Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Távközlési és Médiainformatikai Tanszék

Hálózati hatások kivédése felhő alapú alkalmazások számára

TDK dolgozat

Készítette:
Kovács Levente Tamás
Kocsis Richárd

Konzulensek:
Simon Csaba
Maliosz Markosz

Budapest, 2021 október 28.

Tartalomjegyzék

1. Bevezetés

A felhő rendszerek hatékony és rugalmas erőforrás megosztást biztosítanak a nagy számításigényű feladatokat futtató projektet számára. A felhő környezetre jellemző az elosztott és dinamikus feladatok automatizált, direkt operátori felügyelet nélküli végrehajtása. A felhőszolgáltatásoknak széles spektruma létezik, mivel a piacon levő szolgáltatók próbálnak minden eltérő igények kielégíteni. A legelterjedtebb kereskedelmi felhő kínálat a rugalmas és nagy számítási kapacitás nyújtására koncentrál, de jelentősek a tárolási kapacitást biztosító megoldások is. A változó szolgáltatásokhoz igazodva a felhőszolgáltatások komplexitása is nagyon változatos, a virtualizált infrastruktúra bérléstől az előre megírt programkód futtatását biztosító megoldásig terjed.

A széles körben elterjedt, változatos alkalmazásokat támogató felhő rendszerek olyan, korábban dedikált informatikai infrastruktúrát igénylő feladatok felhőben történő üzemeltetését is lehetővé teszik, mint például a távközlési iparágban megszokott magas rendelkezésre állású, valós idejű szolgáltatások. Ezeket a változatos követelményeket a minőségbiztosítási (Quality of Service – QoS) követelmények írják le. Az ilyen jellegű, sajátos QoS igényű szolgáltatások számára előnyös a felhő rendszerek egyik fontos tulajdonsága, a terhelés függvényében biztosított vízszintes skálázhatóság, amikor is egy adott szoftvermodult több példányban futtat a felhő rendszer, ezáltal több erőforrással támogatva a feladat végrehajtását. A több példányban futó szoftvermodulok hatékonysága szempontjából nem mindegy, hogy milyen módon oszlanak meg a kérések közöttük. A felhő rendszerekben ezt a feladatot a forgalom elosztó (Load Balancing – LB) megoldások látják el, amelyek változatos és gazdag képességekkel próbálnak megfelelni a különböző szolgáltatás típusok igényeinek. A forgalom elosztó megoldások az évek során a gyakorlatban is bebizonyították hasznosságukat, de mivel a felhő rendszerekben túlnyomó többségükben a számítástechnikában elterjedt web alapú szolgáltatásokat üzemeltetnek, a fennebb jelzett távközlési szolgáltatások sajátos QoS igényeit nem tudják maradéktalanul kielégíteni.

A felhőbe telepített szolgáltatások kezelése, a külső feltételek és az erőforrás igények változására automatikusan végrehajtott menedzsment műveletek (amit orkesztrációnak hívnak) biztosítják a hatékony feladatvégzést. Az orkesztráció képes a felhő rendszerbe integrált egyes hálózati (például virtuális kapcsolók, szoftveres forgalom elosztó) funkciókat is kezelni, de a felhő rendszeren kívüli hálózati beállításokra nem lehet hatása. Ennek következményeként egyes, a hálózatban történő változásokra a felhő rendszerben kell

adaptálódni. Amennyiben szigorú QoS igényű szolgáltatások számára a külső hálózati változások gyorsan mennek végbe, az orkesztrációs reakció idő túl hosszú lehet, ezért ebben az esetben a szolgáltatás architektúráját kell úgy megtervezni, hogy a hálózati dinamika ellenére a QoS igények ki legyenek elégítve.

Ebben a dolgozatban megvizsgáljuk egy, a felhő rendszerek hálózati környezetének gyakori elrendezéséből fakadó, a távközlésben megszokott szigorú QoS igényű szolgáltatásokat érintő hatást, javaslatot teszünk a hatás kivédésére, valamint egy minta megvalósítás segítségével értékeljük a rendszer és komponensei teljesítményét.

A következő fejezetben áttekintjük a virtualizációs megoldásokat és a Kubernetes felhő rendszert. Ezek után a hálózat virtualizálási megoldásokról írunk, de itt vezetjük be a terhelés elosztási módszereket is. A 4. fejezetben bemutatjuk egy széles körben használt terhelés elosztási mechanizmus által a Kubernetes klaszterbe telepített szolgáltatásokra gyakorolt negatív hatást, majd a következő fejezetben részletesen ismertetjük megoldási javaslatunkat. A 6. fejezetben bemutatjuk javaslatunk mérés alapú teljesítmény értékelését, végül összefoglaló résszel zárjuk a dolgozatunkat.

2. Felhő rendszerek és virtualizáció

A felhő alapú megoldások minden bizonnyal arról kapták a nevüket, hogy az informatikai szolgáltatások a felhasználók számára érzékelhetetlen vagy kevéssé érzékelhető HW infrastruktúrából, a felhőből jönnek. Ezért az „Internetről jövő” szolgáltatásokat, az „ott kint” futtatott szoftvereket, megoldásokat „cloud computing” vagy „felhő megoldásoknak” hívjuk. Leegyszerűsítve, a felhő alapú szolgáltatás során a szolgáltatást a saját gépünkön vesszük igénybe, lokálisan, de a szolgáltatás nyújtásáért felelős infrastruktúra ettől független helyszínen működik, nem feltétlenül helyben vannak üzemeltetve a szerverek. Továbbá, azok nem is feltétlenül a mi tulajdonunkban vannak, sok esetben nem is tudjuk, hogy pontosan milyen konfigurációjuk van és hol helyezkednek el. Ellenben pontosan definiálva van, hogy milyen erőforrásokat nyújtanak, milyen rendelkezésre állással, biztonsággal és természetesen milyen költségért. A felhő szolgáltatásokat több szempont alapján is kategorizálhatjuk, például annak alapján, hogy a felhő alapú megoldásokat kínáló szolgáltató milyen módszerrel osztja szét az erőforrásait a felhasználók között, hogy teljes platformot kínál-e, esetleg csupán egy adott szoftvert, vagy csak tárhelyet. A modern felhő rendszerek adatköztpontba szervezett szerver klaszterekből épülnek fel. A kulcstechnológia, amely lehetővé teszi, hogy a felhőszolgáltató a szerver klasztereket nem egy-egy különálló számítógépként (bear metal), hanem absztrakt erőforráscsomagként tudja felajánlani, virtualizációnak nevezik. A fejezet további részében

2.1 A virtualizáció

A virtualizáció [1] egy absztrakciós megoldás, ami elválasztja a fizikai hardvert az operációs rendszertől, és nagyobb erőforráskihasználást és rugalmasságot biztosít. A virtualizáció több virtuális gép használatát is lehetővé teheti, a virtualizáció típusától függően különböző operációs rendszerek futhatnak egyidőben, egymástól teljesen függetlenül, egymás „mellett” ugyanazon a számítógépen. Amennyiben hardver virtualizációs megoldást használunk, a virtualizált egységet virtuális gépnek (Virtual Machine – VM) nevezzük és minden egyes VM saját virtuális hardver-kiépítéssel rendelkezik (pl. RAM, CPU, stb.), a vendég operációs rendszerek ehhez az erőforrás készlethez férnek hozzá, csak ezzel tudnak dolgozni. Az operációs rendszer tehát egy teljes értékű hardverkészletet lát, teljesen függetlenül a számítógép valódi hardver-kiépítésétől. A virtuális gépek által használt adatokat (beleértve a programozási könyvtárakat is) fájlokban tárolja a gazdarendszer, ezáltal lehetővé téve azt, hogy a teljes környezet képfájlként (VM image) gyorsan lementhetőek, másolhatók

és kezelhetők legyenek. Teljes rendszerek (teljesen konfigurált alkalmazások, operációs rendszerek, BIOS és virtuális hardver) mozgatható át rövid idő alatt egyik fizikai szerverről a másikra, miközben a külső szemlélők (felhasználók) csupán néhány másodperces leállást tapasztalnak.

A virtualizáció gyökerei az 1960-as évekig nyúlnak vissza, amikor is a nagy (és költséges) mainframe számítógépek erőforrásait próbálták több feladat (job) közt megosztani. Idővel, a kisebb méretű gépek megjelenésével hatékonyabb, olcsóbb megoldás kínálkozott a számítási teljesítmény elosztására, így az 1980-as évekre a személyi számítógépek és nagyobb teljesítményű munkaállomások és szerverek elterjedtségéhez képest a virtualizáció nem volt széles körben alkalmazott technológia.

Az 1990-es években a szakmai közösség megtapasztalta, hogy a virtualizáció miként képes megoldani néhány aktuális problémát. Meglepő módon, ezen problémák egy rész éppen az olcsó hardverek elterjedésével jelent meg: az alacsony kihasználtság, a növekedő fenntartási költség és a sebezhetőség (támadhatóság, gyenge informatikai biztonság).

Az elmúlt húsz év fejlődésének köszönhetően a virtualizáció egy érett technológia lett, ezáltal napjainkban a virtualizációnak kiemelt szerepe van az informatikai folyamatokban, rugalmasságával, biztonságával segíti a vállalkozásokat. Azáltal, hogy a virtualizálásakár egyetlen számítógépet vagy kiszolgálót használva több, különállóerőforrás csoportot különít el, javítja a skálázhatóságot és a számítási feladatok hatékonyságát, miközben kevesebb kiszolgálót használ, kevesebb energiát fogyaszt, így csökkennek az infrastrukturális és a karbantartási költségek.

2.2 Virtualizáció típusok

A virtualizálás többféleképpen csoportosítható, a legelterjedtebb megosztás alapján négy fő kategóriába sorolható. Az első az asztali virtualizálás, amelynek eredményeképp egyetlen központi kiszolgáló egyedi asztali rendszereket tesz elérhetővé, és kezeli is azokat. A második a hálózati virtualizálás, amely a hálózati sávszélességet egymástól független csatornákra osztja fel, majd ezeket meghatározott kiszolgálókhoz vagy eszközökhöz rendeli. A harmadik kategória a szoftvervirtualizálás, amely elkülöníti az alkalmazásokat a hardvertől és az operációs rendszertől. Végül a negyedik kategória a tárolási virtualizálás, amely több hálózati tárolási erőforrást egyesít egyetlen tárolóeszközzé, amelyet több felhasználó is elérhet. Mivel a munkánk során az operációs rendszer szintű virtualizációt használó Docker konténerekkel foglalkoztunk, ezért a továbbiakban ezt részletezzük.

2.3 Operációs rendszer szintű virtualizáció

Ez a virtualizációs technika [2] lehetőséget biztosít arra, hogy egy operációs rendszeren belül több egymástól izolált, önálló környezetet hozzunk létre. A különböző típusai más és más szintű izolációt valósítanak meg és eltérő szinten szabályozható bennük a fizikai, és az operációs rendszer szintű erőforrásokhoz való hozzáférés is. Ezt a technológiát szokás több különböző néven is emlegetni: sandbox, jail, particionálás stb. Több fajta operációs rendszer szintű virtualizációs technológia létezik. Vannak, amik az operációs rendszer szerves részét képezik, mások kiegészítéseként érhetőek el. Közös jellemzőjük, hogy úgy egészítik ki az alap operációs rendszert, hogy az önmagán belül képes legyen létrehozni több virtuális környezetet (Virtual Environment - VE). Ezek a VE-k valójában ugyanannak az operációs rendszernek egy-egy virtuális példányai. Ezért ez a technika nem képes az alaprendszer operációs rendszer változatától eltérő rendszert futtatni. Cserébe a virtualizációból adódó teljesítmény-csökkenés 1-3 % környékén tartható mind számítási, mind pedig I/O hozzáférés esetén valamint akár több száz virtuális környezetet is ki lehet alakítani ugyanazon a fizikai eszközön. Ebből adódon ennek a technikának a legjelentősebb felhasználói a tárhely (hosting) szolgáltatók, akik Virtuális Privát Szerver (VPS) szolgáltatásokat nyújtanak e környezet segítségével.

2.4 Linux konténerek

Egy Linux konténer [3] egy operációs rendszer szintű virtuális környezet, mellyel képesek vagyunk egy gépen több izolált Linux rendszert futtatni. Egyebek mellett saját fájlrendszere, hálózati és folyamat stack-je van, viszont a kernelen osztozik a host géppel és a többi futó konténerrel (szemben a tradicionális értelemben vett virtualizációval), aminek köszönhetően lényegesen kisebb többletmunkával, gyorsabban létrehozhatóak, mint egy virtuális gép. Egy konténert a Linux kernel izolációt biztosító funkcióival lehet létrehozni, de nem kernel szintű fogalom. A használt kernelfunkciókhoz interfészt nyújtó LXC [4] az, ami a mai értelemben vett konténer fogalmát megalapozta (ezért is hívjuk őket LinuX Container-eknek).

A Linux Daemon (LXD) egy bővítmény az LXC rendszerhez, ami egy felhasználóbb barát felületet biztosít a könnyebb menedzselés érdekében. Biztosít egy REST API-t, amin keresztül a az LXC instance-ket lehet menedzselni. A kliensek interfészekhez csatlakoznak és az LXD démonon keresztül vezérlik a konténereket [4].

2.5 Docker

A Docker [5] jelenleg az egyik legelterjedtebb container framework, amelynek implementációja erősen támaszkodik a Linux kernel nyújtotta szeparációs lehetőségekre (namespacek, control groupok). A Docker fejlesztők és sysadminok számára egyaránt egy nagyon egyszerű lehetőséget biztosít hordozható alkalmazások létrehozására és menedzselésére. Segítségével bárki fejleszthet és becsomagolhat egy alkalmazást például a saját laptopján, és biztos lehet benne, hogy az egy teljesen másik környezetben, tipikusan egy cloud szerveren futtatva is ugyanúgy működni fog. A másik komoly előnye a sebesség. A technológiából fakadóan a Docker containerek kicsik és gyorsak, hiszen tulajdonképpen csak egy, a kernelen futó sandbox környezetről van szó, amely kevés erőforrást emészt fel. További két előnye, amelyet érdemes megemlíteni, az a modularitás és a skálázhatóság. A Dockerrel nagyon egyszerűen "feldarabolhatjuk" az alkalmazásunkat különböző konténerekbe.

A Docker alapvetően három részből áll: a daemon, a kliens, és egy REST API. A daemon a host gépen fut és parancsokat fogad a klienstől, az API segítségével pedig közvetlenül kommunikálhatunk a daemonnal. A felhasználók a klienst használják és segítségével adják ki a különböző parancsokat. A containerek definiálására a dockerfile szolgál, ez instrukciókat tartalmaz arra vonatkozóan, hogyan épüljön fel a container. Ezekkel a parancsokkal például előírhatjuk az operációs rendszer verzióját, bizonyos Linux csomagok installációját, portok megnyitását, és további, a mindennapi üzemeltetési gyakorlatban szükséges funkciót.

2.6 Kubernetes

A Kubernetes [6] nyílt forrású orkesztrációs platform a konténer alapú alkalmazások menedzseléséhez. A Kubernetes egy gazdag és növekvő nyílt forráskódú közösségi termék, ami biztosítja a szoftvert a megbízható, skálázható és elosztott rendszerek építéséhez és telepítéséhez. A Kubernetes számos absztrakciót és API-t biztosít, amelyek megkönnyítik az elosztott architektúrák kiépítését. A podok és konténerek csoportja a különböző csapatok által kialakított image-eket egyetlen telepíthető egységbe csoportosítja. A podok egy vagy több konténernek egy csoportja, az ezeken belül futó alkalmazások közös erőforrással rendelkeznek. A Kubernetes a szolgáltatások számára terheléselosztást, menedzsmentet és elhatárolást biztosít [6]. A Kubernetes-nek kétfajta csomópont típusa van a mester csomópont (master node), ami menedzseli a clustert és a végrehajtó csomópontok (worker node-k),

amiken az applikáció fut. Minden node-ba be van építve egy Kubelet nevű ügynök (agent), ami menedzseli a node-t és kommunikál a masterrel.

2.6.1 A Kubernetes komponensei

A következőkben röviden összefoglaljuk a Kubernetes rendszer fontosabb komponenseit.

pod: A Kubernetes elemi, menedzselt virtualizált egysége; a konténereket alkalmazás jellege alapján egy közös csoportba, a pod-ba szervezik; pod-on belül könnyen elérhetik egymást a konténerek, külső alkalmazások számára csak a port számuk alapján lehet különbséget tenni közöttük.

API server: A Kubernetes frontend-je, web interfészt és API-t szolgáltat. Ezzel kommunikál a felhasználó menedzser, és a többi UI eszköz is.

etcd: Elosztott kulcsérték tároló, a klaszter menedzseléséhez szolgáló információknak.

Scheduler: Az elosztott működést biztosítja meg a klaszter csomópontjain, mivel minden új feladat számára meghatároz egy csomópontot, ahol az fog futni.

Controller Manager: Figyeli a csomópontokat és beavatkozik ha változások miatt adaptálni kell a rendszert.

Container runtime: A konténerek keretszoftvere, leggyakoribb esetben a Docker. A legfrissebb Kubernetes verzióban már nem a Docker a keretszoftver, de a Docker képfájlok (image) változtatás nélkül továbbra is futtathatóak.

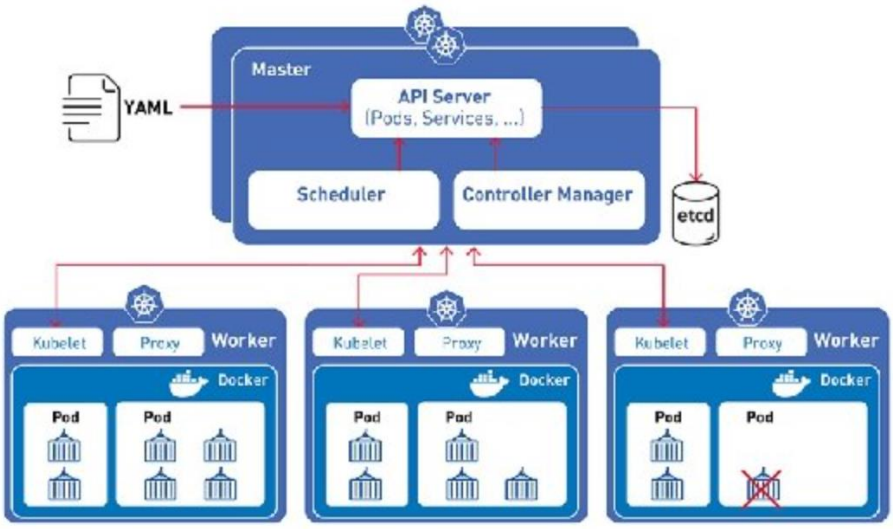
Kubelet: Ez egy ügynök (agent) és minden klaszterbeli csomóponton futnia kell; a feladata, hogy a csomópontra ütemezett konténerek az elvárt módon működjenek.

Kube proxy: a külső kérések számára biztosítják a podok klaszteren belüli egységes elérését (kérés routolása), valamint terhelés elosztó feladatokat is végeznek

Kube control (kubectl): parancssori interfész (command line interface – CLI), amin keresztül az adminisztrátor tud szolgáltatásokat indítani és cluster menedzsment feladatokat ellátni.

A Kubernetes felépítését és a fenti komponensek kapcsolatát a 4. ábra szemlélteti. Az látható, hogy a teljes klaszter egy mester (master) csomópontra (node) telepített funkciók segítségével van vezérelve. Ezeket a funkciókat a Scheduler, etcd, API server és a Controller Manager komponensek valósítják meg. Minden további csomópont számítási feladatokat lát el (worker

node), ezeken futnak a podokba szervezett konténer példányok. A worker node-ok klaszter szintű feladatait valósítja meg a proxy és a kubelet ügynök.



1. ábra Kuberntes klaszter felépítése [6]

3. Felhő rendszerekhez kapcsolódó hálózati megoldások

A gyakorlatban üzemeltetett felhő rendszerek sokrétű hálózati feladatot kell kezeljenek. Ebben a fejezetben, mielőtt még rátérnénk az általunk vizsgált probléma bemutatására, áttekintjük azokat a fontosabb mechanizmusokat, amelyek egy ilyen integrált rendszerben előfordulhatnak. A rendszerünk Linux alapú szerverekre épül, így először a Linux rendszer által biztosított hálózati mechanizmusokról beszélünk. A felhő rendszerünk, igazodva a modern távközlési rendszerekhez, különös tekintettel az új generációs 5G mobil rendszerekre, Kubernetes alapú, ezért röviden bemutatjuk a Kubernetes hálózati megoldásait is. Végül, a felhő rendszert kiszolgáló, az adat forgalmat a kiszolgálást biztosító szoftver egységhez (például egy Kubernetes podban, konténerbe szervezett webszerver) irányító terhelés elosztó megoldásokról nyújtunk áttekintést.

3.1 Linux beépített hálózati megoldások

A Linux gazdag virtuális hálózati képességekkel rendelkezik, amelyek alapjául szolgálnak a virtuális gépek és konténerek, valamint a felhő környezetek kialakításához, használatához. A következő részben a különböző technikákat mutatom be röviden [7].

3.1.1 Bridge (híd)

A híd hasonlít egy switchre, a hozzákapcsolt gépek ethernet interfacei között teremt kapcsolatot. A csomagok továbbításához az Ethernet címet használja nem pedig az IP-címet, ez azért jó, mert a továbbítás a 2. rétegben történik és az összes többi protokoll átláthatóan tud áthaladni rajta.

3.1.2 Bond

A Linux bond illesztőprogram segítségével több hálózati csatolót logikailag egyetlen interfésszé tudunk összekapcsolni. A kapcsolt interfészek viselkedése függ attól, hogy milyen módban vannak, nyújthatnak készenléti vagy terheléselosztási szolgáltatásokat. Továbbá a kapcsolat sebességének növelésére vagy monitorozására is alkalmasak.

3.1.3 Team

A team nagyon hasonlít a bond-hoz, a célja, hogy több portot az L2 rétegben egyetlen logikai egységbe csoportosítson, ez az úgynevezett teamdev. A bond-hoz képest ugyanazt a problémát más megközelítéssel oldja meg, például egy zár nélküli (RCU) TX / RX útvonalat

és moduláris felépítést alkalmazva. Funkcionális különbség a bond és a team között, hogy egy team újabb funkciókat is támogat (pl. más terheléselosztási mechanizmust, IPV6 kapcsolatfigyelést, a D-Bus interfészt).

3.1.4 VLAN (virtualís LAN)

A virtuális helyi hálózati kapcsolatok (Virtual LAN - VLAN) segítségével át tudjuk alakítani a hálózati topológiát igény szerint, anélkül, hogy a fizikai kábeleket mozgatnunk kellene. Lehetővé teszik, hogy a 2. rétegben lévő hálózatok ugyanazt a fizikai kapcsolatot használják, ezáltal rugalmasabb és költséghatékonyabb kábelezési topológiát tudunk kialakítani. Jellemzően akkor használjuk a VLAN-t, ha különböző alhálózatokba szeretnénk rendezni a virtuális gépeket, névtereket vagy a gazda (host) gépeket.

3.1.5 VXLAN (Virtual eXtensible LAN)

A virtuálisan kiterjesztett VLA (Virtual eXtensible LAN - VXLAN) egy alagútprotokoll, ami (névhasonlóság ellenére) nem a VLAN-t egészíti ki, de a korlátozott VLAN azonosítók problémájára nyújt megoldást. A VXLAN-nal a hálózati azonosító 24 bitre bővül, ami azt jelenti, hogy 224 virtuális LAN interfészünk lehet, ami 4096-szorosa a VLAN-nal elérhető kapacitásnak. A protokoll UDP felett működik, egyetlen cél port segítségével.

3.1.6 MACVLAN

A (MAC alapú virtuális LAN) MACVLAN lehetővé teszi egyetlen fizikai interfész számára, hogy egyszerre több 2. rétegbeli MAC és 3. rétegbeli IP címe legyen, ezeket MACVLAN alinterfészeknek nevezik. A különbség a VLAN és a MACVLAN által létrehozott alinterfészektől között, hogy a VLAN alinterfészekkel minden alinterfész egy másik 2. rétegbeli (layer 2 – L2) tartományhoz fog tartozni és mindegyiknek ugyanaz lesz a MAC címe. A MACVLAN esetében viszont minden alinterfész egyedi MAC és IP címeket kap. A MACVLAN interfészt általában a virtualizációs alkalmazásokhoz használják, és mindegyik MACVLAN interfész egy tárolóhoz vagy virtuális géphez csatlakozik.

A MACVLANnak 5 típusa van: privát, VEPA, Bridge, Passthru, és a forrás. A privát típus nem engedélyezi a kommunikációt a MACVLAN példányok között ugyanazon a fizikai felületen, még akkor sem, ha a külső kapcsoló támogatja. A VEPA esetében két MACVLAN példány között az adatok ugyanazon a fizikai felületen kerülnek továbbításra. A Bridge típus lehetővé teszi, hogy a gazdagépen lévő végpontok képesek legyenek egymással beszélni anélkül, hogy a csomagok elhagynák a gazdagépet. A Passthru (a pass through-ból egyszerűsítve) lehetővé teszi egyetlen virtuális gép közvetlen csatlakoztatását a fizikai

interfészhez. Végül a forrás módot használják a forgalom szűrésére az engedélyezett MAC címek alapján.

3.1.7 IPVLAN

Az IPVLAN hasonló a MACVLAN-hoz, azzal a különbséggel, hogy a végpontoknak ugyanaz a MAC-címe. Az IPVLAN-nak két fajta módja létezik a L2 és L3 mód. Az IPVLAN L2 módban úgy működik, mint egy MACVLAN bridge módban. IPVLAN L3 módban a szülő interfész úgy működik, mint egy útválasztó és a csomagok a végpontok között kerülnek továbbításra, ami jobb skálázhatóságot biztosít.

3.1.8 VETH

A VETH (virtuális Ethernet) eszköz egy lokális Ethernet alagút. Az eszközök párban állnak és a pár egyik tagjának kézbesített csomagokat azonnal megkapja a másik eszköz is. Ha egyik eszköz sem működik, a pár kapcsolati állapota inaktív lesz. Akkor használunk VETH konfigurációt, ha a névtereknek kommunikálniuk kell a gazdagép névterével vagy egymás között.

3.1.9 Dummy

A *dummy* interfész teljesen virtuális, mint például a loopback interfész. A dummy interfész célja, hogy olyan eszközt biztosítson, amely csak egy utat biztosít a csomagok számára, de ténylegesen nem küldünk rajta csomagot. Főként tesztelésre és hibakeresésre használjuk manapság.

3.2 Kubernetes hálózat

A hálózat központi része a Kubernetesnek, ezért nagyon fontos, hogy helyesen telepítsük és konfiguráljuk a klászt megfelelő működéséhez. A hálózat kiépítésekor 4 eltérő hálózati problémát kell megoldanunk [8]:

1. Konténerek közti kommunikáció.
2. Podok közötti kommunikáció.
3. Podok és a szolgáltatások közötti kommunikáció.
4. Külső szolgáltatások közötti kommunikáció.

Az első problémát megoldják maguk a Kubernetes podok és egy podon belül a konténerek a helyi hálózati interface-n keresztül tudnak kommunikálni.

A 2. probléma megoldására a Kubernetesben kifejlesztettek egy saját hálózati modellt. Minden pod kap egy saját IP címet, ezért nincs szükség külön kapcsolatos kiépítésre a podok között. Nem kell foglalkoznunk a konténer portok és a gazda (host) portok összehangolásával. A modellnek köszönhetően, úgy tudjuk kezelni az egyes podokat, mint a virtuális gépeket vagy fizikai hostokat a hálózat szempontjából. A modell implementációjához szükséges néhány alapvető követelményt megvalósítanunk.

- A csomópontban (node) lévő összes podnak egymással NAT nélkül kell tudniuk kommunikálni.

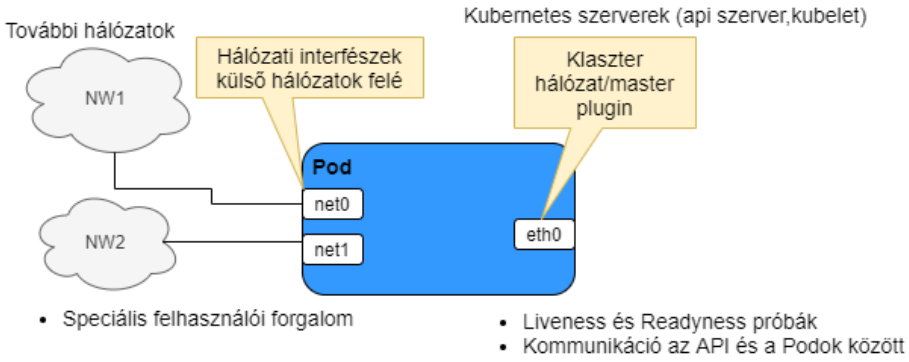
- A csomópontban található ügynöknek tudnia mindegyik poddal kommunikálni azon a csomóponton.
- Azok a podok, amik a host hálózat csomópontján találhatóak, azoknak tudniuk kell kommunikálni az összes csomópont összes podjával NAT használata nélkül.

A modell ugyan összeségében komplex, de így a Kubernetes lehetővé teszi az alkalmazások egyszerű és könnyű portolását a virtuális gépekről a konténerekre.

Az utolsó két problémára maguk a Kubernetes szolgáltatásoknak biztosítanak megoldást. A szolgáltatások és a podok közötti kapcsolat kiépítése címkék és ugynevezett selectorok segítségével történik. A selectorok szabályokat készítenek a kube-proxi komponensben, ami minden node-n megtalálható a klászteren belül, ezért hozzáférést fog tudni biztosítani bármilyen szolgáltatáshoz bármelyik node-n. A kube-proxy 3 különböző módban képes működni user space (felhasználói), iptables és IPVS módban.

3.2.1 Multus CNI

A Multus egy konténer hálózati interfész (CNI) plugin a Kubernetes számára, amely lehetővé teszi több hálózati interfész csatolását a podokhoz [9]. Jellemzően a Kubernetes-ben minden pod-nak csak egy hálózati felülete van (a visszacsatoláson kívül) - a Multus-szal létrehozhatunk egy pod-ot, amely több felülettel rendelkezik. Ez úgy valósul meg, hogy a Multus "meta-plugin" -ként működik, ami egy olyan CNI plugin, amely több más CNI plugint is meghívhat.



2. ábra Pod hálózati interfészei Multus alkalmazása esetén

3.3 Terhelés elosztás

Felhő rendszereknél a felhasználói kérések ütemezése NP-nehéz optimalizálási problémának számít [10][11]. A felhasználási igények szerint az egyes erőforrásokhoz hozzá tudunk rendelni terheléseket és az egyes erőforrások ezek alapján lehetnek alul vagy felül terheltek. Az alul- és túlterhelés különböző hibákat okozhat a rendszerben, ezek elkerülése miatt van szükség a terhelés elosztásra. A felhő rendszereknél többféle típusú terhelést kell figyelembe venni például, memória, számítási, hálózati terhelés. Először az alul vagy felül terhelt csomópontokat kell tudnunk észlelni, majd a felhő rendszer skálázhatóságának és mozgathatóságának kihasználásával tudjuk a terhelést könnyen kiegyensúlyozni. A felhő rendszerekben lévő terhelés elosztás viszont nehezen integrálható egy kliens alkalmazásba, ezért érdemes egy külső terhelés elosztót alkalmazni (ECMP, proxy) vagy meg valósítani egy terhelés elosztást az alkalmazáson belül.

3.3.1 Az Egyenlő Költségű Többutas Terjesztés

Az Egyenlő Költségű Többutas Terjesztés (Equal-cost multipath – ECMP) [12] egy hálózati útválasztó protokoll, ami kihasználja, hogy két pont a között az IP hálózatban, nemcsak egy legrövidebb út létezhet és ezeken az utakon osztja szét a forgalmat, így csökkentve az egyes utak terheltségét. Amikor továbbítunk egy csomagot az útválasztónak el tudnia dönteni, hogy mi lesz a következő állomás (next-hop). ECMP ezt metrikus számításokat és hash algoritmussokat használva álapítja meg, mert azok a next-hoppok, amik egyenlő távolságra vannak, azoknak egyenlő a metrikus értékük és a költségük a hálózatban. ECMP, tehát egy csoport routert azonosít, amiknek érvényes azonos költségű next-hopja van a cél felé. Az ECMP útvalasztó protokoll a legtöbb másik fajta útválasztó protokollal együtt használható.

Az ECMP protokollnak is vannak problémái, az egyik ilyen, hogy a használt azonos költségű utaknak nagy valószínűséggel nem ugyanaz lesz a karakterisztikája. Például az egyik útnak nagyobb a sávszélessége és a késleltetése, mint a másiknak. Ha az útválasztó nincs tisztában ezzel és mindegyik útvonalra ugyanúgy osztja szét a forgalmat például mindegyik útra ugyanannyi csomagot küld, akkor a stratégiánk nem fog jól működni. A nagyobb késleltetésű útvonalon küldött csomagok később fognak megérkezni a célba, mint a kisebb késleltetésű útvonalon küldöttek, ennek a kezeléséhez több CPU kapacitásra van szükség. Manapság az ECMP-vel már úgy nevezett flow-based csomag elosztással használják, ami azt eredményezi, hogy ugyan arról a hostról vagy subnetről érkező csomagok ugyanaz a célállomás felé ugyan azon az útvonalon kerülnek továbbításra. Így a csomag sorrend nem változik és a késleltetés által okozot problémák sincsenek.

3.3.2 IPVS

Az IPVS [13] a Linux kernelbe épített csomagszűrő és fordító keretrendszer, a netfilter része. A terhelés elosztás legnehezebb része, hogy hogyan kezeljük a forgalom mennyiségét. Az IPVS-nek három működési módja van, az egyszerű, de alacsony áteresztőképességű NAT mód, a DR mód és a komplex, de skálázható IPIP mód.

A legegyszerűbb működési mód a NAT / Masquerade mód . A terheléselosztónak egy dedikált IP-címe (VIP; vagy virtuális IP-t) van és ezen keresztül hallgatózik, ha csatlakozási kérelem vagy egy meglévő kapcsolathoz tartozó csomag érkezik, akkor a csomag cél IP-jét megváltoztatja az egyik háttérszerverének az IP-jére. A csomagot az új IP-címmel fogja továbbítani a háttérprogramba. Ennek a működési módnak a fő előnye az egyszerűsége, a hátránya pedig, hogy az összes visszaküldött csomag áthalad a terheléselosztón, így a csomagok forrás IP-címét visszaválthatja a VIP-re.

A második működési mód az átjáró (DR) [14], vagy közvetlen útválasztás vagy közvetlen válasz mód. Itt is a terheléselosztó fogadja a bejövő csomagot, de nem változtatja meg a bejövő csomag cél IP-jét. Az IP helyet a célhoz tartozó MAC címet fogja megváltoztatni, hogy megfeleljen a háttéralkalmazások egyik hálózati kártyájának. A csomag a hálózaton keresztül eljut a háttérszerverhez, aminek ugyanúgy be van konfigurálva a VIP, így eltudja fogadni a csomagokat a terheléselosztótól. A válaszcsomagok közvetlenül a helyi útválasztónak (router) leszenek küldve, a terheléselosztó érintése nélkül. Az hogy a terhelés elosztónak és a háttérprogramoknak ugyanaz a VIP van bekonfigurálva normális esetben IP konfliktushoz vezetne. Azonban ha beállítjuk, hogy a háttéralkalmazások ne hirdessék az VIP-jüket a hálózaton, akkor nem lesz probléma, mert csak a terhelés elosztó fogja a VIP-t

tartani és így minden bejövő forgalom a terhelés elosztón fog áthaladni. A mód fő előnye, hogy a terheléselosztónak kevesebb a terhelése, mivel csak a bejövő csomagokkal kell foglalkoznia, a kimenő válaszokkal nem. Ennek a módnak a hátránya a bonyolultabb beállítás, valamint az a tény, hogy csak a helyi hálózaton működik.

Az utolsó működési mód, amelyet az IPVS támogat, az IPIP [14]. Ez az üzemmód hasonlít az előző módhoz, de a cél MAC cím megváltoztatása helyett a bejövő IP csomagokat egy alagút(tunnel) segítségével juttatja el a háttérprogramnak. Előnye a módnak skálázhatóság, de problémákat okoz, ha a bejövő csomag túl nagy, és eléri az MTU-határt, mert IP széttöredezést okozhat.

3.3.3 Reverse Proxy

A proxy olyan eszköz vagy szerver, amely más eszközök nevében jár el. A proxik, olyan eszközök vagy szoftverek, amik a kliensek és a szerverek között helyezkednek el és a köztük történő kommunikációt tudják kezelni. A fordított proxi általában a webszerverek előtt helyezkedik el és a kliensek kéréseit továbbítja a webszervernek. A kért erőforrásokat megkapja a kliens, de számára úgy tűnik mintha a proxi szervertől kapta volna az erőforrásokat. A reverse proxi segítségével képesek vagyunk a kérések többféle paraméter alapján például felhasználói eszköz, hálózat állapot vagy akár az appplikáció állapota alapján történő irányítására. A reverse proxit főleg terhelés elosztó szolgálatások kialakítására használjak, mert egyenletesebb webes élményt, nagyobb biztonságot lehet vele elérni. Felhő rendszerben használva lehetővé teszi a felhő burst és split alkalmazás architektúrák használatát, amelyek gazdasági előnyt jelenthetnek, a biztonság vagy az irányítás veszélyeztetése nélkül [15].

3.4 Külső terheléselosztás Kubernetesbe telepített szolgáltatások számára

Egy Kubernetes szolgáltatás esetében a külső kéréseket egy, minden klaszter csomóponton futó kube-proxy elem fogja a megfelelő podhoz irányítani, amely terheléselosztó funkciót is megvalósít. További megoldás, hogy felsőbb rétegbeli szolgáltatás-szintű terheléselosztást valósítanak meg a Kubernetes klaszterben. Ezek a felső szinten (alkalmazás szinten) működő megoldások az úgynevezett szolgáltatási háló (service mesh) részét képezik és szolgáltatás szempontjai szerint választják ki a kiszolgáló podokat. Amennyiben egy külső terheléselosztó van a forrás (a szolgáltatáshoz kérést intéző kliens szoftver) és a szolgáltatás közé iktatva, akkor az további forgalmat generálhat, mert a külső irányítást végző logika eltérhet (úgymond „szembe megy”) a klaszteren belüli logikától.

További, és a szolgáltatások számára fontosabb következményekkel bíró probléma, hogy a külső terheléselosztás a klaszterből nem látható külső változás hatására megváltozhat egy útvonal, ezáltal megszakad egy aktív kapcsolat. Ez a probléma már nem csak a távközlési szolgáltatók, hanem más (például streaming szolgáltatás nyújtó) hagyományos számítástechnikai cégek számára is gondot jelent.

Nem véletlen, hogy az elmúlt néhány évben több, jelentős piaci súllyal rendelkező cég által támogatott megoldási javaslat született. A támogatók piaci súlya és a javaslatok közösségbeli híre alapján a Google's Maglev, Github's GLB, Microsoft's Ananta, Facebook's Shiv és a Yahoo's L3DSR megoldásait emelnénk ki [16][17][18]. A megoldások célja, hogy már a klaszteren kívűl megbízható hálózati terheléselosztást biztosítsanak.

Egy alternatív utat követ a MetalLB [19], amelyik a Kubernetes és fizikai hálózati eszközök együttműködését próbálja összehangolni, a Kubernetes klaszter hálózati megoldását ráépítve a hardver alapú terheléselosztókra. A megoldás során a MetalLB veszi át a külső IP címek kiosztásának feladatát és biztosítja a hálózaton történő hirdetését. A megoldás úgy a 2. rétegben, mint – a Border Gateway Protocol routing mechanizmus segítségével – a 3. réteg szintjén működik.

4. A hálózati mechanizmusok negatív hatása

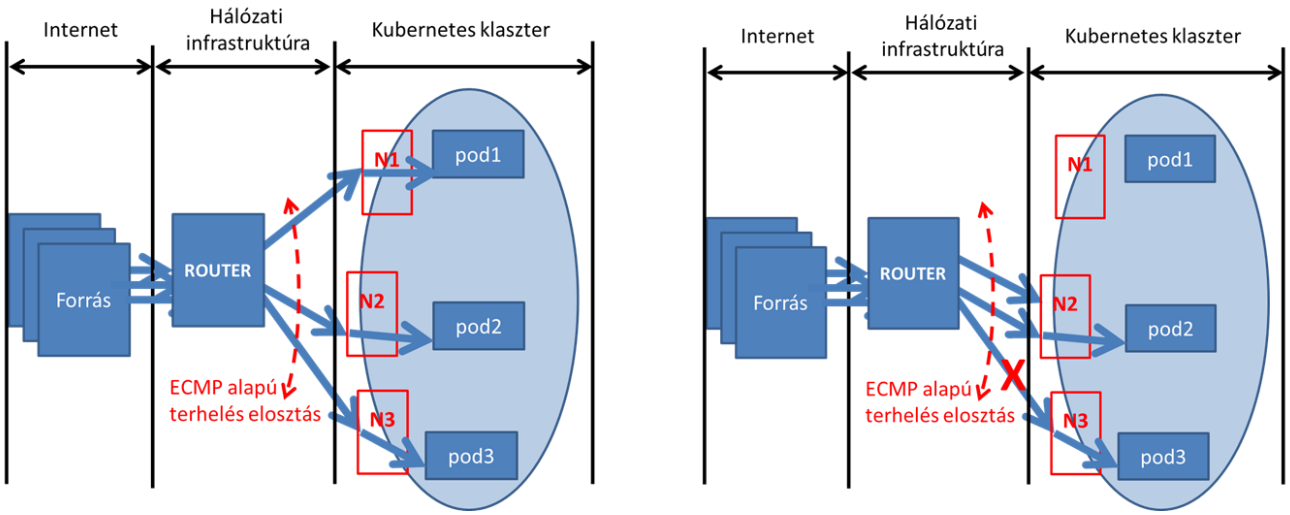
Az előző fejezet végén bemutatott külső hálózati terheléselosztási mechanizmusok korlátai képezik dolgozatunk témáját. Amint a bevezető fejezetben felvezettük, a távközlési szolgáltatások esetében különösen fontos a folytonos, minőségbiztosított (QoS) kapcsolat. Az alacsony késleltetés nemcsak hatékony hálózati továbbítást követel meg, hanem a kapcsolat megszakítása után az új kapcsolat felépítésére szükséges többletkésleltetést sem tolerálja. Ugyanakkor az igazi gondot nem a klasszikus QoS paraméter értékek romlása, hanem a kapcsolat megszakítása jelenti. Ebben a fejezetben bemutatjuk a probléma hátterét, részletesen kifejtjük a hálózati hatást, fizikai hálózati eszközök felhasználásával előállítjuk azt és mérések segítségével számszerűsítjük azt.

A problémafelvetés egy ipari kutatási projekt kapcsán merült fel. A távközlési szektorban dolgozó ipari partner hívta fel figyelmünket erre a sajátos problémára, amellyel a korábbi felhő megoldásokról a Kubernetes klaszterekre való áttérés során, annak sajátos (például a podok kezelését, a hálózatkezelési rendszerét érintő) architektúrájából szembesültek. Napjainkban a távközlési szektor egyik legfontosabb ága, úgy a piaci-gazdasági súlya, mint az innovatív technológiák alkalmazása miatt, az ötödik generációs mobil kommunikációs rendszer, közismertebb nevén az 5G. Habár a korábbi, 4G rendszerek érettebb változataiban is hangsúlyos szerepet kaptak a virtualizációs megoldások, továbbá az első 5G rendszerek architektúráját már felhő alapú rendszerekbe tervezték, a mostanában kidolgozás alatt levő 5G architektúrák azok, amelyeket teljesen felhő natívnak lehet nevezni és – a korábbi felhő rendszerek helyett- már Kubernetes klaszterbe telepítik a virtualizált funkciókat [20]. Egy ilyen környezetben az évtizedek során megszokott architektúrákat újra kell értelmezni. Ha csak a hálózat kezelést vizsgáljuk, akkor az alábbi szempontok mind fontos eltérést jelentenek a korábbi, megszokott módszerektől: a podokban alapértelmezés szerint már csak egy hálózati interfész áll rendelkezésre, a teljes klasztert egy közös overlay (átfedő) hálózat szolgálja ki, a szolgáltatás háló (service mesh) jellemzően alkalamzási rétegbeli forgalom kezelést biztosít, stb. Ezt egészíti ki az az általános szervezési elv, hogy a korábban megszokott operátori környezettől eltérően elvárás, hogy az új rendszer már egy külső virtualizált infrastruktúrában fusson, már egyes Kubernetes konfigurációs opciókat sem tud a telepített rendszer befolyásolni (nem csak a hálózati beállításokat), azokat külső adottságnak kell elfogadnia. A mi munkánkat a hálózati dinamikából eredő hatások motiválták, de ez is egy tág problémahalmazt jelent.

Amint a korábbi fejezetekben már körvonalazódott, mi a külső hálózati terhelés elosztó mechanizmus negatív hatásait vizsgáljuk. A továbbiakban részletesebben feltárjuk ebben a fejezetben, hogy pontosan miként tud váratlan hatást okozni ez a hálózati elrendezés. Azért foglalkoztunk a témával, mert meglepőnek tűnt, hogy egy látszólag kipróbált, a hálózati megoldások terén kiforrottnak tekinthető megoldás is olyan szintű problémát jelent a távközlési szolgáltatások számára, amely – megfelelő ellen intézkedések hiányában – a szolgáltatás ellehetetlenülésével fenyeget. A távközlési gyártók a rendszereikre jellemző komplexebb, saját rendszerük architektúráját követő megoldással el tudják kerülni ezt a csapdát, de mi abban láttuk a kihívást, hogy egy olyan megoldást nyújtsunk, ami lehetőleg önállóan működőképes és csak nyílt, nyilvánosan elérhető technológiákat használ. Ez a választás lehetővé teszi, hogy nem csak távközlési szolgáltatások, hanem általános célú web szolgáltatások is alkalmazhassák.

4.1 Az ECMP probléma

Napjainkban a sok nagyteljesítményű routerben ECMP alapú terhelés elosztást alkalmaznak, de a megvalósítás során azok állapot-mentesek (stateless). Emiatt, ha a beállított alternatív útvonalak egyike érvénytelen lesz, az ECMP algoritmus újra oszthatja a forgalmat a még aktív kimenő linkek közt, ami aktív kapcsolatok megszakítását eredményezi. Amennyiben ez a router egy Kubernetes klaszterbe továbbította a forgalmat, a Kubernetes rendszerben futó pod hiába lenne képes a változást lekövetni, a váratlanul „eltérített” kapcsolat csomagjai el lesznek dobva.



3. ábra Az ECMP probléma szemléltetése

A 3. ábra szemlélteti ezt a helyzetet, amelyen egy szolgáltatást illusztrálunk. Különböző források az Interneten keresztül tartják fent a kapcsolataikat a Kubernetes klaszterbeli podokban futó alkalmazásokkal. Jelen ábrán a klaszter három worker csomópontból (N1, N2 és N3) áll, és az ovális azt jelképezi, hogy az erőforrásaikon üzemel a klaszter. Jelenleg három pod példány (pod1, pod2 és pod3) szolgálja ki a kéréseket. A forgalmat egy olyan külső hálózati router osztja szét a klaszter elemei közt, amely nem tagja a klaszternek, egy független infrastruktúra szolgáltató üzemelteti. A router ECMP alapú forgalom elosztást végez, amely, amíg a három csomópont felé linket aktívnak látja (az ábra baloldalán), az elvárásoknak megfelelően osztja szét a kapcsolatok csomagjait. Amennyiben a hálózatbeli esemény során egy link megszakad (például az N3 csomópont felé az ábra jobboldali fele), akkor elvileg a másik két irányban változatlan módon aktívnak kellene maradjon a korábban is élő kapcsolat. Ugyanakkor, amint az ábrán is jelöltük, az ECMP algoritmus egy hálózati változás hatására újra osztja a forgalmat, és előfordulhat, hogy a korábban az N1 felé továbbított kapcsolat most már az N2 csomóponthoz fog érkezni, pedig az N1-hez vezető link nem volt érintett a hálózati változásban (hibában).

Tehát a Kubernetes felhőben futó alkalmazás (N1-en futó pod1) számára egy semleges hálózati változás (a router és N3 közti linkszakadás) olyan hatással van, amely megszakítja a szolgáltatást. A dolgozatunkban ezt a konkrét negatív hálózati hatást vizsgáljuk.

4.2 ECMP probléma mérés alapú visszaigazolása

Mielőtt rátértünk volna a javaslatunk kidolgozására, szerettük volna visszaigazolni ennek a negatív hatásnak a létezését, ezért egy teszthálózatot terveztünk, amelyben valós ipari környezetben is használt fizikai routereket felhasználva, mérések segítségével ellenőrizhettük az ECMP problémát.

A teszthálózatunk a 3. ábrán látható elrendezést követte (a Kubernetes master csomópont nincs ábrázolva). A Kubernetes klaszter négy darab, második generációs intel i5 processzorral és 8 GB RAM-mal, valamint Linux Ubuntu 18.04 LTS operációs rendszerrel rendelkező asztali számítógépekből valósítottuk meg. Mindegyik PC-nek két hálózati interfész kártyája volt, de az egyiket csak a menedzsment forgalom számára tartottuk fel. A Kubernetes klaszter hálózata WeaveNet volt, és a PC-k 1 Gbps Ethernet hálózati interfészeit használták. A forrást egy Linux Ubuntu 18.04 LTS asztali PC helyettesítette.

A fogadó oldalon egy három példányban futó podot indítottunk. A podban egy HTTP szerver futott, amelyet a Google Kubernetes-t tanító segédanyagaiban *hello* szerverként

használ és annyit módosítottunk rajta, hogy egy HTTP kérésre válaszként megadja a fogadott kérés IP címét, port számát, illetve a pod saját IP címét és a node nevét, amelyen fut. A gyors tesztelés érdekében a három podot (mindegyik node-on egy-egy) egy közös Kubernetes service-be szerveztük, ezt a klaszter címre beállítottuk a forrás és a router routing tábláját is, így a service IP címére be tudtuk küldeni a kérést a klaszterbe. A forrásként használt PC-n a küldő egy bash szkript segítségével több IP címet generálva, a curl segédprogrammal küldtük ki a HTTP kéréseket. A routeren a router által lehetővé tett ECMP alapú terhelés elosztást állítottuk be, a három csomópont felé mutató három linkre. A tesztek menete a következő volt. Először mind a három link aktív volt, kiküldtük a kéréseket és feljegyeztük, hogy melyik forrás címet melyik csomópont felé routolta a router. A második lépésben megszakítottuk (kihúztuk a portból) az N3 felé vezető linket, újra feljegyeztük, hogy melyik forrás címet melyik csomópont felé továbbította a router. Végül az N3 felé vezető linket újra aktívvá tettük, é újra feljegyeztük, hogy melyik forrás címet melyik csomópont felé továbbította a router. Minden esetben, lépésenként összesen 250 kérést küldtünk ki, mindegyik fenntartotta a TCP kapcsolatot a kísérlet teljes időtartama alatt (kivéve, ha megszakadt a vizsgált hatás miatt).

A fenti tesztet három routerrel végeztük el: egy gazdag funkcióhalmazzal rendelkező Juniper SRX1500 routerrel [21], egy kisebb adatközpontba tervezett Mikrotik CCR1036-12G-4S routerrel [22], valamint egy Linux routerrel. A Linux esetében a 4.11-es kernelverziótól kezdve a *multipath* opció alapesetben ECMP-vel osztja szét a forgalmat a több alternatív next-hop közt, a fizikai routerek esetében a gyártó által biztosított legfrisebb firmware-ben szereplő funkciót használtuk.

1. táblázat – ECMP probléma mérés alapú igazolása

ECMP router típusa	**Az 1. lépésben az N1 kapcsolatainak száma**	**A 2. lépésben az N1-ről N2-re átirányított kapcsolatok aránya [%]**	**A 2. lépésben az N2-ről N1-re átirányított kapcsolatok aránya [%]**
Mikrotik	84	50	50
Juniper	90	43.2	48.5
Linux	85	24.7	0

A tesztjeink alapján mindhárom fenti router ECMP megvalósítása az elvárt módon elosztja a forgalmat a három csomópont felé, de a hálózati hiba hatására átrendezi a

kapcsolatokat. Az 1. táblázat 3. oszlopában azt mutatjuk, hogy hány, eredetileg az N1 felé továbbított kapcsolat kerül át az N2-re az N3 linkszakadás következményeként. Az 1. táblázat 4. oszlopában pedig azt mutatjuk, hogy hány, eredetileg az N2 felé továbbított kapcsolat kerül át az N1-re az N3 linkszakadás következményeként. Ennek az a konkrét oka, hogy minden egyes fizikai csomóponton (N1, N2 vagy N3) az aktív kapcsolatokat a Linux kernel kapcsolat követő (connection tracking – conntrack) mechanizmusa nyilvántartja és ha olyan TCP kapcsolat csomagjait észleli, amelynek a felépítéséről nem tud, eldobja azokat. Látható, hogy majdnem a kapcsolatok fele érintett ebben az elrendezésben. Amennyiben több alternatív útvonal lenne, valószínű, hogy egy linkhiba kevesebb változást okozna, de több kapcsolat akkor is érintve lenne (lásd a Linux ECMP megvalósítás hashing stratégiáját magyarázatként [22]).

Tehát mérésekkel sikerült visszaigazolni, hogy a távközlési területen dolgozó kollégák által jelzett ECMP probléma létezik és nem elszigetelt eset. Ebből kiindulva a következő fejezetben egy megoldást javasolunk a negatív hatás (ECMP kapcsolat átrendeződés miatti szolgáltatás megszakadás) kivédésére.

5. A negatív hálózati hatás kiküszöbölése

5.1 A javaslatunk peremfeltételei

A korábbi fejezetben részletesen ismertetett ECMP hatás elhárítására olyan javaslatot próbáltunk adni, amelyhez elég a Kubernetes klaszteren belül módosításokat végrehajtani. Ugyanis egy operátori környezetben, valós üzleti mobil hálózatokban a gyártó által fejlesztett virtuális funkciók, a szolgáltatásokat biztosító podok egy olyan Kubernetes klaszterbe vannak telepítve, amelynek az adminisztratív felügyeletére már nincs hozzáférése a gyártónak. Ez egy éles váltás a korábban, évtizedekig megszokott felálláshoz. Még a felhő alapú 4G rendszerek esetében is a felhőt (telco cloud) a gyártó telepítette. Most azonban már a gyártó csak elvárásokat támaszthat a telepítés előtt (például hálózat kapacitása, processzor teljesítménye, stb.), de úgy a hálózati beállítások, mint a Kubernetes worker node-ok (csomópontok) konfigurációja kívül esik a hatáskörén. Ez a korlátozás nem jelent teljes „lebutítást”, mert a telepítendő podok kiemelt (privileged) hozzáférést kaphatnak, továbbá lehetséges új pod interfészek hozzáadása (például a korábban már tárgyalt multus segítségével), de ezek nem módosítják érdemben az infrastruktúrát. Praktikusan a javaslatunk csak olyan eszköztárral dolgozhat, amely a podok, szolgáltatások és egyéb Kubernetes elemek deklarációs fájljaival létrehozható (ezeket a Kubernetes közösség gyakran *yaml fájloknak* hívja, mert a YAML leírók szintakszisa szerint kell megírni), valamint a podokban futó konténerekbe telepíthető.

Kiindulásként fontos leszögezni, hogy megoldásunkban nem támaszkodhatunk a széles körben elterjedt és a hagyományos web szolgáltatások számára megfelelő szolgáltatási hálóra (service mesh). Ezek a megoldások ugyanis felső rétegbeli forgalomra vannak kidolgozva, de nagyon sok távközlési szolgáltatás alacsonyabb (3. vagy 4. rétegbeli) kapcsolatokat feltételez. Mivel a Kubernetest kiegészítő technológiai ökoszisztéma nagyon változatos és dinamikus, nem zárható ki, hogy egy új megoldás nem fog megjelenni, de jelenleg főleg az istio és envoy, esetleg linkerd megoldások erre a célra nem alkalmasak. Jellemző példa a gazdag terhelés elosztó funkciókkal rendelkező Ambassador, amelynek a logikája (az LB stratégiája) ígéretes, de sajnos az is csak a felső rétegbeli (7. rétegbeli) forgalmat kezelő envoy-ra épül.

5.2 A FrontEnd pod szerepe

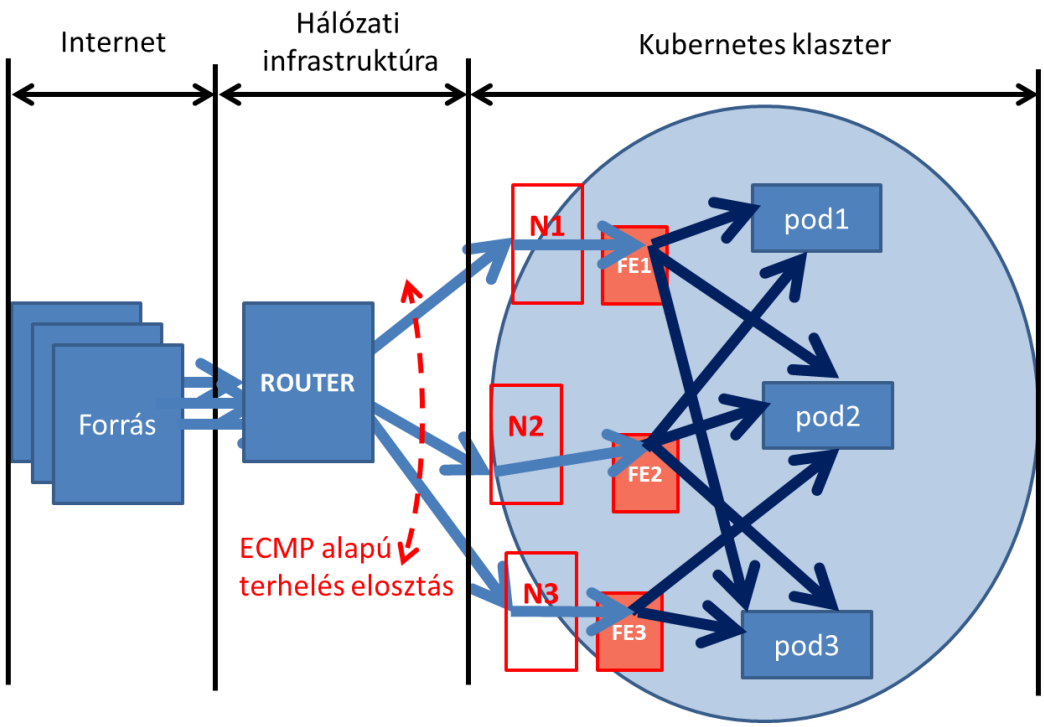
Mivel a célunk az volt, hogy leválasszuk az infrastruktúra oldaláról, kívülről érkező hatásokat a tulajdonképpeni kiszolgáló működését, a kézenfekvő, először a gyakran alkalmazott proxy megoldást próbáltuk alkalmazni. A proxy megoldás lényege, hogy egy, a külső változásokat követni tudó logikával rendelkező egység kezeli a forgalmat, a szolgáltatást képviselve terminálja a kapcsolatot, majd a fogadott hasznos adatot a szolgáltatás logikája és elvárásai szerinti kapcsolatban továbbítja a szolgáltatást megvalósító podnak. Mivel a szolgáltatást megvalósító pod, a külső szereplők számára a proxy „háta mögött végződtetik" a kéréseket. ezért ezeket gyakran BackEnd-ként hivatkozzák.

Számunkra ez azért nem volt követhető, mert alapesetben a szolgáltatást megvalósító logika saját információit a fejlécbe is beleírja (pl. a szolgáltatás meghirdetett IP címe), vagy titkosítás miatt nehézkes a proxy konfigurációja. Megjegyezzük, hogy a fentebb hivatkozott envoy alapú működés is a proxy elvét követi.

Annak ellenére, hogy a proxy által végződtetett kapcsolatra épülő megoldást el kellett vetnünk, azt az elvi gondolatot követtük, hogy egy elem „levédje" a külső hálózati hatásokat („frontolja"), ezért azt néztük meg, hogy egy FrontEnd-nek nevezett pod beiktatásával meg tudjuk-e oldani a feladatunkat.

Mivel a kapcsolatot nem akarjuk megszakítani sem a proxy-nál, sem a FrontEnd-nél, ezért azt vagy 2. vagy 3. rétegben kell továbbítanunk. A második rétegbeli továbbítás előnyeit egy virtuális kapcsolóval (virtual switch) lehetett volna kiaknázni, de – amint később kifejtjük – a FrontEndnek fejlettebb logikai funkciót is szántunk. Továbbá a peremfeltételek miatt egyszerűbb és általánosabb megoldásnak tűnt egy podban megvalósítani a FrontEnd funkciókat. A podban pedig routing során találtuk egyszerűbbnek a számunkra fontos kernel mechanizmusokat kiaknázni, ezért végül 3 rétegbeli megoldást választottunk.

A fenti architekturális döntések után kialakult a 4. ábrán bemutatott elrendezés, valamint a FrontEnddel (FE) szembeni elvárás lista. A külső kapcsolatok egy közös, a szolgátatást azonosító IP címre (virtuális IP – VIP) küldik a kéréseiket, amely megszakítás nélkül el kell érjék az egyik BackEnd (BE) podot. A 4. ábrán a nyilak vége nem kapcsolat terminálást, hanem elvi csomagtovábbítási döntést jelölnek. A kérés bármelyik BackEnd podhoz bejuthat, de egy kapcsolat élettartama alatt a kapcsoalt csomagjai végig ugyanahhoz a podhoz kell érkezzenek. Tehát a router negatív ECMP hatása miatt megváltozott útvonal ellenére a kapcsolat aktív kell maradjon.



4. ábra A FrontEnd alapú megoldás

5.3 A FrontEnd pod kialakítása

A FrontEnd podban kell tehát megoldani, hogy a kapcsolat konzisztens maradjon. Amint láttuk, az ECMP perzisztens, tehát statikus hálózati feltételek mellett az egymás után következő csomagok adott kapcsolaton belül ugyanahhoz a BackEndhez lesznek továbbítva. Ugyanakkor az ECMP nem konzisztens, mert hálózati dinamika hatására megváltoztatja a kapcsolat útvonalát. Erre sok válasz született a szakirodalomban, de a széleskörű szolgáltatói konszenzus, hogy a legjobb megoldás a Google által kidolgozott Maglev hashing algoritmus. Például a korábban hivatkozott az Ambassador-ra 7. rétegbeli megoldás is es használja, de értelemszerűen a Google szolgáltatások, vagy elosztott alkalmazás cache szolgáltatók is használják. Ennek több előnye is van. Egyrészt, ez egy érett, alaposan tesztelt megoldás. Másrészt, már a legújabb Linux kernelek (a 4.18-as változattól kezdve) is megvalósítják azt.

Mi a Magelev alapú megoldást a szakirodalom tanulmányozása során ismertük meg, mint egy potenciális válasz a perzisztens hashing kérdésre. Majd az alternatívák ismeretében a távközlési iparban dolgozó rendszermérnökökkel történt egyeztetések, valamint az iparági

beszámolók [23][24][25] alapján visszaigazoltnak láttuk sejtésünket, hogy a Maglev alapú megoldás a jó választás.

A Linux kernelben az IPVS mechanizmus egyik opciója Maglev alapú továbbítást valósít meg. Teszteltük a Linux Maglev implementáció perzisztenciáját (lásd később, illetve a D. Függeléket) és megfelelően működött. Megjegyezzük, hogy a Maglev is hibázhat (nem 100%-osan perzisztens), de a távközlési alkalmazásokkal szemben támasztott elvárásoknak megfelel.

Mivel a Linux kernelben az IPVS megvalósította a Maglevet (de például az alapértelmezett multipath megoldás nem), a FrontEnd-ben IPVS alapú továbbításra esett a választás. Az IPVS lehetővé teszi a címfordításos (masqueade) továbbítást, de mi pont ezt akarjuk elkerülni, így a másik megoldást („direct forwarding”) választottuk. Ez az IPVS megoldás feltételezi, hogy egy helyi (a gazdagépen) levő IP címre jönnek a csomagok, különben nem alkalmazza a mechanizmusokat, emiatt egy hamis („dummy”) interfészt kellett minden FrontEndben létrehozmi a VIP címével. Az IPVS szabályt persze mi úgy állítottuk be, hogy a csomagot majd egy másik interfészen (a BackEnd felé) továbbítjuk, de ezt az IPVS rendszer már elfogadta.

5.4 A FrontEnd és BakcEnd podok hálózati interfészei

Eddig nem említettük, de a FrontEnd megoldásunk esetében különösen fontos volt a hálózati interfészek megfelelő kezelése, mert a Kubernetes hálózatkezelési és kapcsolat nyilvántartása nincs felkészítve több pod közti kommunikáció ilyen szintű felhasználó általi „eltérítésére”. Továbbá a FrontEnd több interfészes konfigurációja is meghaladja a Kubernetes által kezelt hálózati interfész rendszer (Container Networking Interface – CNI) képességeit. Vizsgálataink során úgy a Flannel-t, mint a WeaveNet-et használtuk a teszt klasztereinkben, a 4. fejezet tesztjeit például Flannel hálózaton végeztük, a 6. fejezetben bemutatott tesztek pedig egy WaveLan hálózatban vannak kimérve. De a tapasztalataink egybevágnak a szakirodalom megállapításaival, a CNI megoldás típusa nem befolyásolja ezt a jelenséget.

Tehát szükség volt a multus megoldásra, hogy a FrontEnd megfelelően konfigurálva legyen. A FrontEnd „bemeneti” és „kimeneti” interfészét egyránt a multus segítségével konfiguráltuk, mindkettő a már ismeretett macvlan típusú interfész lett. Ez lehetővé tette a két interfész hálózati forgalmának szeparációját. A két macvlan interfész (továbbiakban net1 és net2) ugynarra a fizikai interfészre lett rákötve, mint amelyik a router felé biztosítja a

kapcsolatot. Így már egyértelmű, hogy a net1 interfész alkalmas arra, hogy az ECMP router számára next-hopként jelenjen meg (azaz a 4. ábrán a fizikai node-ok és a FE-k közt a csomag nem routolva lesz).

A net2 interfész a BackEnd felé továbbítja a forgalmat. Ugyanakkor a BackEnd pod interfésze is ebbe a tartományba kell essen, ezért az is multus macvlan interfészt kapott. Ezáltal a ForntEnd és BackEnd kapcsolat elkerüli a Kubernetes CNI hálózat menedzsmentet, ami rugalmasságot biztosít számunkra.

Ami a BackEnd podot illeti, a kapcsolatok zökkenőmentes kezelése érdekében azon is létre kellett hoznunk egy dummy interfészt ugyanazzal a VIP címmel. Egyrészt a beérkező csomagokat így feladta a kernel a 7. rétegnek, másrészt a kimenő válaszcsomagokra is így a VIP cím került forrás címnek (source IP). Ez utóbbi a távközlési protokollok számára fontos.

A válasz csomagok ugyanazon a net2 interfészen távoznak, ahol beérkeznek a kérések. Azért, hogy a router és a FrontEnd ne zavarj össze a klaszterbe irányuló, valamint a klaszterből távozó forgalmat, a két interfészt 2. rétegben elválaszotottuk, külön-külön VLAN címkét rendelve hozzájuk (100-as VLAN a net1-hez, 200-as VLAN a net2-höz).

Látható, hogy a hálózati beállítások véglegesítése után minden FrontEnd és BackEnd podon van egy ugyanolyan IP című hálózati interfész (a VIP címmel rendelkező dummy interfészek). Ez összezavarhatja a címfelderítést, ezért külön paranccsal letiltottuk az erre a címre vonatkozó ARP forgalmat, az erre a címre vonatkozó ARP táblákat pedig szkriptben konfiguráltuk minden podunkban.

Ezt az elrendezést az 5.6 alfejezetben bemutatott prototípus megvalósítás kapcsán szemléletesen is bemtutatjuk.

5.5 A FrontEnd podok kapcsolatkezelésének szinkronizálása

Az eddig leírt megoldás képes a VIP címre kapcsolatokat kiépíteni, de még egy lépés hiányzik ahhoz, hogy elfedje a negatív hálózati hatásokat. Ugyanis az eddigi leírás nem biztosítja a FrontEnd-ek közti állapot szinkront, ennek hiányában pedig egy útvonal váltásnál pont ugyanaz a hiba fog jelentkezni, mint amit a fizikai csomópontok (N1, N2 és N3) esetében leírtuk a 4.2 alfejezetben. Azaz hiába képes a Maglev alapú kapcsolat perzisztens hashing műveletre, ha a FrontEnd pod kerneljében megszakítják a csomag továbbítást. A mi esetünkben tovább bonyolítja a helyzetet, hogy a kernel szintű kapcsolati tábla (conntrack) mellett az IPVS mechanizmus egy saját táblát is fenntart. Ebbe csak azok a kapcsolatok kerülnek bele, amelyek helyben végződnek. Ebben az esetben a dummy interfész alkalmas

konfigurációjával mi már sikeresen biztosítottuk, hogy a számunkra érdekes kapcsolatok ebbe, az IPVS-hez rendelt conntrack táblába kerüljenek.

Tehát a megoldásunk akkor lesz teljes, ha az IPVS-hez rendelt connntrack táblát szinkronizálni tudjuk a FrontEnd példányok közt. Fontos megjegyezni, hogy amennyiben több (külön-külön VIP címmel rendelkező) szolgáltatást telepítünk ugyanabba a Kubernetes klaszterbe, akkor mindegyik számára külön-külön FrontEnd példányokat kell létrehozni, és a szinkronizálást csak az ugyanahhoz a szolgáltatáshoz tartozó FrontEnd-ek közt kell elvégezni.

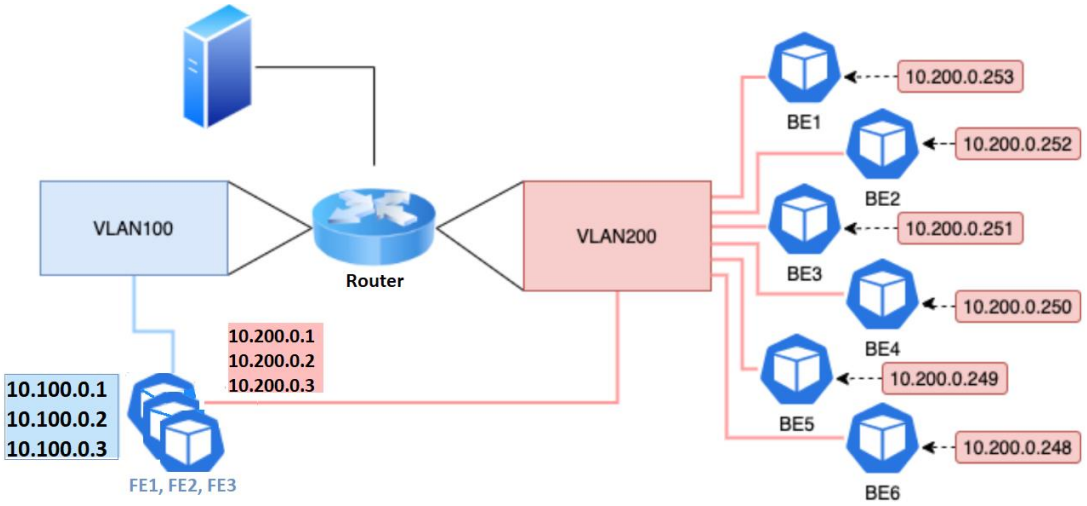
Szerencsére az IPVS mechanizmus egy érett, gazdag funkcionalitású megvalósítással rendelkezik, többek közt egy alapos szinkronizálási folyamattal is. Ez úgy működik, hogy egy szerverben (esetünkben egy FrontEnd podban) aktív IPVS-t master-nek kell konfigurálni, a többit pedig slave-nek, ezután pedig multicast kapcsolat segítségével folyamatosan lekövetik a slave-k a master állapotát. Az IPVS lehetővé teszi, hogy egy IPVS példány egyszerre legyen master és slave, ezáltal minden FrontEnd master-ként szolgál a többi számára azokra a kapcsolatokra, amelyek nála épültek fel.

Ezt a mechanizmust teszteltük, és ennek a tesztnek az időtartamára a router beállítását megváltoztattuk, hogy két FrontEnd-en (fe1 és fe2) keresztül érjük el ugyanazt az egy BackEnd pod-ot (be4). A forrás folyamatosan TCP kéréseket küldött a be4-re, a kapcsolat először az fe1-n keresztül jött létre. A kísérlet közben a fe1 felé vezető linket megszakítottuk, így a routernek a fe2-felé kellett küldenie a csomagokat és a be4 sikeresen tudott válaszolni (a szolgáltatás nem szakadt meg). A conntrack szinkronizációs folyamat lépéseiről készített képernyőmentéseket a B. Függelékben ismertetjük.

5.6 A javaslatunk alapján készített podok áttekintése

Megoldásunk értékelése céljából a 4.2 alfejezetben leírt tesztrendszeren felül a Wigner adatközpontban (Wigner DC) öt Quanta S810-X52L szerveren is megvalósítottuk a teszthálózatot. Ezek a szerverek már megfelelően nagy teljesítményűek (intel E5-2600 processzor, 256 GB RAM, 10 Gbps hálózati interfész), hogy egy valós távközlési szolgáltatónál elérhető infrastruktúrához hasonló tesztelési környezetet nyújtsanak számunkra.

A szerverekből a 4.2 alfejezetben már leírt hálózatot valósítottuk meg. Négy szerverre Kubernetes klasztert telepítettünk, tehát három worker node-unk volt. Az egyik szerver a forgalmat generálta. A hatodik szerver pedig a Linux multipath megoldással biztosította az ECMP router funkciókat. A tesztrendszerben 3 FrontEnd podot (FE1, FE2 és FE3), valamint 6 BackEnd podot (BE1-BE6) indítottunk. Az így létrehozott rendszert az 5. ábra szemlélteti.



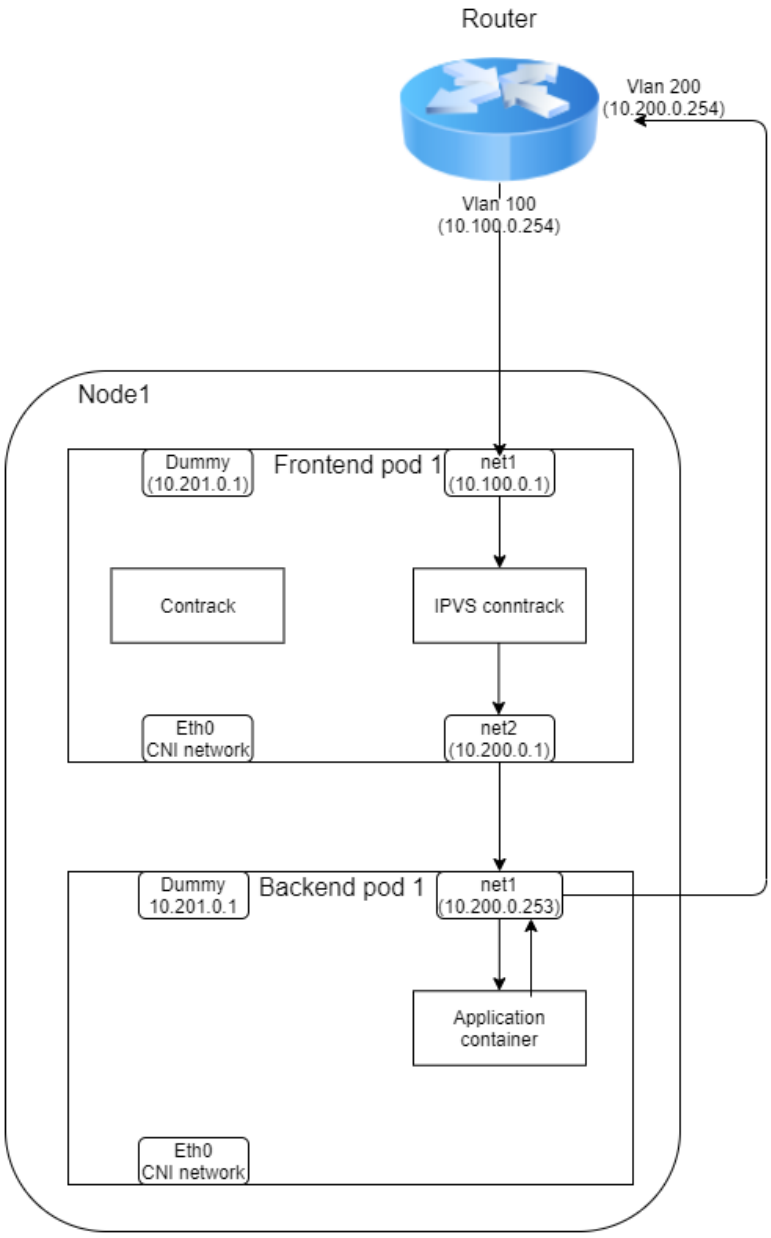
5. ábra – A Wigner DC-ben megvalósított tesztrendszer

A tesztrendszerbe telepített FrontEnd podokat létrehozó YAML leíró fájlt a C Függelékben, míg a BackEnd podokat létrehozó YAML leíró fájlt a D Függelékben is megtekinthető. A router routing tábláját kilistáztuk és az E Függelékben megtekinthető. A két pod egymáshoz való viszonyát és a hálózati kapcsolataikat a 6. ábrán illusztráltuk és tulajdonképpen az 5. ábra router-ének, valamint FE1 FrontEndjének és BE1 BackEndjének a részletesebb bemutatása.

Minden worker node-n és a routeren szükség volt 2 VLAN felkonfigurálására (VLAN 100, VLAN 200). A 100-as VLAN-on a kérések és a bejövő forgalom közlekedik, a 200-as VLAN-on pedig a válaszok és a kimenő forgalom. A beállításokat a Netplan konfigurációs fájlának módosításával végeztük el minden gépen. A routerről érkező forgalmat a FE1 osztja tovább a BE1 podnak, azon fut az a szoftver, amely a kívánt szolgáltatást megvalósítja. A szolgáltatás Virtuális IP (VIP) címe, amelyet a külső szerverről a kliensprogramok meg kell címezzenek a 10.201.0.1. Amint korábban már részleteztük, nem a Kubernetes CNI által biztosított interfészt használtuk, ezeket az ábrán az Eth0 interfészekkel jelöltük és sem az FE1-n, sem a BE1-en nincs szerepük a forgalom továbbításban.

Az általunk használt interfészeket a multus segítségével hoztuk létre (lásd a C és D Függeléket). A FrontEndre a net1 interfészen érkezik be a csomag, majd az IPVS mechanizmus továbbítja a BackEndhez. Az IPVS működéshez szükség volt, hogy minden podhoz hozzá legyen adva a VIP címmel konfigurált dummy interfész. Ezzel azt is elérjük, hogy az IPVS conntrack tábláját használjuk (az FE1 kernelbeli conntrack helyett). A FE1 pod IPVS conntrack tábláját szinkronizálni tudjuk a többi node-n futó FrontEnd podok IPVS

táblájával, így egy frontend kiesésekor a másik kettő át tudja venni a forgalmat a kiesett FE1-től. A BE1 a választ a 200-as VLAN-on közvetlenül küldi vissza a routernek.



6. ábra – A megvalósított FrontEnd és BackEnd podok hálózati interfészei

6. A javaslatunk teljesítményének mérés alapú értékelése

6.1 A javaslatunk funkcionális tesztelése

Az 5. fejezetben bemutatott javaslatunkat megvalósítottuk és telepítettük a 4.2 alfejezetben bemutatott klaszterünkbe. A tesztekhez mindegyik worker csomópontra egy-egy FrontEnd podot, és két-két BackEnd podot telepítettünk. Egy külső forrás PC-ről folyamatosan, 200 különböző forrás IP címmel kérdeztük le a VIP címet. A teszt során megszakítottuk, majd újra visszaállítottuk a fe3-ra mutató linket. A kapcsolat folyamatos volt, tehát a szolgáltatás nem szakadt meg.

A teszt során megmértük a rendszer teljesítményét is. Minden egyes címről egyszerre 5 párhuzamos szálon, forrás IP címenként összesen 500 kérést küldtünk. A teljesítmény teszt során mind a három link a router és a három FrontEnd között aktív volt.

Egy első kísérletben mérést először csak egy BackEnd pod címére küldtük a kéréseket, ekkor a BackEnd multus macvlan interfészén keresztül a routerről direkt a BackEnd podhoz érkeztek a kérések, elkerülve a FrontEnd-et. Ez a mérés viszonyítási alapot szolgáltatott számunkra. A medián válaszidő 8 ms volt, ami egy jó eredmény, de az átlag késleltetés már 25.8 ms lett. Mivel a kérések torlódtak a BackEnd podban, ezért nagy volt a szórás értéke: 104.7 ms.

A második kísérletben a teljes megoldásunkat használtuk, az előző kísérletben használt kérés mennyiséget megtartottuk, de a cél cím a BackEnd pod címe helyett a VIP cím volt. Az előző kísérlettel szemben a második kísérlet során - figyelve a forgalmat - azt láttuk, hogy a megoldásunk sikeresen szét terítette a kéréseket a hat BackEnd pod közt. A mért medián késleltetési érték 5 ms-re csökkent, az átlagos válaszidő pedig kevesebb, mint a felére (10.2 ms-re) csökkent. A szórás is ennek megfelelően nagyon lecsökkent, 17.2 ms-ra. Ezek a mért értékek is azt bizonyítják, hogy sikeresen működik a terhelés elosztás, a torlódás csökken, ennek hatásaként a később érkező és sorba állított kérések száma sokkal kevesebb, emiatt a kiugró kiszolgálási idők értéke is sokkal alacsonyabb.

Összefoglalva, a funkcionális tesztek igazolták a megoldásunk működőképességét és a ezeket a teszteket kísérő méréseink is megerősítették ezt az eredményt.

.

6.2 Az IPVS alapú FrontEnd teljesítményének összehasonlítása az alternatív kiszolgálási lehetőségekkel

Habár az előző alfejezetben már bemutattuk a megoldásunk előnyeit, ebben az alfejezetben egy részletesebb teljesítmény elemzéssel elemeztük a megoldásunk során a FrontEnd pod teljesítményét. Arra keressük a választ, hogy ha eltekintenénk az eredeti feladatunktól és csak az IPVS alapú, FrontEnddel növelt ugrásszámú megoldást használjuk, ez mekkora többlet késleltetést (teljesítmény romlást) jelent a két kézenfekvő alternatívához képest. Az egyik ilyen alterntíva a Kubernetes CNI interfész használata (a mi esetünkben WeaveNet). A másik alternatíva a multus macvlan interfész alkalmazása. Tehát három szcenáriót hasonlítottunk össze. Mindhárom esetben egy általános célú HTTP teszter segédprogramot, a hey-t használtuk forgalom generálásra [26]. Mivel nem az ECMP hatását vagy külső hálózat teljesítményét vizsgáltuk, ezeket a kísérleteket mind az 5. ábrán router-nek jelölt szerverről indítottuk.

A méréseink során a párhuzamos kérés szám (*concurrency* - *c*) paraméter értékét változtattuk, ez terheli meg a rendszerünket. Ezért egy nagyon kevés párhuzamos kérés számból indultunk ki, amelyeknél láttuk, hogy a rendszer gyorsan tud válaszolni. Méréseink során láttuk, hogy veszteség nélkül 10 ezer párhuzamos kérést ki tud szolgálni a rendszer, ezért ebben maximáltuk a *hey* konkurrencia értékét. Minden egyes mérés esetén az összes kérés számát (*number of requests* - *n*) 100 ezer kérésben határoztuk meg. Ez azt jelenti, hogy ha egy párhuzamos kérés szál kap egy választ és az össz kérés szám nem érte el ezt a korlátot, akkor azonnal indít egy új kérést.

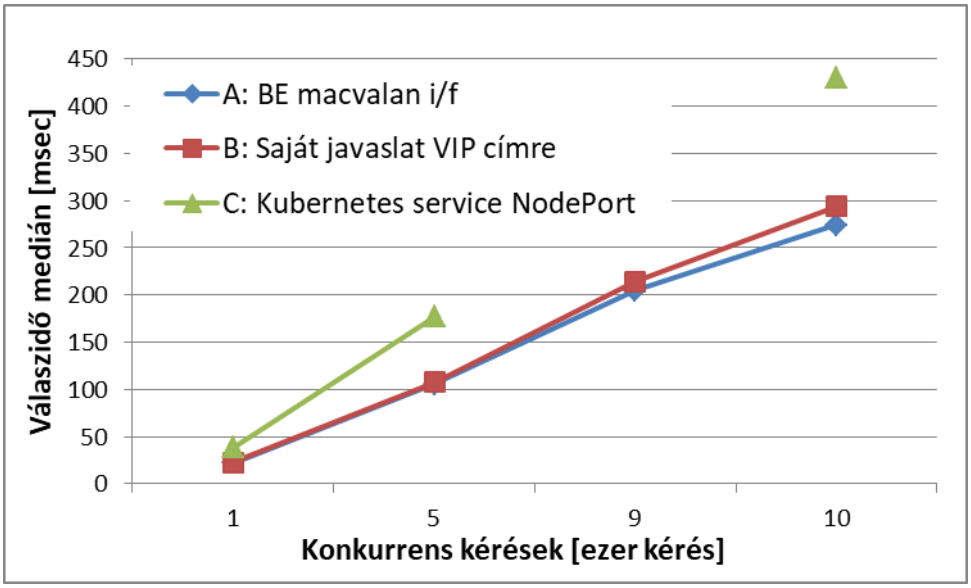
Az első esetben („*A*” szcenárió) a router, kikerülve a Kubernetes CNI mechanizmusait, a multus macvlan interfészen keresztül direkt kommunikál a BackEnd poddal. Szigorúan késleltetés szempontjából ez tűnik a leggyorsabb és leghatékonyabb alternatívának. Mi ezt úgy teszteltük, hogy a BackEnd net2 macvlan interfészét címeztük meg a forrás szerverről.

A második esetben („*B*” szcenárió) az általunk kidolgozott megoldást használtuk, ekkor a VIP címet kellet címezni a forrás oldalon. Csak egy FrontEnd és BackEnd podot használtunk ebben az esetben a méréseinkhez, hogy összehasonlítható legyen az eredmény a másik két szcenárióval.

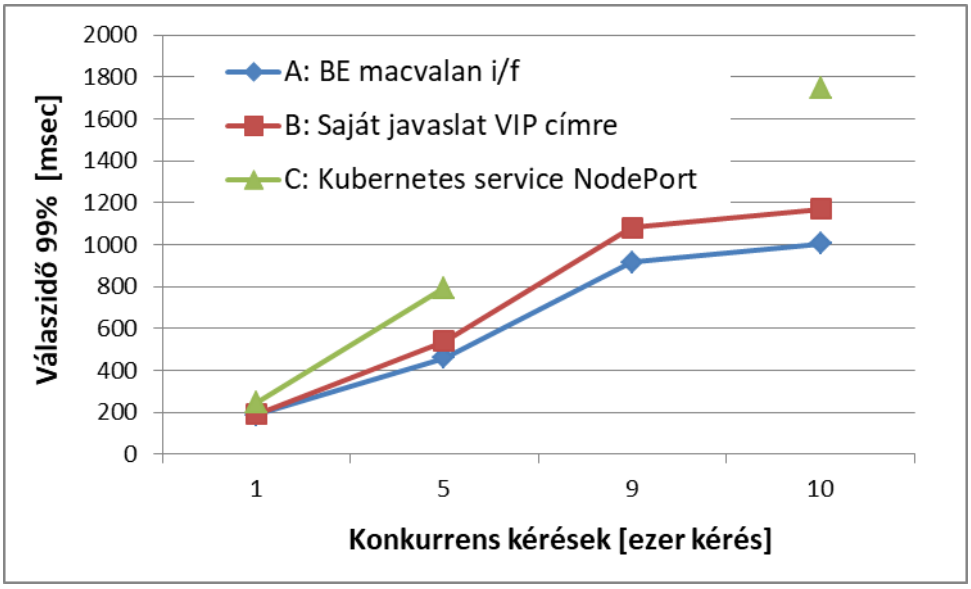
Végül, a harmadik esetben („*C*” szcenárió) azt vizsgáltuk, hogy milyen teljesítményt nyújt a Kubernetes CNI megoldása. Ekkor egy BackEnd podra egy *LoadBalancer* típusú Kubernetes *service*-t definiáltunk. A forrás oldalról így a Kubernetes klaszter egy fizikai

worker node-ját kellett címezzük, a NodePort-ot is megadva. Csak egy BackEnd podot használtunk ebben az esetben a méréseinkhez, hogy összehasonlítható legyen az eredmény a másik két szcenárióval.

A méréseink késleltetési értékeit a 7. és 8. ábrák tartalmazzák. Az első ábra a medián értékeket mutatjuk, mert egy jól működő rendszerben ez mutatja, hogy milyen kiszolgálást kapnak a jobb QoS-t elérő kérések. A második ábrához a 99%-os percentilis értékeket választottuk, mert a gyakorlatban ez az elfogadható tolerancia szint. A késleltetés medián értékei majdnem lineárisan növekednek a terheléssel, de a 99%-os percentilis értékei jól láthatóan kimutatatják, hogy a rendszer telítődik, a késleltetés meghaladja az 1 másodpercet. Az a jelenség, hogy ennek ellenére a medián görbe „laposodik”, azzal magyarázható, hogy a „gyorsan” kiszolgált kérések mellet a „lassan” kiszolgált kérések eloszlása eltolódott a nagyon lassú kiszolgálás irányába. Tehát a terhelés hatására nemcsak a válaszidő nő, hanem a különösen hosszabb ideig a rendszerben „ragadt” kérések arány nő. Látható, hogy előzetes várakozásainknak megfelelően a C szcenárió késleltetés értékei a legrosszabbak. Ugyanakkor ígéretes eredmény, hogy a megoldásunk (B görbe) kevéssel okoz nagyob késleltetést, mint az *A* referencia elrendezés. A C szcenárióhoz képest még egy mérési ponton vizsgáltuk ezt a két szcenáriót, hogy meggyőződjünk erről, de nem kaptunk a várttól eltérő értékeket ott sem. Az látszik, hogy a terhelés növelésével „nyílik az olló”, de nem jelentős mértékben. Ennek alapján azt a következtetést vontuk le, hogy megoldásunk jól skálázódik.

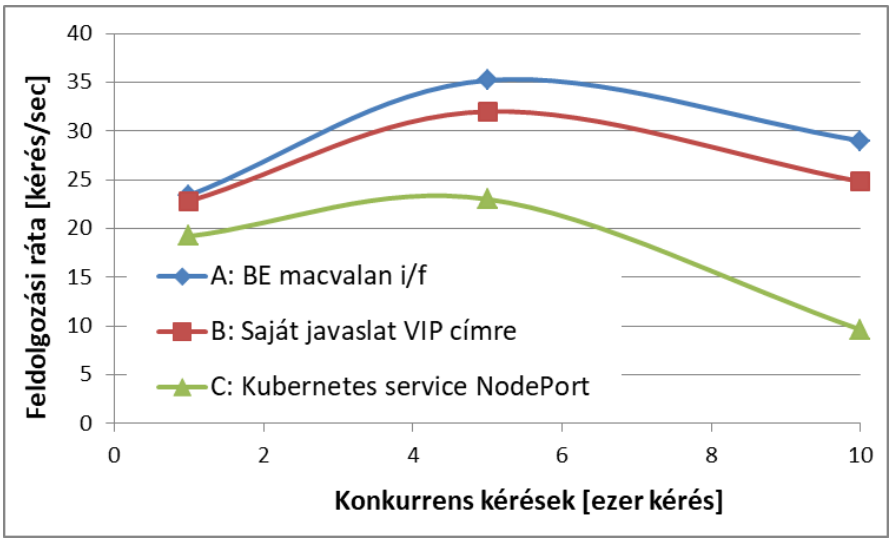


7. ábra – A lekérdezésekre adott válaszok válaszidejeinek mediánja



8. ábra – A lekérdezésekre adott válaszok válaszidejeinek 99% percentilis értéke

Végül összehasonlítottuk a három elrendezés kiszolgálási rátáit. A 9. ábrán bemutatott grafikon megerősíti a korábban tett megállapításainkat a három elrendezés teljesítményének egymáshoz mért viszonyát illetően. Továbbá a feldolgozási ráta csökkenése a 10 ezres párhuzamos lekérdezés esetén is igazolja, hogy ekkor a tesztrendszer telítődik és a feltorlódott kéréseket már csak lassabban tudja a BE kiszolgálni.



9. ábra – A kérések feldolgozási rátája

A méréseink alapján megállapíthatjuk, hogy jól választottuk ki az IPVS mechanizmust és a Maglev implementációt, mert a rendelkezésre álló alternatívákhoz képest jó teljesítményt tud nyújtani valós ipari környezetben is és ezt a szakirodalom is alátámasztja [24][27].

7. Összefoglalás

Dolgozatunkban egy, a modern távközlési rendszerekbe telepített, felhő alapú szolgáltatásokat hátrányt okozó hálózati hatást vizsgáltunk. Munkánk során az ECMP alapú. a Kubernetes klaszterekbe telepített szolgáltatásokra gyakorolt következményeit elemeztük. Kimutattuk, hogy a várakozásinknak megfelelően, a külső hálózati terhelés elosztók egyes hálózati eseményekre a távközlési rendszerekben megszokott elvárásokat olyan mértékben sértik, hogy az szolgáltatás kiesést jelenthet.

Figyelembe véve a távközlési felhő natív rendszereinek telepítési környezete által meghatározott korlátokat, javasoltunk egy FrontEnd alapú megoldást a fenti hatás kiküszöbölésére. Részletesen leírtuk tervezési döntéseink hátterét, a fontosabb mechanizmusok hasznosságát mérésekkel visszaigazoltuk.

A javaslat kidolgozása során használt, gyors tesztelést biztosító környezet után a javaslatunkat egy, az iparban megszokott adatközpont szerverein is megvalósítottuk, majd mérés alapú teljesítményelemzéssel jellemeztük a teljesítményét.

Összefoglalásul megállapíthatjuk, hogy sikerült egy, az elvárások szerint működő javaslatot kidolgozni. Mivel a megoldásunk nem csak a távközlési szektor szereplői, hanem a szélesebb, web szolgáltatást használó fejlesztők számára is hasznos, a továbbiakban a megoldást publikációban dokumentálni, valamint és a Kubernetes közösség számára nyilvánossá szeretnénk tenni.

Köszönetnyilvánítás

Ezúttel fejezzük ki köszönetünket az Ericsson munkatársainak, Papp Olgának, Szabó Örsnek, Saminathan Vijayabaskarannak és Anders Franzennek, akik hasznos meglátásaikkal és tanácsaikkal segítették munkánkat.

Irodalomjegyzék

Ellenőriztük, hogy az irodalomjegyzékben szereplő minden Interneten elérhető forrás 2021. október 28-án elérhető volt, a hivatkozott tartalmat le lehetett tölteni.

[1] https://ieeexplore.ieee.org/abstract/document/7570950
N. Jain, S. Choudhary, "Overview of virtualization in cloud computing." In 2016 Symposium on Colossal Data Analysis and Networking (CDAN), pp. 1-4. IEEE, 2016.

[2] https://phoenixnap.com/kb/what-is-hypervisor-type-1-2
S. Simic: What is a hypervisor?

[3] https://www.w3schools.in/cloud-virtualization/os-virtualization/
OS level virtualization

[4] https://linuxcontainers.org/lxc/introduction/
LXC documentation

[5] https://docs.docker.com/get-started/overview/
Official Docker documentation

[6] https://kubernetes.io/docs/home/
Official Kubernetes documentation

[7] https://developers.redhat.com/blog/2018/10/22/introduction-to-linux-interfaces-for-virtual-networking/
Hangbin Liu: Introduction to Linux interfaces for virtual networking

[8] https://kubernetes.io/docs/concepts/cluster-administration/networking/
Kubernetes Cluster networking

[9] https://github.com/intel/multus-cni
Official Multus CNI documentation

[10] https://doi.org/10.1186/s13677-019-0146-7
Afzal, S., Kavitha, G. Load balancing in cloud computing – A hierarchical taxonomical classification. Journal of Cloud Comp 8 (22) 2019.

[11] https://avinetworks.com/what-is-load-balancing/
What is load balancing

[12] https://tools.ietf.org/html/rfc2992
RFC2991: Equal-Cost Multi-Path routing analysis

[13] https://pasztor.at/blog/ipvs-the-linux-load-balancer/
Janos Pasztor: IPVS: The Linux Load Balancer

[14] https://vincent.bernat.ch/en/blog/2018-multi-tier-loadbalancer
Vincent Bernat: Multi tier loadbalancing with Linux

[15] https://www.f5.com/services/resources/glossary/reverse-proxy
What is a reverse proxy?

[16] https://medium.com/i0exception/rendezvous-hashing-8c00e2fb58b0
Aniruddha: Rendezvous hashing: an alternative to Consistent Hashing

[17]]https://github.blog/2018-08-08-glb-director-open-source-load-balancer/
Theo Julienne: GLB: Github's open source load balancer

[18] https://storage.googleapis.com/pub-tools-public-publication-data/pdf/44824.pdf
Maglev: A Fast and Reliable Software Network Load Balance

[19] https://metallb.universe.tf/
MetalLB, bare metal load-balancer for Kubernetes

[20] https://www.eurecom.fr/publication/6417/download/comsys-publi-6417.pdf
O. Arouk, N. Navid, "Kube5G: A Cloud-Native 5G Service Platform", GLOBECOM 2020-2020 IEEE Global Communications Conference, 2020.

[21] https://www.juniper.net/documentation/en_US/junos/topics/topic-map/switches-interface-resilient-hashing.html#jd0e134
Juniper Resilient Hashing

[22] https://wiki.mikrotik.com/wiki/Manual:IP/Route#Multipath_.28ECMP.29_routes
Official Mikrotik documentation on mutlipath

[23] https://blog.memcachier.com/2017/09/01/maglev-our-new-consistent-hashing-scheme/

Maglev consistent hash used
[24] https://blog.cloudflare.com/high-availability-load-balancers-with-maglev/
Terin Stock: High availability load balancers with Maglev
[25] https://www.usenix.org/system/files/conference/nsdi16/nsdi16-paper-eisenbud.pdf
Maglev Performance Study
[26] https://github.com/rakyll/hey
hey HTTP Load generator
[17] https://www.projectcalico.org/comparing-kube-proxy-modes-iptables-or-ipvs/
IPVS Performance

Függelék

A. A Hello teszt konténer Dockerfile-ja

```
FROM golang:1.8-alpine
ADD . /go/src/hello-app
RUN go install hello-app

FROM alpine:latest
COPY --from=0 /go/bin/hello-app .
COPY ./interfacesetup.sh /
RUN apk --no-cache add iptables tcpdump iproute2 net-tools
ENV PORT 80
CMD ["./hello-app"]
```

10. ábra - A tesztelések során a BackEnd podban használt konténert leíró Dockerfile

B. A Linux IPVS conntrack tábláinak szinkronizációja

Az alábbi mérés a 4.2 alfejezetben leírt tesztrendszeren készült.

A *fe1* FrontEnd podon belüli IPVS conntrack táblája (kiemeltük a követett kapcsolatot) (11. ábra).

```
Every 2.0s: cat ip_vs_conn_sync                                    fe1: Wed

Pro FromIP   FPrt ToIP     TPrt DestIP   DPrt State       Origin Expires
TCP C0A86E04 A739 0AC90001 0050 0AC800FA 0050 ESTABLISHED LOCAL      64
TCP C0A86E04 D601 0AC90001 0050 0AC800FA 0050 ESTABLISHED SYNC       899
```

11. ábra – A fe1 FrontEnd pod conntrack táblája

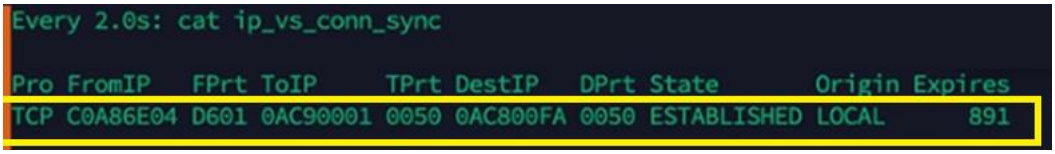
A *fe2* FrontEnd podon belüli IPVS conntrack táblája szinkronizáció előtt (12. ábra).

```
Every 2.0s: cat ip_vs_conn_sync

Pro FromIP   FPrt ToIP     TPrt DestIP   DPrt State       Origin Expires
TCP C0A86E04 E86D 0AC90001 0050 0AC800FA 0050 ESTABLISHED LOCAL      238
TCP C0A86E04 D77F 0AC90001 0050 0AC800FA 0050 CLOSE       SYNC       6
```

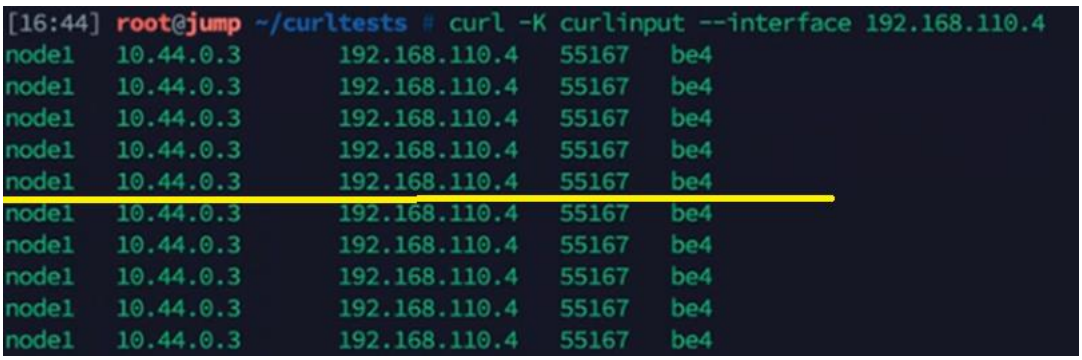
12. ábra – A fe2 FrontEnd pod conntrack táblája szinkronizáció előtt.

A *fe2* FrontEnd podon belüli IPVS conntrack táblája után (kiemeltük a követett kapcsolatot) (13. ábra).



13. ábra – A fe2 FrontEnd pod conntrack táblája szinkronizáció után.

Az útvonalváltás (sárga vízszintes vonallal jelöltük) előtt és után is folyamatos a szolgáltatást biztosító pod válasza, a pod ugyanaz marad (14. ábra).



14. ábra – A szolgáltatást nyújtó be4 BackEnd pod folyamatos lekérdezése.

C. A teszthálózat egyik FrontEnd podjának kialakítása

```
apiVersion: v1
kind: Pod
metadata:
  name: fe1
  annotations:
    k8s.v1.cni.cncf.io/networks: '[
            { "name": "macvlan100",
              "ips": [ "10.100.0.1/24" ],
              "mac": "c2:b0:57:49:47:f1",
              "gateway": [ "10.100.0.254" ],
              "default-route": ["10.100.0.254" ]
            },
            { "name": "macvlan200",
              "ips": [ "10.200.0.1/24" ],
              "mac": "c2:b0:57:49:47:f2"
            }
            ]'
spec:
  nodeName: worker1
  hostAliases:
  - ip: "10.200.0.253"
    hostnames:
    - "hello"
  containers:
  - name: teszt
    image: frontendubuntu:latest
    imagePullPolicy: Never
    ports:
    - containerPort: 80
#    lifecycle:
#      preStop:
#        exec:
#          command: ["/usr/sbin/nginx","-s","quit"]
    securityContext:
      capabilities:
        add:
          - ALL
      privileged: true
```

15. ábra - A 6. fejezetben leírt tesztelések során használt FrontEnd pod YAML leíró fájlja

D. A teszthálózat egyik BackEnd podjának kialakítása

```
apiVersion: v1
kind: Pod
metadata:
  name: be1
  annotations:
    k8s.v1.cni.cncf.io/networks: '[
            { "name": "macvlan200",
              "ips": [ "10.200.0.253/24" ],
              "mac": "c2:b0:57:49:47:aa",
              "gateway": [ "10.200.0.254" ],
              "default-route": ["10.200.0.254" ]
            }]'
spec:
  nodeName: worker1
  containers:
    - name: be1
      image: "mybackend:latest"
      imagePullPolicy: Never
```

16. ábra - A 6. fejezetben leírt tesztelések során használt BackEnd pod YAML leíró fájlja

E. A WIGNER DC teszthálózat ECMP router konfigurációja

```
ubuntu@compute-13-kubernetes-router:~$ ip route
default via 172.16.232.252 dev enp6s0f1 proto static
10.100.0.0/24 dev vlan100 proto kernel scope link src 10.100.0.254
10.200.0.0/24 dev vlan200 proto kernel scope link src 10.200.0.254
10.201.0.0/24
        nexthop via 10.100.0.1 dev vlan100 weight 1
        nexthop via 10.100.0.2 dev vlan100 weight 1
        nexthop via 10.100.0.3 dev vlan100 weight 1
172.16.232.0/24 dev enp6s0f1 proto kernel scope link src 172.16.232.5
```

17. ábra - A WIGNER DC-be telepített ECMP routerként használt Linux szerver routing táblája