Önálló laboratórium beszámoló

Távközlési és Médiainformatikai Tanszék

készítette: **Váradi Richárd Tamás**
ricsi19981007@gmail.com
neptun-kód: **XA5OZH**
ágazat: **Médiainformatika**
konzulens: **Dr. Rétvári Gábor**
retvari@tmit.bme.hu

Téma címe: A Kubernetes és a szolgáltatáshálók hálózati kérdései

Feladat:
A Kubernetes és szolgáltatáshálók hálózati megoldásainak feltérképezése volt. Az egyik olyan probléma, amit én és a konzulensem találtunk, hogy az Istio, mint szolgáltatásháló nem képes UDP (User Datagram Protocol) csomagok fogadására. Viszont az Istio-ban használt Envoy proxy egy újabb verziója már képes UDP csomagokat továbbítani így megvalósítható az is, hogy az Istio-ba ilyen forgalmat irányítsunk be. A félév során egy olyan megoldást kellett találnom, amivel az Istio elé ezzel a proxy-val képes legyek egy úgynevezett átjárót létrehozni. Ennek az átjárónak UDP csomagokat kell tudnia fogadni és átalakítani őket olyan formátumúvá, amit az Istio képes lekezelni.

Tanév: 2019/20. tanév, II. félév

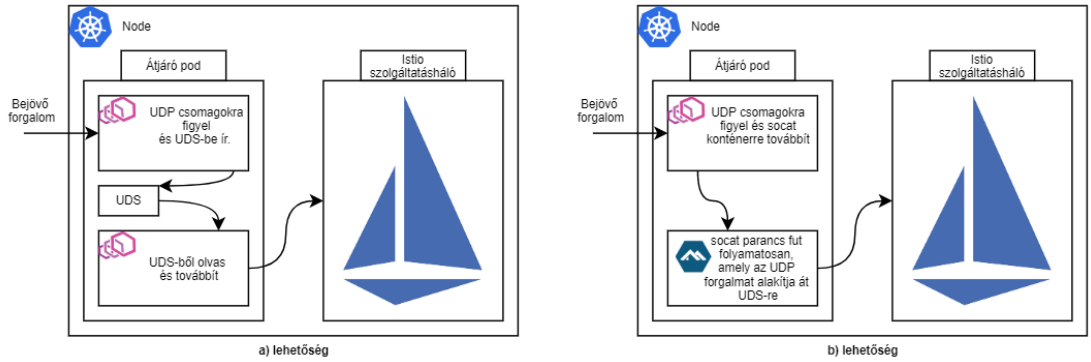
1. A laboratóriumi munka környezetének ismertetése, a munka előzményei és kiindulási állapota

1.1. Bevezető

Az ötlet alapvetően egy GitHub-os hibajegyből született. A GitHub egy webalapú verziókezelő és együttműködésű felület szoftverfejlesztőknek. A hibajegyek úgy születnek, hogy projektekhez lehet őket hozzáírni és erre jó eséllyel a fejlesztők vagy más felhasználók reagálnak. Számunkra most ez a jegy lesz a fontos: https://github.com/istio/istio/issues/1430. Pár példát szeretnék idézni és fordítani, hogy miért kellene ez a támogatottság:

- Publikus felhők, ahol egy titkosított szolgáltatáshálót hoznának létre, amely képes kezelni olyan dolgokat, mint a DNS (Domain Name System) és NTP (Network Time Protocol).
- IoT (Internet of Things) területen.
- Telekommunikáció terén is jó lehet a 3G-től használják az UDP protokollt adattovábbításra.

Ennek megvalósítására a konzulensemmel úgy gondoltuk, hogy érdemes lenne egy átjárót felépíteni az Istio elé, amelyet az alábbi ábrákon be is mutatok.



Az **a** lehetőség nem teljesen biztos, hogy megvalósítható az Envoy korlátai miatt. Az Envoy egy nyílt forráskódú perem és szolgáltatás proxy, ami natív felhőalkalmazásokhoz lett írva C++-ban. Ez a megvalósítás elméleti szinten működőképes lenne, viszont nagyon kevés dokumentáció található arról, hogy az UDS (Unix Domain Socket) pontosan hogyan működik az Envoy-ban.
A **b** megoldás annyiban egyszerűbb, hogy ott kettő konténert hozunk létre egy pod-ban, ahol az Envoy UDP csomagokat figyel és UDP csomagokat továbbít egy Alpine alapú konténernek, amiben fut egy socket parancs, ami képes UDP forgalmat UDS-re irányítani. A Pod a kubernetes legkisebb egysége, amelyekben konténereket lehet létrehozni. A konténereket érdemes úgy létrehozni, hogy konténerenként egy folyamat fusson benne. Az Alpine egy nagyon kicsi erőforrás igényű Unix alapú operációs rendszer.

1.2. Elméleti összefoglaló

1.2.1. Docker

A Docker egy szolgáltatáskészlet Paas (Platform as a Service) termékcsalád, amelyek operációs rendszer szintű virtualizációt végeznek, hogy a szoftvert csomagokban, úgynevezett konténerekben lehessen létrehozni.

A konténerek egymástól elválasztva és saját szoftverüket futtatva léteznek külön könyvtárakkal és konfigurációs fájlokkal. Ezek a konténerek képesek kommunikálni egymással jól definiált csatornákon keresztül. Érdemes azt az elvet szem előtt tartani, hogy minden konténerben csak egyetlen folyamat fusson így sokkal egyszerűbb esetleges hibánál a forrást megtalálni és javítani.

Minden konténer egyetlen operációs rendszer kernelt használ, ezért sokkal kevesebb erőforrást igényelnek, mint a rendes virtuális gépek. Viszont olyan hátulütője van ennek a tulajdonságának, hogy mondjuk a

kiszolgáló egy Linux operációs rendszer, akkor csak Linux alapú konténerek hozhatóak létre, míg egy Windows alapú kiszolgálónál csak Windows alapú konténerek születhetnek. Bár az utóbbi állítás a Hyper-V újítása révén már nem probléma érdekességként ajánlom a Microsoft dokumentációját erről [1]

Ezek a konténereket képfájlokból lehet kialakítani, amiket saját magunk is megírhatunk, de böngészhetünk is közülük a DockerHub-on [2], amely hasonlóan működik, mint a GitHub.

Mikor egy ilyen képfájlt létrehozunk, akkor mindig kell egy alap képfájl, amelyet az említett DockerHub oldalról letudunk tölteni a docker alkalmazás segítségével.

További információ gyűjtésére erről a témáról a következő oldalakat ajánlom: [3] [4]

1.2.2. Kubernetes

A kubernetes egy nyílt forráskódú rendszer, amely a fejlesztés automatizálására, skálázásához és konténer alapú alkalmazások kezelésére való. Eredetileg a Google fejlesztette, de jelenleg a CNCF (Cloud Native Computing Foundation) tartja karban.

Azért érdemes használni, mert mikroszolgáltatások lehet benne létrehozni, amelyek tudnak egymással kommunikálni, de a hálózati megvalósítások nem ezekbe a szolgáltatásokban kell létrehozni, mert a kubernetes erről gondoskodik nekünk olyan hálózati technológiákkal, amelyek megtalálhatóak egy átlagos hálózatban is. Ilyen például a DNS, Routing táblák, IP (Internet Protocol) táblák.

A következőkben ismertetem a két legalapvetőbb komponenst, amelyek a Pod és a Node.

A **Pod** a legkisebb létrehozható objektum alkalmazás fejlesztésére. Egyetlen Pod egy futó folyamatot reprezentál a klaszterünkben és egy vagy több Docker konténert tartalmaz, amelyek saját tárhelyez igényel és egyedi IP címet. Ezek a konténerek úgy lettek tervezve, hogy ugyanazon a gépen helyezkedjenek el és legyenek egyszerre ütemezve.

A klaszter úgynevezett dolgozó gépek gyűjteménye, amelyeket Node-oknak nevezünk. Minden klaszternek legalább egy dolgozó node-ja van.

Mint már említettem egy node egy klaszterben dolgozó gépet reprezentál, ezek lehetnek fizikai gépek, virtuális gépek vagy bármi más. Esetünkben ez egy virtuális gép lesz, melyet a Minikube nevezetű alkalmazás fog számunkra biztosítani. További információ a Minikube-ról. [5]

Most, hogy a számunkra fontosabb részeket ismertettem áttérek a kubernetes hálózati modelljének bemutatására. Kezdetben három tulajdonságát szeretném bemutatni.

- Az összes pod képes kommunikálni a hálózatban megtalálható összes pod-al NAT (Network Address Translation) használat nélkül
- Az összes node képes kommunikálni az összes pod-al NAT nélkül
- Amilyen IP címet lát a pod a saját interfészéhez rendelve, ugyan azt a címet fogja látni más pod vagy node is a hálózatban.

Így a következő hálózati kihívások jelentkeznek:

1. Konténer -> Konténer adattovábbítás
2. Pod -> Pod adattovábbítás
3. Pod -> Szolgáltatás adattovábbítás
4. Internet -> Szolgáltatás adattovábbítás

Ami most számunkra fontosabb lesz az az első kettő pont.

Konténer -> Konténer kommunikáció

Tudni kell, hogy a podok egy közös névtérben futnak és minden podnak lehet saját egydülálló névtere. Ez azért egyszerűsíti meg a kommunikációt a podon belüli konténerek számára, mert így minden konténernek ugyan az az IP címe lesz és csak a portfoglalások fogják őket megkülönböztetni. Ez nem csak, azért nagyon jó dolog számunkra, mert könnyebb hivatkozni külsőleg ezeket a konténereket, hanem egymás között localhost-on tudnak kommunikálni adott portszámokkal. Így maga a kommunikáció is egyszerűbb lesz közöttük, mert a bennük futó olyan folyamatoknak, amiknek szükségük van más konténerekkel való kapcsolatra nem kell konkrét címeket tudni csak portszámokat, de a sebességre is pozitív hatással van.

Pod -> Pod kommunikáció

A konténerekkel ellentétben egyéni IP címmel rendelkezik minden egyes Pod a hálózatunkban. De ez sajnos még nem elég annak érdekében, hogy könnyen megvalósítható legyen közöttük az adattovábbítás. Ehhez ugyan is a gyökér névteret fogják használni, amiből egy van Node-ként. Ez a névtér rendelkezik minden Pod számára egy Virtuális Ethernet Eszköz interfészel és ezek össze vannak úgymond kötve minden egyes Pod eth0 interfészével. De ez még nem fogja megoldani a továbbítást. Ahhoz még egyetlen elemre van szükségünk, mégpedig egy hídra. Ez a híd lesz, az ami a virtuális ethernet interfészeket összeköti és továbbítja a csomagokat. De felmerül a kérdés, hogy hogyan találja meg a célt ez híd. Erre használja az ARP (Address Resolution Protocol) protokollt, ami IP cím alapján képes megtalálni, hogy milyen MAC (Media Access Control address) címre kell továbbítania a forgalmat. Ez a cím egyedileg azonosítja az interfészeket.

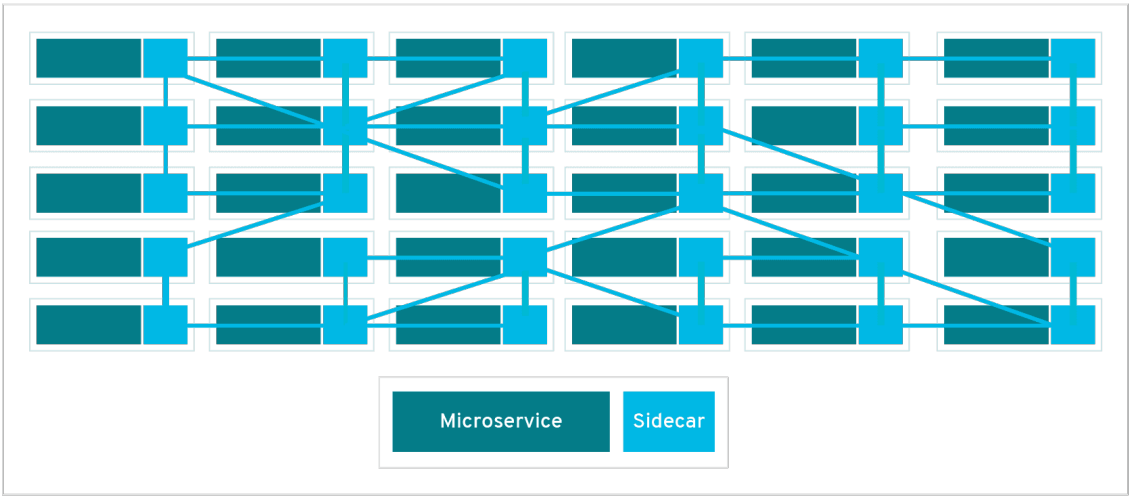
További érdekes információk találhatóak Kevin Sookocheff cikkében [6]

1.2.3. Szolgáltatásháló

A szolgáltatásháló, olyan mint mondjuk az Istio egy módja annak, hogy az alkalmazás különböző részei, hogyan osztanak meg adatot egymás között. Más rendszerekkel ellentétben a kommunikációra a szolgáltatásháló egy dedikált infrastruktúra réteg magában az alkalmazásban. Ez egy "látható" vagyis érzékeljük, hogy ott van, miközben a különböző részek kommunikálnak egymással és így láthatjuk, hogy ezek a komponensek hogyan működnek vagy nem és így egyszerűbbé válik a kommunikáció optimalizálása és elkerülhető a kiesett idő, amíg esetleg az alkalmazásunk nem üzemel.

Minden részét az alkalmazásnak egy szolgáltatásnak hívunk, amelyek különböző folyamatokért felelősek.

Legjobban úgy lehet megérteni, hogy hogyan működnek a szolgáltatáshálók, ha ezt egy ábrával tesszük.



Minden szolgáltatás egy mikroszolgáltatásból és egy sidecar proxy-ból áll. Mikroszolgáltatás alatt az alkalmazás egy adott funkciójának megvalósítását valamilyen nyelven értjük. Fontos kihangsúlyozni, hogy nem kötött milyen programozási nyelven van az adott mikroszolgáltatás írva, mert ez is nagyon rugalmassá teszi az ilyen fejlesztést, abban az esetben, ha különböző csapatoknak más és más nyelven akarnak megírni egyes szolgáltatásokat. Viszont itt jön képbe a sidecar proxy, ami elvégzi az a hálózati kommunikációt a szolgáltatások között. Szóval nem kell a fejlesztőnek a mikroszolgáltatásban azzal foglalkozni, hogy kiknek címezze a kimenő forgalmat vagy, hogy honnét kapja, mert ez mind a két esetben a sidecar lesz.

A kép forrása és további érdekes információk megtalálhatóak még a Red Hat ezen cikkében [7]

1.2.4. Envoy

Ez egy L7 (alkalmazás) rétegen működő proxy és kommunikációs csatorna arra tervezve, hogy nagy szolgáltatás alapú architektúrákat lehessen létrehozni úgy, hogy a szolgáltatások számára a hálózat átlátszó legyen. A proxy egy olyan szerver, amely hálózati forgalmat irányítja. Az előzőleg említett funkcióját úgy

sikerül ellátnia, hogy a szolgáltatáshálók részben lévő szolgáltatásoknál az Envoy lesz maga a sidecar, így a mikroszolgáltatásnak elég mindig csak a localhost-ra küldeni és onnét fogadni adatot.

Képes L3/L4 (hálózati/szállítási) réteg szűrőket is alkalmazni, amivel a különböző protokollokat, mint a TCP (Transmission Control Protocol) és az UDP.

Ezen felül rendelkezik még olyan funkciókkal, amelyek segítségével megvalósítható perem proxy-ként és belső proxy-ként is, felügyelhető, és HTTP (HyperText Transfer Protocol) alapú szűrést is tudunk használni.

További információk ezen az oldalon találhatóak [8]

1.2.5. További technológiák, amelyek szükségesek

UDP - User Datagram Protocol
Az UDP egyszerű kapcsolatmentes összeköttetést hoz létre 2 eszköz között, így nincs kapcsolat ellenőrzés, mint a TCP-nél ezért sokkal gyorsabban felállítható a kapcsolat eszközök között. Viszont nincs garancia arra, hogy a csomagok megérkeznek, sorrendben érkeznek-e vagy, hogy duplikált csomagok jönnek át a csatornán. Ezért olyan szolgáltatásokhoz érdemes használni, ahol nem probléma, ha egy csomag többször jön vagy hiányoznak csomagok. Ilyenek például a telekommunikáció és az élő közvetítések is. További információk találhatóak ezen az oldalon [9]

UDS - Unix Domain Socket
Arra használatos, hogy a folyamatok úgy tudjanak egymással kommunikálni, hogy az minél jobb legyen. A Unix Domain Socket lehet névtelen, vagy hozzárendelve egy fájlrendszerben létrehozott elérési útvonalhoz. További információkat lehet találni ezen az oldalon [10]

iPerf
Az iPerf egy olyan eszköz, amellyel lehet mérni a maximálisan elérhető sávszélességet IP hálózatokban. Támogatja a különböző protokollokat, így használható TCP és UDP mérésre is, de ezen felül lehet még sok máshoz is használni. De betudunk állítani olyanokat is, hogy adott ideig és időközökkel küldjön általunk megadott csomagmérettel és küldési sebességgel, szóval egy nagyon kis hasznos eszköz.

Minden ilyen teszt végén kapunk egy jelentést arról, hogy milyen sávszélességgel érkeztek meg a csomagokat, ebből mennyi veszett el és még más paramétereket is annak tükrében, hogy milyen részletes jelentések szeretnénk látni.

A használata úgy néz, hogy létre kell hozni egy úgynevezett iPerf szervert, amihez beérkeznek a csomagok. Ez lehet bárhol, amit az hálózaton keresztül elérünk. Ezen felül még létre kell hozni egy iPerf klienst is, aki az általunk specifikált módon fog csomagokat küldeni a szerverre.

További érdekes információk találhatóak az weboldalán [11]

socat
A Socat egy parancssori alapú segédprogram, amely két kétirányú bájt-folyam hoz létre és adatot továbbít közöttük. Mivel az adatfolyamok különféle típusú adatgyűjtőkből és forrásokból állíthatóak össze, és mivel sok címbeállítási lehetőséget lehet alkalmazni az adatfolyamokra, a socat felhasználható többféle célra.

Ezt fogjuk arra használni, hogy UDS csomagokat alakítsunk át UDP csomagokká. További információk találhatóak a következő oldalon [12]

1.3. A munka állapota, készültségi foka a félév elején

A témalaboratórium alkalmával már foglalkoztam a szolgáltatáshálók megismerésével és használatával, ahol az Isio és az SMI (Service Mesh Interface) különböző aspektusait hasonlítottam össze. Ennek során mélyebb rálátást sikerült nyernem a Kubernetes használatára és arra, hogy egy jó dokumentáció milyen nagy segítséget nyújthat azoknak, akik ebben az iparágban dolgoznak.

2. Az elvégzett munka és az eredmények ismertetése

2.1. A munkám ismertetése logikus fejezetekre tagoltan

Minden konfiguráció, amit a labor során elvégeztem statikusan működik, vagyis csak egy megadott IP cím és port párosra érvényesek, amelyeket meg kell adni a konfigurációs fájlokban. Később a szakdolgozatban szeretném majd úgy megvalósítani ezeket a feladatokat, hogy dinamikusan tudjanak működni.

2.1.1. Lokális környezet

Minden konfigurációt és mérést Ubuntu 18.04 LTS (Long Term Support) környezetben végeztem el, annak érdekében, hogy konzisztensebb legyen a mérés a későbbiekben is, mert számomra egyszerűbb dockert és kubernetest használni Linux alapú környezetben, mint Windows-ban.

Ahhoz, hogy lokálisan, docker és kurbernetes nélkül tudjam használni az Envoy-t elsőnek is telepíteni kellett, amit a Lyft (akik az Envoy-t fejlesztik) által nyújtott dokumentáció alapján sikerült, viszont egy külső forrás alapján nagyon egyszerűen megvalósítható volt. Ennek a részleteibe nem szeretnék belemenni, mert csak utasításokat kell követni, viszont ezen az oldalon megtalálhatóak [13].

Ehhez és a másik két környezethez is az Envoy legújabb verzióját használtam, ami a 1.14.1-s verziószámot jelenti.

Ahhoz, hogy a proxy-t lehessen konfigurálni YAML (YAML Ain't Markup Language) fájlt/fájlokat kell létrehozni. Ezeket a fájlokat adattárolásra és konfigurációk létrehozására szokás használni, mert könnyű őket olvasni és írni is.

Esetünkben három része lesz, ennek a fájlnak, amire oda kell figyelni és módosítani. Ezek lesznek azok a részek: **admin, listeners, clusters**. Az Envoy dokumentációjában találtam egy példa fájlt, amit alapnak vettem minden alkalommal, ez a fájl megtalálható ezen az oldalon [14]

Az admin rész csak annyiban érdekes számunkra, hogy a 127.0.01:9901-s címen ezáltal elérjük az általunk beállított porxy konfigurációját, logjait és hasonló értékeket, de ezek megtalálhatóak lokálisan is, ahol fut az Envoy. Mivel a kezelőfelületnek nem kell feltétlen UDP alatt működnie így az maradhat TCP is.

Ezután közvetlen jön a Listeners, mint a neve is utal rá, ami ez alatt a blokk alatt fog szerepelni, olyan attribútumú csomagokra fog figyelni. Elméletileg egy ilyen konfigurációban megadható egynél több listener is, de ezt nem ajánlják az Envoy dokumentációjában, mert úgy nehezebb karbantartani és statisztikai adatokat kinyerni. Mivel csak lokálisan fog futni a proxy, ezért localhost-on fogunk figyelni és annak egy általunk választott portján, aminek én most a **4000**-set választottam. Fontos ugye az is, hogy itt már UDP csomagokra kell figyelnünk, ami azt jelenti, hogy erre a címre érkező csomagok csak UDP protkollal rendelkezhetnek. Fontos még, hogy a listener-en belül még be legyen állítva ez az érték is: **reuse_port: true**, mert alapértelmezetten az Envoy nem hoz létre socket-et minden a dolgozó szálaknak, így viszont kitudjuk kényszeríteni, hogy minden ilyen szálnak legyen egy sajátja.

Legutolsó sorban a Klaszter rész jön, ahol statikusan beállítjuk, hogy milyen címre továbbítsa azokat a csomagokat, amit a listener-ben beállítottunk. Ezt én most a 127.0.0.1:5000 címre állítottam be, aminek a portszámát kedvünk szerint módosíthatjuk, de oda kell figyelni arra, hogy ez ne egyezzen a konfiguráció során beállított más porttal, mert akkor sajnos nem fog működni a továbbítás. Ezen felül több végpontot is megadhatunk nem csak egyet így megoldható a terhelés elosztás is különböző pontok között. Ebben az esetben elég számunka az alap ROUND_ROBIN, ami elméletileg sorra haladna végpontokon és sorban küldené ki a csomagokat, de mivel nálunk csak egy végpont lesz így ezt nem fogjuk tapasztalni.

A kész konfigurációs fájl megtalálható lesz, a csatlakozó dokumentumok jegyzékénél.

A konfigurációnkat már csak el kell fogadtatni az Envoy-al, hogy aszerint fogadja és továbbítsa a bejövő forgalmat. Ezt egy egész egyszerű paranccsal megtudjuk tenni: **envoy -c <s fájl elérési útvonala>**

Ezután már nem marad más csak tesztelni, hogy milyen sávszélességgel képes továbbítani a csomagokat. Ennek a mérésére fogjuk használni az iPerf parancsot.

Az iperf szervert a következő paranccsal hozzuk létre.

```
iperf -s -p 5000 -u -U -e -i 1 --realtime
```

Ezzel megadtuk, hogy az 5000-s porton figyeljen UDP forgalomra és csak egy UDP szálon figyelünk. Az -e kapcsoló felel azért, hogy részletesebb adatokat kapjunk, mindezt másodpercekre bontva valós időben.

Ezután létre kell hozni az iperf klienst is, aki a forgalmat fogja biztosítani.

```
iperf -u -c 127.0.0.1 -p 4000 -b 72250000 -l 100 -i 1 -t 10
```

UDP forgalmat továbbítunk a 120.0.0.1:4000-s címre, ugye ezen figyel az általunk beállított proxy. 72.3 Mbit/s sebességgel 100 byte-os csomagokkal másodpercenként küldjük a csomagokat 10 másodpercen keresztül.

A teszt eredménye a szerveroldalon lesz látható.

Ez még kellene.

2.1.2. Dockeres környezet

A dockeres megvalósításnál is igazából hasonlóan fogunk eljárni, mint lokálisan, de csak annyi fog különbözni, hogy nem a mi számítógépünk fog futni, hanem a konténerben. Ez azért egy jó megoldás, mert így akár több ilyen proxy is könnyedén létrehozható más és más konfigurációkkal.

Ahhoz, hogy ezt megtudjuk valósítani elsőnek a Docker egy vverzióját kell telepítenünk. Telepítésnél oda kell figyelni, hogy Snap store-ból nem szabad, mert, akkor nem fog minden funkcionalitása megfelelően működni. Telepítés folyamatát érdemes Docker által nyújtott dokumentáció alapján csinálni [15].

Ha már van Docker a gépünkön, akkor elsőnek le kell tölteni a megfelelő Envoy képfájlt a DockerHub-ról, amit a következő paranccsal egyszerűen megtehetünk.

```
sudo docker pull envoyproxy/envoy:v1.14.1
sudo docker images
```

Az első paranccsal az általunk kívánt képfájl és adott verziója fog letöltődni, míg a másodikkal ellenőrizni tudjuk, hogy megvan-e a képfájl.

Ezután már csak a konfigurációs YAML fájlt kell megcsinálni, ami nagyon hasonló lesz, mint a lokálisnál, csak annyiban fog különbözni, hogy nem localhost-ról fogunk csomagokat kapni és nem is oda fogunk küldeni, hanem a docker konténer saját IP címére fogunk. Ez a cím nekem a **172.17.0.1**, de a **ip a** paranccsal könnyen ellenőrizni tudjuk. De minden más marad a régiben.

Ahhoz, hogy ezt a konténert létre tudjuk hozni a következő parancsot kell kiadni:

```
docker run --net=host --name=proxy-udp -d \
> -p 9901:9901 \
> -p 4000:4000 \
> -v $(pwd)/udpDocker.yaml:/etc/envoy/envoy.yaml \
> envoyproxy/envoy:v1.14.1
```

Az elsősorban megadjuk, hogy milyen hálózatban legyen a konténerünk, aminek most a kiszolgáló hálózata lett beállítva és adtunk neki egy kicsit jobban megjegyezhető nevet, mint amit egyébként a docker adna neki. Az utána következő 2 sorban azt a két portot adjuk meg, amelyeken keresztül a konténerünk elérhetővé válik. Utána jön az izgalmasabb rész, ahol az általunk írt konfigurációs fájl tartalmát betöltjük a konténerben lévőbe. Ezt azért kell megcsinálni, mert az alapképfájlban van egy alapkonfiguráció, ami jelenleg számunkra nem kell szóval felül kell írni. Másrészt, azért is szükség van arra, hogy így megadjuk, mert így nem kell a konténerbe belépni, majd ott megadni a fájlt, ami esetleg több ilyen konténer létrehozásánál elég fárasztó lehet. Az utolsó sornál pedig csak a képfájl nevét adjuk meg elérhetőségnek.

A sebesség mérése csak annyiban fog különbözni az előzőhöz képest, hogy nem a 127.0.0.1:4000-s címre fogunk csomagokat küldeni, hanem a 171.17.0.1:4000 címre.

Ehhez a részhez tartozó konfigurációs fájl is megtalálható a csatlakozó dokumentumok jegyzékében.

Itt is meg kell csinálni a sebességmérést.

2.1.3. Kubernetes környezet

Ahhoz, hogy kubernetes-ben is létre tudjuk hozni ugyan ezt a környezetet, ahhoz kell egy olyan program amellyel virtuális létre tudunk hozni egy dolgozó node-t. Erre tökéletes lesz a Minikube, de ahhoz hogy ezt tudjuk használni kell, hogy a számítógépünk képes legyen virtualizációra. Virtualizálni a legtöbb gép már képes, de ha virtuális környezetben dolgozunk, akkor azok nem képesek arra. Mindemellet kell, hogy legyen valamilyen eszköz, amely az alapot fogja biztosítani a Minikube-nak. Én a VirtualBox-t választottam, amiben virtuális gépeket lehet létrehozni, de ez lehetett volna VMware vagy KVM is.

A Minikube és a Kubernetes könnyedén telepíthető a következő oldal segítségével [16]

Mivel most nem egy nagy erőforrás igényű alkalmazást fogunk létrehozni nem kell a virtuális gép paramétereit beállítani és elég csak szimplán indítani a Minikube-t a következő paranccsal.

```
minikube start
```

Így 2 CPU maggal, 2 GB memóriával és 20 GB tárhellyel fog létrejönni. Szükséges megjegyeznünk a későbbiekhez az ip címét.

```
minikube ip
192.168.99.100
```

Ezután létre kell hoznunk a saját Envoy képfájlunkat, mert ha az alap fájlból szeretnénk létrehozni a pod-ot, akkor nem fogjuk tudni úgy konfigurálni, ahogyan azt mi szeretnénk. Ahhoz, hogy ezt megtudjuk csinálni írnunk kell egy **Dockerfile**-t, amiben specifikáljuk, hogy mi legyen az alap képfájl, milyen konfiguráció legyen beállítva és, hogy milyen parancs fusson le a konténer létrejöttekor. Így néz ki a Dockerfile:

```
FROM envoyproxy/envoy:v1.14.1
COPY envoy_upd.yaml /etc/envoy/envoy.yaml
RUN apt-get update && apt-get install -y tcpdump && apt-get install -y net-tools
```

Mint látszik, az első sorban az alap képfájlt adjuk, meg míg a másodikban a saját konfigurációnkat bemásoljuk, abba a fájlba, amiből dolgozni fog az Envoy, mihelyt elindul. Az utolsó sor már csak, azért kell, hogy esetleges hibákat egyszerűbb legyen kideríteni és megoldani.

Most létre kell hozunk a YAML fájlt, ami nagyon hasonlítani fog az előzőekhez, viszont annyiban el fog térni, hogy itt már bármilyen címről érkező UDP csomagot el fogunk fogadni, ha az 5000-s portról érkezik. Viszont küldeni egy címre fogunk csak, ami az én esetemben a VirtualBox által létrehozott lesz, ami 192.168.99.1. Ezt könnyen kideríthető az **ip a** paranccsal.

Miután mindez megvan, már csak létre kell hozni az általunk megadott képfájlt és azt feltölteni a DockerHub-ra, ahonnét majd később le fogjuk tudni tölteni.

```
sudo docker build .
sudo docker tag <image_id> vidarhun/kubeudpenvoy:v1
sudo docker push vidarhun/kubeudpenvoy:v1
```

Most egy Deploymentet fogunk létrehozni, amire azért van szükség, mert így a Pod véletlenszerű törlődése esetén rögtön létre fog jönni egy ugyan olyan funkcionalitású, de más IP-vel rendelkező Pod. De ugyanakkor később, ha több ilyen Pod-ra lenne szükségünk, akkor a számukat könnyen fel lehet skálázni annak megfelelően, hogy mennyi terhelése érkezik a szolgáltatásunkba. A legfontosabb részét bemutatom és részletesen ismertetem, de ez a fájl is megtalálható lesz csatlakozó dokumentumok jegyzékében.

```
spec:
  containers:
  - name: envoy-container
    image: vidarhun/kubeupdenvoy:v1.0
    ports:
      - containerPort: 5000
        protocol: UDP
```

Itt állítjuk be, hogy a Podunk milyen névvel jöjjön létre és, hogy az általunk létrehozott és felöltött képfájl alapján jöjjön létre. Fontos még az, hogy a konténernek egyezzen a nyitott portja a konfigurációs fájlban a listener-nél megadott porttal, mert így lesz képes a csomag betalálni hozzá. A kimenő forgalom számára, azért nem kell ilyen pontosan specifikálni, mert a Pod teljes mértékben képes kilátni a klaszterből és a VirtualBox csatlakozási pontját megtalálni.

Ezt most már csak a kubernetessel kell elfogadtatni, ami nagyon egyszerűen megtehető az alábbi paranccsal.

```
kubectl apply -f <fajl eleresi utvonala>
```

Miután ezt megtettük meg tudjuk nézni, hogy milyen elemek jöttek létre a mi kis hálózatunkban az alábbi paranccsal.

```
kubectl get all
NAME                                    READY   STATUS    RESTARTS   AGE
pod/envoy-deployment-767c686bcc-rrn2k   1/1     Running   2          10d

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   12d

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/envoy-deployment    1/1     1            1           10d
```

```
NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/envoy-deployment-767c686bcc   1         1         1       10d
```

Ebből jól látszik, hogy létrejött egyetlen egy darab Pod, ami a mi konfigurációnkat tartalmazza és az is, hogy van neki egy kis sorszáma, ami jelzi is nekünk, hogy ez nem egy nagyon permanens elem, mert minden egyes létrejöttekor más számot fog kapni.

De ezen felül még látszik, hogy van egy darab deployment-ünk és egy hozzá tartozó replicaset. A replicaset azt mutatja nekünk, hogy hát kubernetes elem jött létre a deployment futtatása során és abból, hány áll készen használatra. Most egy darabot mutat, mert a méréshez elég egy is.

Viszont a proxy még nem érhető el a klaszteren kívül és így nem is tudunk számára semmilyen forgalmat küldeni, mert nem látjuk. Ezt azzal küszöböljük ki, hogy az imént létrehozott deployment-et ki exponáljuk egy porton keresztül a külvilág számára.

Ezt többféle módon is megtehetjük, de a célnak megfelel a NodePort használata is. Az elképzelése ennek az lesz, hogy van a Node és azon kinyitunk egy portot, ami össze van kapcsolva a klaszterrel. Ez azt eredményezi, hogy ami adat a Node-hoz érkezik az mind az adott klaszterbe lesz irányítva és azt is, hogy ami a klaszterből ki jön az mint ehhez a porthoz lesz irányítva. A következő utasítással ez egyszerűen megtehető.

```
kubectl expose deployment envoy-deployment --type=NodePort
--protocol=UDP --port=32000 --target-port=5000
```

Így megadtuk az utasítás számára azt, hogy mely deployment-hez kell egy NodePort-t nyitnia és azt is, hogy az ezen érkező csomagoknak UDP protokollal kell rendelkezniük. A –port részt nem kötelező megadni, ez csak arra az esetre jó, ha van egy másik klaszterünk vagy szolgáltatásunk, amelynek el kell érnia az Envoy-t. Ezzel szemben a –target-port értékét muszáj arra a számra állítani, amin a Pod figyel, mert másképp nem fog az adat bejutni.

Ezután a szolgáltatás létrejötte ellenőrizhető a **kubectl get services** utasítással, ahol látni fogjuk a klaszer címét és a hozzá tartozó portokat. Itt fontos megjegyezni, hogy 2 portot fogunk ott látni, egyszer a 32000-t és egy a kubernetes által generált portot. Szóval, ha klaszteren belül kell nekünk elérni a proxy-t, akkor a 32000-s kell, ha kívülről, akkor a generált.

Most már csak azt kell kideríteni, hogy milyen címre kell küldeni a csomagokat. Ez egy egyszerű minikube utasítással megoldható.

```
minikube service envoy-deployment
|-----------|------------------|-------------|-----------------------------|
| NAMESPACE |       NAME       | TARGET PORT |             URL             |
|-----------|------------------|-------------|-----------------------------|
| default   | envoy-deployment |       32000 | http://192.168.99.100:32315 |
|-----------|------------------|-------------|-----------------------------|
```

Az URL rész alatti címre kell címezni a csomagokat.

2.2. Összefoglalás

A félévi munka során elért új eredmények ismételt, vázlatos, tömör Ebben a részben az adott félévre vonatkozó, az *Önálló laboratórium tárgy keretében elvégzett munka során* elért **új** eredmények ismételt, vázlatos, **tömör** összefoglalását várjuk, lehetőleg nem felsorolásként. Itt még egyszer ki lehet térni a leglényegesebb eredményekre, valamint a félév során felmerülő nehézségekre, de meg lehet említeni a továbbfejlesztési irányokat, lehetőségeket is.

Ezt a részt tagolható a következő pontok megválaszolásával:

- Mi volt az **aktuális kérdés**, probléma, amivel a félév során foglalkoztál?
- Mi a dolgozat **célja**, miért érdekes egyáltalán ezzel a problémával foglalkozni?
- Milyen **módszereket** használtál a probléma megoldása érdekében?
- Mik a legfontosabb **eredmények**?
- Milyen **következtetéseket** lehet levonni?

Ha valaki elolvassa ezt a részt, képet kell kapnia az egész dolgozatról. Ne legyen az absztrakt szó szerinti ismétlése.

Fontos, hogy az itt megadott sablontól el lehet térni, használata nem kötelező, csak segítséget jelenthet, viszont a fedőlap lehetőleg maradjon ugyanez és tartalmilag egyezzen meg a sablon irányelveivel. A beszámoló felépítésében nem érdemes eltérni a *Bevezető – Féléves munka és eredmények bemutatása – Összefoglaló* hármastól.

3. Irodalom, és csatlakozó dokumentumok jegyzéke

3.1. A tanulmányozott irodalom jegyzéke

[1] Microsoft Corporation, *Linux containers on Windows 10*
https://docs.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/linux-containers

[2] Docker Inc, *DockerHub* https://hub.docker.com/

[3] Docker Inc, *Docker Documentation* https://docs.docker.com/

[4] Wikipedia contributors, *Wikipedia:Academic use*, Wikipedia, The Free Encyclopedia, 2011 Nov 11. Available from:
https://en.wikipedia.org/wiki/Docker_(software)

[5] The Kubernetes Authors, *Minikube documentation* https://minikube.sigs.k8s.io/docs/

[6] Kevin Sookocheff, *A Guide to the Kubernetes Networking Model*
https://sookocheff.com/post/kubernetes/understanding-kubernetes-networking-model/

[7] Red Hat Inc, *What's a service mesh?*
https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh

[8] Envoy Project Authors, *What is Envoy*
https://www.envoyproxy.io/docs/envoy/latest/intro/what_is_envoy

[9] Wikipedia contributors, *User Datagram Protocol*
https://en.wikipedia.org/wiki/User_Datagram_Protocol

[10] Linux man-pages project, *Unix Domain Socket*
http://man7.org/linux/man-pages/man7/unix.7.html

[11] iPerf contributors, *iPerf* https://iperf.fr/

[12] Gerhard Rieger, *socat - Linux man page* https://linux.die.net/man/1/socat

[13] ComputingforGeeks - Home for *NIX Enthusiasts, *How To Install Envoy Proxy on Ubuntu / Debian Linux*
https://computingforgeeks.com/how-to-install-envoy-proxy-on-ubuntu-debian-linux/

[14] Envoy Project Authors, *UDP proxy*
https://www.envoyproxy.io/docs/envoy/v1.14.1/configuration/listeners/udp_filters/udp_proxy

[15] Docker Inc, *Docker Documentation*
https://docs.docker.com/engine/install/ubuntu/

[16] The Kubernetes Authors, *Install Minikube*
https://kubernetes.io/docs/tasks/tools/install-minikube/

[17] *Tájékoztató a Műszaki Informatika Szak önálló laboratórium tantárgyainak 2008/9. tanév I. félévi lezárásáról a BME TMIT-en (VITMA367, VITMA380, VITT4353, VITT4330),* http://inflab.tmit.bme.hu/08o/lezar.shtml, szerk.: Németh Felicián, 2008. november 5.

[18] Wikipedia contributors, *Wikipedia:Academic use*, Wikipedia, The Free Encyclopedia, 2011 Nov 11. Available from:
http://en.wikipedia.org/w/index.php?title=Wikipedia:Academic_use&oldid=460041928

Itt jegyezném meg, hogy a tanulmányozott irodalmat hivatkozni kell a szövegben. Szükség esetén többször is. Az irodalomjegyzék célja (lásd a 3.1 fejezetet) ugyanis kettős[1]:

1. Az olvasó tájékoztatása, hogy a dokumentumban ki nem fejtett dolgoknak, a tudottnak vélt ismereteknek hol lehet bővebben utánanézni, így ott kell meghivatkozni az irodalmat [**?**, **?**], ahová az irodalom kapcsolódik.

2. Megmutatni a tárgyfelelosnek/konzulesnek az elolvasott irodalom mennyiségét

Javasoljuk, hogy a hallgatók tanulmányozzák, hogyan néznek ki a hivatkozások a villamosmérnöki/informatikai szakma vezető szakmai folyóirataiban megjelenő cikkekben. Ebben a témavezető is biztosan tud segíteni. A hivatkozás teljességére és egyértelműségére tessék ügyelni. Például, ha egy könyvnek több, eltérő kiadása is van, akkor azt is meg kell jelölni, hogy melyik kiadásra hivatkozunk. A webes hivatkozások problémásak szoktak lenni, de manapság egyre több az olyan dokumentum, ami csak weben lelhető fel, ezért használatuk nem zárható ki. Itt is törekedni kell azonban a pontosságra és a visszakereshetőségre. A weben található dokumentumoknak is van címe, szerzője, illetve érdemes megadni a letöltés/olvasás időpontját is, hiszen ezek a dokumentumok idővel megváltozhatnak.

A wikipédiás hivatkozások használata nem javasolt, mert a wikipedia másodlagos forrás. Tájékozodjuk a wikipédián, de aztán olvassuk el az adott oldalhoz megadott hivatkozásokat is. A wikipedián külön szócikk foglalkozik azzal, hogy miért nem szerencsés tudományos munkákban a wikipédiára hivatkozni [18].

Nem publikus dokumentumok hivatkozása nem javasolt és csak kivételes helyzetben elfogadható!

3.2. A csatlakozó dokumentumok jegyzéke

Lokális környezet konfigurációs fájlja.

```
admin:
  access_log_path: /tmp/admin_access.log
  address:
    socket_address:
      protocol: TCP
      address: 127.0.0.1
      port_value: 9901
static_resources:
  listeners:
  - name: listener_0
    reuse_port: true
    address:
      socket_address:
        protocol: UDP
        address: 127.0.0.1
        port_value: 4000
    listener_filters:
      name: envoy.filters.udp_listener.udp_proxy
      typed_config:
        '@type': type.googleapis.com/envoy.config.filter.udp.udp_proxy.v2alpha.UdpProxyConfig
        stat_prefix: service
        cluster: service_udp
  clusters:
  - name: service_udp
    connect_timeout: 0.25s
    type: STATIC
    lb_policy: ROUND_ROBIN
    load_assignment:
      cluster_name: service_udp
      endpoints:
      - lb_endpoints:
        - endpoint:
            address:
              socket_address:
                address: 127.0.0.1
                port_value: 5000
```

[1] Akárcsak ennek a fejezet hivatkozásnak, ami a `\aref babel` parancsot demonstrálja

Dockeres környezet konfigurációs fájlja.

```
admin:
  access_log_path: /tmp/admin_access.log
  address:
    socket_address:
      protocol: TCP
      address: 127.0.0.1
      port_value: 9901
static_resources:
  listeners:
  - name: listener_0
    reuse_port: true
    address:
      socket_address:
        protocol: UDP
        address: 172.17.0.1
        port_value: 4000
    listener_filters:
      name: envoy.filters.udp_listener.udp_proxy
      typed_config:
        '@type': type.googleapis.com/envoy.config.filter.udp.udp_proxy.v2alpha.UdpProxyConfig
        stat_prefix: service
        cluster: service_udp
  clusters:
  - name: service_udp
    connect_timeout: 0.25s
    type: STATIC
    lb_policy: ROUND_ROBIN
    load_assignment:
      cluster_name: service_udp
      endpoints:
      - lb_endpoints:
        - endpoint:
            address:
              socket_address:
                address: 172.17.0.1
                port_value: 5000
```

Kubernetes részben tárgyalt Deployment fájl.

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: envoy-deployment
  labels:
    app: envoy
spec:
  selector:
    matchLabels:
      app: envoy
  replicas: 1
  template:
    metadata:
      labels:
        app: envoy
    spec:
      containers:
      - name: envoy-container
        image: vidarhun/kubeupdenvoy:v1.0
        ports:
          - containerPort: 5000
            protocol: UDP
```

<A munka ezen beszámolóba be nem fért eredményeinek (például a forrás fájlok, mindenképpen csatolni akart forráskód részlet, felhasználói leírások, programozói leírások (API), stb.) megnevezése, fellelhetőségi helyének pontos definíciója, mely alapján a az erőforrás előkereshető – értelemszerűen nem nyilvános dokumentumok hivatkozása nem elfogadható.>