Azonban az MPLS és a ráépülő 5G maghálózat (beleértve az adatközponti specifikumokat) nagyon sok protokollt használ, ami komplexé teszi a működését, és nem mellesleg így a network slicing megvalósítása is kihívás elé teszi a távközlési szolgáltatókat. Célszerű lenne olyan forgalomtovábbítási technológiára támaszkodni, amely az MPLS-hez hasonlóan szolgáltatási garanciát képes nyújtani, és egyben képes lenne egy egyszerűbb architektúrális működést lehetővé tenni, és egyben egységesebb protokollhierarchiát valósítana meg. Ehhez nyújtana megoldást a Segment Routing over IPv6 (SRv6) [6], ami képes lehet hálózati szegmensen áthatoló, egységes protokoll architektúrát nyújtani [7]. Ebben a megközelítésben az összes köztes csomópont a csomag útvonala során előre meg van határozva. Ezek egy úgynevezett szegmens listában vannak elhelyezve, ami egy IPv6

protokoll kiegészítés. Maga a szegmens lista utasításokból áll, amelyek tulajdonképpen speciális IPv6-os címek. Ezekből minden egyes router feldolgoz egyet-egyet, és így kerül továbbításra egy adott csomag az útvonala során. Ahhoz, hogy a Network Slicing megvalósításra kerüljön SRv6-al, az MPLS-hez hasonlóan szintén szükséges a Traffic Engineering megvalósítása. Az ehhez szükséges SRv6 alapú kiegészítő funkciók implementálásához jól használható a szintén újszerű extended Berkeley Package Filter (eBPF) technológia [8]. Tulajdonképpen ez egy RISC alapú virtuális gép a Linux kernelben, mellyel programozható az operációs rendszer működése anélkül, hogy a kernel forráskódját módosítanánk, vagy írnánk egy új kernel modult. Ez azért lényeges tulajdonság, mert annak ellenére, hogy egy nyílt forráskódú projekt fejlesztésébe rengetegen bekapcsolódnak, nem olyan ütemben fejlődnek bizonyos kiegészítések a kernel működését tekintve, mint amekkora igény lenne rá. Ezzel a megoldással azonban a kompatibilitási probléma megszűnne, és gyakorlatilag tetszőlegesen írható lenne egy olyan SRv6 logika a csomagfeldolgozásra, ami a jelenlegi modulokban nincsen implementálva. Nem mellesleg ezzel a technológiával a csomagfeldolgozási időn is lehet javítani, köszönhetően annak, hogy az eBPF programok segítségével a csomag nem megy végig a kernel protokoll rétegein, hanem a megfelelő ponthoz érve egyből átirányításra kerül a cél interfészhez. Munkám során ezen technológiai együttes megoldásokat fogom vizsgálni, és javaslatokat tenni a csomagfeldolgozás módjairól felhő alapú környezetben. A dolgozatom első felében összehasonlítom az MPLS, és az SRv6 protokollokat, valamint a szolgáltatásgarancia biztosítására vonatkozó megfontolásokat (Traffic Engineering). Ezt egy rövid elméleti összefoglaló fogja követni a a Kubernetes, mint konténer orkesztrációs szoftver alapvető működéséről. Majd bemutatásra kerül az eBPF, és ennek kapcsán a Cilium működése, architektúrája, valamint, hogy miként képesek támogatni a Kubernetes adatközpontban való működését. Ezután következik a tesztrendszer összeállításának részletezése, melynek során egy adatközpontot reprezentáló környezetet hozok létre. Teljesítménymérést végzek arra vonatkozóan, mennyivel hatékonyabb az eBPF programok használata két végpont közötti kommunikáció során. A tesztrendszerem egy Kubernetes alapú adatközponti hálózati környezet, melyben a Cilium konténer hálózati interfész kerülne implementálásra. Végül elvégzem a mérést SRv6 enkapszulációval is, és kiértékelem az eredményeket.

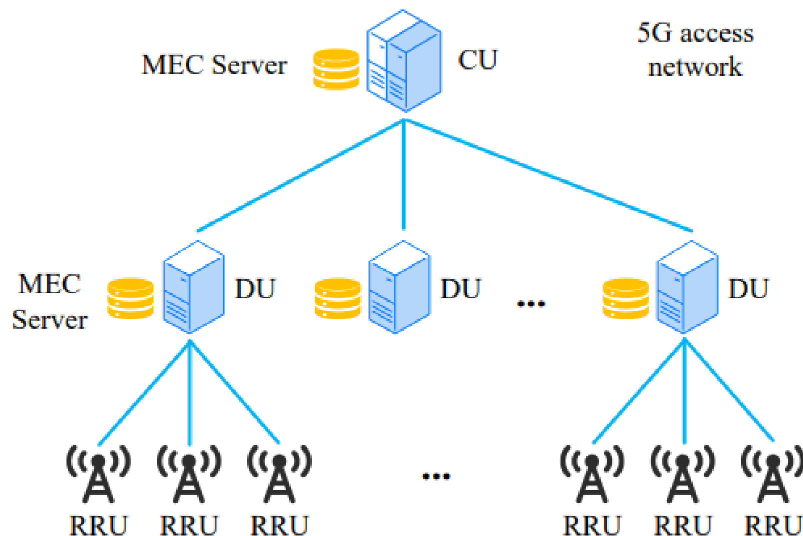
2. Network Slicing

Az ötödik generációs hálózatok egyik legnagyobb technológiai áttörései tartozik a hálózatszeletelés, angol terminológiában pedig Network Slicing néven utalunk rá. Ennek segítségével különböző felhasználói igények folyamatosan kielégíthetők úgy, hogy közben erre megfelelő 5G hálózati komponenseket allokálunk, hogy külön-külön egy adott szolgáltatást, úgynevezett slice-t szolgáljon ki. Többféle szcenárió létezik a komponensek hovatartozását illetően, ugyanis bizonyos NF-ek, mint például az AMF tartozhat több slice alá is, de akár külön-külön egy AMF létezhet szeletenként is (4. ábra). Ez mind implementáció kérdése, hogy milyen követelményeknek kell éppen megfelelni.



4. ábra: Hálózati szeletek realizálása [9]

A technológia megvalósításához szorosan fűződik egy szintén újszerű és egyben elengedhetetlen építőelem, ami nem más, mint az Edge Computing. A fő koncepció, hogy az adatok elosztottan kerülnek tárolásra helyi szervereken, ezáltal a késleltetést csökkentve, és jelentős overhead-et megtakarítva. Így az adatnak nem kell keresztül mennie az aggregációs és maghálózat elemeken. Cserébe a komplexitás növekszik, hiszen plusz réteges felhős szerkezet keletkezik: far-edge, edge, central stb. Ez esetünkben a rádióállomás funkcionális egységeinek a szétválasztását, valamint az UPF-ek lokalizálását fogja jelenteni. A kívánt lefedettség elérése a fejegységek elhelyezésével kezdődik (RRU). Több fejegységet pedig egy úgynevezett elosztott egység (DU) fog össze, egy cellát csak egy DU kezel. Az elosztott egységek koordinálását pedig a központi egység (CU) végzi. Ezen felül ez biztosít interfészt az AMF és az UPF kapcsolódásához (5. ábra).

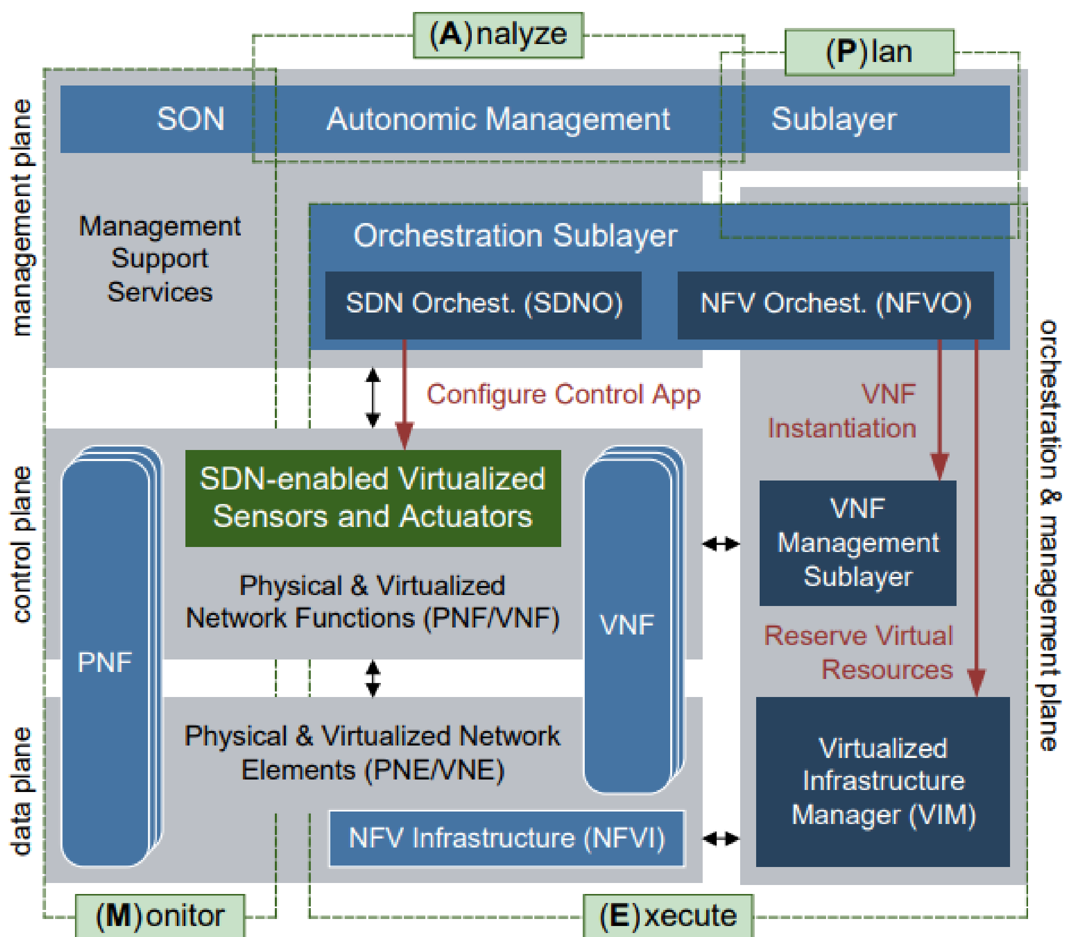


5. ábra: Az 5G-s rádió logikai egységei [10]

A network slicing megvalósításához elengedhetetlen egy virtualizált környezet létrehozása, hiszen a hálózati szeletek orkesztrálásához, üzemeltetéséhez egy jól skálázható rendszerre van szükségünk. Itt jön képbe a Software Defined Networking (SDN) [11] paradigma, amely az 5G infrastruktúrájának egyik leglényegesebb koncepciója, mondhatni fontos alkotóeleme. Alapvetően 3 fundamentális jellemzőt fontos itt megemlíteni:

- A vezérlő és a kontroll sík egyértelmű szétválasztása. Erre már kitértem az ötödik generációs mobilhálózat architektúrájának ismertetése során.
- A hálózat logikájának a hardvertől való elkülönítése, azaz a szoftverben való implementációja.
- Egy központi hálózati kontroller (NOS) jelenléte, amely gyakorlatilag az összes hálózati eszköz forgalomtovábbítását koordinálja.

Maga a virtualizációnak van egy másik aspektusa is, amely a hálózati erőforrások kezelését, felhőbe való kiszervezését, allokálását, újra felhasználását foglalja magába. El is érkeztünk a virtuális gépek, és a konténerek kérdésköréhez. Az 5G NF-eket gyakorlatilag ezen virtuális elemek halmaza alkotja. Ezeket Virtualized Network Function-nek (VNF), a mögöttes fogalom pedig a Network Function Virtualization (NFV) (6. ábra).

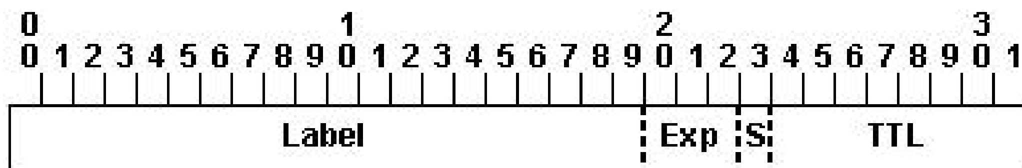


6. ábra: Az SDN és NFV koncepció a hálózatmenedzsmentben [12]

Ennek az infrastruktúrának köszönhetően végpontok közötti, logikailag elkülönített hálózati szegmensek realizálhatók mind a rádió, transzport, és maghálózati elemeken egyaránt. Az utóbbi kettő alapját az iparban jelenleg az MPLS technológiára alapozva valósítják meg. A következő fejezetekben bemutatom ennek a megoldásnak az architektúrális felépítését, működését, valamint az általa biztosítható szolgáltatási garanciákat. Majd kitérek a protokoll hátrányaira, milyen nehézségekkel kell megküzdeni a hálózatszeletelés implementációja során. Ezen keresztül bemutatásra kerülne az ezzel konkurens SRv6 technológia ugyanazon jellemzők mentén, ahogyan ezt az MPLS-nél is tenni fogom. Végül következtetéseket vonok le, hogy milyen előnyökkel jár ez az újfajta megközelítés.

3. Multi Protocol Label Switching (MPLS)

Az MPLS egy olyan csomagtovábbítási technológia, amely Labeleket használ az útvonalválasztáshoz. Ez egy 32 bites helyi azonosító, amelynek adatstruktúrája az alábbiak szerint épül fel (7. ábra).



7. ábra: Az MPLS Label felépítése [13]

Label: Ez maga a Label értéke (20 bit)

Exp: Későbbi használatra van félre téve, most jelenleg a Class of Service (CoS) mező szerepét tölti be, amely tulajdonképpen az adatkapcsolati réteg szolgáltatási garancia implementációja (3 bit)

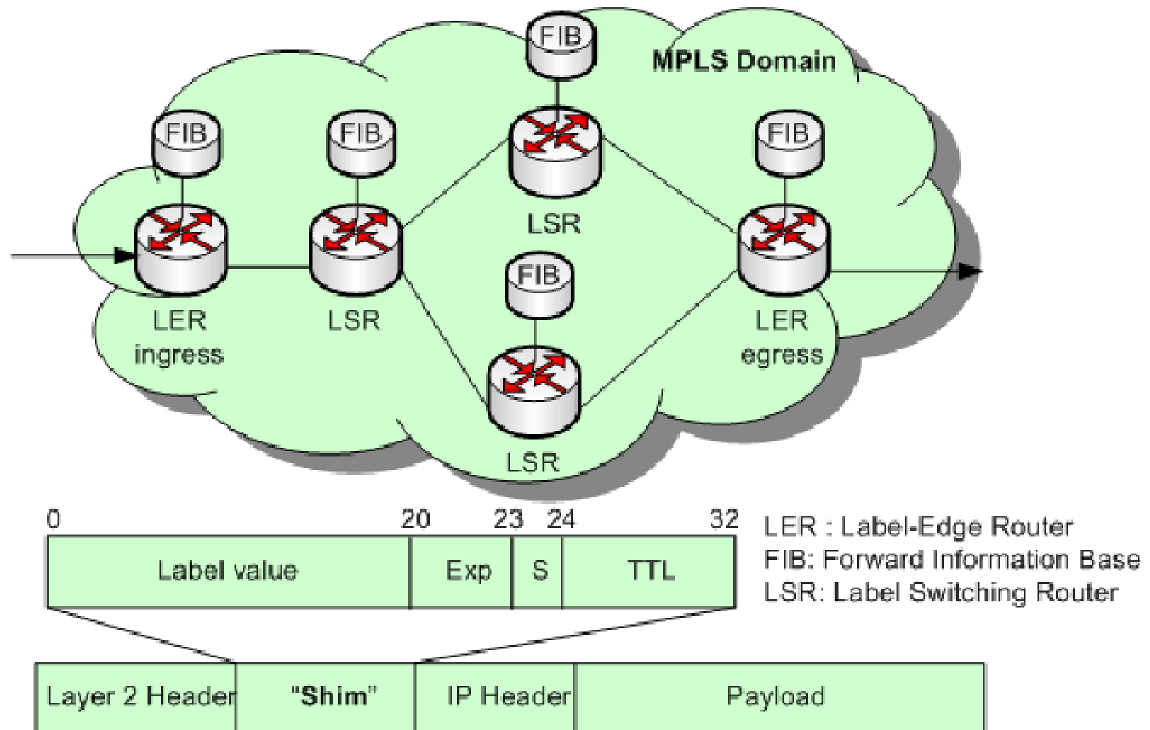
S: A verem végét jelöli (1 bit)

TTL: Time to Live (8 bit)

Ezek a Labelek csomagokhoz vannak hozzárendelve, így minden csomag bekerül ez alapján egy adott osztályba, melyet Forwarding Equivalence Class-nak (FEC) nevezünk. Egy FEC-hez tartozó csomagok összessége ugyanúgy vannak kezelve, azonos útvonalon kerülnek továbbításra, prioritásbeli kezelésük is megegyezik. Ez a hozzárendelés többféle szempont alapján is megvalósítható:

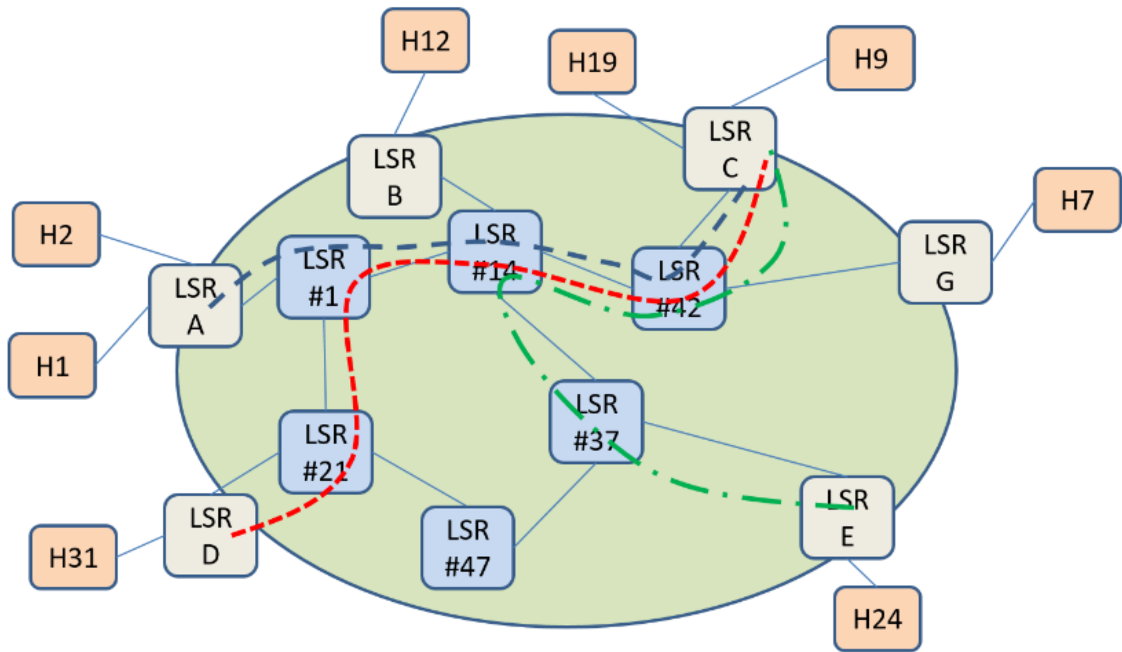
- Cél IP cím
- Forrás IP cím
- TCP/UDP port
- Forrás és cél AS
- CoS
- Applikáció
- Az előzőek kombinációi

Maga a címke (Label) a keret és a csomag fejléce közé ékelődik be (8. ábra). Így megvalósítható, hogy az IP csomag fejlécének feldolgozása csak az MPLS hálózati domén határainál történjen meg (LER), a doménen belül pedig Layer 2-es szinten történjen a csomagfeldolgozás (LSR-ek segítségével).



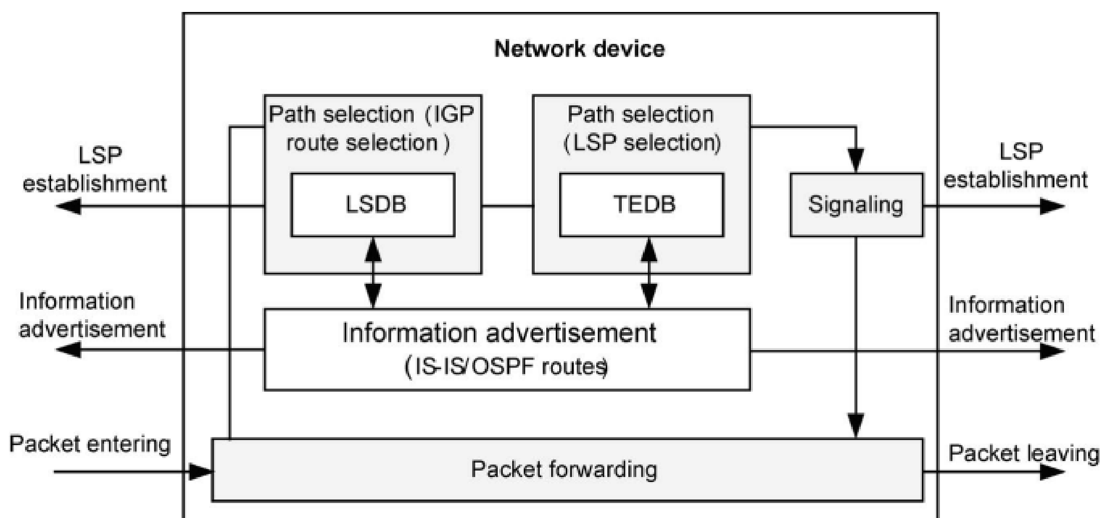
8. ábra: MPLS architektúrája [14]

Az IP csomag a domén határához érve a LER hozzárendeli a címkét. Innentől kezdve a doménen belül címkekapcsolás segítségével kerül továbbításra a forgalom. Ez egy kapcsolótábla segítségével történik (ábrán a FIB lesz), melyben a bemeneti port és címkéhez egyértelműen tartozik egy kimeneti port és címke páros. Ezzel az exakt találat kereséssel nagysebessségű csomagfeldolgozást lehet elérni, nem véletlen, hogy ez terjedt el az ötödik generációs távközlési hálózatokban. A forgalom miután megérkezett a túloldali LER-hez, a címke lekerül, és a csomag továbbításra kerül, elhagyva az MPLS domént. Fontos megjegyezni, hogy a csomag megfelelő FEC-hez rendelése egyszer történik meg, miután beérkezett az MPLS hálózatba. Mint ahogyan ezt már említettük, a FEC-hez egyértelműen tartozik egy útvonal a doménen belül. Ezeket nevezzük Label Switch Path-nek (LSP). Azonban fontos kiemelni, hogy ez mindig egy irányt határoz meg, azaz a visszairányra külön LSP-t kell definiálni. A működés szemléltetése céljából az alábbi ábrán (9. ábra) pirossal és zölddel jelölve van két külön útvonal.



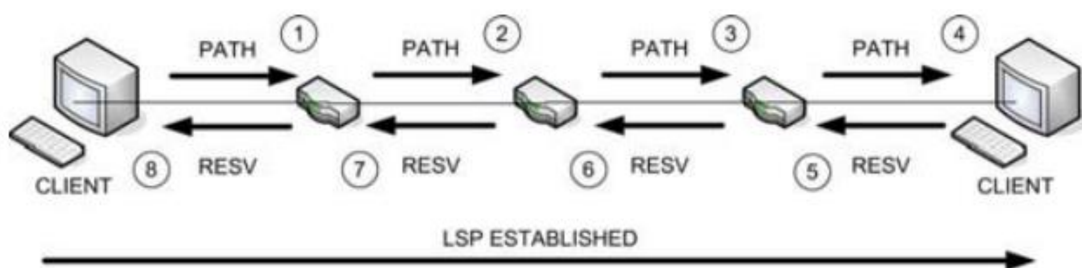
9. ábra: LSP-k szemléltetése [15]

A végpontok közötti szolgáltatási garancia biztosításához úgynevezett Traffic Engineering (TE) megoldásokat kell alkalmaznunk. Ez a fogalom a hálózati teljesítmény optimalizálását jelenti az adatforgalom megfelelő szabályzásával. Itt többféle szempontot is figyelembe kell venni. Elsősorban fontos a hálózati leterheltség alapján történő útvonalválasztás, a hálózat kapacitásának figyelembevétele, valamint a megfelelő erőforrások hatékony kihasználása. Az MPLS TE architektúrája a 10. ábrán látható.



10. ábra: MPLS TE architektúra [16]

Az MPLS doménen belüli routerek meghatározott időközönként státuszinformációkat küldenek a saját linkükről (sávszélesség, maximálisan lefoglalható sávszélesség, aktuális sávszélesség, link állapota). Ezek IGP protokollok segítségével kerülnek elárasztásra az MPLS hálózatban, tipikusan IS-IS [17], vagy az OSPF [18] megoldásokat szoktak használni a megfelelő kiegészítő fejléc alkalmazásával. A kapott információk a Link State Database (LSDB) adatbázisba kerülnek bele. Az LSP meghatározása ezekből az adatokból történik a Shortest Path First (SPF) algoritmus egy továbbfejlesztett változatával, a Constrained SPF-vel (CSPF) [19] használatával. Ez annyiban különbözik az elődjétől, hogy itt a kalkuláció függ az LSDB-ben tárolt sávszélesség adatoktól, esetleges hop szám korlátozástól, vagy bármilyen definiált irányelvtől, melyet alkalmazni kell. Utóbbiak egy úgynevezett Traffic Engineering Database (TEDB) adatbázisba vannak eltárolva. Ezen függőségek együttes figyelembevételével kerülnek kiszámításra az LSP-k. Az útvonalak kihúzása RVSP [20] segítségével valósul meg (11. ábra).



11. ábra: LSP-k kihúzása RVSP segítségével [21]

A protokoll használata erőforrásfoglaláson alapul, a küldő elküldi az ehhez való igényét a vevőnek egy PATH üzenetben. Itt történik a címke kérés, és az erőforrás ellenőrzés is egyben. A tulajdonképpeni erőforrásfoglalás és a címke válasz a RESV üzenet keretében fog lezajlani. Ezek végeztével a címkekapcsolt útvonal sikeresen kiépül. Fontos kiemelni, hogy ez úgynevezett soft state alapon működő jelzési protokoll, ami azt takarja, hogy a küldő és a vevő között periodikus PATH/RESV üzeneteket kell egymásnak küldeniük, hogy továbbra is foglalva legyen az adott erőforrás.

Ezzel áttekintettük az MPLS hálózatok felépítését, működését, és a szolgáltatásgarancia biztosítására vonatkozó megfontolásokat. Megfigyeltük, hogy az MPLS TE implementációja elosztott működésű, azaz minden hálózati elem magának számol útvonalat, és az RVSP protokollal pedig kiépítésre kerül az ennek megfelelő útvonal. Ez a fajta megközelítés a hálózat növekedésével több problémát is felvet.

- Egyrészt mivel minden LER/LSR saját magának számol útvonalat egymástól függetlenül, így nem a domén teljes, legoptimálisabb útvonala kerül kiválasztásra.
- Autonóm MPLS hálózatokat átívelő, end-to-end útvonalszámítás is problémás, mivel akármilyen IGP protokollt is használunk, az mindig egy adott doménen belül működik. Bár létezik olyan implementáció (pl. BGP-nek van ilyen), amely AS-ek között képes link információkat terjeszteni, azonban ez több processzor számítást igényel.
- Nem mellesleg a különböző AS régiókon keresztül kialakított end-to-end szolgáltatás komplex MPLS implementációt igényel. Például egy VPN kapcsolat létrehozása bonyolult, hiszen többszörös enkapszuláció, és dekapszuláció történik.
- Az RSVP protokoll soft state működése miatt nem igazán skálázható nagy hálózatokban. Transzport és maghálózatokban sokkal szigorúbb erőforrás igények jelentkeznek mind CPU, sávszélesség, és a memória vonatkozásában.

Ezen problémakörök megoldását hivatott kiküszöbölni az SRv6 technológia. A következő fejezetben az MPLS-hez hasonló áttekintést fogok tenni mind az architektúrájára, működésére, szolgáltatási garancia biztosítására vonatkozóan. Rávilágítok, hogy miért előnyös ez a fajta megközelítés, hogyan nyújt megoldást a fentebb sorolt komplexitási kérdéskörökre.

4. Segment Routing over IPv6 (SRv6)

Az SRv6 egy olyan csomagtovábbítási technológia, amely alkalmazásával az MPLS-el ellentétben sokkal skálázhatóbb Traffic Engineering valósítható meg. Működését tekintve a csomag útvonala előre meg van határozva a köztes routerek megfelelő interfész IPv6-os címeivel, melyek egyben utasításkészleteket is definiálnak a csomagfeldolgozásra vonatkozóan. Ezeket Segment Identifier-nek (SID) nevezzük, és az IP csomagok megfelelő fejlécébe kerülnek elhelyezésre. Kétfajta SID-et különböztünk meg, ugyanis a 8402-es RFC-ben [22] implementálásra került az MPLS hálózatban való használata is. Így a SID az éppen szükséges követelménytől függően lehet egy 32 bites MPLS címke, amely szintén egy adott routerre specifikus, de lehet egy 128 bites IPv6-os cím is, amely a natív SRv6-os transzport és maghálózatban kerül alkalmazásra. Az utóbbiról fogok részletesebben mesélni, a tesztrendszerem összeállítása során is egy ilyen hálózatot vettem alapul. A SID struktúrája az alábbi (12. ábra).

12. ábra: SID formátuma [16]

A Locator mező azonosítja a hálózati csomópontot, a hálózati eszközök ez alapján címzik a csomagot. Ez egy 64 bites prefixet foglal a címből. A Function mező definiálja az utasításokat, amelyeket a csomaggal végezni kell a megfelelő routereken. Ilyen lehet például az SRv6-os csomag dekapszuláció, csomagtovábbítás és annak megfelelő specifikus kiegészítései. Példaként bemutatok közülük párat:

End: Az egyik legalapvetőbb SID, így kerül felszólításra egy adott csomópont, hogy fejezze be a jelenlegi utasítás végrehajtását, és kezdje el a következőt.

X: Továbbítja a csomagot a megfelelő kimenő interfészen/interfészeken.

T: A csomópont egy adott routing táblát használjon a csomagtovábbításra.

D: Dekapszulálja az IPv6 csomag fejlécét és a kiegészítő mezőket

V: MAC cím alapú csomagtovábbítást definiál, erre van egy külön dedikált tábla, amelyen belül Layer 2-es címeket keres.

B6: Az adott hálózati eszköz, ami ezt az utasítást végrehajtja egy SRv6-os Policy-hez lesz hozzákötve. Ez a szolgáltatásgarancia biztosításánál fog fontos szerepet játszani.

Az Argument mező opcionális. Lehetőségünk van paraméterezni egy adott utasítást abból a célból, hogy egy szolgáltatás, esetleg egy adott applikáció meta adatát tároljuk benne.

Maga a SID-ek egy úgynevezett Segment Routing Header-ben [23] (SRH) vannak tárolva (13. ábra).

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Next Header   |  Hdr Ext Len  | Routing Type  | Segments Left |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Last Entry   |     Flags     |              Tag              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|            Segment List[0] (128-bit IPv6 address)             |
|                                                               |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|                                                               |
                              ...
|                                                               |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|            Segment List[n] (128-bit IPv6 address)             |
|                                                               |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                                                             //
//         Optional Type Length Value objects (variable)       //
//                                                             //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

13. ábra: SRH felépítése [23]

A mezők az alábbi funkciókat töltik be:

Next Header: Ebben van megadva, hogy milyen fajta fejléc/mező fog következni a csomag feldolgozása során.

Hdr Ext Length: A SID hossza 8 oktett egységekben, mellőzve az első 8 oktett értéket.

Routing Type: Ebbe van megadva a routing fejléc típusa. Az SRH típusa decimálisan kifejezve 4.

Segments Left: Itt van nyilvántartva az éppen hátramaradt szegmensek száma, azaz a hopok számát lehet itt követni.

Last Entry: A szegmens lista utolsó elemének indexét tartalmazza.

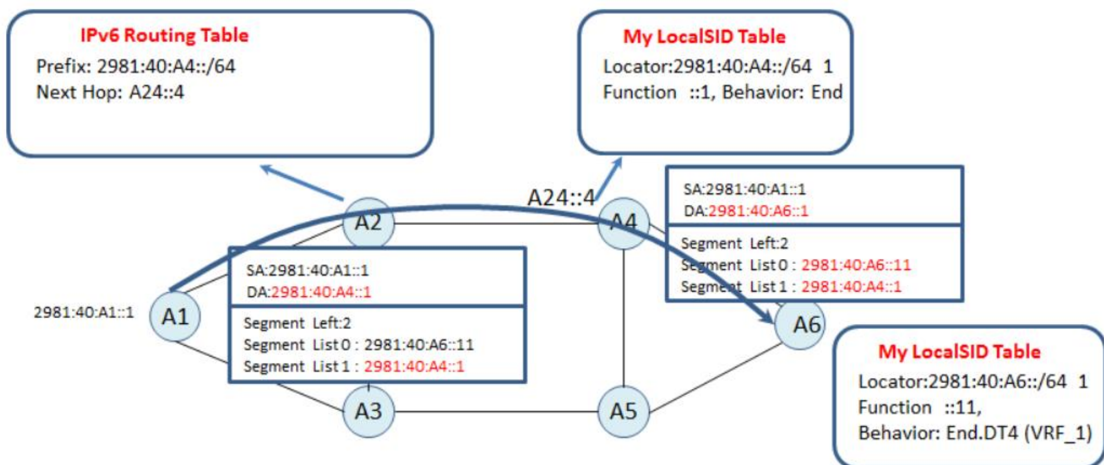
Flags: 8 bites szervezésű flagek (jelzőbitek) halmaza. Az RFC jelenleg ezt a mezőt jövőbeni használatra definiálta. A kommunikáció során az összes flag ilyenkor 0.

Tag: SRv6 csomagok csoportosítására szolgál, amelyek azonos tulajdonságokkal rendelkeznek.

Segment List: Ez a mező gyakorlatilag egy számláló, amely mindig az aktuálisan végrehajtandó SID-re mutat. Ahogyan a fenti ábrán is látható a szegmens lista megfelelő n-dik értéke reprezentálja az n-dik szegmenst. Ahogyan a csomag halad a csomópontokon, úgy csökken a counter értéke.

TLV: Ez egy opcionális mező, meta adatokat tartalmaznak a SID-ek feldolgozására vonatkozóan.

Az SRv6-os csomag útvonala általánosságba véve az alábbi módon néz ki (14. ábra).

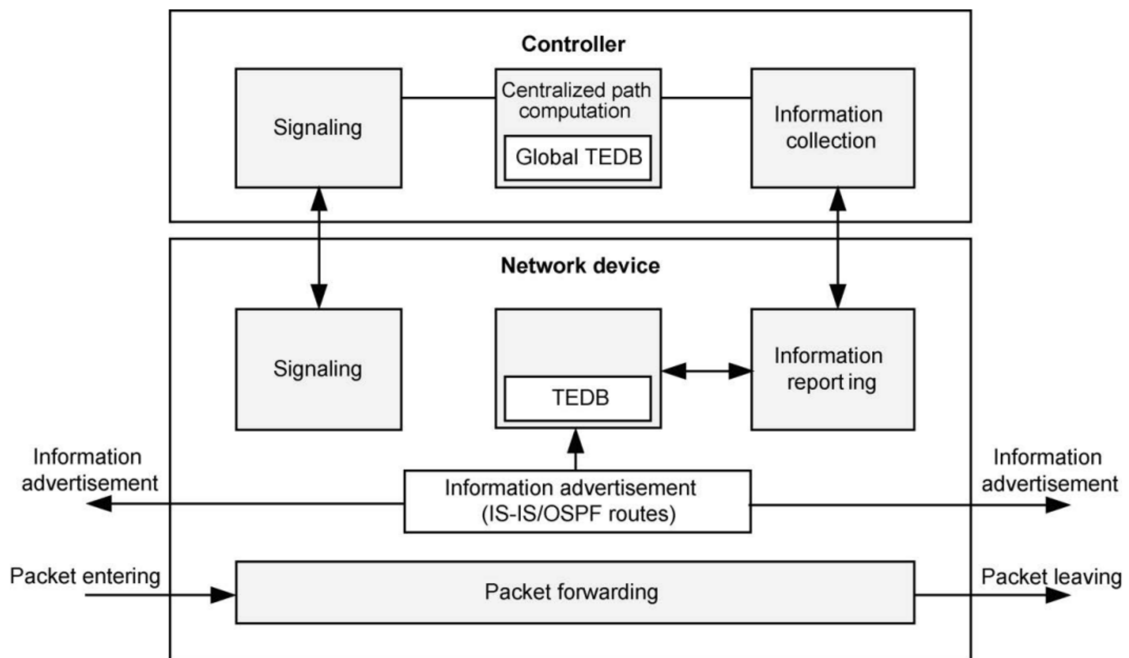


14. ábra: SRv6-os csomag feldolgozása a megfelelő routereken [24]

Az SRv6 hálózati doménbe érkező csomagot a forrás (Source SR Node, A1) becsomagolja és rak rá egy SRH-t, meg egy IPv6 fejlécet. Majd a csomag a cél IP cím alapján a megfelelő routerhez kerül (Endpoint Node, A4 majd A6). Akármilyen csomópontról is beszélünk, az SRv6-os csomagfeldolgozás úgy történik, hogy a router elkülönít egy lokális SID táblát, melyben a szegmensek és a hozzá tartozó utasítások találhatóak. Ha a cél IP cím szegmensként szerepel ebbe a táblában, akkor a csomag feldolgozásra kerül. Amint az End program vagy bármilyen változata végrehajtásra kerül, az SL pointer már a következő SID-re mutat és csökken 1-el a Segment Left mező értéke. Így a router tudja az éppen aktuális szegmens értéket, ami bekerül a külső IPv6-os csomag cél IP cím mezőjébe. Ennek megfelelően mindig a megfelelő útvonal felé lesz irányítva

a forgalom. Előfordul, hogy nem minden hopnál történik SRv6-os csomagfeldolgozás. Ebben az esetben a router normál IPv6-os csomagokként kezeli a forgalmat, azaz a routing tábla bejegyzései alapján kiküldésre kerül a megfelelő interfészen (Transit Node, A2). Ilyenkor nem kerül módosításra a cél IP cím mező a fejlécen belül, nyilvánvalóan a szegmens listában se szerepel ennek a csomópontnak az IP címe.

Belátható, hogy ezzel a működési elvvel a csomag útvonalának meghatározása sokkal egyszerűbb, hiszen az IP csomag alapból tartalmazza az érintett csomópontokat anélkül, hogy bármilyen plusz jelzési protokollt használnánk ezen információk kiosztáshoz. A lokális routerek csupán a szegmens feldolgozására vonatkozóan tárolnak információt a SID táblájában. Mind a Source SR Node, mind az Endpoint Node csomópontok az SRv6 doménben egy központi entitástól kapják meg a csomag útvonalának beprogramozásához szükséges információkat (15. ábra). Ez nagy különbség az MPLS TE architektúrájához képest, hiszen azzal ellentétben itt egyik hálózati csomópont se számol útvonaloptimalizálást, hanem egy központi elem megcsinálja azt, és egy IGP protokoll segítségével ez kiosztásra kerül.



15. ábra: SRV6 TE architektúrája [16]

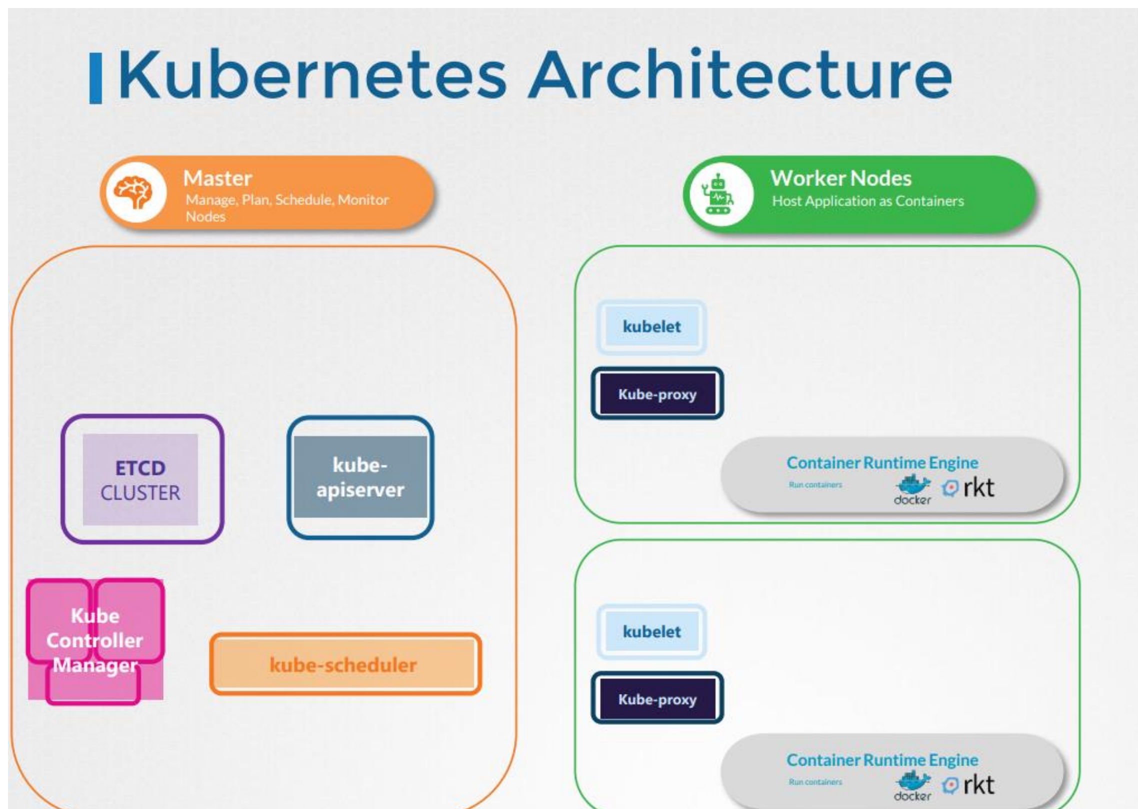
Hasonlóan a link információk hirdetései is így kerülnek a központi kontrollerhez. Ezzel nem csak az RVSP protokoll skálázási problémáját oldjuk meg, hanem az MPLS SDN implementációjának hiányosságai is kiegészülnek. Ebből az következik, hogy lehetőségünk van end-to-end útvonaloptimalizálást végezni, hiszen a kontroller AS-en átívelő útvonalszámítást végez. Egyszerűsödik a protokollhierarchia is, kiküszöböltük az MPLS komplex implementációját, valamint az RVSP protokoll használatát. Mindezen fejlesztéseknek köszönhetően a network slicing is könnyedén megvalósítható.

A következő fejezetekben egy rövid elméleti áttekintést teszek a Kubernetes, mint konténer orkesztrációs szoftverre, felépítésére, működére.

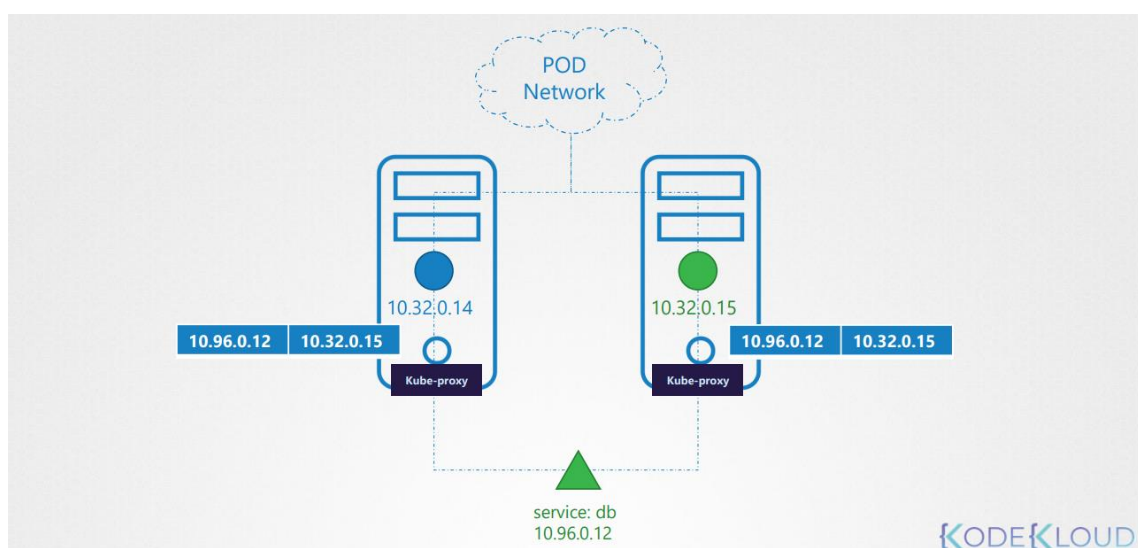
5. Kubernetes

A Kubernetest eredetileg a Google kezdte el fejleszteni, majd az idő haladtával a Google létrehozta a Cloud Native Computing Foundation [25] szervezetet, és azóta ők fejlesztik a mostanra már nyílt forráskódú szoftvert. A konténer-orkesztrációs platformok közül ez emelkedett ki leginkább az idő haladtával, ezt használja a legtöbb távközlési cég is. Fontos megjegyezni, hogy nem a kubernetes végzi a konténerek futtatását, hanem a már említett runtime engine-nek. Többfajta is létezik, a Docker többek között alkalmas erre, és ezt használtam én is. Az architektúra egy klaszterből indul ki, amelyben node-ok kerülnek elhelyezésre, két fajta létezik: master és worker node (16. ábra). A master gyakorlatilag a vezérlősík, ez tartalmazza a kube-apiservert, az ETCD-t, a kube-schedulert és a Kube-Controller-Managert. A kube-apiserver a legfontosabb komponens, minden kommunikáció rajta keresztül megy. Az ETCD egy kulcs-értékpárokat tartalmazó adatbázis, az összes podokkal, node-okkal, service-el, stb. kapcsolatos információk itt találhatók. Podnak nevezzük a legkisebb futtatási egységet. Általában egy konténert tartalmaz, de indokolt esetben előfordulhat, hogy akár 2 is be van ágyazva a podba, ezt nevezzük Multi-Container Pod-nak. Amikor például létrehozunk egyet akkor az ETCD adatbázisba bekerül egy bejegyzés erről a podról: melyik node-ra kerül, mióta fut, státuszinformációkat is tartalmaz, és még sok mást. A kube-scheduler ütemezi, hogy melyik pod melyik node-ra kerüljön, alapértelmezetten törekszik a terheléskiegyenlítésre, magyarán ügyel arra, hogy egyik helyre se legyen allokálva túl sok pod. A Kube-Controller-Manager-nek a feladata a master node komponenseinek folyamatos monitorozása, és szükség esetén beavatkozzon a rendszerbe és minél hamarabb elhárítsa a hibát. A Worker node-okon jönnek létre a konténerek, ezek valósítják meg a felhasználói síkot. Minden egyes ilyen node-ot egy-egy külön kubelet menedzsel, ő regisztrálja fel a node-ot, ő hozza létre a podokat. A podok közti, pod-kliens közötti kommunikáció során egy úgynevezett service-el teremtünk kapcsolatot (16. ábra), ez is egy objektum a kubernetesben, pont mint a podok, node-ok, klaszterek. Itt fontos megjegyezni a kube-proxy-t, szintén egy pod, ami minden egyes node-hoz létrejön. Neki lesz a feladata hogy a kernel netfilter [26] modulját használva elvégezze az útvonalválasztást a megfelelő iptables szabályok, láncok alapján. Gyakorlatilag ezekbe a táblákba találhatóak a pod-service összerendelések. A klaszter hálózatát többféle módon lehet implementálni, ezeket az interfészek specifikációi, valamint a pluginok határozzák

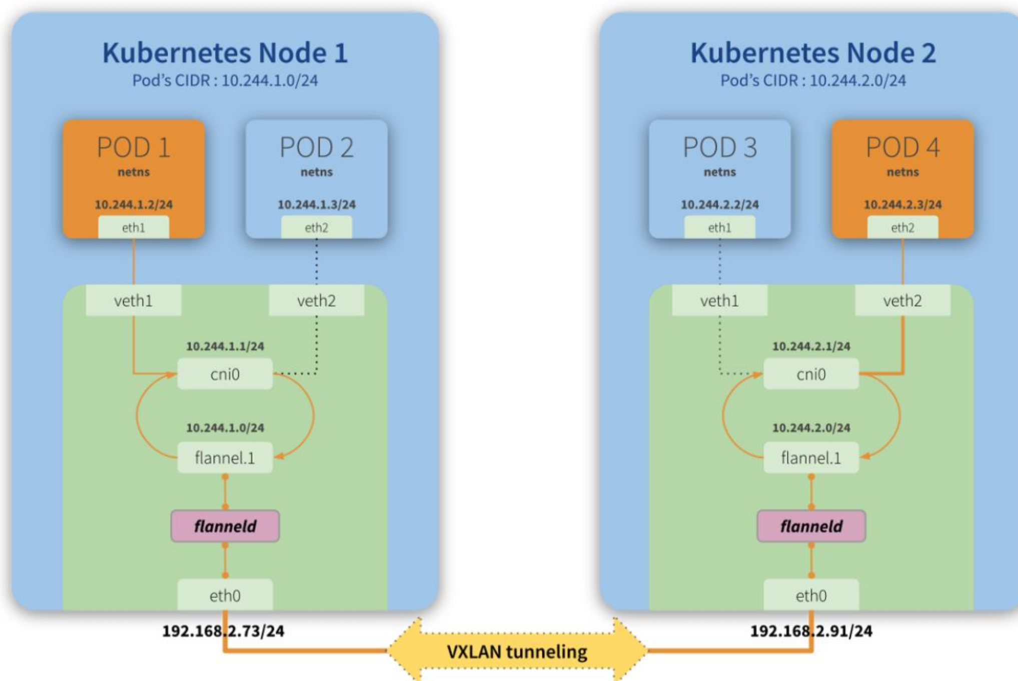
meg. Ezek összefoglalva határoznak meg egy úgynevezett Container Network Interface-t (CNI), amelynek a feladata többek között a hálózati interfészek menedzselése, IP cím allokáció, útvonalválasztás, és a konténerek közötti kapcsolat létrehozása (18. ábra).



16. ábra: A kubernetes architektúrája



17. ábra: Service működése



18. ábra: Példa Kubernetes CNI-ra (Flannel) [27]

Ezzel áttekintettük a kubernetes alapvető felépítését, működését. A következő pontban a Linux tűzfalszoftverét, az iptables-t fogjuk áttekinteni.

6. Iptables

A kernel működését tekintve a gép erőforrásain uralkodik és osztja ki őket a folyamatoknak. Ezen erőforrások közé tartozik a CPU ütemezés, a memória, az állományrendszer, vagy éppen egy adott driver. A Linuxról ismeretes, hogy monolitikus kernel, ami annyit takar, hogy az operációs rendszer egésze a kernel space-ben van implementálva. Ez hordoz magával előnyöket és hátrányokat is. Az, hogy a user space és a kernel space egy közös címtéren helyezkedik el gyorsabb működést tesz lehetővé, a rendszerhívások direkt módon történnek. Ugyebár egy nagy programról beszélünk, ami kizárólagosan a kernel space-ben fut. Egy ilyen operációs rendszer működése elég jól optimalizálható. Azonban pont ebben mutatkozik meg a hátránya is, többek között az, hogy kevésbé flexibilis a fejlesztésre. Ha egy új funkciót szeretnénk hozzáadni a kernel működéséhez az egész rendszer forráskódján módosítani kell. Ennek a kiküszöbölésére nyújt megoldást a kernelbe betölthető modulok, amelyek tulajdonképpen a kernel funkcionalitásainak kiegészítésére szolgálnak. Tárgykódú állományokról beszélünk, amelyek futási időben linkelődnek a kernelhez. Ilyen modulok segítségével van megvalósítva a Linux alapú disztribúciók netfilter tűzfalszoftverének elengedhetetlen komponense, az iptables is. Ez kommunikál a netfilter által definiált helyekkel, amely a kernel protokoll rétegei között találhatóak. Ezekben a pontokban történik a NAT, NAPT, csomagszűrés, csomagmanipulációs eljárások. Öt előre definiált helyen vannak ezek meghatározva a hálózati rétegekben, amelyek az alábbiak:

NF_IP_PRE_ROUTING: Ez a pont egy bejövő forgalom hatására triggerelődik, közvetlenül miután belépett a hálózati verembe. Itt még semmilyen útvonalválasztás nem történik arra vonatkozóan, hogy hova küldjük a csomagot.

NF_IP_LOCAL_IN: Ez a ponton már az útvonalválasztás megtörtént és tudjuk, hogy a hosztnak van címezve a csomag.

NF_IP_FORWARD: Ezen a ponton is megtörtént az útvonalválasztás és tudjuk, hogy továbbítani kell a csomagot.

NF_IP_LOCAL_OUT: Ebben az esetben egy helyileg generált forgalom hatására triggerelődik amint eléri a hálózati vermet.

NF_IP_POST_ROUTING: Itt pedig lokálisan generált, vagy útvonalválasztott kifelé irányított csomagok hatására triggerelődik még mielőtt a forgalom kiküldésre kerülne a kimenő interfészre.

Az iptables segítségével tudunk szabályokat (rule) meghatározni. Ezeket láncokba (chain) rendeződnek, és különböző táblákba (table) kerülnek elhelyezésre. Ezenkívül vannak olyan instrukciók (target) amelyek akkor hajtódnak végre, ha bizonyos szabály érvényre kerül. Láncok tekintetében vannak beépítettek, amely mindenképpen szerepel a táblákban. Ezek gyakorlatilag a fentebb tárgyalt csomagfeldolgozási pontokhoz vannak hozzárendelve (19. ábra):

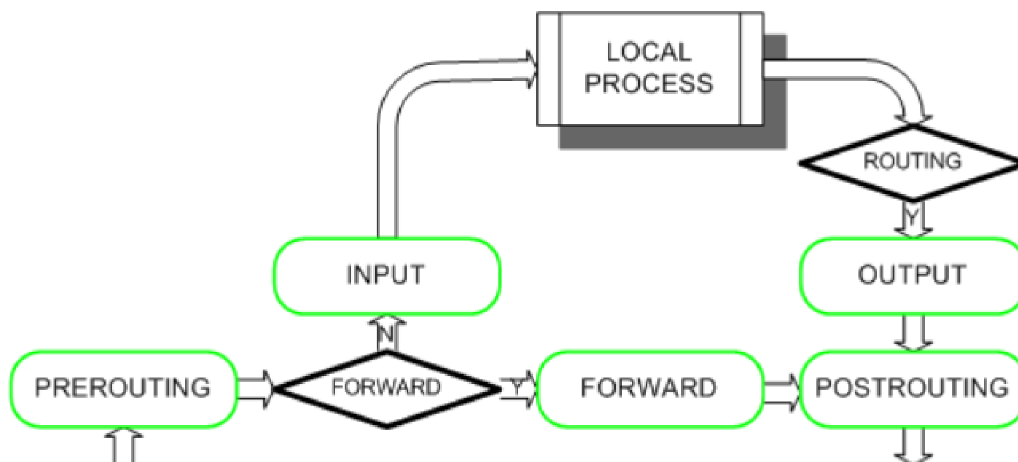
PREROUTING: Az NF_IP_PRE_ROUTING hatására triggerelődik

INPUT: Az NF_IP_LOCAL_IN hatására triggerelődik

FORWARD: Az NF_IP_FORWARD hatására triggerelődik

OUTPUT: Az NF_IP_LOCAL_OUT hatására triggerelődik

POSTROUTING: Az NF_IP_POST_ROUTING hatására triggerelődik



19. ábra: A láncok szerepe a csomagfeldolgozásban [28]

Léteznek felhasználó által definiált láncok is, ilyeneket használ többek között a Kubernetes is, vegyítve persze a fentebb tárgyalt beépített láncokkal. Ezek a láncok kiegészítik az eredeti láncot, tulajdonképpen meghívásra kerülnek általuk. Miután az összes szabály kiértékelődött, a RETURN target után visszatér az eredeti lánchoz. Táblák esetében pedig az alábbiakat különböztetjük meg: **Filter, NAT, Mangle, Raw,** és **Security** tábla.

A target-ek közül kétfajtát különböztetünk meg annak függvényében, hogy a csomagfeldolgozás után mit teszünk vele. Az úgynevezett terminating target-ek esetében, ami egy adott szabályhoz tartozik, nem kerül további szabályok kiértékelésére a csomag. A leggyakoribbak az alábbiak:

ACCEPT: Ez a target beengedi a csomagot az iptables tűzfalán keresztül

DROP: Ilyenkor a csomag eldobásra kerül, mert semmilyen láncon belüli szabályok kritériumainak nem felelt meg

RETURN: Ahogyan már fentebb ezt már megemlítettem, ez a target visszaküldi a csomagot az eredeti lánchoz, hogy további szabályok kiértékelésére kerüljön

REJECT: Az iptables tűzfal elutasítja a csomag beengedését, és a kapcsolódni kívánt eszköznek hibaüzenetet küld. A különbség a DROP mechanizmusához képest az, hogy ott nem fog hibaüzenetet küldeni a kliensnek.

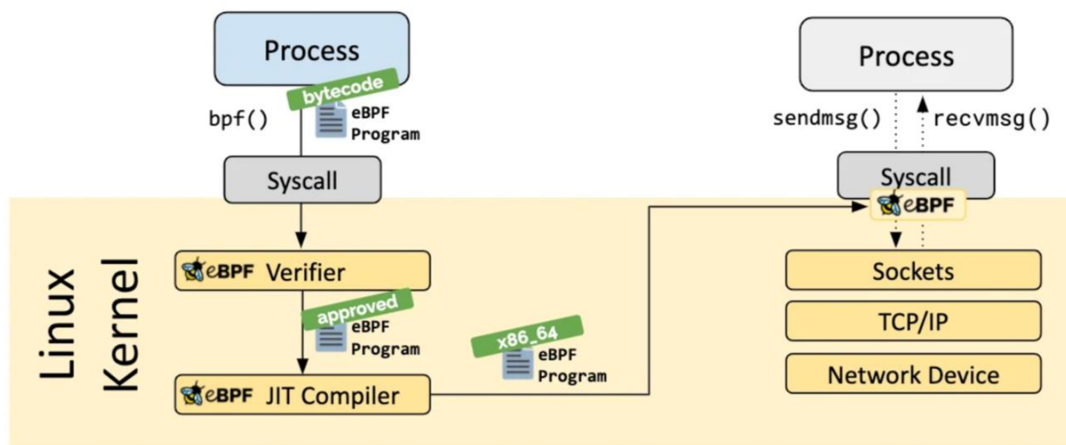
Ezzel áttekintettük az iptables működését. A mai napig ez a legmeghatározóbb tűzfal szoftver Linux környezetben. Azonban idővel több probléma is felmerült a használata során. Kezdjük azzal, hogy nagyon nehezen fejlődik, ami annak tudható be, hogy a kernel modulok fejlesztése nem egyszerű feladat. Ahogyan erről már a bevezetésben is szó esett, a monolitikus kernelek funkcióinak bővítése azért nehéz, mert a különböző komponensei a Linux kernelnek erősen függnek egymástól. A fejlesztés során nagyon kell figyelni arra, hogy adott egy funkció megfelelően működjön, hiszen bármilyen bug vagy probléma előfordulása rendszerhibát eredményez.

Továbbá performancia szempontból se a leghatékonyabb megoldás. Alapvetően az iptables nem olyan alkalmazási területekre lett kitalálva, mint például a hálózatszeletelés, ahol a késleltetés minimalizálása kulcsfontosságú szerepet tölt be. A probléma abból fakad, hogy az IP csomag feldolgozása szekvenciálisan történik, azaz egy táblán belül végig kell keresni, hogy pontosan mely láncokat, és azokon belül is mely szabályokat kell alkalmazni az adott csomagra. Ez összességében nagymértékben növeli a csomagfeldolgozási időt, ami számunkra kedvezőtlen.

Ezekre a problémakörökre nyújtana megoldást az extended Berkeley Packet Filter (eBPF) technológia, amit a következő fejezetben fogok részletesen kifejteni.

7. eBPF

Mint ahogyan ezt már a bevezetőben is felvezettem az extended Berkeley Packet Filter megoldást nyújthat a Linux kernel protokoll rétegeiből felhalmozódó számítási költségek, és az ebből fakadó lassú csomag feldolgozás problémájára, az operációs rendszerrel szemben támasztott és gyorsan változó követelményekre. Működését tekintve egy RISC alapú virtuális gép van beágyazva az operációs rendszerbe, mellyel a kernel programozhatóvá válik anélkül, hogy a kernel forráskódját módosítanánk vagy írnánk egy új modult. A rendszer architektúrája, működése az alábbi (20. ábra).



20. ábra: Az eBPF működése [29]

A legfontosabb elemek közé tartoznak az eBPF programok, melyet a userspace-ből fogunk betölteni a kernelbe. Ezek C nyelven vannak megírva, amelyek előre meghatározott helyekhez vannak hozzácsatolva a hálózati veremben. Ezek lehetnek kernel függvények (kprobes), felhasználó által írt függvények (uprobes) meghívása, rendszerhívások, vagy nyomkövetési pontok (tracepoints) a forráskódban/kernel modulban. Lehet ezenkívül hálózati interfészeknél, de akár socketekhez is lehet rakni. A programok megírásához, és a kernelbe való betöltéséhez több fejlesztői könyvtár is létezik. Ezek közé tartozik többek között a *bcc, bpftrace,* az *eBPF Go,* és a *libbpf C/C++* könyvtára [29]. Ezek típus függően *clang/LLVM* compilert [30] használnak, melynek segítségével a program egy úgynevezett bájtkód sorozattá lesz lefordítva. Tulajdonképpen ezek CPU független Instruction Set-ek, amit majd egy a Just-In-Time (JIT) fordító fog a hardverben használt processzor megfelelő architektúrájú gépi kódjára lefordítani. Előtte azonban ellenőrzésre kerül, hogy a futtatni kívánt eBPF programunk

biztonságos -e. A szűrés során a Verification Engine olyan vizsgálatokat tesz, mint például a folyamat jogosultsága, hogy betöltheti -e a programot a kernelbe. Ellenőrzi azt is, hogy lefut -e a program, nem kerül be egy esetleges végtelen hurokba, feltartva így bizonyos folyamatokat. Nem kerül használatra inicializálatlan változók, illegális memória területhez való hozzáférés. Kivizsgálásra kerül továbbá az is, hogy megfelelő méretű a program, benne van a korláton belül, amit még lehet használni. Ezen felül a processz jogosultság függvényében további lépéseket tesz a rosszindulatú támadások ellen, mint például konstansokba töltött rosszindulatú kódok elleni védelem, az eBPF program memóriaterületének csak olvasható állománnyá tétele, stb. Vannak még további feltételek is, én csupán párat soroltam fel belőlük szemléltetés céljából. Ha mindezen követelményeket sikeresen teljesítette, akkor fogja a JIT compiler a program bájt sorozatát közvetlenül lefordítani gépi kóddá futási időben [31]. Itt meg is állnék egy pillanatra, mert fontosnak tartom kihangsúlyozni miért is jó ez a megoldás.

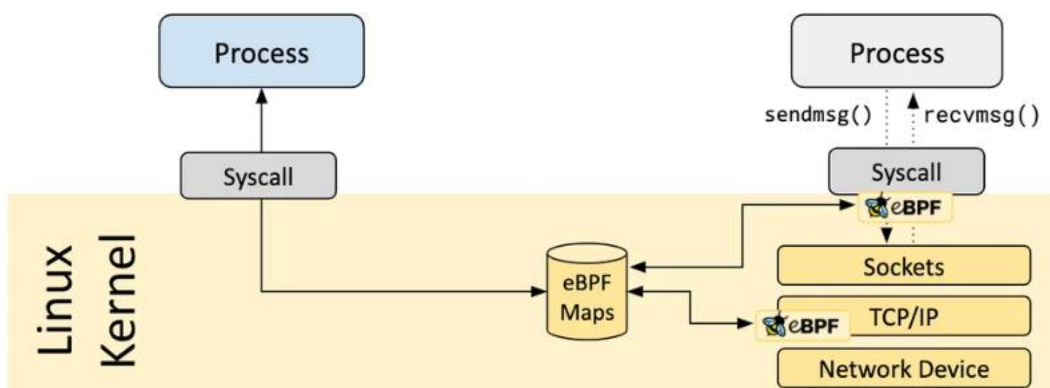
Ehhez érdemes megemlíteni hogyan működik a statikus fordító, és az interpreter. Kezdeném az előbbi működésével, ami először csinál a forráskódból egy olyan adatstruktúrát, ami igazából a kódban lévő összefüggések reprezentálására szolgál. Ezek közé tartoznak a különböző objektumok, osztályok, függvényhívások, különböző műveletek stb. Ez tipikusan egy fa struktúrát szokott jelenteni, amelyet absztrakciós fának nevezünk. Ebből így könnyedén megcsinálja a gépi kód generálást és az optimalizálást. A létrejött objektumfájlokat összelinkeli a forráskóddal, végül készít egy futtatható fájlt. A statikus fordítónak meg van az az előnye, hogy maga a futtatható fájl végrehajtási ideje kisebb lesz, mint egy interpreter által futtatott kód. Azonban főleg nagyméretű forráskódok esetén a fordítás idő jelentős mértékben megnövekszik.

Ezzel szemben az interpreter működését tekintve, nem fordítja le előre az egész forráskódot, nem optimalizál, és nem is csinál linkelést, mivel alapból nem készít futtatható fájlt. A megírt program végrehajtása futási időben történik, az absztrakciós fa elemeinek rekurzív kiértékelésével, azaz itt történik a gépi kód generálása, és végrehajtása egyben. Gyakran nem direktben történik a gépi kód előállítás, hanem egy úgynevezett köztes nyelvre, tipikusan bájtkód előállítás történik először. A bájtkód nem más, mint utasítások (Instruction Set), amelyeket egy virtuális gép fog végrehajtani. Ez a gép pedig a hardveren fut, azaz a processzor onnan már tudja futtatni a VM kódját, persze miután le lett belőle generálva a gépi kód. Ez azért jobb megoldás, mint amikor direkt gépi kóddá fordítottunk mindent, mert mivel egyesével értékelődik ki az absztrakciós fa

elemei, így lényegesen lassabb lesz a futási idő. Ezzel szemben, ha beiktatunk egy virtuális gépet, amelyet a saját utasításkészletével táplálunk, akkor lényegesen gyorsabb lesz a végrehajtás, mivel nem lesz többlet utasítás a rekurzív hívásokból.

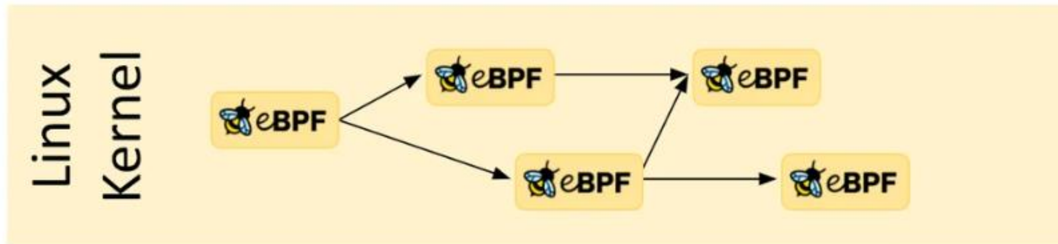
Az alapvető cél az lenne, hogy a compiler és az interpreter bájtkódos implementációjának előnyeit együttesen tudnánk kihasználni. Itt jön képbe a JIT, amely annyiban módosítja az interpreter bájtkódos implementációjának a működését, hogy itt nem kerül végrehajtásra a program, hanem az egész bájtkód sorozat lefordításra kerül. Ezt követően pedig az eBPF program betöltődik a megfelelő pontba a kernelben. Tulajdonképpen ezzel olyan hatékony futási időt lehet elérni, mint egy lefordított kernel forráskóddal, vagy egy kernel modullal. Az eBPF-ben megvalósított JIT fordító támogatott processzor architektúrái [32] jelenleg az *x86_64, arm64, s390, ppc64, sparc64, mips64, arm32,* és az *x86_32.*

Az eBPF programok működésében jelentős szerepet játszik a különféle adatok, státuszinformációk eltárolása egy közös adatbázisba, amelyet eBPF Map-nek hívunk (21. ábra). Ennek az adatstruktúrája különböző lehet, nyilvánvalóan attól függ, milyen adatot szeretnénk benne tárolni. Általában hash tábla, Ring Buffer, Stack Trace, Longest Prefix Match (LPM) adatszervezésű, vagy Least Recently Used (LRU) algoritmust használó adatbázisok szoktak a Map-be kerülni. Ezekhez mind userspace-ből, mind az eBPF programok hozzá tudnak férni. Az előbbinél a *bpf()* rendszerhíváson [33] keresztül, melynek segítségével adatokat kérhetünk le, elemeket módosítani, törölni lehet. Míg az utóbbinál *bpf-helper* függvények [34] segítségével, melynek köszönhetően olyan információkat tudunk lekérdezni, mint például debug üzenetek, mikor bootolt be a rendszer, de használható akár csomagok manipulálására is. Ezen funkciók mind a Linux kernelben vannak implementálva.



21. ábra: eBPF Map [29]

Az eBPF programokat képesek vagyunk eBPF programokkal is triggerelni, köszönhetően annak, hogy ennek megfelelő szintén *bpf-helper* függvények (22. ábra) vannak erre implementálva.



22. ábra: eBPF programok összekapcsolása [29]

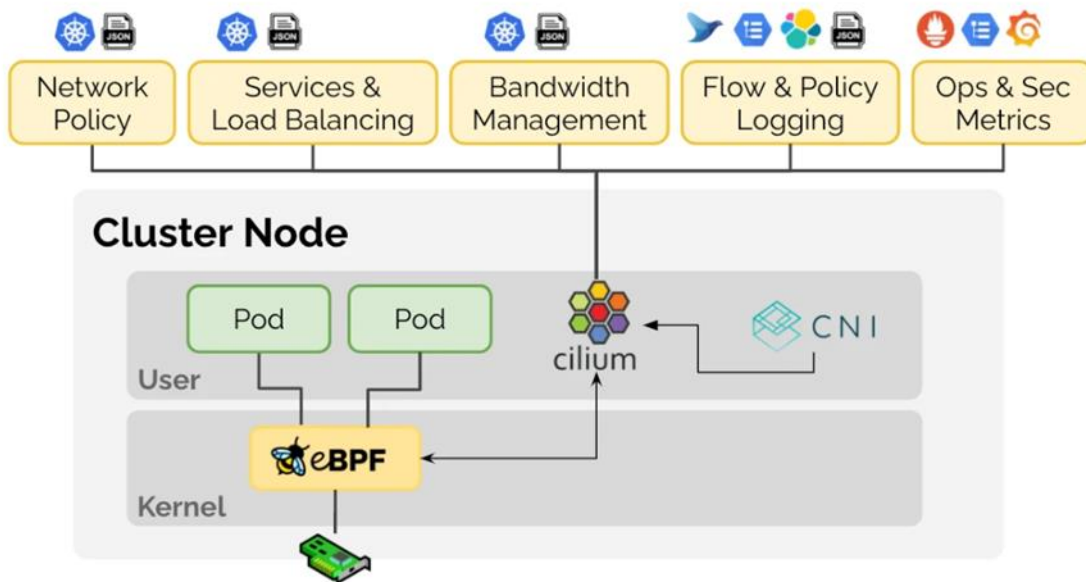
Belátható, hogy többféle szempontból is igen ígéretes irányt képvisel ez a technológia, nem véletlen hogy napjainkban is vannak már aktív ipari fejlesztések, amelyek erre épülnek (23. ábra).

23. ábra: eBPF ipari fejlesztéseket képviselő cégek [35]

Több alkalmazási területe van. Mivel az eBPF programoknak több kapcsolódási pontja is lehet a kernelben, így elég gyakran használják különböző folyamatok monitorozására, hibakeresésre. A Verification Engine használatán felül mi is képesek vagyunk olyan eBPF programokat írni, amivel további biztonsági irányelveket tudunk implementálni, amire éppen számunkra szükség van. Használható továbbá különböző kernel funkciók kiegészítésére/megvalósítására, nem véletlen, hogy egyes Kubernetes CNI-ban így implementálják például az SRv6 enkapszulációt/dekapszulációt, amire később ki is fogok térni a dolgozatom során. Nem utolsó sorban, csomagfeldolgozás optimalizálására is használják, ez az az alkalmazási terület, aminek a segítségével kikerülöm az iptables okozta szekvenciális csomagfeldolgozást. A következő fejezetben a Ciliumot fogom bemutatni, mely egy olyan CNI, amelyben az eBPF implementálva van.

8. Cilium

A Cilium egy nyílt forráskódú CNI, amely mint a többi konténer hálózati interfészhez hasonlóan, hálózati interfészeket menedzsel, IP cím allokációt végez, útvonalválasztást kezel, és a konténerek közötti kapcsolat létrehozását biztosítja. Ami miatt ez különleges, hogy eBPF technológiát használ, amely segítségével több újszerű alkalmazási terület nyílik meg. Az előző pontban tárgyalt eBPF felhasználási esetek gyakorlatilag a Ciliumban is alkalmazhatók (24. ábra).



24. ábra: Cilium architektúrája [36]

Az eBPF programok a kernel megfelelő pontjaihoz való hozzácsatolásával képesek vagyunk egy Kubernetes klasztert sokkal hatékonyabb monitorozás, megfigyelés alatt tartani, ezáltal könnyebben detektálni az esetleges felmerülő hibákat. Sokkal specifikusabb biztonsági irányelvek alkalmazására is lehetőségünk nyílik. Míg maga a Kubernetes L3/L4 szinten implementálja ezt, a Cilium kiegészíti ezt L7 specifikus megszorításokkal. A Loadbalancer implementálására is egy robusztusabb és biztonságos megoldás nyílik, köszönhetően az eBPF-nek, és az eXpress Data Path-nek (XDP) [37], amire majd hamarosan kitérek. A Border Gateway Protocol-t (BGP) [38] is meg lehet valósítani, ennek a segítségével lehetőség nyílik útvonalválasztási, és más fontos információk átvitelére különböző autonóm hálózatok között. Ez majd egy lehetséges SRv6 alapú hálózat link információk, és SID-ek kiosztásánál játszik szerepet, mint IGP protokoll. Ugyanúgy SRv6 alapú enkapszuláció/dekapszuláció, valamint hálózat is

implementálva van a Cilium CNI-on belül, azonban az ezzel kapcsolatos fejlesztések még tesztfázisban járnak. Épp ezért ennek megvalósítása jelenleg a Cilium-on belül még nem igazán kivitelezhető, de erre kitérek majd bővebben a tesztkörnyezetek összeállításánál. Amit még fontosnak tartok megemlíteni, az a kube-proxy helyettesítése, ugyanis ő végzi az útvonalválasztást a Kubernetes hálózaton belüli kommunikációja során. Ugyebár Linuxos operációs rendszer alatt futó szoftverről beszélünk, amely a már korábban említett iptables tűzfal segítségével végez csomagfeldolgozást, azaz a megfelelő táblákba lévő Kubernetes által definiált láncokat, és az ahhoz tartozó szabályokat a kube-proxy kezeli. Ezzel szemben a Ciliumban lehetőség nyílik eBPF programokkal helyettesíteni a kube-proxy-t, és implementálva van egy olyan logika, amely nem megy végig a kernel rétegein keresztül, hanem kikerüli azt, és egyből mehet a CNI interfészén keresztül a pod interfészéhez a forgalom. A Cilium több forgalmi útvonalat is definiál a kernelen belül, melyek az alábbiak:

XDP: Amint a csomag megérkezik a network driverhez, az oda csatolt eBPF program lefut, és feldolgozza a csomagot. Ezzel lehet a legoptimálisabb csomagfeldolgozást elérni, hiszen itt még semmilyen iptables általi előfeldolgozás nem történt meg. Ez tipikusan rosszindulatú csomagok szűrésére használható.

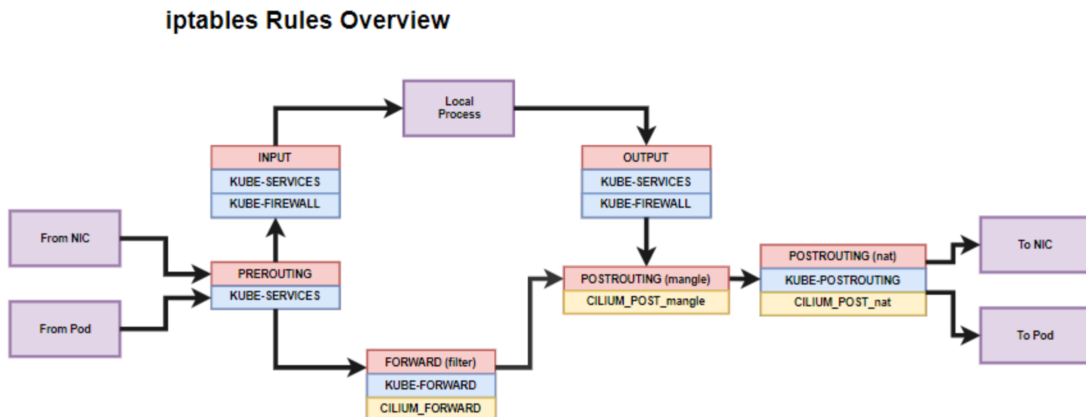
Traffic Control Ingress/Egress: Az eBPF programok ebben az esetben a hálózati interfész hálózati/szállítási rétegéhez lesz a program becsatolva, azaz itt már a TC ingress (bemenet) esetén történik előfeldolgozás a csomaggal. Gyakorlatilag a node forgalmának az átirányítására alkalmas egy helyi végpontra úgy, hogy kikerüli a kernel iptables által felállított útvonalat. Ezzel fogjuk megvalósítani a kube-proxy helyettesítését is.

Socket Operations: A socket operations, az egy olyan pont a kernelben, ahol socket műveletek történnek meg. Ez egy specifikus cgroup-hoz van kötve, ami ugyebár TCP események esetén fut le. Az eBPF program is az ehhez a cgroup-hoz van kötve, és ezzel lehet monitorozni a TCP állapotváltozásokat, például hogy mikor került bontódott/épült fel egy kapcsolat.

Socket send/recv: Ebbe a pontba elhelyezett eBPF program abból a célból lett implementálva, hogy a különböző socket műveleteket figyelje, és annak alapján eldobja a kontroll üzeneteket, vagy tovább engedje, de akár másik socket felé is átirányíthatja.

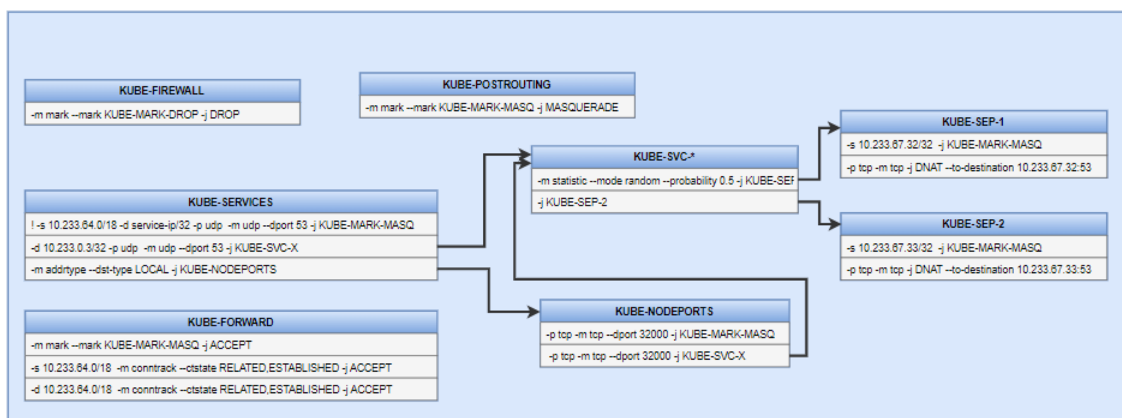
9. Adatforgalom útvonala a kernelben kube-proxy vs. eBPF esetén

Cilium CNI használata esetén az alábbi alapértelmezett, és a Kubernetes, Cilium által definiált iptables láncok, szabályok vannak jelen (25. ábra).



25. ábra: Alapértelmezett és kube-proxy által létrehozott iptables láncok [39]

Számunkra a CILIUM prefix-szel ellátott láncok irrelevánsak, mert mi nem fogunk vele találkozni a mérés során. Azonban a Kubernetes láncok fontos szerepet töltenek be, melyekre az alábbi példát hoztam a dokumentációból (26. ábra).

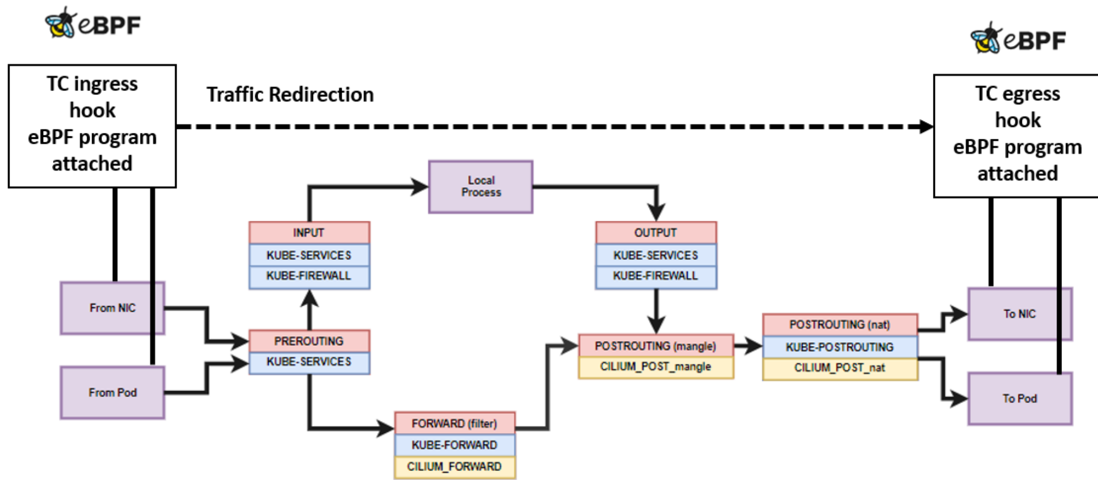


26. ábra: A Kubernetes láncokhoz tartozó szabályok [39]

A rendszer PREROUTING láncát használva eldöntésre kerül, hogy helyi forgalomról beszélünk, vagy továbbítani kell -e a csomagot. Helyi forgalom esetén amikor a KUBE-FIREWALL láncban lévő csomag eldobás érvényre kerülhet, ha érvényre kerül a benne lévő szabály. Két esetben kerülhet megjelölésre (KUBE-MARK-DROP) a csomag. Az

egyik, hogy LoadBalancer service használata esetén, ha a csomag forrás IP címe nincs a megengedett hálózatok között. Másik eset pedig az, ha egy adott szolgáltatásnál használjuk az *ExternalTrafficPolicy=Local* [40] annotációt a service leírójában. Ezt fogjuk is használni a mérések során, szóval erre később ki is térek. A KUBE-SERVICES láncban olyan szabályok vannak, mint például SNAT végrehajtása olyan egy adott forrás IP cím tartomány, adott cél port szám, adott szállítási rétegbeli protokoll használata (TCP/UDP) esetén. De hasonló kritériumok alapján össze lehet kapcsolni egy service lánccal, ami majd a későbbiekben meghívódik. Ha meghívásra kerül egy ilyen lánc (KUBE-SVC-*, KUBE-NODEPORTS), akkor service típustól függően változhatnak a szabályok. Például LoadBalancer esetén definiálva vannak a végpontok, és annak előfordulási valószínűségük. NodePort esetén is mondjuk egy végpont, meg a nyitott port a node IP címéhez. A végpontoknak is van saját láncuk (KUBE-SEP-*), amiben a DNAT, SNAT/MASQUERADE szabályok megfelelő paraméterezéssel jelen vannak. Ezután helyi forgalom esetén még végigmegy a rendszer INPUT láncán is. Helyi forgalomgenerálás esetén ugyanez lejátszódik visszafelé is, csak most a beépített OUTPUT lánc hívódik meg a kimenő forgalom kezelésére. Forgalomtovábbítás esetén pedig a szokásos FORWARD után meghívódik a KUBE-FORWARD lánc is, ahol hasonlóan útvonválasztással kapcsolatos szabályok, és különböző módon megjelölt csomagok kezelésére vannak bejegyzések. A POSTROUTING láncok végrehajtása után, meghívódik a KUBE-POSTROUTING is, ami megintcsak hasonló bejegyzéseket tartalmaz az ezzel analógiába hozható POSTROUTING szabályokkal.

Ezzel szemben a TC ingress/egress pontokba elhelyezett eBPF programok használata kiküszöbölné az iptables láncok okozta lassú csomagfeldolgozást, hiszen egy az egyben a forgalom átirányításra kerül a CNI kimenő interfészhez (ez itt az ábrán nem került feltüntetésre), majd onnan a cél interfészhez (27. ábra).

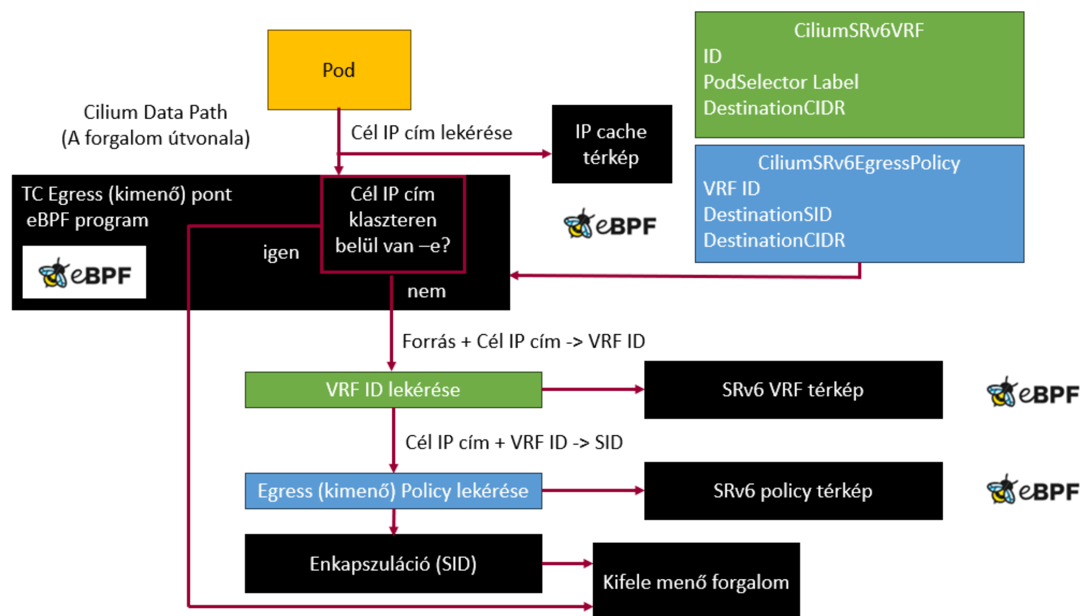


27. ábra: Csomagfeldolgozás eBPF-el [39]

10. SRv6 csomagfeldolgozás implementálása Ciliumban

Ahogyan ezt már a Cilium alkalmazási területeinél is már megemlítettem, az SRv6 csomagfeldolgozás implementálása eBPF programokkal történik. Ráadásul a kube-proxy helyettesítéséhez hasonlóan, itt is a TC ingress/egress pontokhoz van becsatolva a program, ami az enkapszulációt, és a dekapszulációt végzi. Az SRv6-os funkció engedélyezését a Cilium konfigurációs yaml fájljának (cilium-configmap) megfelelő paraméterezésével érhető el.

A Linux kernelben lehetőségünk van egy úgynevezett Virtual Routing and Forwarding (VRF) doméneket [41] létrehozni. Ennek segítségével hálózati rétegbeli (L3) szeparációt tudunk megvalósítani az eszközök interfészei között. Ez ad lehetőséget közös útvonalválasztási táblák, és alapértelmezett átjáró nyilvántartásához egy adott névteren belül. Így valósítjuk meg, hogy több SID-et egy útvonalhoz tudjunk társítani (28. ábra). Azaz az SRv6-os csomagfeldolgozása során a VRF-ek fontos szerepet fognak játszani.

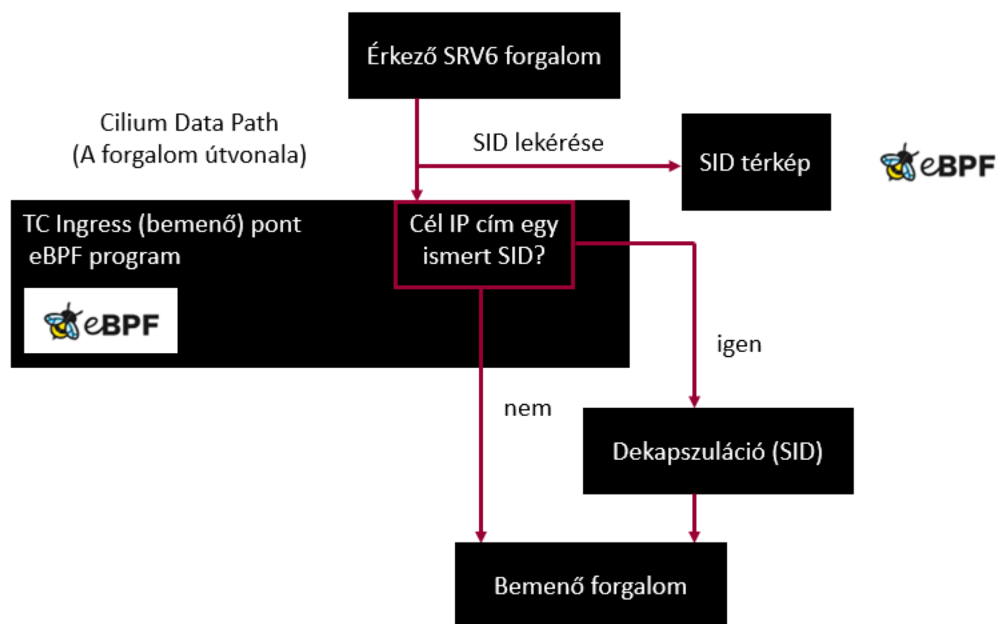


28. ábra: Az enkapszuláció folyamata [42]

Maga az enkapszuláció első lépése, hogy egy megfelelő eBPF program lekérdezi az IP cache térképről a cél IP címet. Ennek a segítségével a program megállapítja, hogy a klaszteren belüli, vagy kívüli forgalomról beszélünk. Utóbbi esetén elkezdődik a becsomagolás folyamata. Elsőként a forrás, és a cél IP cím alapján lekéri az ehhez tartozó VRF ID-T a megfelelő térképről. Majd ezt felhasználva, plusz a cél IP cím segítségével

meghatározza a SID-ket az SRv6 Policy térkép használatával. Majd a megfelelő SRH, és IPv6 fejléc enkapszulációja történik, és onnantól kezdve a csomag kiküldésre kerül (28. ábra).

A kicsomagolás pedig úgy történik, hogy a TC Ingress ponthoz csatolt eBPF program a cél IP címet lekéri a SID térképről, hogy van -e ilyen bejegyzés. Találat esetén a dekapszuláció megtörténik, ellenkező esetben csak a megfelelő útvonalon továbbküldésre kerül a forgalom a klaszter hálózatába (29. ábra).



29. ábra: A dekapszuláció folyamata [42]

Fontos azonban kiemelni, hogy ennek az implementálása elég nehézkes, mivel az SRv6-os funkció megvalósítása a Ciliumban még gyerekcipőben jár, ebből adódóan az ezzel kapcsolatos dokumentáció eléggé hiányos még. A fő problémát az okozza, hogy nincs leírás arra vonatkozóan hogyan lehetne szerkeszteni az eBPF térképeket amelyeket a csomagfeldolgozás során használatra kerülnek. Így például olyan alapvető konfigurációt nem lehet elvégezni, hogy a SID térképbe bekerüljenek a megfelelő szegmens bejegyzések. Remélhetőleg ennek a fejlesztése minél hamarabb lezajlik, viszont ebből adódóan az SRv6-os csomagfeldolgozásnál a linux kernel implementációjára kell hagyatkoznunk.

11. Performanciaelemzések áttekintése

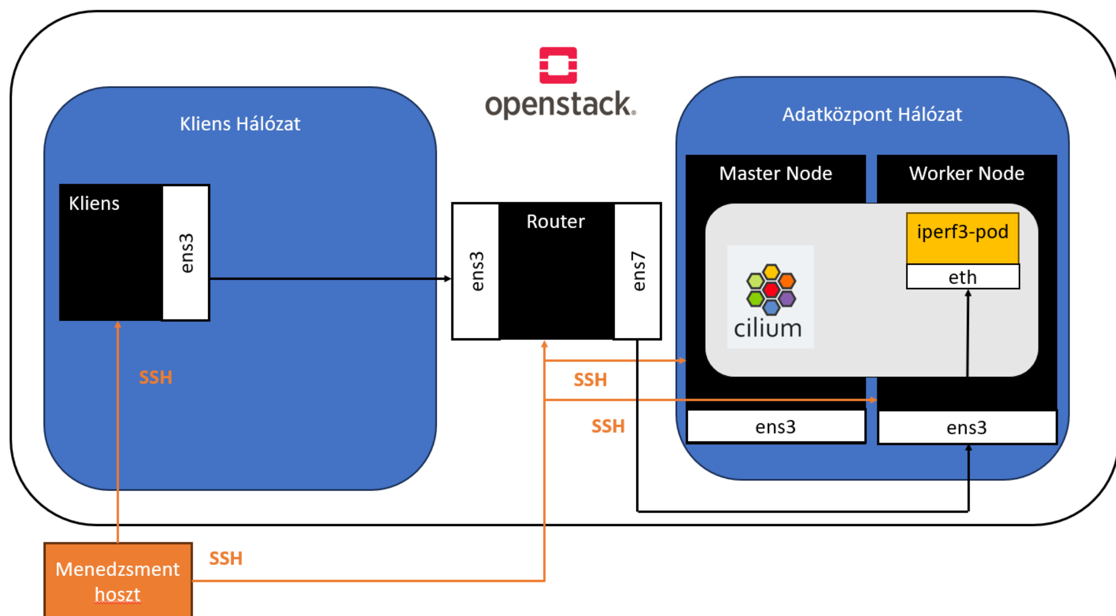
Mérésem célja, hogy bemutassam, miként helyettesíthető a Kubernetes klaszterben a csomagtovábbítás céljára használatos kube-proxy pod a worker node interfészéhez csatolt eBPF programmal. Mint ahogyan ezt a korábbi fejezetekben már megemlítettem, azt várjuk, hogy az eBPF használatával kisebb késleltetést, vagy másképp fogalmazva nagyobb átviteli sebességet fogunk kapni eredményül a kube-proxy-s megoldáshoz képest. Ezt követően tervben volt egy SRv6 alapú tesztrendszer kialakítása is, azonban ennek megvalósítása során több problémába is ütköztem. Egyrészről már említettem, hogy jelenleg a Cilium CNI keretein belül béta fázisban van az SRv6 alapú csomagfeldolgozás implementálása, azaz nincs egy összefoglaló, részletes dokumentáció a megfelelő konfigurációhoz. Ebből adódóan más megoldást kellett keresni ennek megvalósításához. Itt jött képbe a Linux kernel implementációja [43], melynek segítségével sikeresen beállítottam az SRv6 képes hálózati eszközöket a tesztrendszeremben. Gyakorlatilag egy L3 VPN csatornát integráltam a hálózatba. A két végpont között így kialakult az SRv6-os csatorna, azonban a performanciamérés elvégzése során egy újabb problémába ütköztem. Sajnálatos módon a szerver pod elérése SRv6-os csatornán keresztül Nodeport használatával nem érhető el. Felmerült, hogy akkor esetlegesen hozzunk létre egy Load Balancert, amely kívülről elérhető IP címet biztosítana a kliensek számára. Ez egy olyan megoldás lenne, amely forgalmi típusokat tekintve lényegesen szélesebb skálát enged meg, és alkalmas lehetne a forgalom mérésére. Azonban ennek az összes aspektusát körül kell majd járni, mennyire jó, hogy ezt a megoldást választjuk, ugyanis itt felmerül a többleterőforrás használata. Azonban, ha direktbe a pod IP címét adtuk meg célnak, akkor a kliens sikeresen el tudta érni a szervert. Az SRv6 domén összeállításáról a következő alfejezetekben szó esik, a létrehozásához szükséges konfigurációk a függelékben találhatóak.

A natív IPv6-os kommunikáció méréséhez a kliens gépre és a szerver podra telepített *iperf3* [44] Linux-os csomagot használtam. Ez egy nagyon hasznos eszköz átviteli sebesség méréshez, sok kiegészítő paraméterezéssel használható. Be lehet állítani, hogy UDP/TCP forgalmat generáljon, továbbá, hogy melyik porton érje el az iperf3 szervert, be lehet állítani fix értékre a sávszélességet is, és akkor késleltetést lehet ebből mérni. Paraméterererezhető továbbá a csomagméret is, csomagmennyiség, milyen időközönként küldje ki a csomagokat, stb. Én a méréshez TCP forgalmat generáltam, 100

IP csomagot küldtem ki másodpercenként és a mérési eredményeket Excel táblázatban kiértékeltem.

11.1. Performanciaméréshez használt környezet összeállítása

A tesztrendszer kialakításához az Openstack [45] által rendelkezésre bocsájtott erőforrásokat használtam, köszönhetően a Nokia Bell Labs-nak. A méréshez 4 virtuális gépet használtam: egy klienst, egy routert, egyet a master node-nak, egyet a worker node-nak (30. ábra). A klienst és a routert egy hálózatba tettem, a kubernetes klaszter node-okat pedig egy másikba. A klaszter telepítéséhez szükséges yaml fájljai, és a Cilium CNI bekonfigurálása a függelékben találhatóak.



30. ábra: A tesztrendszer sematikus ábrája

Maga a forgalom IPv6 alapú kommunikáción megy, ehhez a virtuális gépek IP címeit manuálisan be kellett állítanom. Továbbá statikus útvonalakat vettem fel a kliens, valamint a worker node számára. A routernek azért nem kellett külön útvonalat adni, mivel már alapból rendelkezik mindkét hálózat felé egy-egy interfésszel.

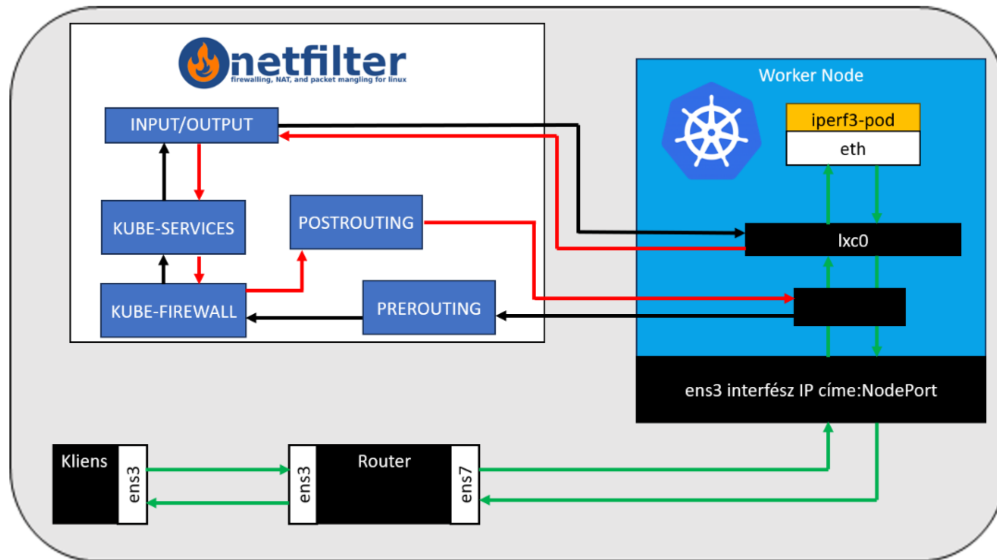
Azt sem szabad elfelejteni, hogy az IPv6 forgalom továbbítást engedélyezni kell egy kernel beállítással (és ezt persze a csomagtovábbításban részt vevő összes gépen engedélyezni kell!).

```
echo "1" >  /proc/sys/net/ipv6/conf/all/forwarding
```

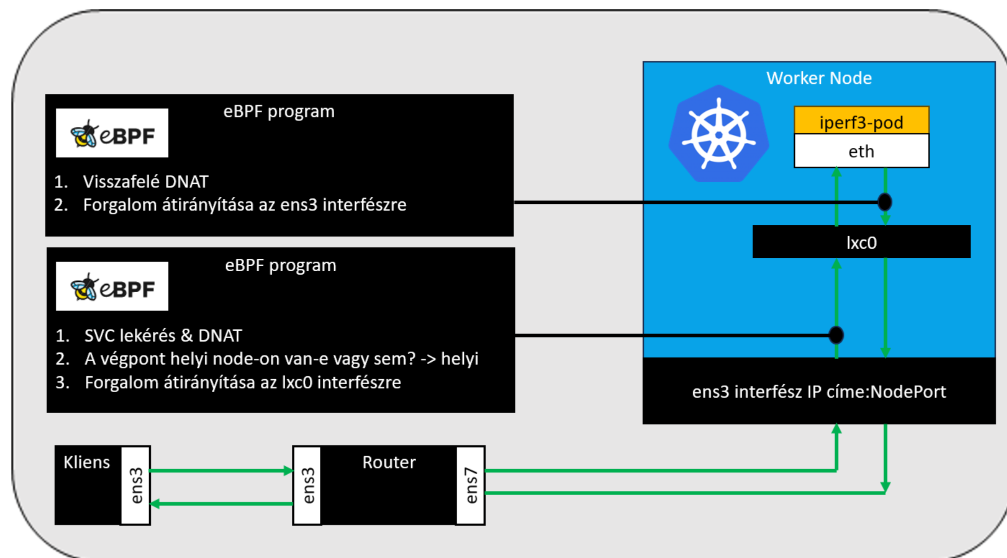
A megfelelő netplan és kernel konfigurációk szintén a függelékben találhatóak.

11.2. Átviteli sebesség mérés IPv6-os csomagok esetén

Az alábbi ábrákon látható a két tszteset összeállítása, a forgalom útja, valamint a csomagfeldolgozás módja (31-32. ábra).



31. ábra: Csomagtovábbítás kube-proxy használatával



32. ábra: Csomagtovábbítás eBPF programok használatával

A szerver podot Nodeport service [46] segítségével tudtam elérhetővé tenni a kliens számára, amely azt csinálja, hogy nyit a worker node ens3 interfészén egy portot 30000-32767 között. Ez megfelelő annotációkkal ellátva a pod és a service yaml fájljában a pod által nyújtott szolgáltatás kívülről elérhetővé válik. Gyakorlatilag az összes olyan

csomagfeldolgozási lépést, amelyek eddig a kube-proxy által definiált iptables láncokban voltak leírva, eBPF esetén pedig már egy annak megfelelő programban van implementálva. A csomagfeldolgozás eBPF használatával a következő ebben a mérési szcenárióban. A TC ingress ponthoz csatolt eBPF program lekéri a Nodeport service-hez tartozó IP címet egy hash-ben tárolt service térképről. Ha talál bejegyzést erre a service-re, akkor az ehhez tartozó végpont IP címre DNAT-ol. Majd kivizsgálásra kerül, hogy a végpont ezen a node-on van -e, vagy sem. Ha igen, akkor a forgalom átirányításra kerül a megfelelő CNI interfészen keresztül a podhoz, egyébként a csomag eldobásra kerül (Ez egy *externalTrafficPolicy: Local* annotációval valósítható meg). Majd a válasz úgy néz ki, hogy szintén egy eBPF program használatával a DNAT visszairányban, és átirányításra kerül a forgalom az ens3 interfészre.

A két mérést úgy végeztem el, hogy a Kubernetes Service számot növeltem a különböző mérési pontokban, és megmértem az összes ponthoz tartozó sávszélességet. Ugyanis az a feltevés, hogy a szekvenciális csomagfeldolgozás miatt, mivel egyre több service van létrehozva a klaszterben, a csomagfeldolgozási idő jelentősen meg fog nőni. Ez azt eredményezi, hogy a mért átviteli sebességek a service szám növelésével lecsökkennek. Ezzel szemben, mivel az eBPF hash és LRU adatszervezésű térképeket használ a service pod összerendelések tárolására, lényegesen jobb teljesítményre számítunk nagy service szám esetén.

11.3. Mérési eredmények kiértékelése

Többfajta mérési forgatókönyvet is végigjártam.

Első teszteset:

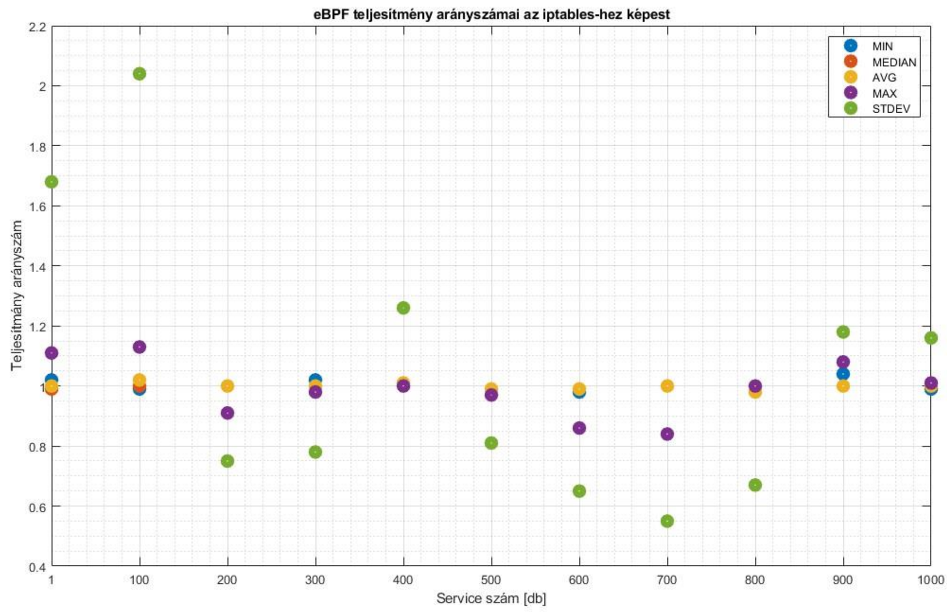
A Kubernetes Service számot 1, 100, 200…1000-ig vittem fel, 100-tól 100-as lépésközzel haladtam.

Cél:

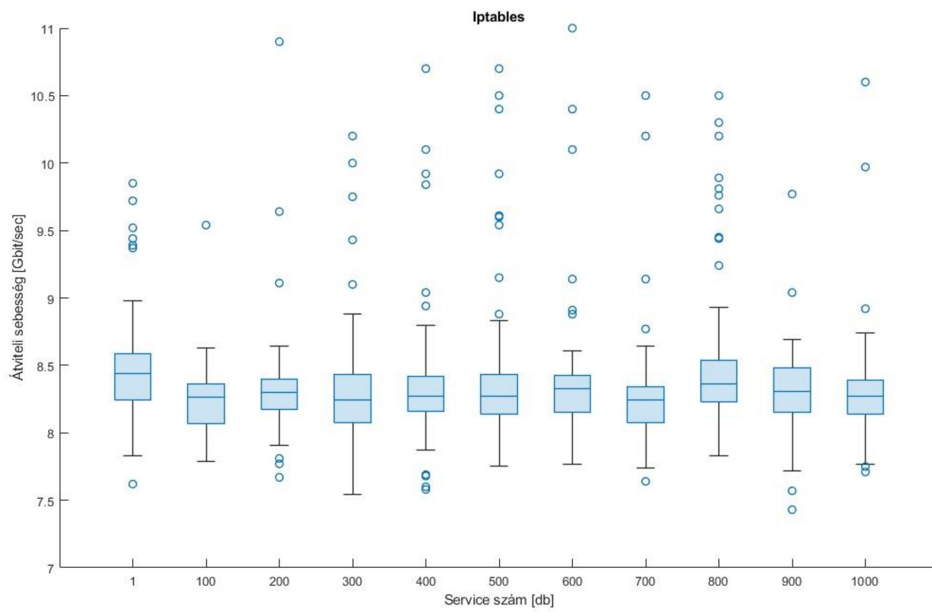
Ezekhez a pontokhoz tartozó átviteli sebesség mérése.

Kiértékelés:

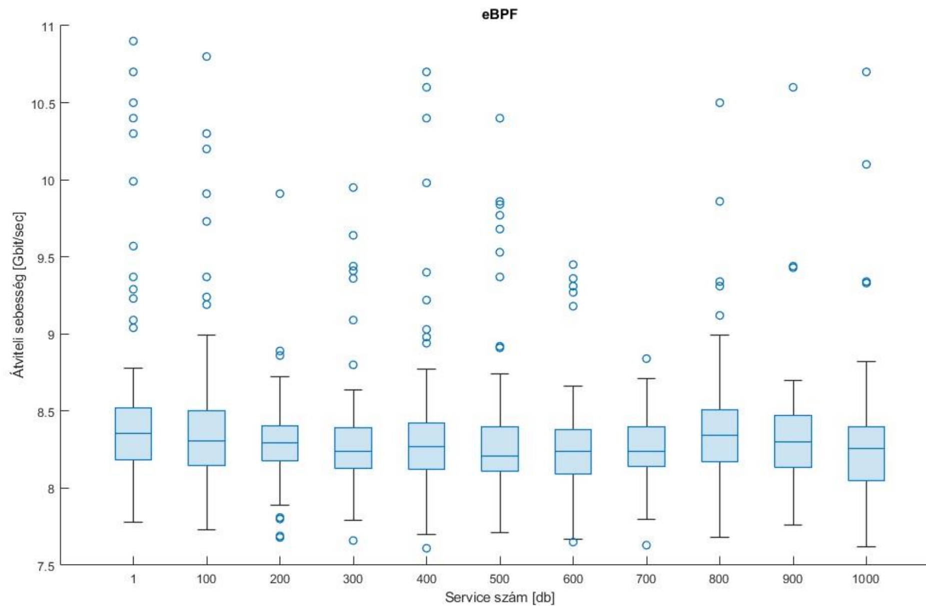
Első körben nem tapasztaltam különösebb performanciabeli különbséget, mind az iptables, mind az eBPF ugyanolyan teljesítményt nyújtott (33-35. ábra).



33. ábra: eBPF teljesítmény arányszámai az iptableshez képest



34. ábra: iptables box plot ábrák a megfelelő pontokhoz



35. ábra: eBPF box plot ábrák a megfelelő pontokhoz

Ami kirívóan eltér az eBPF esetében, az a szórás ingadozásának mértéke (33. ábra), az iptables esetében sokkal szűkebb ez a tartomány. A kapott eredmény feltehetően azért nem egyezett a hipotézisünkkel, mert elképzelhető, hogy a szekvenciális csomagfeldolgozás teljesítményromlása nagyobb service számnál fog bekövetkezni. Épp ezért változtattam a mérési skálán, ezúttal 1-től 10000-ig mentem fel, és 1, 1000, … 10000-ig 1000-es léptékkel növeltem a service számot. Azonban még ennél az esetnél se tapasztaltam különösebb eltérést, az eredmények az elsőhöz hasonlóak lettek.

Van még egy faktor, amelyet végül figyelembe kellett venni a mérés során. A KUBE-SVC lánchoz tartozó szabályok közül mindig az az utolsó, amelyet legutóbb létrehoztak. Így a szekvencián mindig akkor fog csak végigmenni, ha az utolsó service-hez tartozó Nodeporton keresztül próbálunk kommunikálni a poddal. Ugyanis, ha a service keresés során találat van a bejegyzések között, akkor nem fog tovább keresni a szabályok között, hanem visszatér a megfelelő alllánchoz. Ezt is figyelembe véve az eredmények az alábbiak szerint alakultak (36-39. ábra).

Második teszteset:

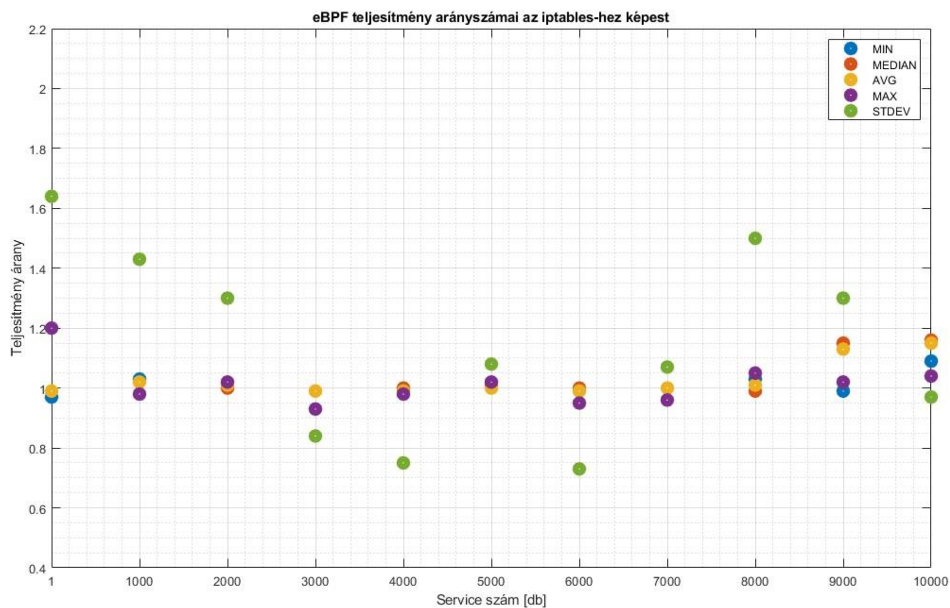
A Kubernetes Service számot 1, 1000, 2000…10000-ig vittem fel, 1000-tól 1000-es lépésközzel haladva. A mérés során mindig az utoljára hozzáadott NodePorton keresztül értük el a podot.

Cél:

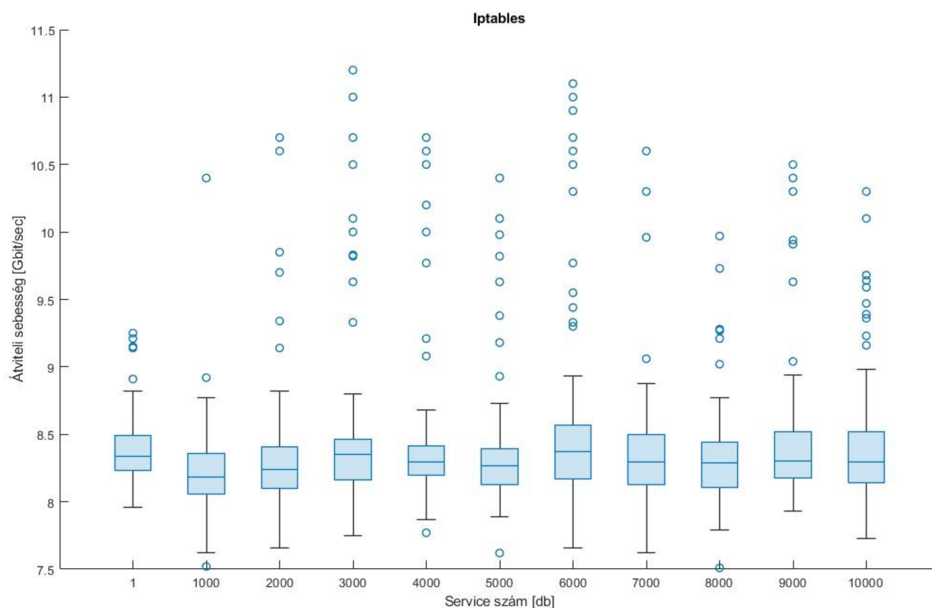
Ezekhez a pontokhoz tartozó átviteli sebesség mérése.

Kiértékelés:

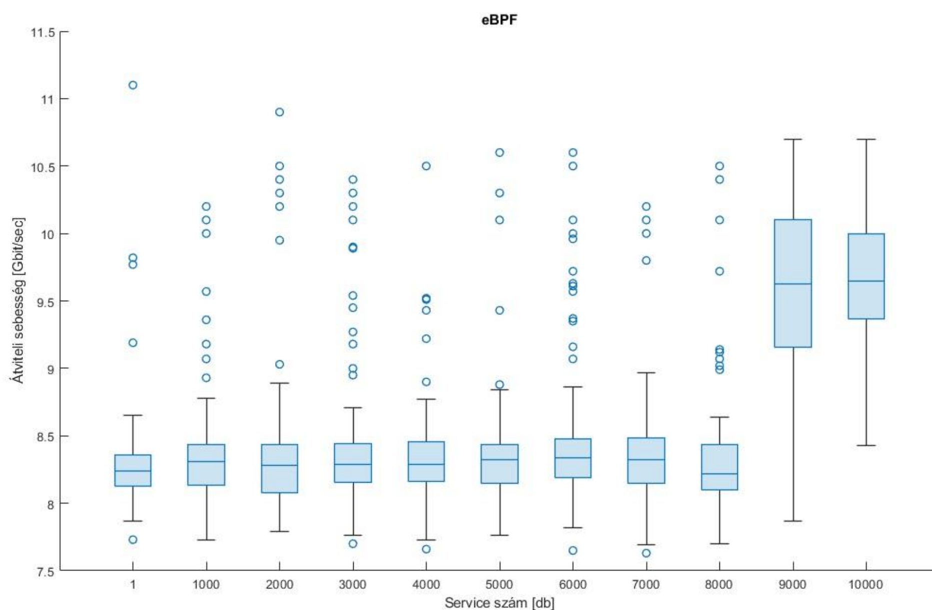
Itt már szembetűnőbb eredményekkel találkozhatunk, hiszen látható, hogy 9000-10000-es service nagyságrendben már jelentkezik számottevő teljesítménybeli különbség (39. ábra).



36. ábra: eBPF teljesítmény arányszámai az iptableshez képest



38. ábra: iptables box plot ábrák a megfelelő pontokhoz



39. ábra: eBPF box plot ábrák a megfelelő pontkohoz

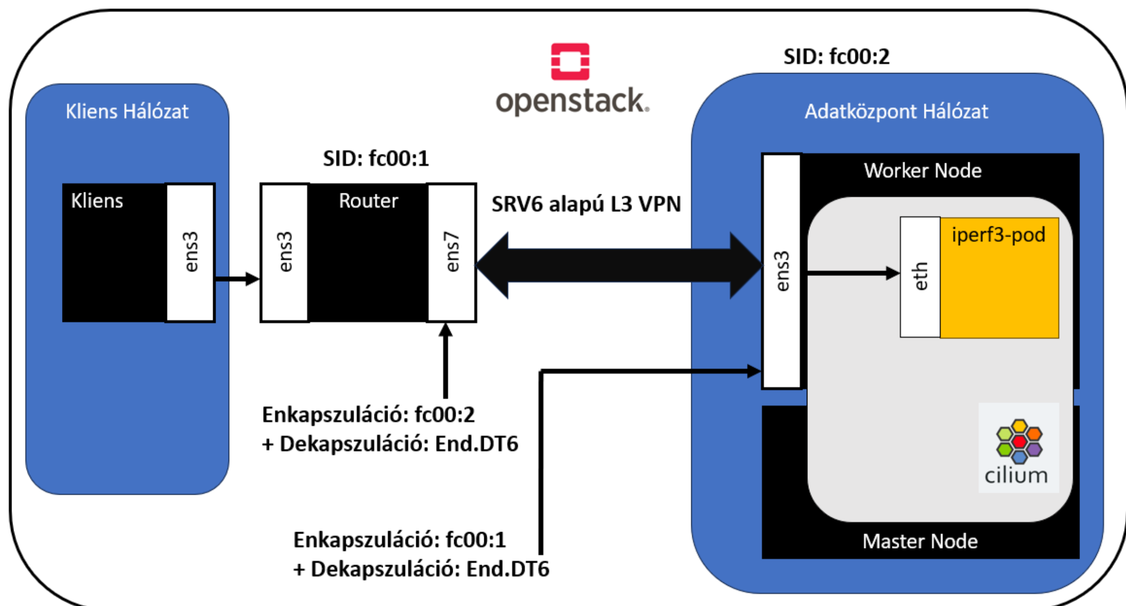
Azaz, ebből az a következtetés vonható le, hogy ipari körülmények között egyértelműen jobb performancia érhető el vele, szemben az iptables megoldáshoz képest.

Azt sem szabad elfelejteni, hogyha csak azt vesszük, hogy kisebb service számban hogyan teljesített az adott módszer, akkor nem tapasztaltunk lényeges teljesítménybeli

degradációt. Ennek a technológiának az alkalmazása gyakorlatilag nem növeli a csomagfeldolgozás műszaki költségét. Ezt azért fontos kiemelni, mert bizonyos hálózati megoldások implementálása eBPF-el lényegesen egyszerűbb lenne, és közben ugyanolyan performanciát nyújtana, mint a Linux kernele. Ezért összességében azt gondolom, hogy ennek a technológiának több aspektusa is ígéretes irányt képvisel, és érdemes ezzel foglalkozni a későbbiekben is.

11.4. SRv6 alapú hálózat létrehozása Openstack-en

A tesztkörnyezet összeállítása annyiban különbözik az előzőtől, hogy ebben az esetben a router és a worker node között egy SRv6 alapú, hálózati rétegbeli VPN csatorna került megvalósításra, azaz csak kiegészítő konfigurációkat kellett végezni a megfelelő hálózati eszközökön. A tesztrendszer sematikus ábrája az alábbi ábrán látható (38. ábra).



37. ábra: SRv6 alapú tesztrendszer

A hálózat abból a szempontból leegyszerűsített, hogy nem kerültek implementálásra köztes csomópontok, csupán az SRv6-os csomag becsomagolását/kicsomagolását végző végpontok lettek megvalósítva. Ebből az következik, hogy mindkét oldal enkapszulálása során egy-egy SID került elhelyezésre az SRH-ban (fc00:1 a routeren, és fc00:2 a worker node-on). A csomag dekapszulálása pedig az End.DT6 utasítás alapján történik, azaz az IPv6 fejléc, és az SRH kicsomagolása után a belső IPv6-os csomag továbbítása egy beállított IPv6-os routing tábla alapján történjen. Ez esetünkben megegyezik azzal a táblával, amit a Linux kernel alapértelmezetten

használ. A router, és a worker node szükséges konfigurációi a függelékben található. A kliens és a pod közötti ICMPv6-os [47] üzenetek forgalmának adatait *tcpdump* segítségével összegyűjtöttem, majd a Wireshark [48] szoftver segítségével pedig kielemeztem. A csomagok felépítése az alábbi ábrákon látható (39-43. ábra).

A kliens IPv6 címe: 2001:db8:1234:5679::1/64

A pod IPv6 címe: 2001:db8:1234:5678:8:2:1:841/104

```
Internet Protocol Version 6, Src: 2001:db8:1234:5679::1, Dst: 2001:db8:1234:5678:8:2:1:841
    0110 .... = Version: 6
    .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... 1011 0100 0100 1010 1001 = Flow Label: 0xb44a9
    Payload Length: 64
    Next Header: ICMPv6 (58)
    Hop Limit: 64
    Source Address: 2001:db8:1234:5679::1
    Destination Address: 2001:db8:1234:5678:8:2:1:841

Internet Protocol Version 6, Src: 2001:db8:1234:5678:8:2:1:841, Dst: 2001:db8:1234:5679::1
    0110 .... = Version: 6
    .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... 1001 0010 0011 1110 1000 = Flow Label: 0x923e8
    Payload Length: 64
    Next Header: ICMPv6 (58)
    Hop Limit: 62
    Source Address: 2001:db8:1234:5678:8:2:1:841
    Destination Address: 2001:db8:1234:5679::1
```

38. ábra: Kliens interfészénél lévő forgalom

```
Internet Protocol Version 6, Src: 2001:db8:1234:5679::1, Dst: 2001:db8:1234:5678:8:2:1:841
    0110 .... = Version: 6
    .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... 1011 0100 0100 1010 1001 = Flow Label: 0xb44a9
    Payload Length: 64
    Next Header: ICMPv6 (58)
    Hop Limit: 64
    Source Address: 2001:db8:1234:5679::1
    Destination Address: 2001:db8:1234:5678:8:2:1:841

Internet Protocol Version 6, Src: 2001:db8:1234:5678:8:2:1:841, Dst: 2001:db8:1234:5679::1
    0110 .... = Version: 6
    .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... 1001 0010 0011 1110 1000 = Flow Label: 0x923e8
    Payload Length: 64
    Next Header: ICMPv6 (58)
    Hop Limit: 62
    Source Address: 2001:db8:1234:5678:8:2:1:841
    Destination Address: 2001:db8:1234:5679::1
```

39. ábra: router ens3 interfészénél lévő forgalom

```
Internet Protocol Version 6, Src: fc00::1, Dst: fc00::2
    0110 .... = Version: 6
  > .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... 1011 0100 0100 1010 1001 = Flow Label: 0xb44a9
    Payload Length: 128
    Next Header: Routing Header for IPv6 (43)
    Hop Limit: 63
    Source Address: fc00::1
    Destination Address: fc00::2
  > Routing Header for IPv6 (Segment Routing)
> Internet Protocol Version 6, Src: 2001:db8:1234:5679::1, Dst: 2001:db8:1234:5678:8:2:1:841
Internet Protocol Version 6, Src: fc00::2, Dst: fc00::1
    0110 .... = Version: 6
  > .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... 1001 0010 0011 1110 1000 = Flow Label: 0x923e8
    Payload Length: 128
    Next Header: Routing Header for IPv6 (43)
    Hop Limit: 62
    Source Address: fc00::2
    Destination Address: fc00::1
  > Routing Header for IPv6 (Segment Routing)
> Internet Protocol Version 6, Src: 2001:db8:1234:5678:8:2:1:841, Dst: 2001:db8:1234:5679::1
```

40. ábra: router ens7 interfészénél lévő forgalom

```
Internet Protocol Version 6, Src: fc00::1, Dst: fc00::2
    0110 .... = Version: 6
  > .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... 1011 0100 0100 1010 1001 = Flow Label: 0xb44a9
    Payload Length: 128
    Next Header: Routing Header for IPv6 (43)
    Hop Limit: 63
    Source Address: fc00::1
    Destination Address: fc00::2
  > Routing Header for IPv6 (Segment Routing)
> Internet Protocol Version 6, Src: 2001:db8:1234:5679::1, Dst: 2001:db8:1234:5678:8:2:1:841
Internet Protocol Version 6, Src: fc00::2, Dst: fc00::1
    0110 .... = Version: 6
  > .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... 1001 0010 0011 1110 1000 = Flow Label: 0x923e8
    Payload Length: 128
    Next Header: Routing Header for IPv6 (43)
    Hop Limit: 62
    Source Address: fc00::2
    Destination Address: fc00::1
  > Routing Header for IPv6 (Segment Routing)
> Internet Protocol Version 6, Src: 2001:db8:1234:5678:8:2:1:841, Dst: 2001:db8:1234:5679::1
```

41. ábra: worker node ens3 interfészénél lévő forgalom

```
Internet Protocol Version 6, Src: 2001:db8:1234:5679::1, Dst: 2001:db8:1234:5678:8:2:1:841
    0110 .... = Version: 6
  > .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... 1011 0100 0100 1010 1001 = Flow Label: 0xb44a9
    Payload Length: 64
    Next Header: ICMPv6 (58)
    Hop Limit: 62
    Source Address: 2001:db8:1234:5679::1
    Destination Address: 2001:db8:1234:5678:8:2:1:841
Internet Protocol Version 6, Src: 2001:db8:1234:5678:8:2:1:841, Dst: 2001:db8:1234:5679::1
    0110 .... = Version: 6
  > .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... 1001 0010 0011 1110 1000 = Flow Label: 0x923e8
    Payload Length: 64
    Next Header: ICMPv6 (58)
    Hop Limit: 64
    Source Address: 2001:db8:1234:5678:8:2:1:841
    Destination Address: 2001:db8:1234:5679::1
```

42. ábra: Pod interfészénél lévő forgalom

Az IPv6-os csomagok becsomagolása/kicsomagolása nagyon jól megfigyelhető mind a router ens7, mind a worker node ens3 interfészén, emellett persze az is nagyon jól látszik, hogy a forgalom többi útvonalán natív IPv6 kommunikáció történik.

11.5. További fejlesztési irányok, összegzés

- A tesztrendszer teljessé tételéhez első körben mindenképpen implementálni kell a későbbiekben legalább egy megoldást arra, hogy a Kubernetes service-ek elérhetőek legyenek SRv6 L3 VPN csatornázás esetén is, akár a Linux alapú implementáció, akár a Cilium CNI keretein belül.
- Továbbá nagyobb szkópba nézve, ha a megvalósítás ezen lépcsőfokát sikerül meglépni, akkor jó lenne az SRv6-os csomagfeldolgozás végpontjait a pod közelébe hozni a klaszter oldalát tekintve. Ennek implementálásához alkalmas lehetne a pod egy sidecar konténerrel való kiegészítése, amely végezné az SRH és az IPv6 header enkapszulációját/dekapszulációját.
- Fejlesztésre szorul továbbá a kube-proxy vs. eBPF mérési szcenárió is, ugyanis meggyőződésem, hogy kimutatható szemmel még jobban látható különbség a két eset teljesítményére vonatkozóan. Érdemes lenne a későbbiekben a csomag/másodperc csomagfeldolgozási rátát kimérni, azonban ennek implementálása nehézkes, ugyanis a TCP protokoll beszabályozza a forgalmat az ehhez szükséges paraméterek változtatására vonatkozóan. UDP használatával pedig még nem feltárt okok miatt nem tudtam elérni az iperf podot a kliensről kube-proxy jelenlétével, eBPF használatával viszont igen.

Összességében úgy gondolom, hogy az eBPF technológia nagyon jól integrálható a Kubernetesbe, sőt mi több, szinte akármilyen területen alkalmazni lehet, ahol csomagfeldolgozás folyik. Belátható, hogy ezzel lényegesen lehet javítani az átviteli sebesség, vagy a késleltetés teljesítményén. Ez különösen fontos olyan területeken, ahol adatközpont hálózatok vannak jelen, mint például a transzporthálózatokban létrejött hálózati szeletek kezelésénél.

Ábra- és táblázatjegyzék

1. ábra: Végpontok közötti kritikus késleltetés követelmények [1] 5

2. ábra: A mobil adatforgalom exponenciális növekedése [2] 5

3. ábra: Network Slicing alkalmazási területei [4] 6

4. ábra: Hálózati szeletek realizálása [9] 8

5. ábra: Az 5G-s rádió logikai egységei [10] 9

6. ábra: Az SDN és NFV koncepció a hálózatmenedzsmentben [12] 10

7. ábra: Az MPLS Label felépítése [13] 11

8. ábra: MPLS architektúrája [14] 12

9. ábra: LSP-k szemléltetése [15] 13

10. ábra: MPLS TE architektúra [16] 13

11. ábra: LSP-k kihúzása RVSP segítségével [21] 14

12. ábra: SID formátuma [16] 16

13. ábra: SRH felépítése [23] 17

14. ábra: SRv6-os csomag feldolgozása a megfelelő routereken [24] 18

15. ábra: SRV6 TE architektúrája [16] 19

16. ábra: A kubernetes architektúrája 22

17. ábra: Service működése 22

18. ábra: Példa Kubernetes CNI-ra (Flannel) [27] 23

19. ábra: A láncok szerepe a csomagfeldolgozásban [28] 25

20. ábra: Az eBPF működése [29] 27

21. ábra: eBPF Map [29] 29

22. ábra: eBPF programok összekapcsolása [29] 30

23. ábra: eBPF ipari fejlesztéseket képviselő cégek [35] 30

24. ábra: Cilium architektúrája [36] 31

25. ábra: Alapértelmezett és kube-proxy által létrehozott iptables láncok [39] . 33

26. ábra: A Kubernetes láncokhoz tartozó szabályok [39] 33

27. ábra: Csomagfeldolgozás eBPF-el [39] 35

28. ábra: Az enkapszuláció folyamata [42] 36

29. ábra: A dekapszuláció folyamata [42] 37

30. ábra: A tesztrendszer sematikus ábrája 39

31. ábra: Csomagtovábbítás kube-proxy használatával 40

32. ábra: Csomagtovábbítás eBPF programok használatával ... 40
33. ábra: eBPF teljesítmény arányszámai az iptableshez képest ... 42
34. ábra: iptables box plot ábrák a megfelelő pontokhoz ... 42
35. ábra: eBPF box plot ábrák a megfelelő pontokhoz ... 43
36. ábra: eBPF teljesítmény arányszámai az iptableshez képest ... 44
378. ábra: iptables box plot ábrák a megfelelő pontokhoz ... 45
38. ábra: SRv6 alapú tesztrendszer ... 46
39. ábra: Kliens interfészénél lévő forgalom ... 47
40. ábra: router ens3 interfészénél lévő forgalom ... 47
41. ábra: router ens7 interfészénél lévő forgalom ... 48
42. ábra: worker node ens3 interfészénél lévő forgalom ... 48
43. ábra: Pod interfészénél lévő forgalom ... 48

Irodalomjegyzék

[1] Végpontok közötti kritikus késleltetés követelmények

Elérés időpontja: 2023. 11. 01.

Elérés:https://www.itu.int/dms_pub/itu-s/opb/jnl/S-JNL-VOL3.ISSUE3-2022-A42-PDF-E.pdf

[2] A mobil adatforgalom exponenciális növekedése

Faisal Tariq; Muhammad R. A. Khandaker; Kai-Kit Wong; Muhammad A. Imran; Mehdi Bennis; Merouane Debbah, "A Speculative Study on 6G", IEEE Wireless Communications, August 2020, doi: 10.1109/MWC.001.1900488

[3] A Network Slicing elméleti áttekintése

Shunliang Zhang, "An Overview of Network Slicing for 5G", IEEE Wireless Communications, June 2019, doi: 10.1109/MWC.2019.1800234

[4] Network Slicing alkalmazási területei

Elérés időpontja: 2023. 11. 01.

Elérés:https://media-bell-labs-com.s3.amazonaws.com/pages/20190729_1641/figure1_network_slicing.jpg

[5] Az MPLS architektúra szabványban való definiálása

E. Rosen, A. Viswanathan, R. Callon, „Multiprotocol Label Switching Architecture"

RFC 3031, RFC 3031 - Multiprotocol Label Switching Architecture (ietf.org)

January 2001, doi: 10.17487/RFC3031

[6] Az SRv6 működésének szabványban való definiálása

C. Filsfils, Ed., P. Camarillo, Ed., J. Leddy, D. Voyer, S. Matsushima,

"Segment Routing over IPv6 (SRv6) Network Programming", RFC 8986,

https://www.ietf.org/rfc/rfc8986.html#name-introduction

February 2021, doi: 10.17487/RFC8986

[7] Az SRv6 mobil felhasználói sík implementációja

S. Matsushima, Ed., C. Filsfils, M. Kohno, P. Camarillo, Ed., D. Voyer,

"Segment Routing over IPv6 for the Mobile User Plane", RFC 9433,

RFC 9433 - Segment Routing over IPv6 for the Mobile User Plane (ietf.org)

July 2023, doi: 10.17487/RFC9433

Elérés időpontja, 2023. 11. 02.

Elérés:https://medium.com/intel-tech/intel-mwc-barcelona-blog-srv6-mobile-user-plane-srv6-mup-technology-demo-1e5e5ea9ae22

[8] Az eBPF működése

Elérés időpontja: 2023. 10. 30.

Elérés: https://ebpf.io/

[9] Hálózati szeletek realizálása

Elérés időpontja: 2023. 11. 02.

Elérés:https://sdn.ieee.org/newsletter/december-2017/network-slicing-and-3gpp-service-and-systems-aspects-sa-standard

[10] Az 5G-s rádió logikai egységei

Ming Yan; Chien Aun Chan; Wenwen Li; Ling Lei; Qianjun Shuai; Andre F. Gygax; I. Chih-Lin, "Assessing the Energy Consumption of 5G Wireless Edge Caching",

2019 IEEE International Conference on Communications Workshops (ICC Workshops),

May 2019, doi: 10.1109/ICCW.2019.8756642

[11] SDN architektúra

Marcin Markowski; Przemysław Ryba; Karol Puchała,

"Software Defined Networking Research Laboratory-Experimental Topologies and Scenarios", 2016 Third European Network Intelligence Conference (ENIC), September 2016, doi: 10.1109/ENIC.2016.044

[12] Az SDN és NFV koncepció a hálózatmenedzsmentben

Elérés időpontja: 2023. 11. 02.

Elérés:

https://www.researchgate.net/publication/317002083_Combined_NFV_and_SDN_Appl

ications_for_Mitigation_of_Cyber-Attacks_Conducted_by_Botnets_in_5G_Mobile_Networks

[13] Az MPLS Label felépítése

Elérés időpontja: 2023. 11. 02.

Elérés:https://www.cisco.com/c/en/us/support/docs/multiprotocol-label-switching-mpls/mpls/4649-mpls-faq-4649.html#anc1

[14] Az MPLS architektúrája

Elérés időpontja: 2023. 11. 02.

Elérés:https://www.researchgate.net/figure/MPLS-network-and-components-22_fig8_255608741

[15] LSP-k szemléltetése

Elérés időpontja: 2023. 11. 02.

Elérés: https://www.grotto-networking.com/BBMPLS.html

[16] MPLS TE architektúrája

Li Zhenbin, Zhibo Hu, and Cheng Li,

"SRv6 Network Programming: Ushering in a New Era of IP Networks",

CRC Press, 2021

[17] IS-IS protokoll TE kiegészítései

H. Smit, T. Li, „Intermediate System to Intermediate System (IS-IS) Extensions for Traffic Engineering (TE)", RFC 3784,

https://datatracker.ietf.org/doc/html/rfc3784

June 2004, doi: 10.17487/RFC3784

[18] OSPF protokoll TE kiegészítések

S. Giacalone, D. Ward, J. Drake, A. Atlas, S. Previdi, "OSPF Traffic Engineering (TE) Metric Extensions", RFC 7471,

https://www.rfc-editor.org/rfc/rfc7471

March 2015, doi: 10.17487/RFC7471

[19] Röviden az CSPF algoritmusról

Elérés időpontja: 2023. 11. 02.

Elérés: https://doc.omnetpp.org/inet/api-3.6.0/neddoc/index.html?p=cspf-algorithm.html

[20] Az RSVP LSP kiegészítése

Daniel O. Awduche , Lou Berger , Der-Hwa Gan , Tony Li , Dr. Vijay Srinivasan , George Swallow, „RSVP-TE: Extensions to RSVP for LSP Tunnels”, RFC 3209

https://datatracker.ietf.org/doc/rfc3209/

December 2001, doi: 10.17487/RFC3209

[21] LSP-k kihúzása RVSP segítségével

Raymond Peterkin; Dan Ionescu, "A Hardware/Software Co-Design for RSVP-TE MPLS", 2006 Canadian Conference on Electrical and Computer Engineering, May 2006,

doi: 10.1109/CCECE.2006.277796

[22] SID fajták

Clarence Filsfils , Stefano Previdi , Les Ginsberg , Bruno Decraene , Stephane Litkowski , Rob Shakir, "Segment Routing Architecture", RFC 8402,

https://datatracker.ietf.org/doc/rfc8402/

July 2018, doi: doi: 10.17487/RFC8402

[23] SRH felépítése

Clarence Filsfils , Darren Dukes , Stefano Previdi , John Leddy , Satoru Matsushima , Daniel Voyer, "IPv6 Segment Routing Header (SRH)", RFC 8754,

https://datatracker.ietf.org/doc/html/rfc8754

March 2020, doi: doi: 10.17487/RFC8754

[24] SRv6-os csomag feldolgozása a megfelelő routereken

Elérés időpontja: 2023. 11. 02.

Elérés: http://www.fabricplane.com/srv6-node-types

[25] A kubernetes szoftver története, fejlődése, Elérés időpontja: 2023. 05. 21.

Elérés:https://kubernetes.io/blog/2018/07/20/the-history-of-kubernetes-the-community-behind-it/

[26] A linux netfilter framework, Elérés időpontja: 2023 10. 30.

Elérés: netfilter/iptables project homepage - The netfilter.org project

[27] Példa Kubernetes CNI-ra (Flannel)

Elérés időpontja: 2023. 11. 02.

Elérés:https://mvallim.github.io/kubernetes-under-the-hood/documentation/kube-flannel.html

[28] Az iptables láncok szerepe a csomagfeldolgozásban

Elérés időpontja: 2023. 11. 02.

Elérés:https://www.globo.tech/learning-center/linux-native-firewall-introduction-to-iptables/

[29] Az eBPF működése

Elérés időpontja: 2023. 11. 02.

Elérés: https://ebpf.io/what-is-ebpf/

[30] Clang/LLVM fordító, Elérés időpontja: 2023. 10. 30.

Elérés: GitHub - llvm/llvm-project: The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.

[31] JIT fordító, Elérés időpontja: 2023. 05. 27.

Elérés: https://www.freecodecamp.org/news/just-in-time-compilation-explained/

[32] Az eBPF támogatott processzor architektúrái

Elérés időpontja: 2023. 11. 02.

Elérés: https://www.man7.org/linux/man-pages/man8/tc-bpf.8.html

[33] BPF függvények

[34] BPF helper függvények

Elérés időpontja: 2023. 11. 02.

Elérés: https://man7.org/linux/man-pages/man2/bpf.2.html

Elérés időpontja: 2023. 11. 02.

Elérés: https://www.man7.org/linux/man-pages/man7/bpf-helpers.7.html

[35] eBPF ipari fejlesztéseket képviselő cégek

Elérés időpontja: 2023. 11. 02.

Elérés: https://ebpf.io/case-studies/

[36] Cilium architektúrája

Elérés időpontja: 2023. 10. 30.

Elérés: https://cilium.io/

[37] Express Data Path (XDP) a Ciliumban

Elérés időpontja: 2023. 10. 30.

Elérés: https://docs.cilium.io/en/latest/bpf/progtypes/#xdp

[38] Border Gateway Protocol szabványban való definiálása

Yakov Rekhter , Susan Hares , Tony Li, „

A Border Gateway Protocol 4 (BGP-4)", RFC 4271;

https://datatracker.ietf.org/doc/html/rfc4271

January 2006, doi: 10.17487/RFC4271

[39] Az alapértelmezett és a kube-proxy által létrehozott iptables láncok, Elérés időpontja: 2023. 10. 30. , Elérés: Architecture — Cilium 1.8.90 documentation (cilium-genbit.readthedocs.io)

[40] ExternalTrafficPolicy=Local annotáció a Nodeport service yaml fájlban, Elérés időpontja: 2023. 10. 30. , Elérés: Using Source IP | Kubernetes

[41] Virtual Routing and Forwarding a Linux kernelben, Elérés időpontja: 2023. 10. 30.

Elérés: https://docs.kernel.org/networking/vrf.html

[42] SRv6 enkapszuláció/dekapszuláció Ciliummal, Elérés időpontja: 2023. 10. 30.

Elérés: https://www.youtube.com/watch?v=ncYG-wScuL8&t=1268

[43] SRv6 implementálása a Linux kernelben, Elérés időpontja: 2023. 10. 30.

Elérés: https://segment-routing.org/

[44] Iperf3 hálózati performancia elemző Linuxos szoftver dokumentációja, Elérés időpontja: 2023. 10. 30.

Elérés: iPerf - iPerf3 and iPerf2 user documentation

[45] Openstack felhőszolgáltató, Elérés időpontja: 2023. 10. 30.

Elérés: Open Source Cloud Computing Platform Software - OpenStack

[46] Kubernetes NodePort service definíció, Elérés időpontja: 2023. 10. 30.

Elérés: https://kubernetes.io/docs/concepts/services-networking/service/

[47] Internet Control Message Protocol for Internet Protocol version 6 (ICMPv6)

Mukesh Gupta , Alex Conta, "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification", RFC 4443,

March 2006, doi: 10.17487/RFC4443

Elérés: https://www.rfc-editor.org/rfc/rfc4443

[48] Wireshark forgalomelemző program, Elérés időpontja: 2023. 30. 31. ,

Elérés: Wireshark · Go Deep

Rövidítések jegyzéke

MPLS: Multi Protocol Label Switching

IPv6: Internet Protocol version 6

SRv6: Segment Routing over IPv6

eBPF: extended Berkeley Package Filter

V2X: Vehicle-to-everything

RISC: Reduced Instruction Set Computer

TE: Traffic Engineering

5G: fifth-generation mobile network

NF: Network Function

AMF: Access and Mobility Management Function

UPF: User Plane Function

RRU: Remote Radio Unit

CU: Central Unit

DU: Distributed Unit

SDN: Software Defined Networking

VNF: Virtualized Network Function

NFV: Network Function Virtualization

LER: Label Edge Router

LSR: Label Switch Router

LSP: Label Switched Path

IP: Internet Protocol

FIB: Forwarding Information Base

FEC: Forwarding Equivalence Class

TCP: Transmission Control Protocol

UDP: User Datagram Protocol

AS: Autonomus System

Cos: Class of Service

TTL: Time To Live

LSDB: Link State Database

TEDB: Traffic Engineering Database

IS-IS: Intermediate System - Intermediate System

OSPF: Open Shortest Path First

RSVP: Resource Reservation Setup Protocol

CPU: Central Processing Unit

IGP: Interior Gateway Protocol

VPN: Virtual Private Network

BGP: Border Gateway Protocol

RFC: Request For Comment

SID: Segment Identifier

SRH: Segment Routing Header

SR: Source Routing

SL: Segment List

CNI: Container Networking Interface

NAT: Network Address Translation

NAPT: Network Address Port Translation

JIT: Just-In-Time compiler

VM: Virtual Machine

LPM: Longest Prefix Match

LRU: Least Recently Used

XDP: eXpress Data Path

TC: Traffic Control

SNAT: Source NAT

DNAT: Destination NAT

VRF: Virtual Routing and Forwarding

L3: Layer 3

ICMPv6: Internet Control Message Protocol version 6

Függelék

A virtuális gépek netplan konfigurációi:

Kliens:

```
network:
   version: 2
   ethernets:
       ens3:
           dhcp4: true
           dhcp6: false
           addresses:
                     - 2001:db8:1234:5679::1/64
           routes:
                   - to: 2001:db8:1234:5678::/64
                     via: 2001:db8:1234:5679::2
           match:
               macaddress: fa:16:3e:0f:21:be
           mtu: 8950
           set-name: ens3
```

Master Node:

```
network:
    version: 2
    ethernets:
        ens3:
            dhcp4: true
            dhcp6: false
            addresses:
             - 2001:db8:1234:5678::1/64
            routes:
                    - to: 2001:db8:1234:5679::/64
                      via: 2001:db8:1234:5678::3
            match:
                macaddress: fa:16:3e:15:76:9c
            mtu: 8950
            set-name: ens3
```

Worker Node:

```
network:
    version: 2
    ethernets:
        ens3:
            dhcp4: true
            dhcp6: false
            addresses:
                    - 2001:db8:1234:5678::2/64
            routes:
                    - to: 2001:db8:1234:5679::/64
                      via: 2001:db8:1234:5678::3
            match:
                macaddress: fa:16:3e:85:3a:6f
            mtu: 8950
            set-name: ens3
```

A klaszter feltelepítése:

Az alapvető koncepció az lenne, hogy minden IPv6-on működne, azonban a Cilium CNI korlátot ad erre. Egészen a legfrissebb verzióig sajnos a csatorna (tunneling) használata IPv4-es node IP címet igényel, ugyanis csak IPv4-es VXLAN enkapszulációt/dekapszuláció van implementálva a Cilium forráskódjában. Ebből adódóan a master, és a worker node-nak is egyaránt szükséges IPv4-es cím, tehát összességében dual-stack interfésze lesz a node-oknak. Azonban csak a worker node-nak szükséges elsődleges IPv4-es node IP-t adni, a master node működik elsődleges IPv6 címmel is. Ebből az következik, hogy csakis kizárólag a worker node-on lévő cilium-operator, cilium-agent, és a kube-proxy podok fognak IPv4 címet kapni. A klasztert a már korábban is használt *kubeadm* csomaggal telepítettem fel. A konfiguráció yaml fájlja:

```
apiVersion: kubeadm.k8s.io/v1beta3
kind: InitConfiguration
localAPIEndpoint:
  advertiseAddress: 2001:db8:1234:5678::1
nodeRegistration:
  kubeletExtraArgs:
    cluster-dns: 2001:db8:1234:5678:8:3:0:a
    node-ip: 2001:db8:1234:5678::1,172.16.69.5
---
apiServer:
  extraArgs:
    advertise-address: 2001:db8:1234:5678::1
    bind-address: '::'
    etcd-servers: https://[2001:db8:1234:5678::1]:2379
    service-cluster-ip-range: 2001:db8:1234:5678:8:3::/112
apiVersion: kubeadm.k8s.io/v1beta3
controllerManager:
  extraArgs:
    allocate-node-cidrs: 'true'
    bind-address: '::'
    cluster-cidr: 2001:db8:1234:5678:8:2::/104
    node-cidr-mask-size: '120'
    service-cluster-ip-range: 2001:db8:1234:5678:8:3::/112
etcd:
  local:
    dataDir: /var/lib/etcd
    extraArgs:
      advertise-client-urls: https://[2001:db8:1234:5678::1]:2379
      initial-advertise-peer-urls: https://[2001:db8:1234:5678::1]:2380
      initial-cluster: dome-ipv6-master-node=https://[2001:db8:1234:5678::1]:2380
      listen-client-urls: https://[2001:db8:1234:5678::1]:2379
      listen-peer-urls: https://[2001:db8:1234:5678::1]:2380
kind: ClusterConfiguration
networking:
  serviceSubnet: 2001:db8:1234:5678:8:3::/112
scheduler:
  extraArgs:
    bind-address: '::'
---
apiVersion: kubelet.config.k8s.io/v1beta1
cgroupDriver: systemd
clusterDNS:
- 2001:db8:1234:5678:8:3:0:a
healthzBindAddress: ::1
kind: KubeletConfiguration
```

A kiadott parancsok:

```
kubeadm init –config config.yaml
```

A kube-proxy nélküli esetnél a master node felhúzása után ki kell törölni a kube-proxy-t definiálóló deamonset-et, és az összes kube-proxy-val kapcsolatos beállításokat a klaszter konfigurációjából:

```
kubectl -n kube-system delete ds kube-proxy
# Delete the configmap as well to avoid kube-proxy being reinstalled during
a Kubeadm upgrade (works only for K8s 1.19 and newer)
kubectl -n kube-system delete cm kube-proxy
```

A worker node esetében is persze manuálisan meg kell adnunk a node IP címeket, a többi paraméter megadásánál pedig ugyanarra kell figyelni, mint amire már korábbi klaszter felhúzása során is kellett: A yaml fájl:

```
apiVersion: kubeadm.k8s.io/v1beta3
kind: JoinConfiguration
discovery:
  bootstrapToken:
    apiServerEndpoint: "[2001:db8:1234:5678::1]:6443"
    token: "j9oah9.woam4y0i4nq73s89"
    caCertHashes:
    - "sha256:af64acff5d17907ab9c2bc4b111bd98ed7e037c74ed6ef534e56e1d98af381a9"
    # change auth info above to match the actual token and CA certificate hash for
your cluster
nodeRegistration:
  kubeletExtraArgs:
    node-ip: 172.16.69.17,2001:db8:1234:5678::2
```

A kiadott parancs:

```
kubeadm join --config config.yaml
```

Itt mégegyszer felhívnám a figyelmet a node IP címek sorrendjére, ami ugyebár különbözik a master node esetén.

<u>Cilium CNI konfigurálása:</u>

A Cilium CNI feltelepítéséhez a Flannel-hez hasonlóan *helm chart*-ot használtam. Ahhoz, hogy a Cilium státuszinformációihoz hozzáférjek továbbá feltettem a virtuális gépre egy Cilium Command Line Interface-t (CLI). Ez kommunikál a cilium-agent podokkal, akik szolgáltatják folyamatosan a CNI-al kapcsolatos információkat. A telepítéshez használt parancsok:

```
CILIUM_CLI_VERSION=$(curl -s https://raw.githubusercontent.com/cilium/cilium-
cli/main/stable.txt)
CLI_ARCH=amd64
if [ "$(uname -m)" = "aarch64" ]; then CLI_ARCH=arm64; fi
curl -L --fail --remote-name-all https://github.com/cilium/cilium-
cli/releases/download/${CILIUM_CLI_VERSION}/cilium-linux-
${CLI_ARCH}.tar.gz{,.sha256sum}
sha256sum --check cilium-linux-${CLI_ARCH}.tar.gz.sha256sum
sudo tar xzvfC cilium-linux-${CLI_ARCH}.tar.gz /usr/local/bin
```

```
rm cilium-linux-${CLI_ARCH}.tar.gz{,.sha256sum}
```

Maga a Cilium telepítése a kube-proxy használatával:

```
helm repo add cilium https://helm.cilium.io/

helm install cilium cilium/cilium \
--namespace kube-system \
--set global.k8sServiceHost=2001:db8:1234:5678::1 \
--set global.k8sServicePort=6443 \
--set ipv4.enabled=false \
--set ipv6.enabled=true \
--set ipam.mode=cluster-pool \
--set ipam.operator.clusterPoolIPv6PodCIDRList="2001:db8:1234:5678:8:2::/104" \
--set ipam.operator.clusterPoolIPv4MaskSize=24 \
--set ipam.operator.clusterPoolIPv6MaskSize=112 \
--set bpf.masquerade=false \
--set bpf.tunnel=false \
--set enableIPv4Masquerade=true \
--set enableIPv6Masquerade=true
```

Az eBPF használatával ez annyiban különbözik, hogy be kell állítanunk egy paramétert, ami a kube-proxy használatára vonatkozik (pirossal jelölt). Mivel alapjáraton a kube-proxy-ra hagyatkozik a klaszterünk, így ezt nem kellett állítani az első esetben:

```
helm install cilium cilium/cilium \
--namespace kube-system \
--set global.k8sServiceHost=2001:db8:1234:5678::1 \
--set global.k8sServicePort=6443 \
--set kubeProxyReplacement=true \
--set ipv4.enabled=false \
--set ipv6.enabled=true \
--set ipam.mode=cluster-pool \
--set ipam.operator.clusterPoolIPv6PodCIDRList="2001:db8:1234:5678:8:2::/104" \
--set ipam.operator.clusterPoolIPv4MaskSize=24 \
--set ipam.operator.clusterPoolIPv6MaskSize=112 \
--set bpf.masquerade=true \
--set bpf.tunnel=true \
--set enableIPv4Masquerade=true \
--set enableIPv6Masquerade=true
```

A pod és service leírófájlai:

Az iperf3 podot egy úgynevezett Nodeport service segítségével tudom elérni a klaszter hálózatán kívülről. Ennek során a worker node interfészén (ens3) nyitunk egy portot, amin keresztül az iperf3 szerver szolgáltatás elérhetővé válik. Ahhoz hogy a podot összekössük a Nodeport-tal az alábbi annotáció szükséges pirossal jelölve a service yaml fájlban:

```
apiVersion: v1
kind: Service
metadata:
  name: iperf3
  labels:
    app: iperf3
spec:
  ipFamilies:
  - IPv6
  ipFamilyPolicy: SingleStack
  ports:
  - port: 5201
    protocol: TCP
    targetPort: 5201
  selector:
    app: iperf3
  externalTrafficPolicy: Local
  type: NodePort
```

Ez pedig a pod leíró yaml fájljában:

```
apiVersion: apps/v1
kind: Deployment
metadata:
   name: iperf3-deployment
spec:
  selector:
    matchLabels:
      app: iperf3
  template:
    metadata:
      labels:
        app: iperf3
    spec:
      containers:
      - name: iperf3
        image: leodotcloud/swiss-army-knife
        ports:
        - containerPort: 5201
```

A podban leírjuk, hogy milyen címke név segítségével kerüljön összekötésre a service-el, a service-ben pedig megadtuk, hogy milyen címke név alapján keresse a podot. Amit lényeges még megjegyezni, hogy ha IPv6-os kommunikációhoz akarunk használni bármilyen service-t, akkor ezt a yaml fájlban ennek megfelelően definiálni kell (kékkel

jelölt). A zölddel jelölt paraméterezés pedig azért fontos, mert így nem történik SNAT egy adott forgalomra (a node IP címére), ha a cél IP cím helyi node-on van, egyébként eldobja a csomagot. Ezt nem kötelező egyébként beállítani, csupán azért tettem ezt meg, mert ezzel is lehet a csomagfeldolgozási időn javítani.

Az SRv6 alapú hálózat összeállítása:

Router beállítások:

Első körben engedélyezni kell az SRv6-os forgalmat a kernel paraméterezésével:

```
echo '1' >  /proc/sys/net/ipv6/conf/all/seg6_enabled
echo '1' >  /proc/sys/net/ipv6/conf/ens3/seg6_enabled
echo '1' >  /proc/sys/net/ipv6/conf/ens7/seg6_enabled
```

Enkapszuláció:

```
ip -6 route add 2001:db8:1234:5678::/64 encap seg6 mode encap segs 1 [
fc00::2 ] dev ens7
```

Enkapszulációnál meg kell adni a router SID-jét:

```
ip -6 sr tunsrc set fc00::1
```

Dekapszuláció:

Szabályként (rule) meg kell adni hogy az adott SID esetén kérje le a paraméterben megadott táblát, ugyanis ebbe lesz a csomag kicsomagolásához tartozó SRv6-os függvény. Azonban a táblát először létre kell hozni:

```
echo 100 localsid >> /etc/iproute2/rt_tables
```

```
ip -6 rule add to fc00::1/128 lookup localsid
```

Az SRv6-os függvény hozzáadása a táblába:

```
ip -6 route add fc00::1/128 encap seg6local action End.DT6 table 254 dev
ens7 table localsid
```

A table 254 a fő (main) routing táblára utal, amelyet a router alapértelmezetten használ, ez kell majd a kicsomagolt IPv6-os csomag útvonalválasztására.

Ezen kívül be kellett még állítani, hogy a worker node szegmensét melyik útvonalon érjük el.

```
ip -6 route add fc00::2/128 via 2001:db8:1234:5678::2 dev ens7
```

Router beállítások:

Szintén végig kell menni a konfigurációkon, ahogyan a router esetében is kellett.

Kernel paraméterek:

```
echo '1' >  /proc/sys/net/ipv6/conf/all/seg6_enabled
echo '1' >  /proc/sys/net/ipv6/conf/ens3/seg6_enabled
echo '1' >  /proc/sys/net/ipv6/conf/ens7/seg6_enabled
```

Enkapszuláció:

```
ip -6 route add 2001:db8:1234:5679::/64 encap seg6 mode encap segs 1 [
fc00::1 ] dev ens7

ip -6 sr tunsrc set fc00::2
```

Dekapszuláció:

```
echo 100 localsid >> /etc/iproute2/rt_tables

ip -6 rule add to fc00::2/128 lookup localsid
```

Az SRv6-os függvény hozzáadása a táblába:

```
ip -6 route add fc00::2/128 encap seg6local action End.DT6 table 254 dev
ens3 table localsid
```

A szegmens statikus útvonala:

```
ip -6 route add fc00::1/128 via 2001:db8:1234:5678::3 dev ens3
```