



## DIPLOMATERV FELADAT

**Szabó Csaba**

szigorló villamosmérnök hallgató részére

### Hálózati eszközök összekötése SDN hálózatokkal

A kommunikációs hálózatok egyes területein egyre inkább teret nyertnek a szoftvervezérelt hálózati megoldások (SDN). Ugyanakkor természetesen a meglévő, a kapcsoláshoz hardverszintű támogatást biztosító és ezért hatékony hálózati kapcsoló berendezések sem tűnnek el a szolgáltatók hálózataiból.

A hagyományos és SDN elemeket egyaránt tartalmazó hibrid SDN hálózatok elemeinek együttműködése szükséges, de a lehetőségek korlátozottak. A cél az együttműködés megvalósítása az OSPF dinamikus útvonalválasztó protokoll egyes elemeinek felhasználásával, és az elkészült funkció elemzése. A hallgató feladatai a következők:

- Tekintse át a hibrid SDN együttműködésre javasolt megoldásokat.
- Valósítsa meg az együttműködést statikus útvonalak alkalmazásával egyszerű hálózati esetekre (pl. stub network).
- Dolgozzon ki OSPF funkciókon alapuló együttműködési sémát.
- Implementálja a szükséges funkciókat az SDN vezérlőhöz kapcsolódó programként.
- Elemezze és értékelje a megoldás viselkedését kisebb, kézzel módosítható teszhálózatokban.

**Tanszéki konzulens:** Dr. Zsóka Zoltán, docens

**Külső konzulens:**

Budapest, 2019. március 8.

Dr. Imre Sándor

egyetemi tanár  
tanszékvezető

**Konzulensi vélemények:**

Tanszéki konzulens: ☐ Beadható, ☐ Nem beadható, dátum:

aláírás:

Külső konzulens:

aláírás:





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

# Hálózati eszközök összekötése SDN hálózatokkal

DIPLOMATERV

*Készítette*  
Szabó Csaba

*Konzulens*  
Dr. Zsóka Zoltán

2020. december 20.

# Tartalomjegyzék

<b>Kivonat</b>	<b>i</b>
<b>Abstract</b>	<b>ii</b>
<b>1. Bevezetés</b>	<b>1</b>
<b>2. A Hibrid SDN világ felé</b>	<b>3</b>
2.1. Klasszikus hálózatok . . . . .	3
2.1.1. Klasszikus hálózati protokollok . . . . .	4
2.1.2. Az OSPF protokoll . . . . .	5
2.1.2.1. OSPF Neighborhood adjacencies . . . . .	6
2.2. Software Defined Network . . . . .	8
2.2.1. SDN kontrollerek . . . . .	10
2.2.2. OpenFlow protokoll . . . . .	11
2.3. Virtualizálás . . . . .	13
2.3.1. Containerok . . . . .	14
2.4. Hibrid SDN hálózati logika . . . . .	15
<b>3. Technológiai lehetőségek és megoldások</b>	<b>17</b>
3.1. ESXi virtualizáció . . . . .	17
3.2. Linux . . . . .	17
3.3. Virtualizálás és Docker . . . . .	18
3.3.1. Docker networking . . . . .	20
3.4. OpenFlow protokoll alkalmazása . . . . .	22
3.4.1. Mininet és OpenvSwitch . . . . .	22
3.4.2. Az alkalmazott SDN controller: Ryu SDN framework . . . . .	25
3.4.2.1. Ryu alkalmazás írása . . . . .	27
3.5. OSPF helye az SDN logikában . . . . .	29
3.6. Hálózat monitorozása . . . . .	29
<b>4. Hibrid SDN hálózat implementálása</b>	<b>31</b>
4.1. SDN hálózat deploy . . . . .	31
4.1.1. Hálózati terv . . . . .	32
4.1.2. Cisco routerek konfigurálása . . . . .	35
4.1.3. ESXi virtualizálás és Docker containerok megtervezése . . . . .	36
4.1.3.1. Docker hálózati logika kialakítása . . . . .	39
4.1.4. SDN konfigurálás . . . . .	40
4.1.5. Névterek létrehozása virtuális host-ként . . . . .	40
4.1.6. Bash script . . . . .	42
<b>5. Hibrid SDN programozás</b>	<b>43</b>

5.1. Statikus kapcsolat kialakítása, próbaprogramok futtatása . . . . .	44
5.2. RyuOspf alkalmazás programozása . . . . .	44
5.2.1. RyuOspf main főosztály . . . . .	45
5.2.2. OpenFlow handle modul . . . . .	48
5.2.3. OSPF handle modul . . . . .	50
5.2.4. NetworkDiscovery modul . . . . .	53
5.2.5. RyuOspf alkalmazás futtatása . . . . .	55
<b>6. Eredmények</b>	<b>56</b>
<b>7. Összegzés</b>	<b>59</b>
7.1. Tapasztalatok . . . . .	59
7.2. Fejlesztési lehetőségek . . . . .	60
<b>Köszönetnyilvánítás</b>	<b>61</b>
<b>Irodalomjegyzék</b>	<b>62</b>

## HALLGATÓI NYILATKOZAT

Alulírott *Szabó Csaba*, szigorló hallgató kijelentem, hogy ezt a diplomaterv meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2020. december 20.

---

*Szabó Csaba*  
hallgató

# Kivonat

A Diplomaterv a *Hibrid SDN* paradigmának megismerését és megvalósítását tűzte ki célul Enterprise hálózaton belül IGP, azon belül is OSPF protokollal. Cél a klasszikus hálózat összekötése az újfajta gondolkodást megvalósító Software Defined Network hálózattal.

Az SDN világában újfajta megközelítést használnak szemben a klasszikus dinamikus protokollokkal, mint például az EGP és IGP protokollokkal, amik még nincs megvalósítva az újfajta logikában. A hagyományos, legacy routerek nagy része nem támogatják az SDN hálózatokban megvalósított kommunikációs formákat, ezért szükséges megoldást keresni az SDN világban is, hogy miképp lehetséges összekötni és klasszikus routing protokollt implementálni a kontroller northbound interfészén keresztül.

A Diplomamunka megvizsgálja a Hibrid SDN lehetőségeit, eddig megvalósított formáit, majd megvizsgálja az IGP routing protokoll jogosultságát az SDN világában, megvalósít egy OSPF implementációt, valamint megvizsgálja azt és kiértékeli. Megvizsgálja a különböző Hibrid SDN megközelítéseket, de elsősorban a megoldásra koncentrál és a Network Operating System logikát és annak megvalósítását tűzi ki elsősorban célul.

A megvalósítás során szóba kerül a virtualizálás, a klasszikus hálózatok protokolljai, az SDN rövid történelme és az átmenet a két világ között. A virtualizálásra az újfajta container-es megoldást nyújtja, az SDN fontos részeként szolgáló Open Flow protokollt felhasználó kontrollerek közül pedig a Ryu-t ismertetni meg részletesebben. Elemzi az OSPF logikáját és azáltal kibővíti az SDN által nyújtotta lehetőségeket.

# Abstract

This thesis research the Hybrid SDN paradigm and try to give a solution how to merge legacy network and SDN inside an Enterprise computer network with IGP, OSPF protocol.

In the SDN world there is a new prospective how to build network opposed to the classic legacy networks, and there are no solutions for IGP and EGP protocols, which is used for branch or economical network dynamic routing. The most of legacy routers doesn't support the SDN communication forms, so we have to find a way how can we implement legacy protocols into the SDN controller with a northbound API or how to connect the two logic into one system.

The thesis watches the possible solutions of Hybrid SND, check he validity of an IGP implementation, it will create OSPF communication inside SDN controller and verify it. It will consider different hibryd SDN solutions, but the main purpose should be the Network Operatin System logic solution and how to reach it.

During the research this thesis will consider the different virtualization solutions, the legacy network protocols and shortly the history of the SDN networks. It will consider how we can go through the different logic, how we can use containers as virtualization platforms and how Open Flow protocol works. During the thesis I will use Ryu OpenFlow controller and implement OSPF logic to extend the possibilities of SND.

# 1. fejezet

## Bevezetés

A számítógépes hálózatok és az internet megjelenése számos új technológiai innovációra adott lehetőséget. A kezdeti időszakban a telekommunikációs szakemberek nem gondolhatták, hogy mennyire széleskörűen alkalmazott és kiterjedt eszköz lesz, mekkora vívmányokat tud felmutatni és nem utolsósorban mennyire fontos része lesz a hétköznapiaknak akár az iparban, akár a civil szférában. Nem tervezhettek azzal sem, hogy mekkora adatforgalmat fog generálni, milliárdnyi aktív csomópont kell a kiszolgálásához és ezáltal hány eszközt kell beilleszteni a rendszerbe, azokat felkonfigurálni, menedzselni és kezelni, hogy létrejöjjön ez a komplex, bonyolult és robosztus rendszer, ami összeköti a számítógépeket, IoT eszközöket, embereket és rendszereket.

Azóta sokat változtak a lehetőségek és az igények. A jelenlegi megoldások az egyszerűség, a központi vezérlés, az automatizálás és okos funkciók felé haladnak. Egyre nagyobb kapacitású hardverek állnak rendelkezésre, megjelenhettek a virtualizálási technológiák, aminek köszönhetően hardverfüggetlen megoldások is születhetnek, segítve az átjárást a különböző környezetek között és nagyobb szabadságot adva a fejlesztők kezébe. Jelenleg többféle szoftveres megoldás is létezik a korábban hardverfüggő alkalmazásokra is, általános célú PC is képes futtatni szervereket, routereket, switcheket és más alkalmazásokat. A fejlődéssel megjelent a hálózatvirtualizálást és központosítást megvalósító technológia is: a Software Defined Network (SDN).

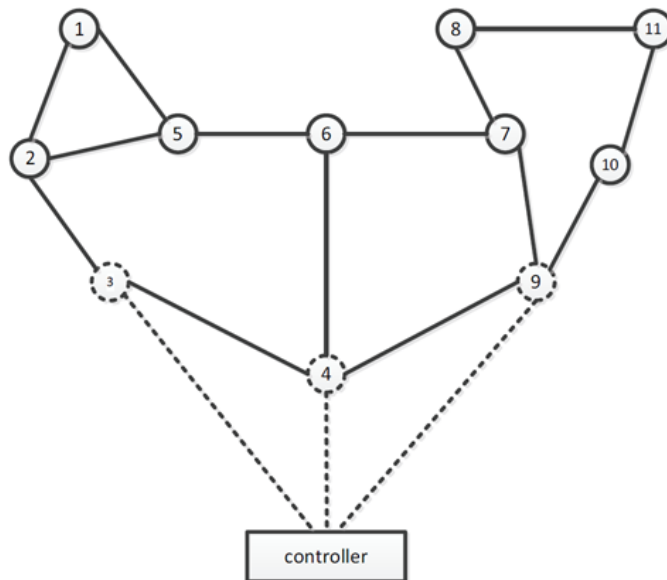
Az SDN egy innovatív és újfajta megközelítést vezet be a networking gondolkodásba. Felhasználja a virtualizálási technológiákat, szétválasztja az adatkapcsolati és vezérlési réteget. Központosítja a network managementet, ezáltal egyszerűbbé teszi a vezérlést, nem kell csomópontonként konfigurálni minden elemet, bevezetni a network OS fogalmát, átlátja a teljes hálózatot, könnyebben automatizálható, analizálható és kezelhető.

A diplomamunka kiindulása egy korábban elkezdett feladat volt. Megyeri Csaba 2017-ben Diplomamunkájában[1] hasonló témával foglalkozott és sok oldalát feltérképezte az ötletnek. Feladatom a Hibrid Software Defined Network témakörrel kapcsolatos fogalmak megismerése, megértése és vizsgálata és a megoldás implementálása.

Az új technológiák megjelenésénél mindig új kérdések merülnek fel és újfajta megoldások kerülnek elő. Viszont az átállás sosem könnyű és nem valósul meg pillanatok alatt. Az SDN megjelenésekor felmerült kérdésként, hogy mi lesz a hagyományos legacy eszközökkel, technológiákkal, hogy lesz az áttérés, hogy kommunikál a két rendszer egymással. Más elgondolásra épül a két technológia, más fajta kommunikációt valósítanak meg és más eszközöket használnak.

A Diplomaterv téma feldolgoz többféle hibrid SDN megoldási javaslatot, célja volt a valódi fizikai megoldásokhoz legközelebb álló hálózat megtervezése és megvalósítása virtuális tesztkörnyezetben, illetve a megtervezett hibrid SDN megvalósítása és elemzése a rendelkezésre álló eszközökkel és ESXi hypervisor alapú virtualizáción.





**1.1. ábra.** Hibrid SDN kialakítása[2]

A feladat során megismerhetünk sokféle modern hálózatokkal kapcsolatos technológiát, szakmai cikkek segítségével keresünk újfajta megoldásokat, illetve megvizsgálunk és felhasználunk meglévő eszközöket, hogy megvalósítsuk az átmenetet az SDN és a legacy hálózatok között. Megismerjük az új fogalmakat, megkeressük és kiválasszuk a használható alkalmazásokat.

Célunk a téma kidolgozása és megvalósítása, a korábbi diplomamunka el nem végzett feladatainak feldolgozása és megvalósítása, a saját környezetben megvalósított hálózatok átvitele ESXi alapú Hypervisorra, illetve a szimulált fizikailag elkülönített csomópontokkal implementált hibrid SDN létrehozása, beüzemelése és programozása.

## 2. fejezet

# A Hibrid SDN világ felé

Mielőtt *in medias res* belekezdenénk a virtuális hálózatok tárgyalásába tekintsük át, honnan is indult el ez a gondolat és miként jutottunk el mára oda, hogy a klasszikus fizikai elemeket várhatóan a virtualizált technológiák, a Cloud és az SDN cserélik le, vagy legalábbis jelentősen átalakítják a ma ismert rendszereket. Mára az internet, a Cloud és a virtualizálás a fogalma beépült a köztudatba, beszélünk róla, tudunk a létezéséről.

Bár az Software Defined Network széles körben kevsésbé ismert, legalább annyira innovatív és jövőbemutató. Az ipari alkalmazásokban, mint például az OpenStack alapú Cloud rendszerek hálózati részéért felelős egysége (nova) által gyakran felhasznált technológia. Azonban nem szabad összekeverni a kettőt, az SDN és a Cloud függetlenül álló paradigma annak ellenére, hogy az alapgondolatuk megegyezik.

### 2.1. Klasszikus hálózatok

A számítógépes hálózatok[3] mibenlétét mostanra már nem kell különösebben bemutatni. Az internet a hétköznapijaink részévé vált, használjuk a munkában, otthon, kikapcsolódásra, ügyintézésre, informálódásra és sok egyéb dologra is, rákapcsolódhatunk okostelefonról, laptopról, óráról, táblagépről, vagy akár már az autónkról is. A teljesség igénye nélkül vizsgáljuk meg alaposabban, mit is jelent a legacy networking, hogy eljuthassunk a fizikaitól a teljesen szoftveres világig.

A hálózat általános értelmezése szerint egymáshoz csatlakozó embereket, vagy dolgokat jelent. A számítógépes hálózat alatt egymáshoz csatlakozott számítógépek. Az IoT (Internet of Things) korszakában az asztali gépek, szerverek mellett nyomtatók, biztonsági kamerák, okos telefonok, szenzorok, kütyük és mindenféle fizikai eszközök csatlakoznak egymáshoz. Mivel milliárdnyi és még annál is több eszköztől beszélünk azért, hogy ezek nagy számban tudjanak csatlakozni egymáshoz és képesek legyenek kommunikálni egymással (ezt hívjuk scalability-nek - kiterjeszthetőség, bővíthetőség) szükség van hálózati eszközökre, protokollokra és technológiákra. Az alapvető hálózati funkciókat megoldásokat biztosító routerek, switchek és access pointok sem teljesen ismeretlenek egy laikus felhasználónak.

A hálózat lehet LAN (Local Area Network) és WAN (Wide Area Network): az első egy lokális, például otthoni hálózat, míg a második egy nagyobb kiterjedésű, LAN-okat összekötő hálózat.

Az internet rétegeinek elemzéséhez OSI modell 7 szintjét vették alapul, a TCP/IP-ben csak 5 rétegben gondolkodunk. Legalacsonyabb szintről indulva, az L1 réteg a fizikai szint, az L2 a Data-Link réteg (ethernet, switchek), az L3-as a Network (IP réteg, routing, routerek), az L4 a Transport míg az L5-7 az alkalmazási rétegek az Internet Engineering

Task Force (IETF) szabványaiban. A hagyományos hálózatokban a routerek és a switchek elsősorban L2 és L3 szinten működnek.

A forgalom elosztásáért és irányításáért a klasszikus internetes világban fizikai csomópontok felelnek. A protokollokat és a különböző funkciókat ezeken a csomópontokon egyenként kell konfigurálni. Könnyen belátható, hogy az adminisztráció és a nagyobb hálózat felkonfigurálása a hálózatkezelő szakembereknek számára nagyon sok munkát és erőforrást jelent, ráadásul sokkal több fizikai eszközt is. A virtualizálási megoldások sok újdonságot hoztak, de az IPv4 és IPv6 hálózatok nagy része még jelenleg is legacy csomópontokkal működik. Vizsgáljuk meg, milyen protokollok vannak és miket érdemes felhasználni a diplomamunka során.

### 2.1.1. Klasszikus hálózati protokollok

A protokollok leírják, hogy milyen típusú csomagot miképpen kell kezelni, mi a célja, mit módosít, hogyan kell konfigurálni, vagy éppen merre kell továbbítani. Rengeteg szabványt találunk ami nagyon sokféle konfigurációt és beállítást tud kezelni.

Az L2 szint a switchek világát tárja fel. Ilyenek például az *Ethernet*, *VLAN*, *Spanning Tree* vagy a *Etherchannel*. Ha SDN logikával gondolkodunk mondhatjuk, hogy ez az adatkapcsolati réteg. Az SDN, amint látni fogjuk elsősorban ezen szinten kezeli a csomagokat, de persze programozható L3 routingolás is.

Érdemes kitérni a *VLAN* (virtual LAN) protokollra is, ami – amint nevében is benne van – virtuális LAN-okra, subnetekre tudja bontani az L2-n megvalósított hálózatot is. Ha a switch egy portja több VLAN-t is kezel *trunk* portnak nevezzük.

A routerek magasabb, L3 szinten kezelik a csomagokat. Ebben a rétegben van a különálló hálózatokat összekötő routingolás, illetve az IPv4 és az újabb IPv6 protokoll. Az IP hálózatokban egymástól különálló és független alhálózatokat, subneteket lehet létrehozni, amik egymással kommunikálnak, ezen kívül adatvédelmi funkciókat valósítanak meg (pl. *VPN*), ipari alkalmazásokban jelennek meg (p. *PPP*), vagy akár az IPv4 korlátosságát kiterjesztő protokollokat alkalmaznak (pl. *NAT*). SDN szemmel ez a control plane.

A routing irányítja az egymástól függetlenül álló hálózatok között futó csomagokat. Az routing utakat manuálisan is lehet konfigurálni, ezáltal létrehozni statikus routot, vagy akár dinamikus routing protokollokat is használhatunk, ami alapján a router dönti el, merre küldi tovább a beérkező package-t.

Az dynamic routingnak két fő típusa van: az EGP (Exterior Gateway Protocol), ami kifejezetten WAN és még annál is nagyobb, ISP szintű backbone hálózati kapcsolatokat biztosít (pl. *BGP*), illetve az IGP (Interior Gateway Protocol), ami AS-en (Autonomous System) belül routingol, például egy ipari vagy nagyobb otthoni hálózaton belül köt össze különálló subneteket. A diplomamunka kifejezetten IGP hálózatokra koncentrál.

Az IGP-n belül találunk link-state (*RIP*, *EIGRP*) és datapath típusú (*OSPF*, *IS-IS*) routingolást. Manapság az egyik leginkább használt protokoll az *OSPF*, ezért ezt alkalmazó megoldás keresése volt a cél.

Fontos megemlíteni, hogy az SDN világban - bár nem a klasszikus routing, hanem úgynevezett active networking megoldást használ - implementálva vannak a legacy routing protokollok bizonyos elemei. Például az OpenDaylight (ODL) kontroller képes kezelni a *BGP*-t, vagy látni fogjuk, a Ryu ismeri a *BGP* és az *OSPF* headereket és csomagtípusokat is, de maga a kezelés nincs lekódolva. A *BGP* túl robosztus egy IGP hálózathoz, ezért a ezek a megoldások figyelmen kívül lettek hagyva.

### 2.1.2. Az OSPF protokoll

Az Open Shortest Path First[4] (OSPF) az egyik legnépszerűbb routing protokoll manapság a klasszikus legacy routerek világában. Egy széleskörűen támogatott link-state alapú IGP protokollról beszélünk. A link-state logika annyit jelent, hogy minden egyes router feltérképezi a teljes hálózatot, ezáltal ismeri a szomszédait, azok szomszédait, az egymás között működő linkeket, azoknak a sebességét, állapotát és egyéb adatokat. Minden egyes csomópont ugyan azt a hálózati képet tárolja a saját adatbázisában. Ezt a feltérképezést különböző OSPF csomagokkal oldja meg.

A protokoll első verziója 1987-ben jelent meg, amit '97-ben a 2-es verzió követett, amit az RFC3422 szabványban deklaráltak. 2008-ban megjelent a v3 is, ami az IPv6-os megvalósítása az OSPF-nek. Az elv hasonló, viszont ez már hatékonyan használja ki a 128 bites IPv6 nyújtotta előnyöket. Továbbiakban az OSPFv2-re hivatkozok OSPF néven is.

A link-state logikából fakadóan az OSPF erőforrás igényes, mivel minden egyes routernek tárolnia kell a teljes hálózati térképet, azt egymáshoz viszonyítani kell, majd ha ténylegesen mindenhol ugyan az a topology, abból kell kiszámolni Dijkstra algoritmussal a legrövidebb utat (SPF - shortest path first). Előnye ennek a megoldásnak, hogy a protokoll gyorsabban reagál a hálózat változására, mint például az EIGPR, vagy más distance vektor protokollok. A hálózati adatokat az LSDB-ben (Link State Database) lévő entry-k, az LSA-k (Link State Advertisement) tartalmazzák. Ezeket az LSA-kat cserélik a routerek, ha felismerik, hogy hiányzik valamelyik elem a saját adatbázisából.

Egy LSA tartalmazza a Router ID-t (RID), ami minden egyes routernek önálló azonosítója, az IP hálózatok címeit, a linkek költségadatait, timereket és egyéb információkat az egyes nodeokról és a hálózatról. A Router ID kiválasztása történhet manuálisan, vagy automatikusan a legmagasabb értékű aktív loopback interfész lesz, aminek híján a legmagasabb IP-jű aktív interfész kerül kiválasztásra. A cost, tehát a link költségét – ha nem manuálisan van beállítva – egy referencia érték (alapérték 100 000Kbps) segítségével kerül kiszámolásra. Ezt az értéket elosztja a link sebességével és abból kapott szám lesz a cost. Minél kisebb ez a szám, annál könnyebben lesz kiválasztva a link az SPF számolásnál. Az LSA-knál van egy timer, ami az entry elavulását számolja, ezért időszakonként adatbázist kell cserélniük a nodeoknak.

Típus	Név	Cél
1	Hello	Hálózatfelderítés és életbentartás.
2	Database Description (DBD)	LSDB összegzett adatainak a küldése. Arra van használva, hogy ellenőrizzék a routerek egymás között, hogy ugyan azt a képet látják-e a hálózatról. Csak LSA headereket küld.
3	Link-State Request (LSR)	Speciális, részletesebb LSA-t kér.
4	Link-State Update (LSU)	Válaszul az LSR-re küld LSA-t a szomszédjának.
5	Link-State Acknowledgement (LSAck)	Jóváhagyja az LSR-LSU cserét.

#### 2.1. táblázat. Az OSPF csomagok típusai

Az OSPF többféle csomagotípust használ (2.1. táblázat), hogy felvegye az új belépő nodeokat, a link-state logikának megfelelően jelezze, hogy aktív-e a link, vagy éppen valami gond van, esetleg ha csak egyszerűen LSA entry-t kér. Az kapcsolat felvételénél és tartásánál – ahogy a 2.1.2.1. alszakaszban látni fogjuk – első és talán legfontosabb üzenet a *Hello*. Ezzel kommunikálja a router, hogy belépett a hálózatba, illetve később azt, hogy még mindig életben van. Ha a *Hello*val tudomást szereztek egymásról a csomópontok *DBD*, tehát Database Descriptor üzenettel LSDB adatbázist cserélnék. Ha valami különbség van a két eszköz tudásában, *LSU* Link State Update üzenettel kér pontosabb

részleteket egy adott, vagy több LSAról, amire válaszul egy Link State Update (*LSU*) érkezik, hogy aztán egy *LSAck* acknowladge packet érkezésével a kérő tudomásul vegye a változást.

A *Hello* az LSA-hoz periodikus frissítést igényel. A *Hello*-hoz is tartozik timer, ami a node elavulási idejét kívánja számolni. A diplomamunka részeként íródó RyuOspf alkalmazásnak kezelnie kell ezeket az üzeneteket, hogy a többi router mindig lássa az SDN nodeokat.

Bár az alkalmazásunk számára nem lesznek lényegesek, de szükséges megemlíteni az OSPF több erősségét. A link-state logikának köszönhetően a csomópontok és a linkek terheltsége a nodeok számával exponenciálisan növekszik, nagyobb LSDB-k lesznek, több lesz az LSA és Hello csere, ami leterheli a networkot. Ennek megoldásaként létrehozhatunk különböző hálózati területeket, úgynevezett OSPF Area-kat. Az Area 0 minden esetben a backbone hálózat lesz, amihez minden Area-nak csatlakoznia kell legalább egy border routerrel. A teszhálózat szempontjából elég csak a backbone hálózatot implementálni.

Másik említésre méltó fogalom a Designated Router (DR) és Backup Designated Router működése. Amennyiben egy subnethez több router interfész is csatlakozik az üzenetek szintén eláraszthatják a hálózatot, ezért az alhálózaton belül a nodeok választanak egy úgynevezett vezetőt (DR) és egy helyettes vezetőt (BDR), hogy az update üzeneteket elég legyen csak neki küldeni, ő pedig szétossza az egész hálózatnak az adatbázisát.

LSA típus	Név	Egyéb
LSA Type 1	Router LSA	
LSA Type 2	Network LSA	DR által generált
LSA Type 3	Summary LSA	ABR summary route
LSA Type 4	Summary ASBR LSA	ASBR Location
LSA Type 5	AS External LSA	
LSA Type 6	Multicast OSPF LSA	
LSA Type 7	No-so-stubby area LSA	
LSA Type 8	External Attribute LSA for BGP	
LSA Type 9	Intra-Area-Prefix LSA Header	

## 2.2. táblázat. Az LSA egységek típusai

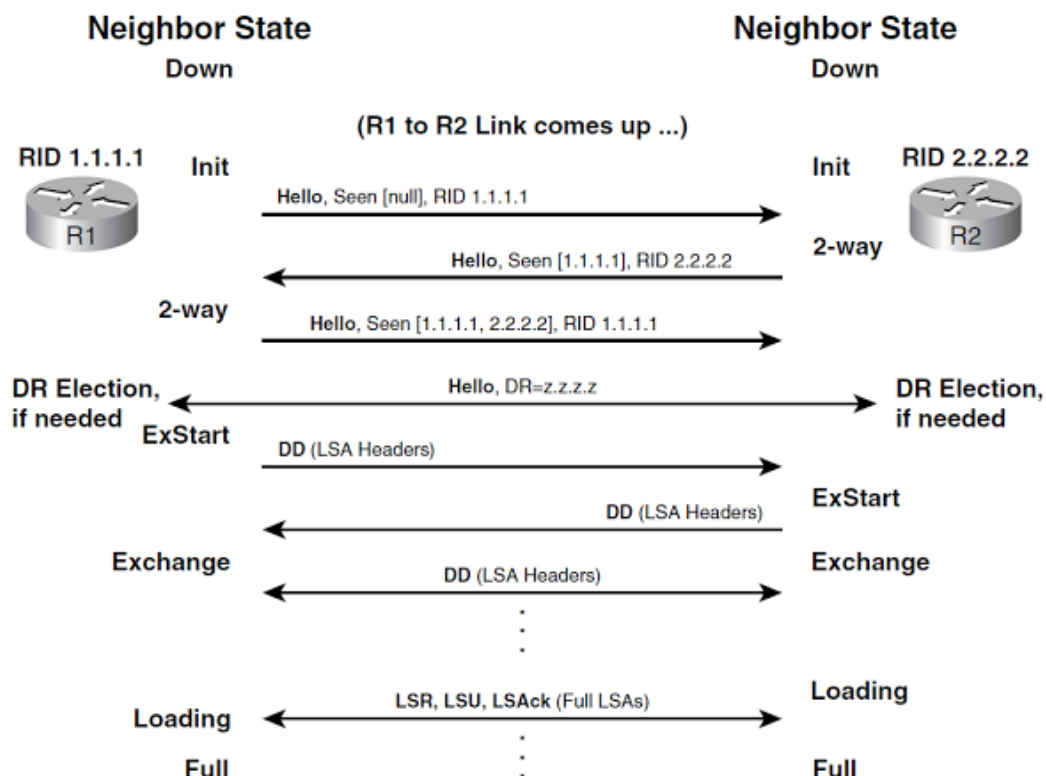
Az Multiarea és egyéb OSPF szolgáltatásai miatt 9 féle *LSA* típust is megkülönböztetünk. Egy single area hálózatban, ahol nincs DR/BDR választás elég lesz csak a *LSA Type 1* Router LSA-ra, ami a router specifikus adatokat osztja meg. Az *LSA Type 2* a DR jelzésére szolgál, a *Type 3-9* pedig a multi-area OSPF speciális routerek jelzésére van.

### 2.1.2.1. OSPF Neighborhood adjacencies

Az alkalmazásnak meg kell valósítani LSDB és adjacency felépítést, így vizsgáljuk meg részletesebben a folyamatot.

A 2.1. ábra segít a folyamat megértésében. A példában két router szerepel: R1, illetve R2. R2 már benne van az OSPF hálózatban, ehhez csatlakozik az R1. Belépéskor az *Init* állapotban van és küld egy *Hello* üzenetet a szomszédjának, aki szintén *Init*-be kerül. Ebben csak a saját *RID*-ja van, de nincsenek látott szomszédok az üzenetben. Az R2 válaszul szintén *Hello* üzenetet küld, amiben benne van a saját *RID*-ja és a szomszédok is. Ekkor lépünk be a 2-way státuszba. Az R1 küld még egy *Hellos*, már a teljes szomszédsággal.

Ezután szükség szerint elkezdődik a *DR - BDR* választás. Beléptünk az *ExStart* statetebe, ahol az *Exchange* statere kiválasztja a Master nodeot (magasabb *RID* lesz a Master) DBD cserével (R1 üzenetet küld, én leszek a master, válaszul R2 korrigálja). Most már megkezdődhet az LSDB-k cseréje immáron az *Exchange* módban. Ezt *BDB* üzenettel kez-



2.1. ábra. OSPF neighborhood felépülése két router között[4]

di meg a master node. Ebben az adatcserében csak az LSA headerek, ezáltal legfontosabb adatok kerülnek megosztásra, majd ha különbség van az LSDB-ben a **Loading** státuszban **LSR** csomagokkal lekérdezik a hibás LSA-t egymástól. **LSU**-val elküldi a részleteket, majd megerősítő **LSAck** üzenetekkel véglegesítik a kommunikációt, majd mindketten a **Full** statebe kerülnek.

A **Full** állapotban sem nem áll meg az adatcsere, a korábban említett módon a **Hello** üzenetekkel alap beállításon 10 másodpercenként updatelni kell az állapotukat, de legalább a Dead timer idejéig (default beállítás: 40mp). Amennyiben a Dead timer lejár, a szomszéd LSA entry-je törlésre kerül az LSDB-ből. Ha változik a hálózat, akkor LSU-val az új LSA-k is kiküldésre kerülnek, hogy mindig megegyezzenek a topology táblák.

## 2.2. Software Defined Network

A Software Defined Networking[5] a számítógépes hálózatok növekedésével, ezáltal a network management növekvő kihívásai mellett, illetve a Cloud és virtuális alapú rendszerek elterjedésével elkerülhetetlen logikai újítás volt. A létrejöttéhez hozzájárult a virtualizálási technológiák fejlődése, az OpenFlow megjelenése és a kialakult újfajta gondolkodásmód.

Az SDN kialakulásának és innovációjának mibenlétének megértéséhez menjünk vissza az időben. Mint azt láthattuk, egy nagyobb hálózat megtervezése és felépítése rengeteg munkát és konfigurációt igényel. Minden nodeot külön kell konfigurálni, megtervezni protokoll szinten, el kell gondolkodni, hol mit használjon a mérnök, eszközöket kell beszerezni, azokat telepíteni, fizikailag kell összekötni. Ez erőforrás és időigényes feladat.

A Software Defined Network kialakulásához az alábbi fő ötleteknek kellett felmerülnie: az *active network*é, a kontrol és adatréteg szétválasztásának, a központi irányításnak, a hálózat virtualizálásnak és az ezekből következő *network OS*-nek (Operating System)[6][7].

A hálózati forgalom elosztásáért, döntések meghozataláért és irányításáért a klasszikus internetes világban az egyes internetes csomópontok felelnek. Azért, hogy módosítsunk a jelenlegi konfigurációt és másképpen irányítsuk a forgalmat ezeket a nodeokat egyenként kell újraállítani és a már meglévő eszközöket és protokollokat használhatjuk fel. Az internet fejlődése során az megjelent az *active networks* fogalma, ami azt célozza meg, hogy ezeknek a nodeoknak a működését saját programokkal tudjuk befolyásolni. Ez könnyebbé és gyorsabbá teszi az új technológiák megjelenését, mivel bármilyen működést beállíthatunk. Viszont vannak ezzel a gondolattal problémák és kérdések is: az egyes nodeok tartalmazzák a kódot? Vagy talán a csomagokban van és a csomópontok bontják ki? Ráadásul a programok futtatása erőforrás igényes. Voltak próbálkozások a tisztán *active networks* megvalósítására (DARPA), de korábban nem lett sikeres implementációja. A gondolat viszont megjelent az SDN és az OpenFlow logikájában is.

Az SDN másik fő, és talán legfontosabb ötlete a *control plane* és *data plane* szétválasztása. A *control plane* a irányítás logikáját tartalmazza, számításokat végez, továbbítja a csomagokat. A klasszikus világban ez a routing. A *data plane* pedig az adatkapcsolatokat jelenti. A legacy routing nem ad közvetlen hozzáférést az adatréteghez. Ugyan azon a hálózaton zajlik a kontroll logika is, amin az adat áramlik, ez nem a legbiztonságosabb megoldás és erőforrás igényes is. Nehezebb a network management, egyenként kell konfigurálni a nodeokat, a meglévő routing protokollokra kell hagyatkoznunk, valamint lassabban lehet új technológiákat bevezetni. A fő előnyei a különválasztásnak: a gyorsabb innováció (kontrol logika nem hardver és protokoll függő), rálátunk az egész hálózatra és sokkal flexibilisebb a vezérlés. Ide kapcsolódik a Network OS fogalma is, ami annyit jelent, hogy a hálózaton úgy mond egy operációs rendszert futtathatunk, ami kernel szinten vezérlelheti az egészet.

A szétválasztást korábban többféleképpen is próbálták megvalósítani: először az IETF FORCES (2003) projekt keretében, ahol a switchek voltak programozhatók. Szabványosítás híján megállt a projekt. 2004-ben a BGP-t felhasználva fejlesztették ki a RCP-t (Routing Control Platform), ahol már egy központi kontrollert használtak a kontrol logikához. Volt Inter-AS kommunikáció is az AS-ek között, de a BGP miatt túl robusztus és korlátolt volt, azonban már közelített az SDN logikájához. Az ethane projekt (2007) már létező switchekre akartak új szoftvert tenni. Itt a hardverek korlátozták a fejlődést. Majd 2011-ben az Open Network Fondation (ONF) fejlesztésében megjelent az OpenFlow protokoll.

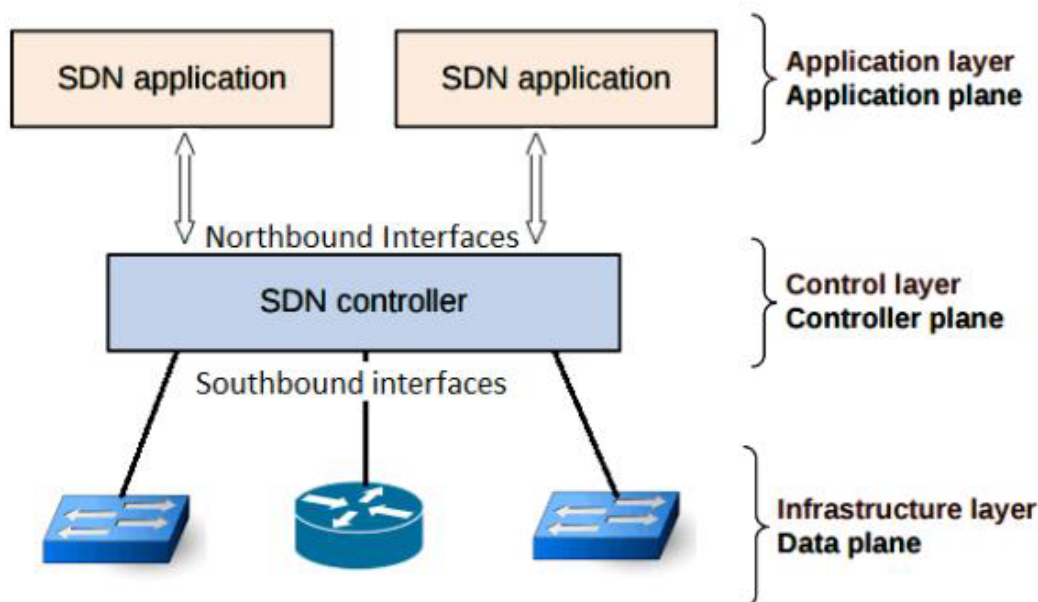
A hálózat virtualizálás a VLAN-oknál már szóba került. Nagy előnye, hogy adott fizikai környezettől függetlenül szoftveresen tudjuk elosztani az erőforrásokat és több alkalmazást tudunk futtatni. Mivel nem függ a fizikai eszközöktől, könnyebben programozható a hálózat és tetszőleges logikai hálózatot alakíthatunk ki ugyan azon az environmenten.

Még említésre méltó a 4D network, ami szétválasztja a döntést (decision), a terjesztési döntéseket (dissemination), felderítést (discovery), és az adatréteget (data).

A *Software Defined Network*[5][8] bevezeti programozhatóságot (active network logika), külön van választva az adatkapcsolati és vezérlési réteg, van egy központi vezérlője, virtualizálásnak köszönhetően szabadon alakíthatjuk ki a hálózatot, bevezethetünk VNF-eket (Virtual Network Function) és NFV-ket (Network Function Virtualization), illetve a hálózatot a kontrolleren keresztül szabadon programozhatjuk (Network OS).

A központosításnak köszönhetően rá lehet látni a teljes hálózatra, könnyebb a network management, saját alkalmazásokat lehet fejleszteni. Az adatkapcsolati réteg és vezérlési sík szétválasztásával lekerül a számítási feladat az egyes nodeokról, központi egységre végezhetjük el a számításokat.

De a megértéshez nézzük meg az SDN "Big picture"-jét, az architektúráját (2.2. ábra).



2.2. ábra. SDN Big Picture (vázlatos architektúra)

Az ábrán látható az adatkapcsolati és a kontroll réteg szétválasztása, valamint a hálózat programozhatósága. Az *data plane* réteg csak a csomagtovábbításért felel, ha egy bejövő packet-et nem tud kezelni, mert nincs benne a saját Forwarding táblájában akkor megkérdezi a feljebb lévő rétegeket, vagy eldobja a csomagot. A kontroll és adatkapcsolati réteget a southbound interfész köti össze. Itt a legelterjedtebb protokoll az SDN világában az OpenFlow, de vannak egyéb megoldások is, mint például OVSDB, NETCONF, SNMP. Az OpenFlow meghatározza a dataplane és a controlplane logikáját is.

Az SDN kontroller foglalja el a control plane szerepét. Az OpenFlow logika meghatározza a működését, de más protokollokat is támogathat. Itt zajlik a döntések kezelése, kalkulációk, forgalom irányítás, management. A központosításnak köszönhetően rálát az egész hálózatra, azt fel is térképezi és létrehoz egy topology adatbázist, amit több kontroller típus esetén akár egy GUI-n (Graphic User Interface) ki is lehet rajzolni és lehet szerkeszteni. A felderítést lldp-vel tudja még hatékonyabbá tenni. A kontroller tárolja a network node-ok adatait és tárol inventory-t, statisztikákat gyűjt, host tracking-ot folytat, tárolja az IP-ket és rengeteg más funkciót is tud kezelni. Egy SDN-ben egy, akár több



redundáns vagy független kontroller is lehet. A kontrollernek nem kell fizikailag egy helyen lenni sem a hálózattal, valamint funkció szerint önmagát is többfelé lehet osztani.

A northbound interfészen keresztül kapcsolódik az application layer-hez, ahol API-kat és alkalmazásokat lehet futtatni. A diplomamunka feladatában itt kell megvalósítanom többek között az OSPF vezérlést is.

Az SDN mivel virtualizált hálózat, ez azt jelenti, hogy bármilyen fizikai környezeten bármilyen logikai hálózatot megvalósíthatunk. Tehát például nem kell ugyanabban az épületen, városban vagy országban lennie a kontrollernek, ahol az összes node, futhat több fizikai eszközön is vagy ugyan azon a teljes hálózat (overlay és underlay network).

Az SDN további előnyökkel is rendelkezik. Képes támogatni multi-tenant, többfelhasználós alkalmazást, skálázható, szabadon kezeli a címeket, jó QoS-t biztosít (Quality of Service), megkönnyíti a már említett módon a traffic engineering feladatokat, szabadon módosítható és irányítható a routing és switching és megkönnyíti az OAM (Operations, Administration, Maintenance) feladatokat. A már említett módon az adatkapcsolati node-ok L2 szinten kommunikálnak, de a kontrollernek köszönhetően L3 kezelése is biztosított, vagy akár magasabb szintű elemeket fel is lehet dolgozni.

### 2.2.1. SDN kontrollerek

Mára már többféle SDN kontrolleres megoldás is létezik, így választani kell, hogy milyen alkalmazásra melyiket érdemes használni. Mindegyiknek van erőssége és hátránya. Döntési szempontok lehetnek a programnyelv, ami a teljesítményét is befolyásolja, a tanulhatóság, a felhasználói bázis és támogatás, illetve, hogy mire fókuszál az adott szoftver (Southbound API, Northbound API, OpenStack támogatás, oktatási, kutatási vagy ipari célok). Nézzünk meg néhány ismertebb és megvizsgálásra érdemes kontrollert, mint például a: NOX/POX, Ryu, Floodlight, ONOS, illetve OpenDaylight kontrollereket[9].

	NOX	POX	Ryu	Floodlight	ODL	ONOS
Language	C++	Python	Python	Java	Java	Java
Performance	Fast	Slow	Slow	Fast	Fast	NA
Distributed	No	No	Yes	Yes	Yes	Yes
OpenFlow	1.0 (-1.3)	1.0	1.0-1.5	1.0	1.0-1.5	1.0-1.5
Multi-tenant Clouds	No	No	Yes	Yes	Yes	Yes
Learning Curve	Moderate	Easy	Moderate	Steep	Steep	Steep

### 2.3. táblázat. SDN kontrollerek összefoglaló táblázat

Az első generációs SDN kontrollerek közé tartozik a NOX[10], ami még az OpenFlow 1.0 megjelenésével debütált 2008-ban, de mára a későbbi verziókat is támogatja. Egyszerű, gyors, könnyen tanulható, de kevés funkcionalitással rendelkező framework. A programozásakor regisztereket vezet be, amik eseményeket generál, hogy a programozó ezeket az eseményeket tudja kezelni és így tudja irányítani a forgalmat. A NOX C++-ban íródott, de készült egy python alapú változata is: a POX[11]. A nyelv miatt ez egy lassabb kontroller, viszont az egyik legegyszerűbben tanulható SDN kontroller, az technológiával ismerkedőknek javasolt ezzel kezdeniük.

A Ryu[12] egy python alapú open source framework, ami az OF mindegyik jelenlegi verzióját támogatja. Beilleszthető az OpenStack Cloudba, moduláris és inkább egy keretrendszert ad, mint egy kész szoftvert. A Network Operating System logikára fókuszál, sokféle megoldásra ad lehetőséget. Hátránya a sebessége programozási nyelvből fakadó sebessége.

A Floodlight[13] szintén egy open source kontroller, Java nyelven íródott. Az OFv1-et támogatja, jó dokumentációval rendelkezik, támogatja a REST API-t és elsősorban production-ra koncentrál, tehát hogy eladható termék legyen. Támogatja az OpenStack és multi-tenant cloudokat, jó a teljesítménye, sokféle technológiát támogat, viszont nehezen tanulható.

Az ipari világban talán a leginkább alkalmazott SDN kontroller az OpenDaylight[14] (ODL). Elsődleges célja az ipari felhasználás és a technológiák széleskörű támogatása, ennek köszönhetően kifejezetten robusztus. Open-source, Java nyelven íródott, gyors sokféle modul és megírt alkalmazás van készen hozzá. Támogatja az OpenStack és egyéb Cloud technológiákat, de robusztussága miatt komplex és nehezen tanulható.

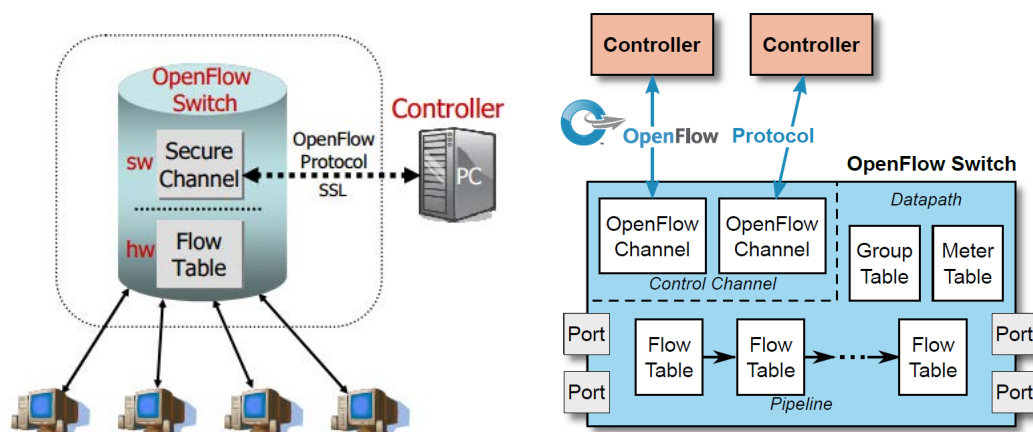
Mivel az Open Network Foundation kezeli az OpenFlow protokollt érdemes megemlíteni az általuk fejlesztett kontrollert is: az ONOS-t[15] (Open Network Operating System). Szintén Java-ban íródott és elsősorban az SDN alkalmazások fejlesztéséhez nyújt hatékony platformot. Jellemző rá az elosztott központi felépítés, támogat többféle northbound és southbound API-kat, illetve ez is moduláris rendszer.

Megyeri Csaba a diplomamunkájában az ODL kontrollert választotta[1], ezáltal a jelenlegi diplomamunka is annak a megismerésével indult, viszont a rendelkezésre álló technológiák korlátai és az azóta még robusztusabbá váló és nehezen érthető dokumentáció szempontjából inkább a könnyebben megérthető, kevesebb kész megoldással, de szabadabb programozási lehetőségekkel rendelkező Ryu kontrollert választottam a hálózat vezérlésére.

A kontrollereknél, de az SDN tárgyalásánál is szóba került a OpenFlow protokoll. A jelenlegi kontrollerek elsősorban ezt a protokollt használják ki, illetve a vezérlési logika is erre épül. Nem árt alaposabban tárgyalni, hogyan működik és mi is ez.

### 2.2.2. OpenFlow protokoll

Az OpenFlow[16] (OF) eredetileg egy kutatásokra fejlesztett eszköz, ami számos protokollt támogat, akár újakat is hozzá lehet adni a környezethez. Ethernet kapcsolókon alapul, belső forgalmi táblával. Az általános működési elvekre épít, így a különböző gyártók különböző megoldásai nem zavarják a működését.



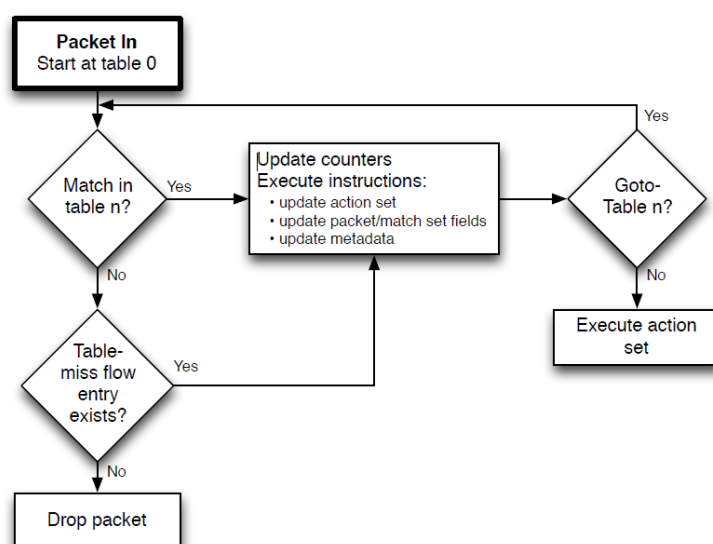
**2.3. ábra.** OpenFlow switch és logika architektúrája és annak fejlődése az OFv1-től és az OFv1.5-ig

Az OpenFlow elsősorban az SDN southbound leíró protokollja, secure vagy control channel-en keresztül tartja a kapcsolatot az OpenFlow Switchek és a kontroller között, de logikája ennél kiterjedtebb. Az OpenFlow sok mindent definiál, így van csomagformátuma, amik alapvetően a hálózat irányításáért és a network management kialakított eszközeit

vezérli és segíti. A protokoll az L2-es SDN csomópontokban kialakít egy irányítási táblát, úgynevezett FlowTable-t, amibe a flow entryken keresztül matcheket és actionöket rendel, hogy megmondja, miként kezelje a bejövő csomagokat. Az OpenFlow a hivatalos megfogalmazásban van a legegyszerűbben leírva: az OpenFlow célja a hálózat viselkedését irányítja[5]. Ha egy a vSwitch nem talál a bejövő üzenethez entryt felküldi a kontrollernek, hogy döntse el, mit kell tenni vele. Az eldönti a forgalomirányítás módját, visszaküldin és a vSwitch (vS, OVS) végrehajtja.

Az OpenFlow alapú switchekben, ahogy azt a 2.3. ábra is mutatja – bár a korábbi verziókban egyszerűbb volt a logika[17] – az 1.5-ös verzióban 3 fő részre osztja a az architektúrát: Pipeline rész, Datapath rész és a Control Channel. Az csomag beérkezésekor egy OpenFlow porton kerül bele a rendszerbe, majd sorban a FlowTable elemeivel próbálja összehasonlítani, hogy milyen parancs vonatkozhat rá. A Group tábla actionöket tárol, amik a csomag kezelését írják le. A OpenFlow Channel pedig a kontrollerrel tart kapcsolatot.

Nézzünk meg egy Flow táblát közelebbről. Mint láthatjuk, egy OVS több Flow táblával is rendelkezhet. Ez a beérkező csomagotat rendeli alá úgynevezett match folyamatnak, ami hogyha sikerrel jár, tehát a switch rendelkezik a beérkező csomagra vonatkozó Flow Table entryvel, akkor a benne található actionnal kezeli. Ha nincs "match" tovább küldi a kontrollernek.



2.4. ábra. Az OpenFlow 1.3-as verziójának matching táblázata

A matching részletes folyamata a verziókkal sokat bonyolódott, a részletes leírást megtalálhatjuk az egyes verziók dokumentációjában. Ami viszont érdekes, hogy ebben a mezőben sokféle feltételt meg tudunk adni. Csak pár példát nézzünk: **switch port**, **MAC src**, **MAC dst**, **Eth type**, **VLAN ID**, **IP src**, **IP dst**, **IP prot**, **TCP sport**, **TCP dport**, **VLAN tag** és az OpenFlow fejlődésével még több is elérhető. Ezeket a feltételeket az egyes FlowTable entry-k tárolják.

Az *action* lehet többek között Forward All (továbbítás az összes portra), Forward Controller (továbbítás a kontrollernek), Forward Local (egy szomszédos switchnek küldje tovább), Forward Table (a Flow Table-ben legyen módosítás), Forward In port (küldje vissza a bejövő porton) vagy Drop (eldobás - ha nincs match alapvetően ez történik). Lehet Modify (tehát módosíthatja a packet headerjét, pl. VLAN ID, set destination IP),

Enqueue (sorba helyezés) egy adott porton. Az újabb verziókban még többféle action is elérhető.

Secure Channel vagy Control Channel-en hallgatnak az vSwitchek a kontroller felé és küldenek fel OF üzeneteket.

Lehet csoportosítani is az actionokat, amit action set-nek hívunk. Egy listát ezekkel Group-nak hívunk. A Groupokat tárolja a Group Table. Az action group-ban többféleképpen lehet lefuttatni az actionokat: összeset egyszerre (execute all action), vagy az indirect group-okban egy meghatározott sorrendben.

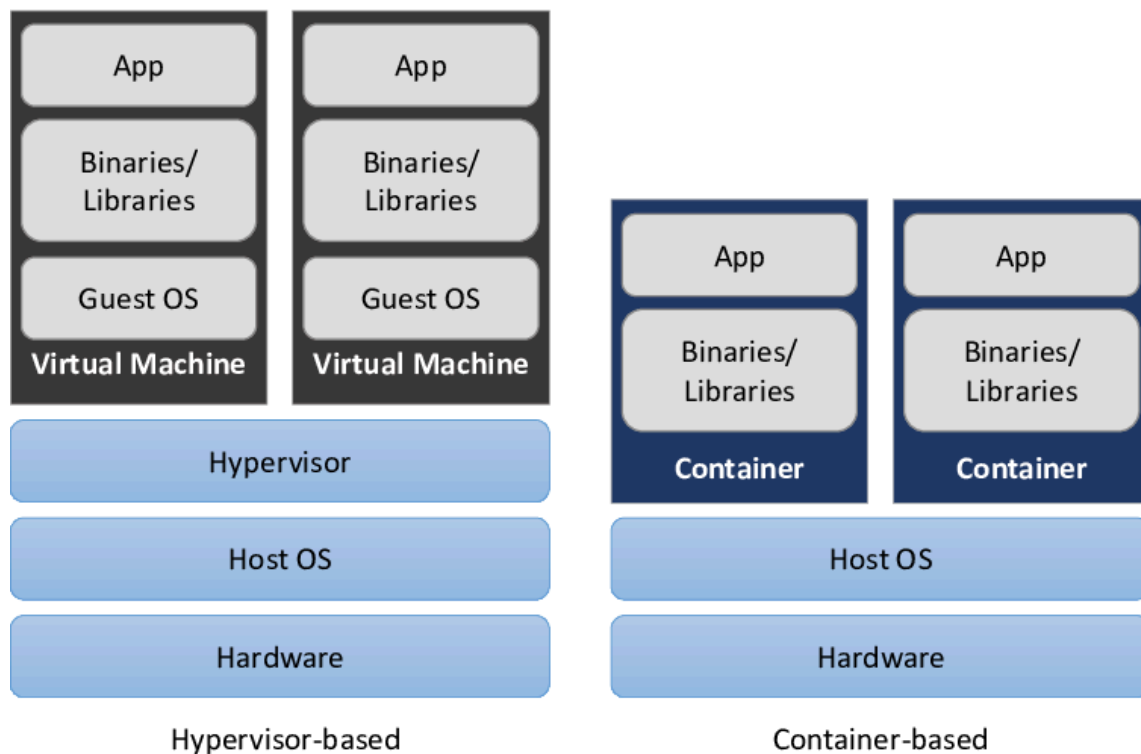
A különböző csomagokat és a hozzájuk tartozó protokollokat három fő csoportba osztjuk: *Asynchronous* csomagok, *Controller-to-Switch* csomagok és *Symmetric* csomagok. Az asszinkron esetben a kontroller eventeket, eseményeket generál. Ilyenek például a Packet In Message, Flow Removed Message, Port Status Message és a Error Message (1.5-ös verzióban még: Controller Role Status Message, Table Status Message, Request Forward Message és a Controller Status Message). A feladat megoldásához a Packet In Message lesz igazán fontos. A szimmetrikus üzenetek az SDN működéséért és kezeléséért felelős funkciókat tölti be (pár példa a teljesség igénye nélkül: Hello, Echo Request, Echo Reply, Exprimenter és az 1.5-ös verzióban Error Message). Programozás szempontjából fontosak lesznek a *Controller-to-Switch* üzenetek: ezek közül is elsősorban az érdekes a Packet-Out message, illetve a Modify State üzenet, azon belül is a Flow\_Mod csomag, de itt találhatunk Handshake, Switch Configuration, Flow Table Configuration, Multipart Messages és többféle switch működését vezérlő csomagstruktúrát.

## 2.3. Virtualizálás

Már érintve volt a virtualizálás témája, azonban érdemes megvizsgálni a témát alaposabban is. Ez a technológia a fizikai processzorok és számítógépek teljesítményének növekedésének köszönhetően jött létre. Azáltal, hogy az alkalmazások már nem terhelik le a teljes számítási kapacitást el lehetett kezdeni azon gondolkodni, hogy miként jöhet létre egy teljesen hardver független, de fizikainak tűnő teljesen szoftveres megoldást. A hardvertől leválasztott alkalmazás számtalan előnyt kínál, többek között azt, hogy bármilyen környezetben ugyan úgy tud működni. Virtualizálni lehet hálózatot, számítógépet, adattárolót, szervert, vagy bármilyen fizikai alkalmazást, hogy környezettől független megoldás szülessen.

A virtualizálás alatt is természetesen van egy, vagy több fizikai eszköz, amit úgy nevezünk, hogy *host*. Ez a host foglalja magában a virtualizált platformot. Ezen fut egy úgynevezett *hypervisor*, ami egy olyan szoftver, amely a rajtuk futó VM-eket (virtual machine) kezeli, azoknak erőforrás kiosztását és életciklus managementjét. A hypervisoron futó virtuális gépek teljes értékű operációs rendszereket és szoftvereket futtatnak (például Linux, Windows, Cisco IOS, stb.). Úgy is hívhatjuk őket, hogy szoftver alapú számítógépek, belülről egy fizikai rendszernek tűnnek.

A hypervisoroknak két típusa van: 1-es típus az úgy nevezett bear metal hypervisor, ami közvetlenül a hardveren fut és ezáltal alacsonyabb szinten kezeli a VM-eket, illetve a 2-es típusú hosted, ami egy már a hardveren futó operációs rendszeren fut és onnan osztja tovább az erőforrásokat virtuális gépeknek. A bear metal megoldás hatékonyabb és jobb az erőforrás kihasználása, mert nem futtat egy host OS-t is a guest OS-ek mellett. A feladat során rendelkezésre álló fejlesztőkörnyezet egy ESXi típusú, vmware által fejlesztett 1-es típusú virtualizációs platform. A második típusban elterjedt technológiák például a VMware, VirtualBox, vagy a linux alapú KVM.



**2.5. ábra.** Virtualizálási és container technológia architektúrája

A virtualizálás előnyei között van a költsécsökkentés, nem kell különböző alkalmazásokra és környezetekre mindig újabb fizikai hardvert vásárolni. Második előnye az agilitás és a gyorsaság, tehát új alkalmazásnál nem kell teljesen újragondolni a környezetet, hanem rögtön fel lehet telepíteni a meglévő rendszerre. Gyorsabban és hatékonyabban lehet automatizálni a deploy-t. Lehet csökkenteni a leállással felmerülő idővesztést is a technológiával, tehát ha a host megáll, a VM-eket gyorsan újra lehet telepíteni egy backup hostra is. Multi-tenant (több felhasználó) esetben hatékonyabb az erőforrás kezelése, mivel ha egy VM csak egy kis részét használja a neki rendelt CPU kapacitásnak egy másik VM többet tud használni, tudja aktivitás szerint felügyelni a managementet.

A hálózati virtualizálás hasonló logikán alakul: a hálózati döntéseket és környezetet virtualizálja. Egy vagy több logikai hálózati topológia ugyan azon az infrastruktúrán többféle módon tud megjelenni, virtuális alkalmazások futhatnak környezettől függetlenül. Többféle logikai router és node futhat (guest OS), hatékonyabb erőforrás elosztás és függetlenek azt a fizikai CPU-tól, memóriától vagy sávszélességtől és ennek köszönhetően könnyen lehet módosítani a topológiát.

### 2.3.1. Containererek

A containerek mára egyre fontosabb szerepet töltenek be. Ez a virtualizálásnak egy alacsonyabb szintű, de hatékonyabb módja, megjelenik a Cloudban, virtuális hálózatok és egyáltalán a virtuális alkalmazások használatakor. Sokaknak a Docker, vagy a Kubernetes ugrik be erről, de a technológia a linuxban korábban megjelent már.

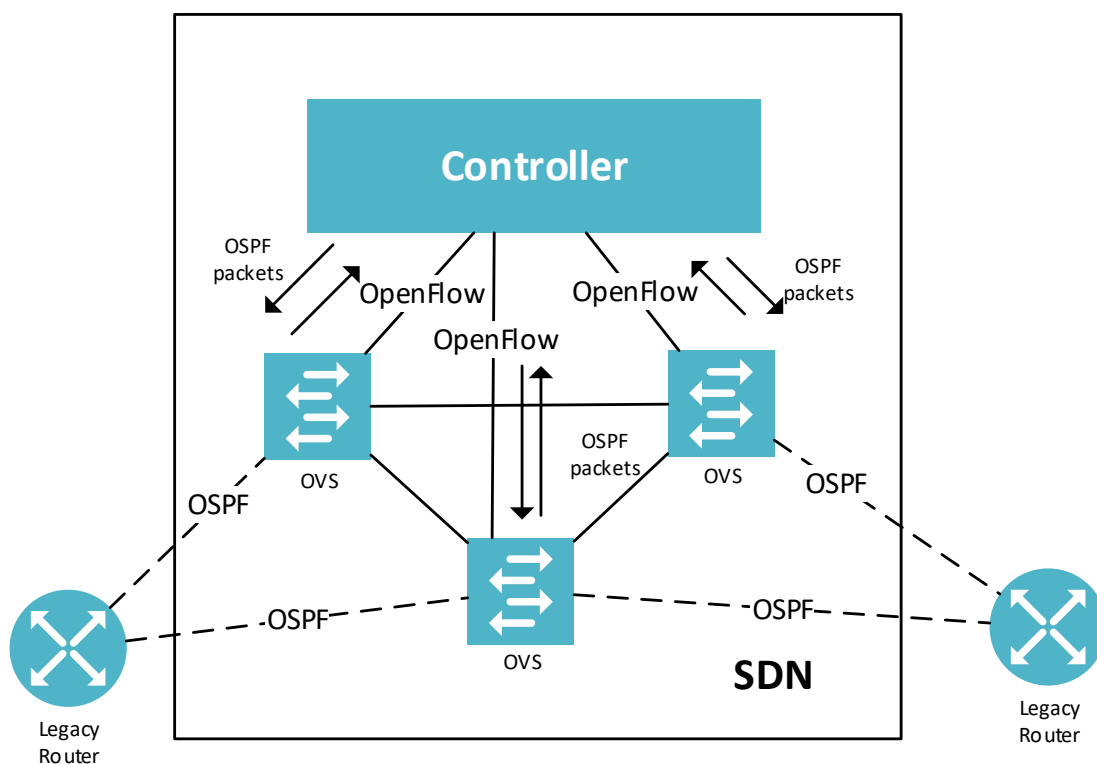
A fő különbség a klasszikus virtualizálás között, ahogy a 2.5. ábra is mutatja, hogy itt nem egy teljes rendszert virtualizálunk, hanem egyes alkalmazásokat és szolgáltatásokat zárjuk keretbe. Ezáltal kevesebb erőforrásra van szükség, nem szükséges teljes OS-t futtatni virtuálisan, elég csak egyes részeit, fájl rendszerét és az alkalmazásnak szükséges funkciókat.

A containerek szintén különálló egységekként futnak, alattuk is host OS van, csak nem hypervisor biztosítja a működésüket, hanem úgynevezett Runtime Engine-n (pl. Docker Engine), illetve a container library-k a host fájlrendszerének egy különálló és elszeparált másolata.

## 2.4. Hibrid SDN hálózati logika

Az új technológiák megjelenésénél - mint ahogy korábban is láthattuk - nem megy gyorsan és zökkenő mentesen, egyik napról a másikra az átállás. Van egy átmeneti időszak, amikor meg kell ismerkedni az újjal, de a működés biztosítása miatt szükség van még a régi rendszerre. A diplomamunka feladata, hogy megvalósítson egy olyan megoldást, ahol a régi és az új megoldásokat valamiképpen ötvözve, egy hibrid működő rendszert hozunk létre és amiben ki tudjuk használni a régi és az új technológia előnyeit.

A hibrid hálózati logika kiválasztásánál segítségünkre van a korábban megírt dolgozat[1]. A benne részletezett megoldásokra csak említés szintjén térek ki, a jelenlegi feladtnál az általa nyújtott megoldási javaslat alapján indultam el.



**2.6. ábra.** A legacy network és az OpenFlow alapú SDN összekötése hybrid SDN-ben.

A hibrid megközelítésnél két irányból közelíthetjük meg: egyrészt, ha minden hagyományos csomópontot felszereljük OpenFlow protokollal, hogy képes legyen kommunikálni a kontrollerrel, ezáltal az egyes node-ok kapják meg és kezelik mind a két technológiát (hibrid csomópontok). Másik irány pedig, hogy a kontrollernek tanítjuk meg a klasszikus megoldásokat. Mivel az SDN egyik alapja a Network OS, vagyis a programozhatóság és a sokszínű irányítás, valamint az egyes nodeokat módosítani sokkal több munkát, erőforrás igényel és sok routernél egyszerűen nem áll rendelkezésre megoldás egyértelmű, mit ér-

demes választani. Az SDN-ben és a kontrollerben kell megoldani a klasszikus protokollok kezelését.

Ezt is többféleképpen valósíthatjuk meg. A *Bridge* architektúrával[18], ami VLAN szerű headert tesz rá az ethernet csomagokra, amit SDN kontrollerből tudunk módosítani. Az *MPLS*-el (Multiprotocol Label Switching), ami OpenFlow segítségével dolgozza fel a csomagokat és alakítja ki a forwarding logikát egy Open vSwitchben. Erre irányult egy ISP (Internet Service Provider) hálózatban megvalósított OSHI (Open Source Hybrid IP/SDN) nevezetű kutatási projekt[19], ami több részletét is kidolgozta a logikának. Ezt a megoldási formát visszük tovább a feladatban. A *Hybridtrace* pedig elsősorban egy SDN és klasszikus IP világban is használható hálózatmonitorozási megoldást ad.

A feladat az OSPF protokoll beillesztése lesz MPLS módon a a kontroller döntési rendszerébe. A gyakorlatban, ahogy a 2.6. ábra is mutatja úgy néz ki, hogy az OVS node-oknak egy legacy routerből érkezik egy OSPF csomag. Azt - mivel a vSwitch nem tudja kezelni - OpenFlow-ba csomagolva felküldi a kontrollernek, ami megvizsgálja az OSPF package-t, beilleszti az általa tárolt topológiába, majd kiszámolja Dijkstra-val a Shortest Path utakat, majd az OVS-nek visszaküldi a next-hop nodeot, amit a vSwitch beépíthet a Flow táblájába, hogy a következőleg érkező csomagokat továbbítani tudjon OSPF routing szerűen. A külső routerek úgy látják, mintha az egyes SDN nodeok OSPF routerek lennének és eltárolja annak az adatait. A kontrollernek tudnia kell kezelni a neighbor adjacency felépítését, a bejövő és kiküldendő csomagokat, illetve az OSPF topology adatbázis legfontosabb adatait is tárolnia kell (LSDB).

## 3. fejezet

# Technológiai lehetőségek és megoldások

Miután megismerkedtünk az alapfogalmakkal keressünk konkrét megvalósításokat. A fejezet célja az általánosságban megtárgyalt hálózati, virtuális és SDN megoldásokra és logikákra konkrét alkalmazásokat kínálni és azoknak a megoldásait részletezni.

Először tárgyalásra kerülnek a diplomamunka során rendelkezésre álló technológiák és fejlesztőkörnyezet, majd szó lesz az ezekre épülő és felhasználható megoldásokról, az SDN és OpenFlow-val kapcsolatos eszközökről és végül a hálózati monitorozásról.

### 3.1. ESXi virtualizáció

A fejlesztés során egy ESXi[20][21] típusú, VMware által fejlesztett, Linux alapú bare-metal hypervisor állt rendelkezésre. Mint korábban említésre került, ez közvetlenül a hardveren fut, nem kell alá operációs rendszer, ezáltal hatékonyabban használja ki az erőforrásokat, nem kell az alkalmazás szempontjából érdektelen Host OS futtatására is pazarolni a hardver kapacitását.

### 3.2. Linux

A hypervisoron a GNU[22] (GNU's not Unix) alapú, Unix szerű Open Source Linux disztribúció, az Ubuntu Server 20.04[23] stable verzió futott virtuális gépekként. Az Ubuntu az egyik leghasználtabb és legelterjedtebb Linux disztribúció, alapja a Debian. A Command Line Interface-ben (CLI) az apt repository-n lehet csomagokat és alkalmazásokat telepíteni. Az Ubuntu Servernek nincs grafikus felülete.

Mivel eleinte kevés storage volt a VM-hez rendelve, később ki kellett bővíteni, ezért virtuális tárolók lettek létrehozva LVM (Logical Volume Management) groupokkal, ezáltal a teljes operációs rendszer egy tárhelyet lát több helyett. Az LVM-nek három szintje van: a physical volume (PV), ami a legalacsonyabb szinten hoz létre block device node-ot tehát fizikainak tűnő nodeokat, a volume group (VG) PV csoport, amik a következő, LV szint containereiként működnek, míg az legmagasabb szinten a logical volume (LV) pedig logikai partícióként működik. Ezen fut maga az operációs rendszer.

A Linuxon lehet scripteket futtatni, a saját CLI parancsaiból vannak összeállítva. Ezeket bash scriptnek hívjuk. A commandokból álló kódot egy .sh fájl-ba kell menteni, amit később parancsként le lehet futtatni. Ez hatékony segítséget nyújt többszöri telepítés, deploy automatizált megoldására.

Létre lehet hozni az operációs rendszeren belül egymástól elkülönített hálózati részeket, úgynevezett namespace-eket. Ezekben a namespace-kben önálló IP, routing és interfé-



szek alakíthatók ki, önálló networking viselkedése van. Ezt fel lehet használni egy különálló host szerű node létrehozásához.

### 3.3. Virtualizálás és Docker

A diplomamunka során többféle virtualizálási technológia is ki lett próbálva. Tesztkörnyezet felépítéséhez futottak Windows-os VirtualBox[24] hypervisoron Linux VM-ek, ESXi rendszerét alapul adó vmware[20] VM-ek. Nagyon hasonlóan működik mind a kettő, fő különbség viszont, hogy a VirtualBox open source, szabadon használható és letölthető virtualizálási technológia ellentétben a vmware alkalmazásaival. A vmware használ egy vmrc (vmware remote consol) protokolt, ami az ESXi szerverhez biztosít távoli elérhetőséget, ha nem áll rendelkezésre ssh kommunikáció.

Mode	VM→Host	VM←Host	VM1↔VM2	VM→Net/LAN	VM←Net/LAN
Host-only	+	+	+	–	–
Internal	–	–	+	–	–
Bridged	+	+	+	+	+
NAT	+	Port forward	–	+	Port forward
NATservice	+	Port forward	+	+	Port forward

**3.1. ábra.** VM Networking különböző alkalmazások szerinti ajánlásával a VirtualBox dokumentációjából

A VM-ek hálózati elérését hasonló módon oldja meg a két technológia. Lehet *not attached* módban, ahol a VM nem rendelkezik hálózati kapcsolattal. *NAT* módban a host IP-jét lefordítja egy saját subnet IP-re, ezáltal kilát az internetre, de kívülről nem lehet elérni a hálózatot. Ennek a NAT-nak módosíthatjuk is a szabályait. A *Bridged networking* egy úgynevezett bridg-eel kapcsolódik a host gép egy interfészére, így közvetlenül használja azt. *Internal networking*nél csak egy belső hálózatot hoz létre, kívülről nem látszik. A Host-only módban pedig a host beállításait használja, ott hoz létre loopback szerű interfészeket. Ez hasznos lehet, ha a VM-ek csak egymással szeretnének kommunikálni, de a külvilágot nem kell látniuk.

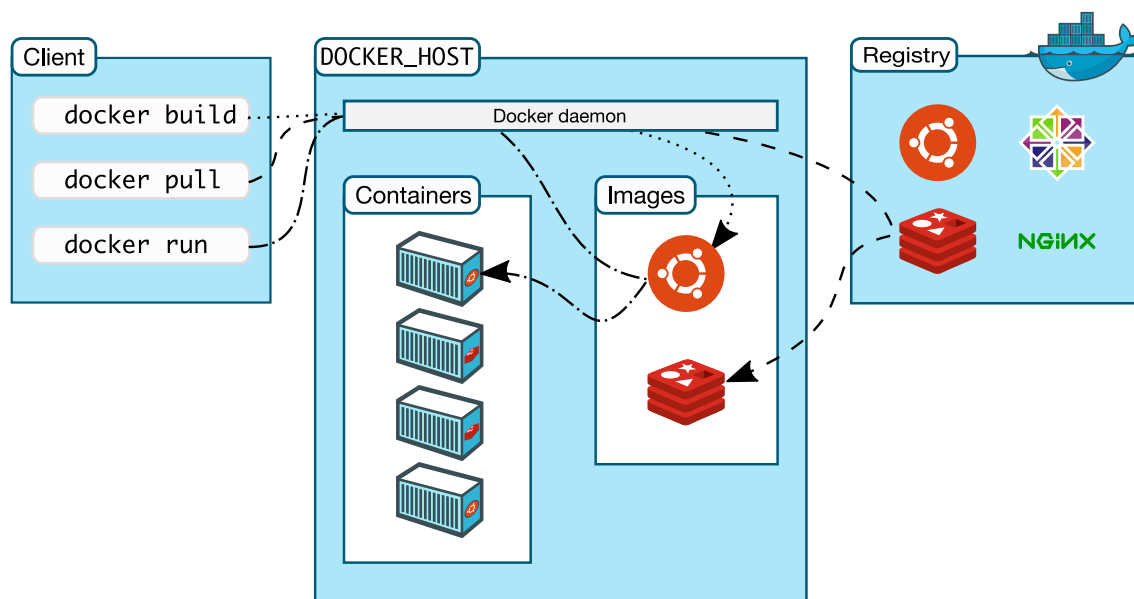
A Linux operációs rendszernek van egy saját virtualizálási technológiája: a KVM[25] (Kernel Virtual Machine). Ez egy teljes értékű, beépített, kernel szintű hypervisor Intel VT és AMD-V processzorokra. A VirtualBox-hoz hasonlóan ez is open source, hatékony és gyors eszköze a Linux alapú virtualizálásnak.

Az erőforrás hiány miatt kevésbé robusztus megoldást kellett választani. Az előző fejezetben már volt szó a konténerekről, ami nem a teljes hardvert és operációs rendszert virtualizálja, hanem a már meglévőt függetleníti el a host OS-től, egyfajta tárolót alakít ki. Itt korlátozottan futnak operációs rendszerek, erőforrásokat és fájlrendszert is a hosttól közvetlenül kap, megfelelően leválasztva arról. Nagy előnye, hogy bármilyen környezetben ugyan abban a konténerben ugyan az az alkalmazás ugyan úgy futni képes, rendszertől független.

A *Docker*[26] a legismertebb container alkalmazás. A Docker Int. által fejlesztett open-source platform egy nagyon sokrétűen és hatékonyan felhasználható fejlesztői tool. Létezik mindenki számára elérhető ingyenes, illetve egyéneknek vagy ipari felhasználásra

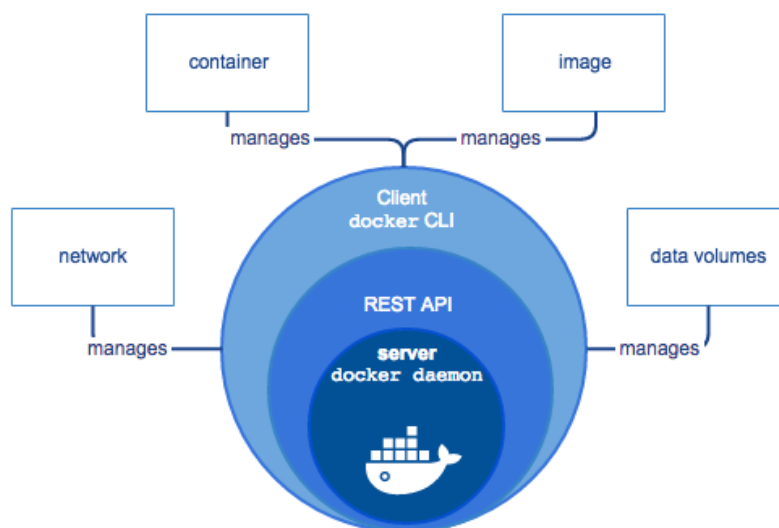
szánt fizetős verziója is. Elsősorban szolgáltatások különválasztására és keretbe zárására hatékony. Kisebb erőforrás igénye van, mint egy vm-nek, ezáltal gyorsabb is, jó micro service architektúrákra is.

Vizsgáljuk meg a Docker fő komponenseit a 3.2. ábra segítségével.



**3.2. ábra.** A Docker architektúrája

Három fő eleme van a Dockernek: Docker Host, Client és a Registry. A docker client az a host számítógép, aminek a termináljába az operátor parancsokat tud kiadni. Ezek a parancsok lehetnek docker pull, docker push, docker run, docker run és sok egyéb más parancs is.



**3.3. ábra.** Docker Engine komponensei és kapcsolatai

Egyik fő komponense a Docker Daemon. Ebben található a Docker Engine, ami a teljes containeres környezetnek az irányításáért felelős. A 3.3. ábra mutatja a különböző rétegeit. A docker daemon server hozza létre, menedzseli és törli a containereket és

image-eket. Efölött fut egy *Rest API*, ami a felhasználói utasításokat és actionokat közvetíti a daemonnak. Ezt akár kívülről is el lehet érni. A használat szempontjából a legérdekesebb része a **docker CLI Client**, ahova a parancsainkat írhatjuk meg. A Docker Engine kezeli a konténereket, image-eket, a hálózatot és a storage megoldásait. Mindig futnia kell, különben nem működik a szolgáltatás. Egy operációs rendszeren fut, ami lehet akár Linux, Windows vagy MacOS.

Az **DOCKER HOST**-on belül találhatjuk meg még a containereket, amiket szabadon hozhatunk létre, módosíthatunk, kezelhetjük az egyébként vm-hez hasonló networking funkciókat, jogosultságokat. Másrészt itt találhatjuk a Docker Image-eket is.

A Docker Image nem egyenlő a containerekkel. Ez egy nem írható és módosítható elem, nem a host használja fel. Ebből épül fel a container, ami lemásolja és kibontja az image-t. Az image-t meg lehet osztani a Docker Hub-on és mások számára is elérhetővé lehet tenni, vagy fordítva: mi is használhatunk mások által létrehozott publikus Image-eket.

Docker Image-eket is van lehetőség létrehozni. Működés közben egy container az image rétegeire húz fel egy újabb réteget, amit módosítani tud és futás közben használ. Amennyiben a használt containerből újra Image-t csinálunk vagy újabb verziót hozunk létre ez a réteg hozzáadódik és a következő felhasználáskor már ez sem lesz módosítható. A Dockerben van verziókövetés - egyszerre több verziót is tárolhatunk a Hub-on.

Az architektúra harmadik fő részében, a *Registry*ben találhatjuk a docker adatbázisát, innen érhetjük el a már szóba került Docker HUB-ot, ahova saját felhasználói fiókkal beléphetünk és saját repository-t hozhatunk létre. Szükség esetén, ha probléma van egy konténerrel, létrehozhatunk Docker file-t, amivel akár saját szabályokat is vezethetünk be.

Nézzünk pár parancsot, amivel CLI-ből vezérelni tudjuk a programot (provision).

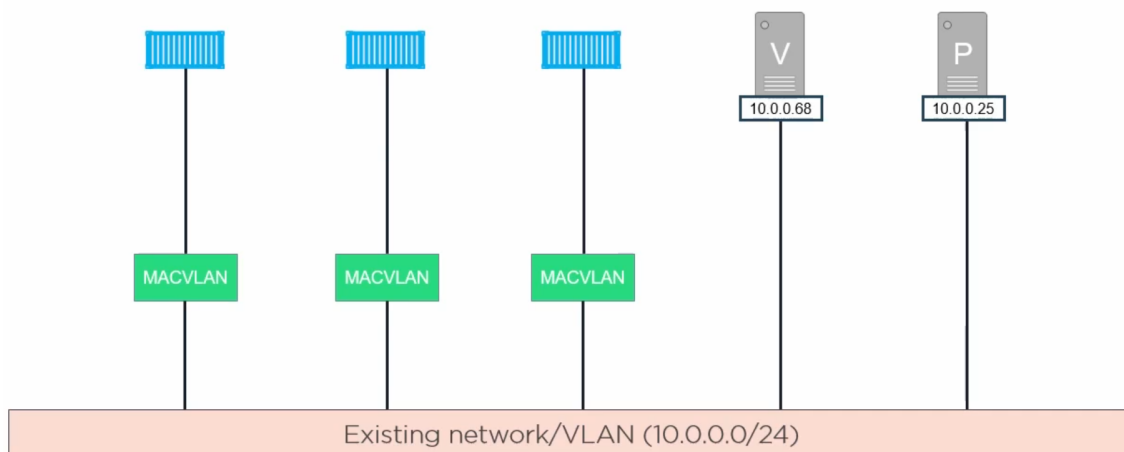
```
$ docker help          # Kilistázza az összes Docker commandot
$ docker version       # Verzió kiírása
$ docker search image  # Docker hubon kereshetünk image-t
$ docker info          # Részletes információ a dockerről
$ docker run *args*    # Ezzel a parancsal egyszerre tudunk letölteni és telepíteni image-t
$ docker pull          # Image letöltése a Docker Hub-ról
$ docker push          # Image feltöltése a Docker Hub-ra. Be kell jelentkezni hozzá
$ docker exec          # Image-ből container futtatása. -itd és többféle paraméter adható meg
                        hozzá
$ docker start <container> # Már létrehozott ontainer indítása
$ docker stop <container>  # Container leállítása
$ docker rm            # Container törlése
$ docker rmi          # Image törlése
$ docker ps           # Containerek listázása (-a: nem futókat is)
$ docker images       # Image-k listázása
$ docker commit <id> <hub/img> # Image létrehozása
$ docker login        # Docker Hub login
```

A Docker containerekben engedélyezni lehet különböző jogosultságokat is a **\$docker run --cap-add** parancssal. Különböző linux capability-t lehet hozzáadni, mint például: **NET\_ADMIN**, **SYS\_ADMIN**, **SYS\_BOOT**, **CHOWN**, **FOWNER** és még másokat. A containerben nem lehet **sudo** parancs beírásával root szinten parancsokat kiadni, ezért kell ezekkel a jogosultság kiterjesztésekkel megoldani néhány megoldást.

Mivel az diplomamunka alkalmazására elég speciális technológiát használunk a hálózat kialakítására, ezért érdemes egy külön fejezetben tárgyalni ezeket.

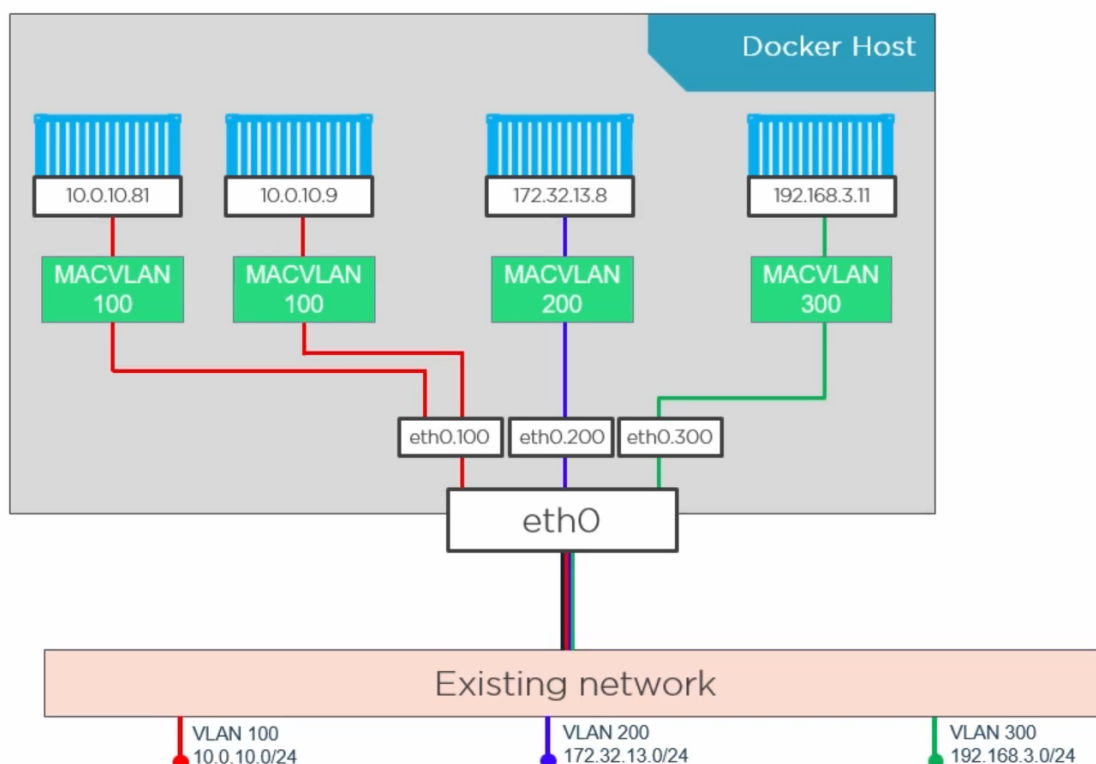
### 3.3.1. Docker networking

A Dockerben is megtalálhatunk a klasszikus virtualizálási technológiákhoz hasonló network kialakítást. Ezek alapján felfedezhetjük például a **bridge** networkot. Ez az alap beállítása a dockernek, ha további parancsok nélkül létrehozunk egy containert. Ez a VM világban a NAT módnak felel meg. A Docker Daemon automatikusan létrehoz egy subnetet (általában



3.4. ábra. A macvlan megoldás ahogy a switch oldalról látszik

172.17.0.0/24), amit NAT szerűen hozzáköt egy host interfészhez. A **host** módban a host hálózati beállításait használja, nem hoz létre külön egységet. Az **overlay** pedig a host-only módnak feleltethető meg. Ez esetben a container hálózat nem lesz kikötve host interfészre, így csak egymással és a host-al tudnak kommunikálni a nodeok. A *swarn* szolgáltatás használatakor ez a megoldás javasolt. Van még **none** beállítás, amikor nincs network driver a containerben.



3.5. ábra. A macvlan megoldás ahogy belülről látszik

Van még egy megoldási lehetőség, amit érdemes megvizsgálni. Mégpedig a **macvlan** kialakítást (3.4 és 3.5. ábra). Ennek az a lényege, hogy úgy alakítja ki a hálózatot a Docker Engine, hogy a külső hálózat számára olyan lenne, mintha switchen keresztül több host is

csatlakozna a hálózathoz L2 szinten. Ezt úgy éri el, hogy minden egyes macvlan interfészhez rendel egy MAC címet, ezenfelül a linkeket úgy választja szét, hogy hozzárendel egy VLAN Tag-et is.

A `macvlan` networkingt háromféleképpen lehet használni: `bridge`, `802.1q trunk bridge mode` és `ipvlan` (vagy `ipvlan`). A különbség annyi ezek között, hogy a `bridge` módban fizikai interfészhez köti közvetlenül a `macvlan` linket és annak az IP címét használja. Az általunk legérdekesebb típus a `trunk` mód, ami a korábban leírt megoldást valósítja meg. A harmadik az `ipvlan`, amikor L3 bridgel köti össze a host-al és nem L2 bridge-t hoz létre.

Ez hasznos lesz a hibrid SDN környezet kialakítására, mivel olyan hálózati topológiát tudunk kialakítani, ami egymástól teljesen független node-okat tesz egy hálózatba. Az ESXi vSwitch-e miatt a VLAN technológiára egyébként is szükség lenne. Egy nehézsége van ennek a megoldásnak, mégpedig az, hogy a host OS-nek és a rendszernek támogatni kell a *Promiscuous Mode*-ot [27], tehát hogy a rendszer minden csomagot a CPU-n keresztül irányít, hogy az dolgozza fel és irányítsa tovább.

Példa az alkalmazáshoz:

```
$ docker network create -d macvlan \ # macvlan docker network elem felvétele, paraméterek megadása
--subnet=172.16.86.0/24 \
--gateway=172.16.86.1 \
-o parent=eth0 \
my-macvlan-net
$
$ docker network ls          # Verify
$ docker network inspect my-macvlan-net # Verify
$
$ docker run --rm -dit \    # Hozzáadjuk a fent létrehozott networkot egy containerhez.
--network my-macvlan-net \
--name my-macvlan-alpine \
alpine:latest \
ash
$
$ docker container inspect my-macvlan-alpine
$
$ docker exec my-macvlan-alpine ip addr show eth0
```

### 3.1. lista. Docker macvlan kiépítése

## 3.4. OpenFlow protokoll alkalmazása

Magát a protokollt a 2.2.2. alszakaszban megismertük. Láttuk, hogy milyen csomagokat és actionokat kell használnunk (OF actionok: `OFPACTIONOutput`, `OF_PACKET_OUT`, `OF_PACKET_OUT`, stb.; üzenet típusok: `PacketIn`, `FlowMOD`, `PacketOut`, stb.; csomag és protokoll típusok: szinkron, asszinkron, controller-to-switch). Általánosságban megtárgyaltuk a gondolkodás-mód lényegét és már van egy elképzelésünk, hogyan működik.

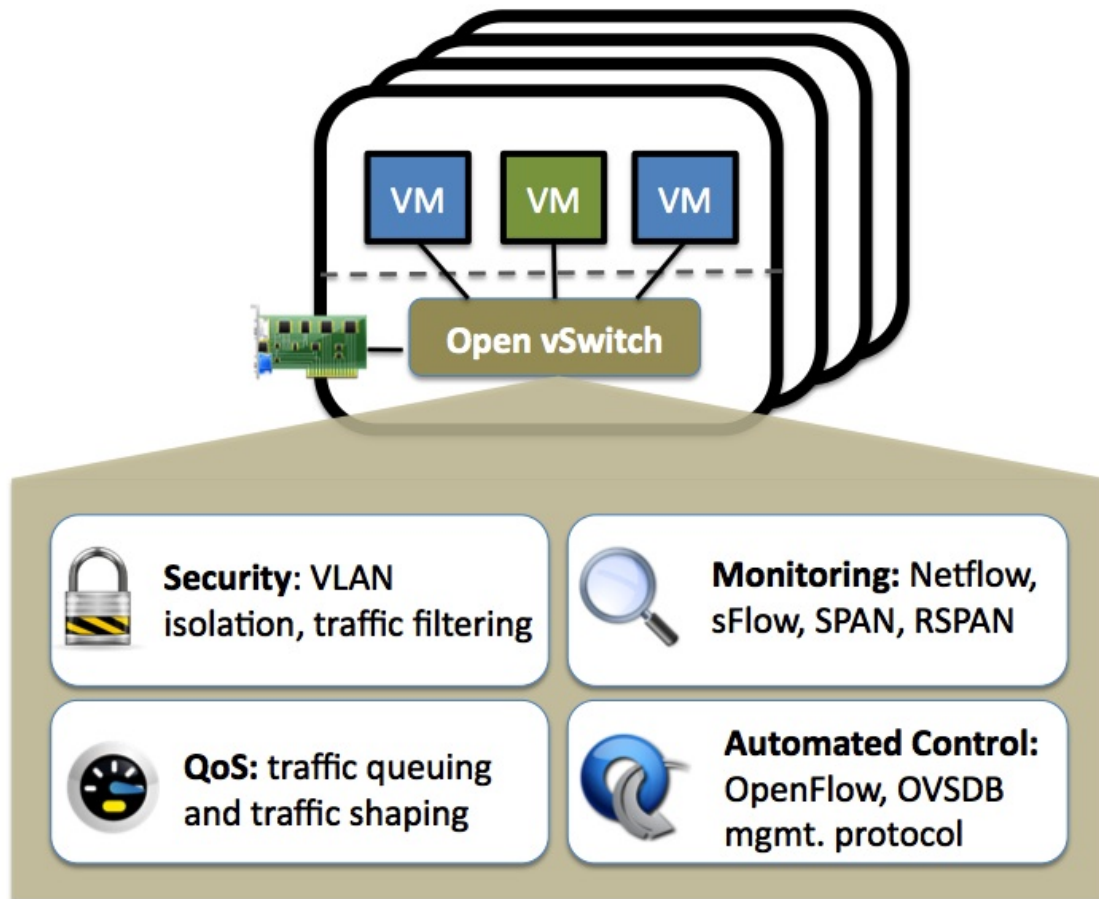
Ahogy a 3. fejezetnek is, cél az alkalmazott technológiák bemutatása. Mivel az OpenFlow meghatározza a kontrol és az adatkapcsolati réteget is, ezért az OF szemszögéből érdemes megvizsgálni az alkalmazott SDN szoftvereket.

### 3.4.1. Mininet és OpenvSwitch

Ahogy megismerhettük, az SDN két részre van választva: `control plane` és `data plane` réteg. Először vizsgáljuk meg, milyen lehetőségek az L2-es adatkapcsolati rétegen.

Az SDN adatkapcsolati rétegének tárgyalásánál szinte megkerülhetetlen a *mininet* [28] szóbaejtése. Ez egy hasznos network emulátor tool virtuális hálózatok kialakítására. Sokféle

funkcióval rendelkezik, létre tud hozni host-okat, network node-okat, közöttük tetszőleges topológiákat, illetve képes network monitoring funkciókat is megvalósítani. Rendkívül hatékony kutatás, fejlesztés és tanulás céljából is. Sok SDN alkalmazás fejlesztése ennek segítségével kezdődik. A [1]-es kiinduló diplomamunka is használja ezt az eszközt. Mivel ebben az esetben kifejezetten cél volt a "fizikai szerű", tehát a fizikai megvalósításhoz legközelebbi alkalmazás kialakítása volt, így egy-egy teszhálózat felhúzásán kívül nem került másra sor.



3.6. ábra. Open vSwitch funkciói

Az OpenFlow alapú switchek közül először az Open vSwitch[29] (OVS) kerül elő (a mininet is ezt használja a switch nodeok felépítésénél). Az OVS egy multilayer szoftver open source Apache2 licenszsel. Az OpenFlow funkcióit már nem szükséges bemutatni, de ez is egy kifejezetten sokrétű program. Sokféle környezetbe implementálható, az interfészek beilleszthetők a virtuális rétegekbe. Sokféle protokollt támogat, mint például VLAN, LACP, NetFlow, sFlow, QoS, GRE, VXLAN, OpenFlow, stb. Elsősorban Linux kernelbe illeszthető be.

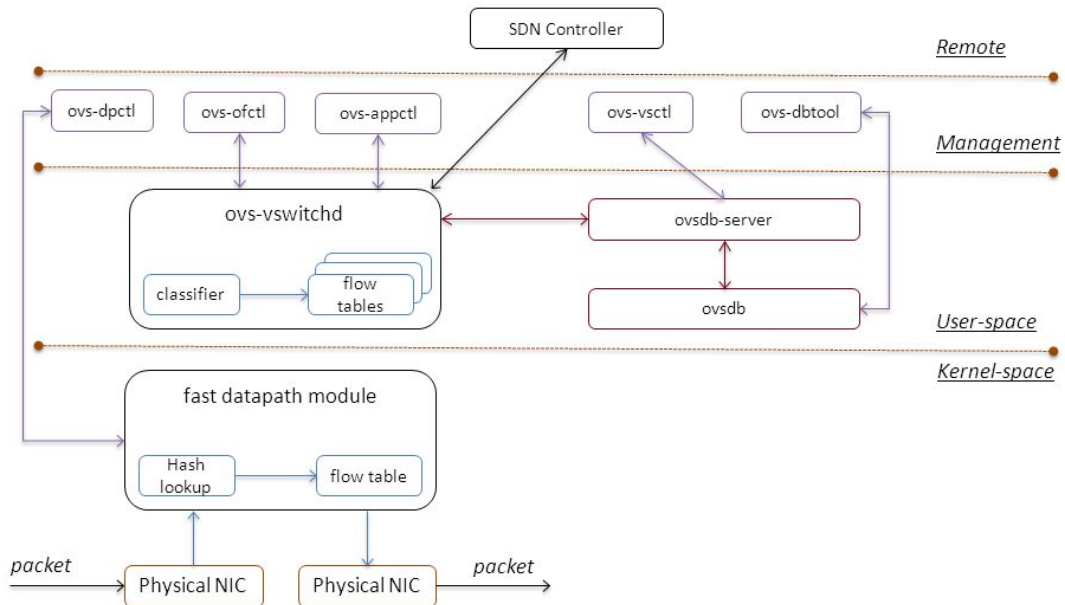
Az OVS fő elemei:

**ovs-vswitchd** az a daemon, ami a switchet implementálja a Link kernel együtt flow alapú switchinghez.

**ovsdb-server** egy egyszerű adatbázis szerver, ahol az OVS a lekérdezésekhez szükséges beállításokat tárolja.

**ovs-dpctl** egy tool a switch kernel moduljának konfigurációjához.

## Open vSwitch (Architecture)



3.7. ábra. Open vSwitch architektúrája[30]

**Scriptek és specifikációk RPM építéséhez** a Citrix XenServerhez és a Red Hat Enterprise Linuxhoz. A XenServer RPM-ek segítségével lehet OVS-t telepíteni Citrix XenServerre.

**ovs-vsctl** egy eszköz az **ovs-vswitchd** lekérdezésére és frissítésére. Ez az alkalmazás lesz számunkra a legérdekesebb.

**ovs-appctl** eszköz, ami commandokat küld a futó Open vSwitch daemonnak.

**ovs-ofctl** eszköz az OpenFlow switch és controller adatainak lekérésére és irányítására. Ennek segítségével létrehozhatunk a switchben FlowTable Entry-eket anélkül, hogy a controller beavatkozna.

**ovs-pki** eszköz public-key infrastruktúra létrehozására és térképezésére az OVS-ekben

**ovs-testcontroller** egyszerű beépített OpenFlow controller, ami hasznos lehet a teszteléshez.

**tcpdump patch**, ami OpenFlow csomagok parsolására használható.

Ezek közül a legérdekesebb az **ovs-vsctl**[31], ami tulajdonképpen a fő control tool. Ezzel lehet módosítani az **ovsdb-server** entry-eket, amik az OVS adatait és konfigurációit tárolja. Ezen keresztül hozhatunk létre új, törölhetünk és módosíthatunk interfészeket, bridge-eket és portokat a vSwitch-ben, valamint ezen keresztül adhatjuk meg a controller elérhetőségét.

Pár **ovs-vsctl** command kiemelve:

```

$ ovs-vsctl add-br <név>           # Bridge-et lehet ezzel létrehozni. ŐKésőbb ezekhez tudjuk
    hozzárendelni a portokat. Megjelenik a host portjai között is
$ ovs-vsctl del-br <név>           # Bridge törlése
$ ovs-vsctl add-port <bridge> <interfész> # Port felvétele adott bridghez.
$ ovs-vsctl del-port (bridge) <port>      # Port törlése
$ ovs-vsctl set-controller <bridge> <interfész> # ŐKüls kontroller beállítása
$ ovs-ofctl <bridge> <flow>          # Bár ez nem ovs-vsctl command, de a korábban említett
    módon ezen keresztül lehet létrehozni FlowTable Entry-ket

```

### 3.2. lista. Open vSwitch főbb parancsok

#### 3.4.2. Az alkalmazott SDN kontroller: Ryu SDN framework

Térjünk át a control plane rétegre. Már korábban említve volt, hogy miért ez a kontroller lett kiválasztva. Kiegészítésképpen fontos ebben a fejezetben megemlíteni, hogy a kiinduló diplomamunka során más fejlesztői környezet állt rendelkezésre, nem az ESXi, ezáltal rendelkezésre álltak WEB alapú modulok, topology viewer-ek, hálózat monitorozó eszközök, alternatív virtuális gépek és operációs rendszerek. Ráadásul a jelenlegi hardveren korlátozott erőforrások miatt nem lehetnek nagyobb kapacitást igénylő alkalmazások. Ez is szerepet játszott, hogy nem az ODL kontroller lett felhasználva. A robusztusságának és beépített alkalmazásainak köszönhetően sok kényelmi funkciót fel lehetett volna használni (a már említett topology viewer, management eszközök), viszont a jelenlegi megoldásra a Ryu tűnt ideális megoldásnak.

A Ryu kontroller[12][32] egy python alapú moduláris, komponens alapú SDN framework. Többféle API-t kínál, ami egyszerűbbé teszi egy új hálózat létrehozását, menedzselését és kontrollását. Támogatja számos új és régebbi protokollok implementálását és tartalmazza az bizonyos elemeit ezeknek. Ilyenek például OpenFlow, NETCONF, OF-config, OSPF, BGP, ETH, stb. Ezek közül csak a legfontosabb, SDN világban alapvetően használt protokollok vannak teljesen lekódolva, de API-kon és a northbound interfészen keresztül könnyedén létrehozhatjuk egyéb protokollok SDN-es megoldását, amire segítségül vannak kontroller beépített moduljai.

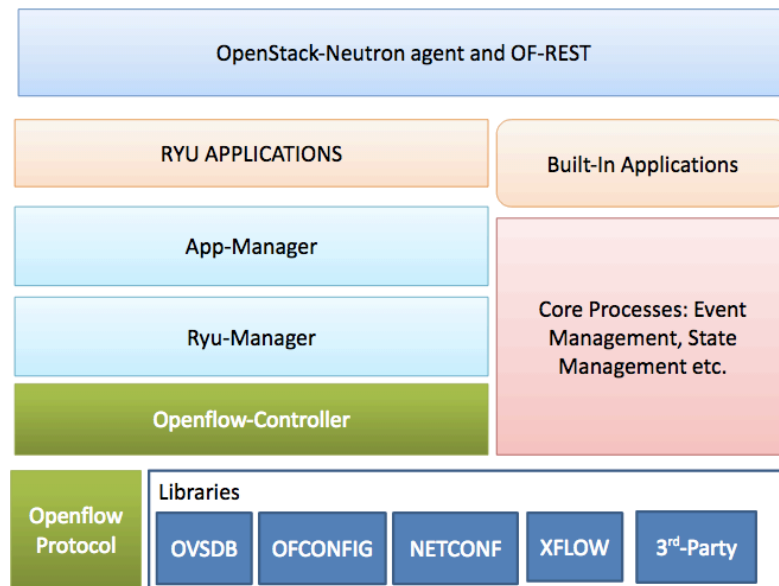
A kontrollert magát az NTT csapata fejlesztette. Érdekesség, hogy maga a Ryu szó Japánul sárkányt, folyamat és iskolát is jelent[33]. A legnagyobb erőssége a sokszínűsége és sok protokoll támogatása. Az ODL-el szemben nem túlságosan robosztus, nagyobb szabadságot és könnyebb fejlesztést enged a fejlesztőknek.

A Ryu architektúráját a modularitás jellemzi. Természetesen megtaláljuk az OpenFlow támogatására szolgáló részeket. A southbound oldalon, a *Libraries* területen találhatjuk a machine-to-machine protokollokat, mint például az OF-Config-ot, Open vSwitch Database Management Protocolt (OVSDb), NETCONF, XFlow és egyéb protokollokat. A *Ryu Packet library* segít parsolni (lefordítani) és építeni egyéb protokollokhoz illeszkedő csomagokat, így a már említett VLAN, MPLS, GRE, OSPF, BGP és más packeteket is felismerhetünk és csomagolhatunk. Ez a programozást jelentősen megkönnyíti.

Az *OpenFlow Protocol és Kontroller* egészen a legutóbbi, 1.5-ös OF verziót is támogatja. Az üzenetek kezelése és az OpenFlow fejezetben (3.4. szakasz) megismert eseménytriggereket is itt kell keresni, valamint a *Controller to Switch* üzeneteket, *Asszinkron* üzeneteket, *Szimmetrikus* üzeneteket és *Struktúra* üzeneteket. Innen az alkalmazás szempontjából legfontosabb elemek a már korábban említett *Flow-modify*, *Packet-in*, *Port-status*, *send\_msg* API, packet builder API és a fő eseménykezelő python decorator, a *set\_ev\_cls()* API fog kelleni.

A *Manager és Core-Process* elem indul el a Ryu indításával. Amikor fut, a megadott IP-n figyel a 6633-as OpenFlow portot. Az OVS-ek ezen keresztül tudnak csatlakozni hozzá. Az *app-manager* a legalapabb komponense az összes Ryu alkalmazásnak. Minden





3.8. ábra. Ryu kontroller architektúrája [33]

fejlesztő által írt alkalmazás a RyuApp osztály leszármazottja. A core-process komponens az architektúrában tartalmazza az esemény managementet, üzenet küldést és fogadást, az in-memory state menedzsmentet és egyéb elengedhetetlen funkciókat. Meg kell jegyezni, hogy bár a Ryu teljes mértékben pythonban íródott, támogatja a más nyelvben megírt komponenseket.

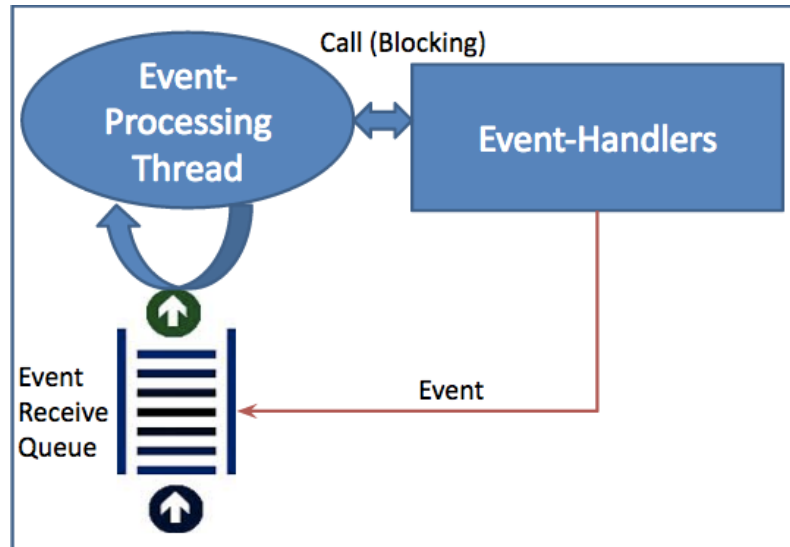
A *Ryu Nourthbound* interfészen API-kon keresztül hozzá lehet csatlakozni OpenStack alapú Cloud rendszerek Neutronjához, támogat GRE alapú overlay-eket és VLAN konfigurációkat, használható REST típusú inerfészen keresztül hozzáférés az OpenFlow-hoz és webalkalmazásokhoz is lehet csatlakozni WSGI technológiával.

Végül pedig vizsgáljuk meg az *Alkalmazási réteget*. A Ryu több elkészített alkalmazással rendelkezik, mint például egy `simple_switch` nevű egyszerű switchet megvalósító alkalmazás, vagy `firewall`, `GRE tunnel`, `topology` alkalmazások, amikből tanulni is lehet a framework működését, másrészt fel is lehet használni részeit egyéni alkalmazásokra.

A már korábban említett módon minden Ryu alkalmazás (3.9. ábra) a RyuApp osztály leszármazottja. Az egyes programok single-thread entryk, tehát egyszerre csak egy funkciót valósítanak meg, viszont egyszerre több is futtatható. A kontroller applikációi eseményekkel kommunikálnak egymással. Ezek FIFO rendszerben követik egymást (fontos megjegyezni, hogy ezáltal az OpenFlow események, mint a PacketIn, vagy hasonló funkciók is ilyen mód, pipeline szerűen kaphatók el az alkalmazásban). A fő ciklus kiemeli az első eseményt és lefuttatja a hozzá kapcsolódó eseménykezelő handlert. A különböző alkalmazásokat a `ryu-manager *app*` meghívásával indíthatók el.

Ryu moduláris struktúrája és főbb elemei kód és fájlrendszer szinten is megfigyelhető[34]. Beleásva mélyebben a kontrollerbe a következő struktúrát találjuk:

- `/ryu/` A fő elemei a kódnak a `/ryu/ /ryu/` mappában érhető el. A különböző funkciókat almappákba vannak elhelyezve.
- `app/` Ebben a csomagban találhatóak a Ryu felett futó alkalmazások. Ide lehet elhelyezni saját programokat is.
- `base/` Itt találhatóak a Ryu alkalmazások úgynevezett base class-ai, tehát minden alkalmazás az itt található osztályok leszármazottjai.



3.9. ábra. Ryu alkalmazás működése

**controller/** A kontroller részben találhatóak olyan alapvető modulok és funkciók, amik például az OpenFlow kezelését menedzselik, illetve a kontrollerhez kapcsolódó eszközöket, switcheket, flow elemeket, hálózati eseményeket, statisztikákat is itt gyűjt a szoftver.

**lib/** Több csomagkönyvtárat tartalmaz, amik a különböző protokollok csomagjait, headerjeit, kezelését és könyvtárait tartalmazza. Például itt találhatóunk egyéb fontos - számunkra kevésbé érdekes, de hasznos protokollokat is, mint Netflow vagy sFlow.

**ofproto/** A csomag tartalmazza az OpenFlow protokoll specifikus információkat, parsereket, valamint az OF több verzióját is támogatja (1.0, 1.2, 1.3, 1.4, 1.5)

**topology/** A topology package fontos lesz a RyuOspf alkalmazás számára, mivel itt találhatjuk a hálózati felderítéssel kapcsolatos függvényeket, az OpenFlow switchek térképét, azok kezelésével kapcsolatos információkat (portok, linkek, stb.). Különböleg használ LLDP protokollt is.

A kontroller beépített moduljait felhasználva el is lehet kezdeni egy alkalmazás megírását. A dokumentációban[12] segítségül találhatóunk kezdő példakódot, illetve segít kiválasztani a számunkra fontos elemeket, mint például a `@set_ev_cls()` OpenFlow eseményvezérlő dekorátort, egy egyszerű `packet_in_handler()` funkciót, leírást ad a `Packet()` osztály alapjairól és használatáról. Sajnos a dokumentáció nem túl részletes, ezért annak megismerését sokszor a példakódokból lehet csak teljesíteni.

#### 3.4.2.1. Ryu alkalmazás írása

Mielőtt nekiállunk saját programunknak (5.2. szakasz), pár sorban vizsgáljuk meg, hogyan javasolt elkezdni alkalmazást írni a RyuSDN kontrollerhez. Miután telepítettük a Ryut a `$pip install ryu` paranccsal létrehozunk egy `myapp.py` modult a `/ryu/app/` könyvtárban.

Segítségül tekintsük meg a hivatalos dokumentációban található mintakódot, ami egy alap funkciójú L2 switchet szimulál, de közelebb vezet minket a programozás megértéshez:

```
from ryu.base import app_manager
```

```

from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_0

class L2Switch(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(L2Switch, self).__init__(*args, **kwargs)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath
        ofp = dp.ofproto
        ofp_parser = dp.ofproto_parser

        actions = [ofp_parser.OFPActionOutput(ofp.OFPP_FLOOD)]
        out = ofp_parser.OFPPacketOut(
            datapath=dp, buffer_id=msg.buffer_id, in_port=msg.in_port,
            actions=actions)
        dp.send_msg(out)

```

### 3.3. lista. myapp.py példakód

A példakódban láthatjuk, hogy a felhasznált modulok importálása után létrehozott osztály a már korábban említett módon az `app_manager.RyuApp` leszármazottja. A kód elején kiválasztja a felhasználandó OpenFlow verzióját, ami jelen esetben a 1.0, majd konstruktorokkal segítjük az objektum példányosítását. Mivel itt nem használunk fel egyedi osztály szintű változókat, ezért minden argumentumot és keyword-el ellátot argumentumot szuper függvényen keresztül továbbítunk az anyaosztálynak.

Érdekes lesz a feladat szempontjából a `set_ev_cls` dekorátor, ami a `packet_in_handler()` függvény előtt található. Ebben látható egy OFP esemény, esetünkben egy `packet_in` üzenet, aminek a bekövetkezése során hívja meg az általunk létrehozott függvényt. A dekorátor második argumentuma a switch-ek által megvalósítandó akciót (action) jelzi, ami ez esetben `MAIN_DISPATCHER`, tehát az OVS-ek a csomagokat a kontroller feldolgozása előtt figyelmen kívül hagyja a csomagot és csak a Ryu válasza után dolgozza fel, vagy illeszti be a Flow táblájába a teendőjét. Ha a kontroller nem rendel akciót a beérkezett csomaghoz, akkor eldobja.

Maga a bejövő csomagokat feldolgozó `packet_in_handler()` függvény már `ev` paraméterként kapja meg az üzenetet, azt szedi szét különböző változókbá, külön választja az üzenetet, datapath-ot és az OpenFlow-al kapcsolatos funkciókat. Majd a függvény meghatározza az action-t, amit az OVS-nek kell tennie. Esetünkben elárasztja a bejövő packageket a hálózatban. Utána `OFPPacketOut` segítségével létrehozuk a kimenő csomagot, hogy azután a `send_msg` függvénnyel visszaküldjük a feldolgozott információkat a switchnek. A kész a kód lefuttathatjuk a programot:

```
$ ryu-manager myapp.py
```

### 3.4. lista. Ryu mintakód futtatása

Továbbiakban fontos lesz a `ryu.topology` modul, ami az SDN hálózat feltérképezésért felelős. Ezáltal lehetséges lekérdezni a kontroller által kezelt hálózatot. Használni fogjuk `EventSwitchEnter` eseményt, amikor OVS csatlakozásakor történik meg, ennek során meg lehet hívni egy `get_topology_data()` függvényt, ahol a `get_switch()` és `get_link()` függvényekkel le tudjuk kérdezni a nodeokat és a hozzá tartozó linkeket[35].

Mindenképp fontos megemlíteni a `ryu.lib.packet` könyvtárat, ami több szempontból is érdekes lehet. A `Packet()` osztály képes kezelni rengetegféle csomag keretét, headert, azokat kibontani, rendszerezni (`parser`) és összecsomagolni (`serialize`). Képes felismerni, hogy milyen csomag érkezett és az alapján tudja rendszerezni. A Packet API rendelkezik `add_protocol`, `get_protocol`, `get_protocols`, `serialize`, `parse` függvényekkel.

Headerek között találhatunk DHCP, BGP, ARP, ICMP, IP, VLAN, ETH osztályokkal és az diplomatervezési témája szempontjából a legfontosabbal: *OSPF* headerrel. Az `ospf.py`-ban találhatunk LSA, LSU, LSAck, Hello és az OSPF-hez kapcsolódó csomagstruktúrákat. A protokoll kezelése nincs benne a kódban, de mindenképp jó kiindulást ad a protokoll implementálásához.

Említésre méltó még az OpenFlow kezelő API, ami az OpenFlow metódusok kezelésére szolgál. Ki lehet választani az OF verziót, e modul segítségével lehet reagálni az eventekre, lehetőséget ad a Controller-to-Switch, aszinkron, szimmetrikus, *modification* és egyéb csomagok kezelésére, fogadására és küldésére. A dokumentáció segítségével könnyedén meg lehet fejteni a használatát.

### 3.5. OSPF helye az SDN logikában

A korábban részletezett routing protokollt (2.1.2. szakasz) kód szinten, python nyelven kell beágyazni a Ryu *Framework* northbound oldalára. Az SDN világban a döntési helyzeteket nem az SDN nodeok kezelik, hanem amennyiben ismeretlen csomag vagy kérés jön hozzájuk OpenFlow protokollon keresztül továbbítják a kontrollernek, ami eldönti, hogy mit hajtson végre a feladó node[36].

A Ryu adta lehetőségeket kihasználva nincs szükség a különböző packet és LSA típusokat lekódolni, hiszen van beépített osztály ezekre a `ryu/lib/packet/ospf.py` modulban. A csomagkezelést és időzítéseket viszont önállóan kell megvalósítani az általunk választott SDN kontrollerben.

Magához a protokoll fejlesztéséhez több segítséget is találhatunk az interneten. Ilyen segítség lehet többek között a *University of Cambridge* egyik projektje, ahol egy routert akartak létrehozni, operációs rendszerrel és routing protokollokkal együtt. Ennek az eredményeképp jött létre a PW OSPF[37] (Pee-Wee OSPF), ami egy OSPFv2 szerű protokoll, de nem rendelkezik az eredeti protokoll teljes funkcionalitásával. Minden esetre jó kiindulás az önálló fejlesztéshez.

Fontos még megjegyezni, hogy célunk, hogy kívülről, a legacy hálózathoz minden egyes OVS látható legyen, hogy a routerek ismerjék a teljes hálózati térképet. Úgy mond a kontrolleren keresztül "átverjük" a legacy nodeokat, hogy a virtuális nodeokat valós csomópontokként mutatja.

### 3.6. Hálózat monitorozása

Ahhoz, hogy fel tudjuk térképezni a hálózat tényleges működését szükségünk van monitoring toolokra is. Ezeknek a feladata ellenőrizni, hogy van-e kapcsolat két host között, milyen típusú csomagok futnak a hálózaton, esetleg milyen adatforgalommal. A legegyszerűbb és legszéleskörűbben használt a ping ICMP üzenet, ami ha megérkezik a címzethez visszaküldi a feladónak, hogy minden oké, megkapta a csomagot.

Ennél vannak robusztusabb alkalmazások is. Az egyik ilyen a szintén széles körben Wireshark[38], ami egy protokoll analízáló open-source szoftver. 1998-ban kezdődött a fejlesztése, rengeteg protokollt felismer, sokféle platformon fut (Linux, Windows, macOS, Solaris, FreeBSD, NetBSD) és egy UI-n keresztül ad betekintést a host hálózati kommuni-

kációjára. Ez a UI-nak köszönhetően hatékony, viszont mivel a szerveren nincs GUI, ezért más alkalmazást is kell keresni.

A `tcpdump`[39] és `libcap` cli alapú open-source packet analyzer szoftver. Segít a TCP/IP és egyéb szabványos küldött vagy fogadott üzenetet feltérképezni. Unix alapú rendszereken működik, libpcap Linuxos könyvtárat használja az üzenet elfogásához. Létezik Windowsos megoldása is, amit WinDump-nak hívunk és WinPcap-ot használ. A libpcap formátumot a Ryu kontroller is támogatja.

A két hatékony eszközt össze is lehet kötni[40], ezáltal a Wireshark előnyeit kihasználva, úgymond remote módban is meg lehet oldani a hálózatmonitorozást. SolarWinds toolokkal pedig akár élő adatforgalmat is lehet követni távolról.

## 4. fejezet

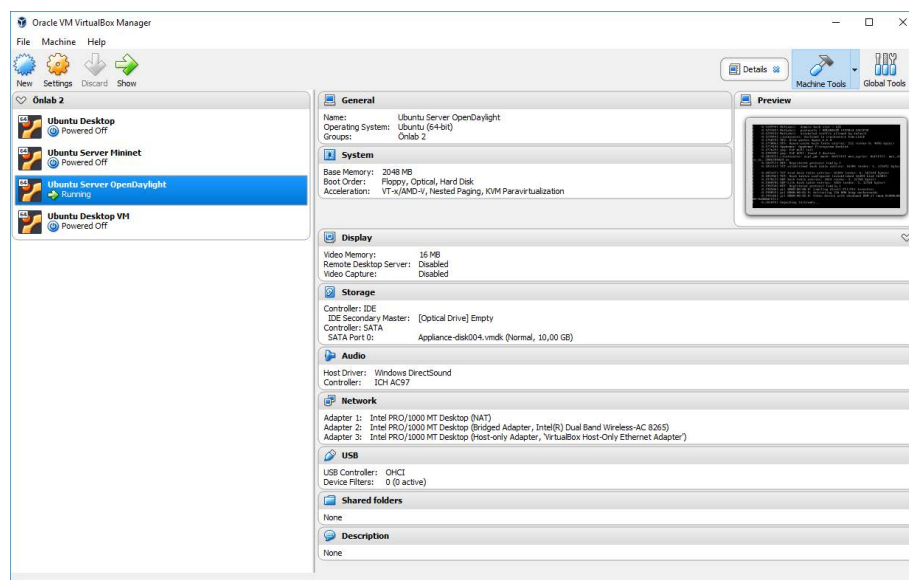
# Hibrid SDN hálózat implementálása

A hibrid SDN hálózat kialakításának két fő része van: a hálózat felépítése (deploy), illetve magának a kontrollernek a programozása. Specifikálni kellett a problémát, meg kellett tervezni az egyes elemeket, hogy aztán meg lehessen valósítani, hogy aztán a végén tesztelni tudjam.

A hálózat felépítését először saját környezetben kezdtem, hogy megismerhessem a technológiákat és gyakorlatban is alkalmazni tudjam. Ezután a megismert megoldásokat az ESXi környezetre vittem át. Amikor elkészült a hálózat el lehetett kezdeni a kontroller programozásának megtervezését, majd a kód megírását. Az elkészült alkalmazást ezek után tesztelni kellett.

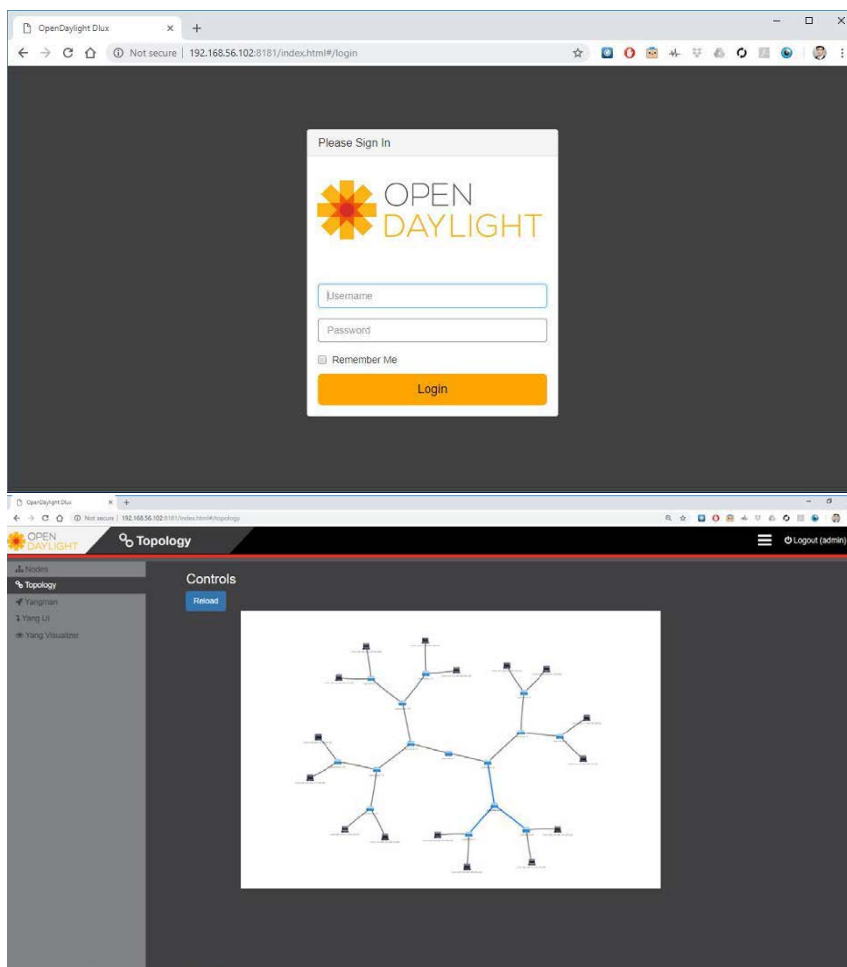
### 4.1. SDN hálózat deploy

Az SDN implementálását VirtualBox alapú Linux VM-eken kezdtem. Mivel a Linuxot sem ismertem mélyebben, ezért azzal is meg kellett ismerkednem. Magával a Software Defined Network-el Megyeri Csaba által megvalósított feladatán keresztül kezdtem a tanulást.



4.1. ábra. VirtualBox VM-ek

Felépítettem egy SDN hálózatot ODL kontrollerral és mininetel. Mivel ez Windows-os virtualizációval működött volt lehetőségem az OpenDaylight GUI, grafikus felületét is megvizsgálnom.



4.2. ábra. OpenDaylight GUI

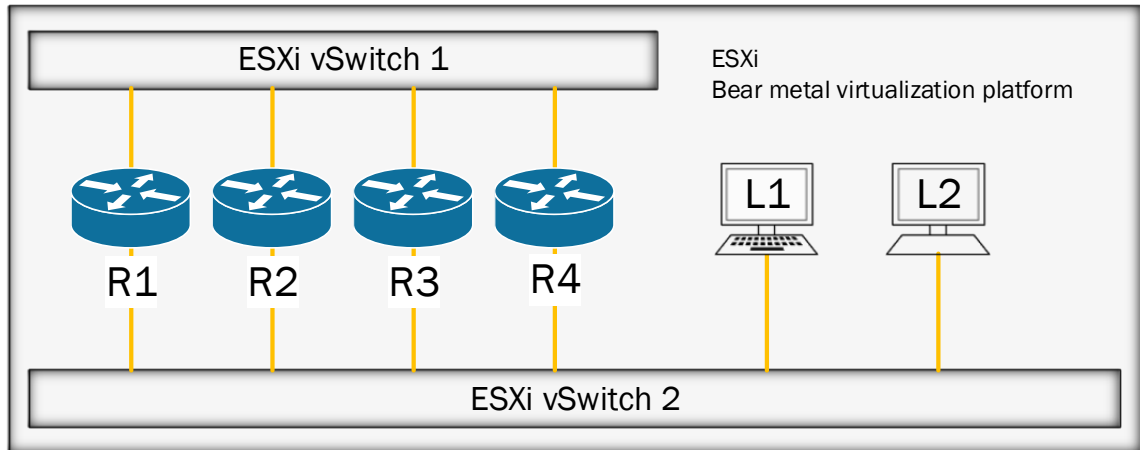
Maga a saját virtualizációs platformos megoldás nagyobb szabadságot és jó lehetőséget adott a technológia, Linux, SDN és virtualizálás megismerésére. Későbbiekben is tudtam használni és ebben tesztelni, ha valami nem működött az ESXi környezetben.

#### 4.1.1. Hálózati terv

Miután a kezdeti kísérletezésen továbbjutottam el lehetett kezdeni megalkotni a hálózati tervet. A felhasználható hálózati eszközök és kialakítás a kapott ESXi VM-ektől függött. Rendelkezésre állt négy darab Cisco IOS image és két Linux VM. Ez alapján kezdtem el gondolkodni, hogy milyen megoldásokat érdemes a hálózatban választani.

Az ESXi hálózatban a VM-ek egy virtuális trunkolt switchel voltak összekötve L2 szinten (4.3. ábra). Azért, hogy virtuálisan LAN-okat, tehát különálló és szétválasztott hálózatokat tudjak kialakítani VLAN-okat kellett bevezetni. Ezáltal bármilyen logikai topológiát meg lehet valósítani, nem függ a fizikai és virtuális linkektől.

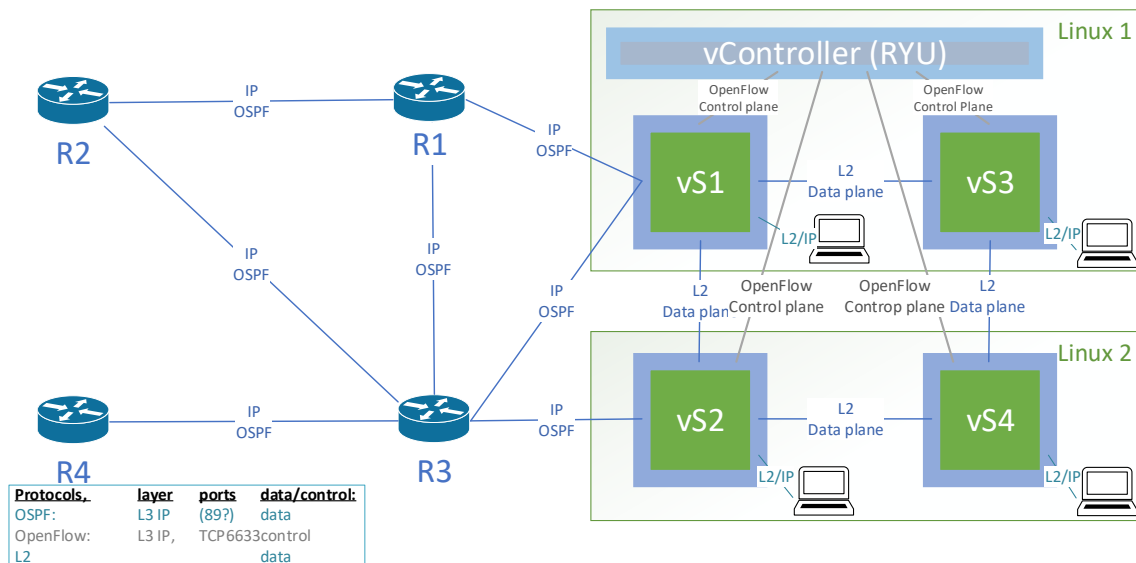
A hálózatba ezáltal 4 routert, Linuxonként 2-2 OVS-t, mindegyik vSwitchhez 1-1 host-ot terveztem a kontrollert pedig egyik Linuxban helyeztem el. A hálózat tervezésénél figyeltem, hogy többféle hálózati helyzet jelenjen meg. Ezáltal szükséges, hogy legyenek redundáns linkek, tehát olyan utak, amikor többféle utat is választhat a csomag, legyen



4.3. ábra. ESXi VM kialakítása

különálló node (stub network), ahova csak egyféleképpen jut el és legyen hurok is, amit a routing protokoll ki tud majd küszöbölni.

A megtervezett hálózathoz először megvizsgáltam a felhasználandó protokollokat, melyik node hogyan fog kommunikálni a szomszédjaival. A 4.4. ábra mutatja, hogy a routerek és az SDN border nodejai L3-as OSPF protokollal oldják meg a routingot. Látható, hogy a linuxokban telepített virtuális switchek (vS) egymással L2 szinten (data plane), a kontrollerrel pedig OpenFlow protokollal (control plane) vannak összekötve. A host-ok a switcheken keresztül szintén az adatkapcsolati rétegen kapcsolódnak rá a hálózathoz.



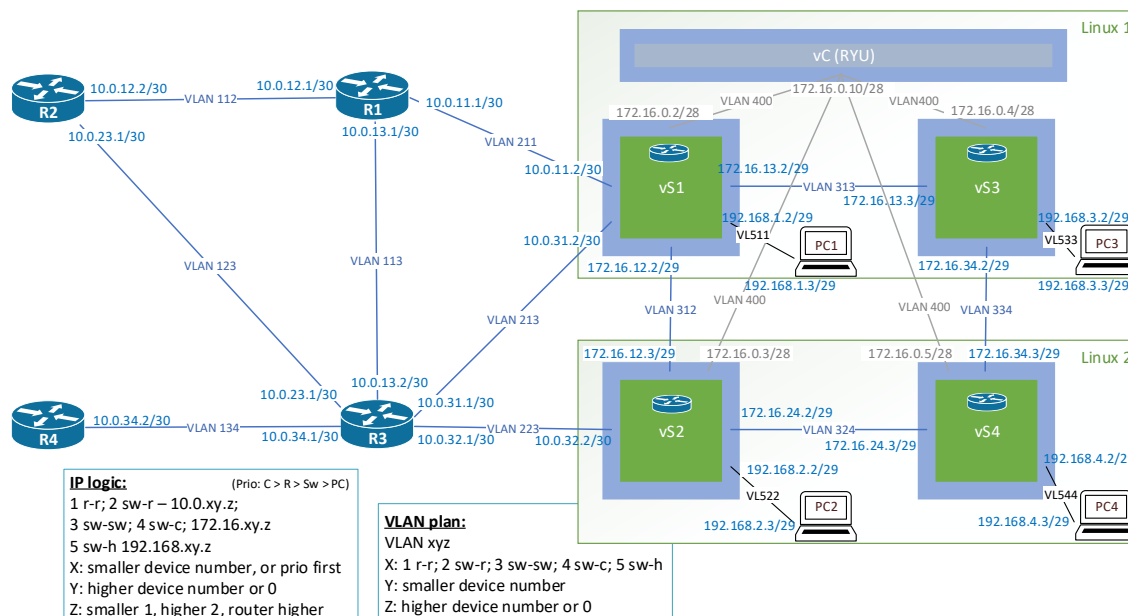
4.4. ábra. Hálózati routingolásra és vezérlésre felhasználandó protokollok és kommunikációs csatornák tervezése

Magát az OSPF vezérlést az SDN-en belül a már említett módon a Ryu-n futó program fogja megvalósítani. A legacy routereknek úgy lesz kommunikálva a kontrolleren keresztül az SDN hálózat, mintha az egyes switchek routerek lennének, tehát a legacy hálózathoz közvetlenül kapcsolódó vS1 és vS2 úgy hirdeti a szomszédjait, mintha azok is OSPF képes routerek lennének.



Ha egy OSPF üzenetet kap egy border vSwitch, rögtön felküldi a kontrollernek. Az megvizsgálja és az alapján frissíti a saját adatbázisát majd a teljes hálózatának a Flow tábláit. Az HELLO és LSDB update üzeneteket szintén a kontroller fogja irányítani.

Az adatkapcsolatok megtervezése után megnézhetjük az IP tervet (4.5. ábra).



4.5. ábra. Network IP és VLAN terv

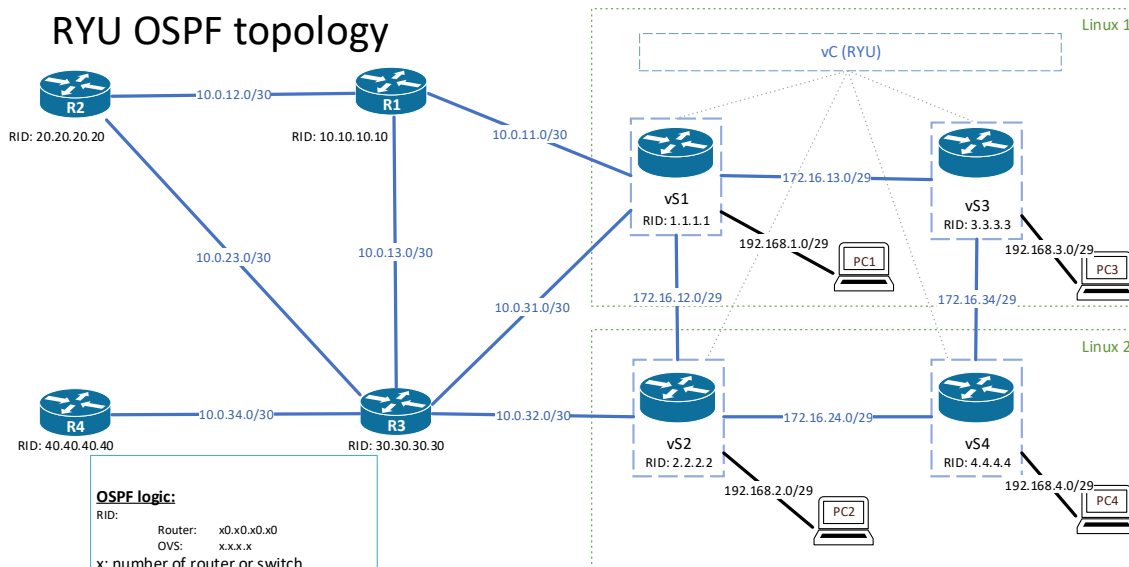
Az ábrán látható, hogy négy féle point-to-point kapcsolat van:

1. router és router
2. vSwitch és router
3. vSwitch és vSwitch
4. vSwitch és kontroller
5. vSwitch és host PC

A IP címek és subnetek tervezésénél azt a logikát követtem, hogy a hálózat méretétől függően alakítom ki a subnet típusát. A logika szerint a Routerok, tehát a legacy része a hálózatnak WAN (Wild Area Network) vagy MAN (Metropolitan Area Network) szerepet tölt be az alkalmazásban, így az azzal kapcsolatos subnetek a Class A típusú IP-ket kapnak, tehát 10.0.xy.z/30 IP-t. Az SDN rész már egy kisebb, belső business hálózatot testesít meg, ezáltal Class B típusú lesz, 172.16.xy.z/29 IP-je lesz. A hostok pedig a megszokott Class C, 192.168.xy.z/29 IP-jű interfészekkel fognak rendelkezni. Az x mindig a subnet kisebb azonosítójú, vagy magasabb prioritású elemét fogja jelölni, y pedig a magasabb azonosítójú, vagy 0-t (például Router 1 és Router 2 subnetben x=1, y=2; R1 és R3 subneten x=1, y=3; R3 és vS2 subneten x=3, y=2), z pedig a kisebb elemnél, vagy magasabb prioritásúnál 1, magasabb értéknél 2. Prioritási sorrend: Kontroller > Router > vSwitch > Host.

A 4.5. ábra tartalmazza a VLAN terveket is. A logika hasonló elven működik, hasonlóan 5 féle összeköttetés alapján számozza a VLAN ID-t. A VLAN xyz azonosítóban az x jelenti a kapcsolat típusát, y a kisebb azonosító számát, z pedig a magasabb azonosítót, vagy kontroller esetén x azonosítja a VLAN-t, több node is van a subneten, így y és z maradhat 0.

Végül a hálózattal kapcsolatos tervezés utolsó része az OSPF neighborship és RID tervezése volt.



4.6. ábra. Hálózat OSPF neighborhood terve

A 4.6. ábra jelzi, hogy ebben a megközelítésben nem külön legacy és SDN node-okban gondolkodunk, hanem OSPF routerekben és az általuk hirdetett hálózatokban. A könnyebb megértés érdekében jelölve vannak a virtuális logikai elemek, de lényegét az OSPF logika adja. Látható, hogy ha x a Router vagy vSwitch azonosítóját jelöli a Router ID (RID) megválasztása az OVS-ek esetében x.x.x.x, routerek esetén pedig x0.x0.x0.x0 azért, hogy mindig a Routerek legyenek a magasabb RID-jú elemek és a master, vagy adott esetben a Designated Router szerepét ők töltsék be. Ezáltal nem kell implementálni a controller szoftverébe az ezzel kapcsolatos feladatokat. A vSwitchek természetesen nem hirdetik a controllerhez kapcsolódó linkjeit, így az az OSPF szomszédokat tartalmazó listában rejtve maradnak.

#### 4.1.2. Cisco routerek konfigurálása

Mielőtt konfiguráltam a Linuxokat beállítottam a routereket is egyesével. Beállítottam a VLAN-okat, Up státuszba állítottam az interfészeket, beállítottam az OSPF area-t és meghirdettem a networköket. Példának a Router 1-es konfigurációt a 4.1 kódrészlet mutatja. Pingeléssel ellenőriztem a kapcsolatot.

```
enable
configure terminal
!
hostname R1
ip name-server 152.66.248.12
line console 0
logging synchronous
ip route 0.0.0.0 0.0.0.0 12.255.255.1
no ip domain lookup
ip default-gateway 10.255.255.1
!
interface GigabitEthernet1
no shutdown
!
interface GigabitEthernet1.112
```

```

encapsulation dot1Q 112
description R1_to_R2
ip address 10.0.12.1 255.255.255.252
no shutdown
!
interface GigabitEthernet2
no shutdown
!
interface GigabitEthernet2.113
encapsulation dot1Q 113
description R1_to_R3
ip address 10.0.13.1 255.255.255.252
no shutdown
!
interface GigabitEthernet3
no shutdown
!
interface GigabitEthernet3.211
description R1_to_OVS1
encapsulation dot1Q 211
ip address 10.0.11.1 255.255.255.252
no shutdown
!
interface GigabitEthernet4
no shutdown
!
interface GigabitEthernet4.999
encapsulation dot1Q 999
ip address 10.255.255.15 255.255.255.0
!
router ospf 1
network 10.0.12.0 255.255.255.252 area 0
network 10.0.13.0 255.255.255.252 area 0
network 10.0.11.0 255.255.255.252 area 0

```

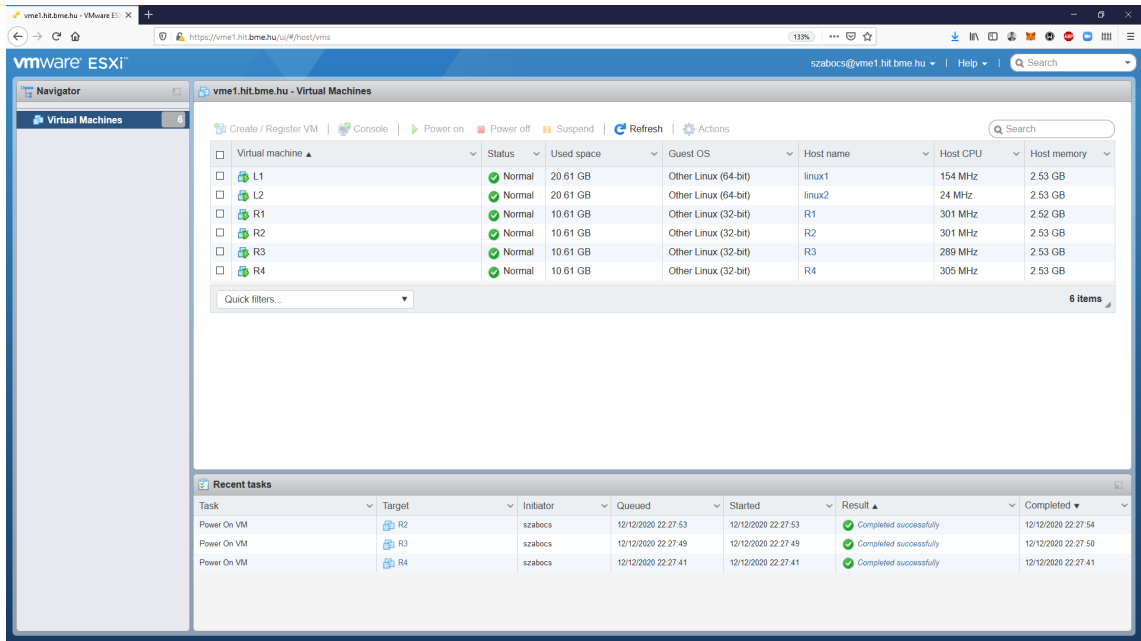
**4.1. lista.** Router 1 konfigurálás

#### 4.1.3. ESXi virtualizálás és Docker containerek megtervezése

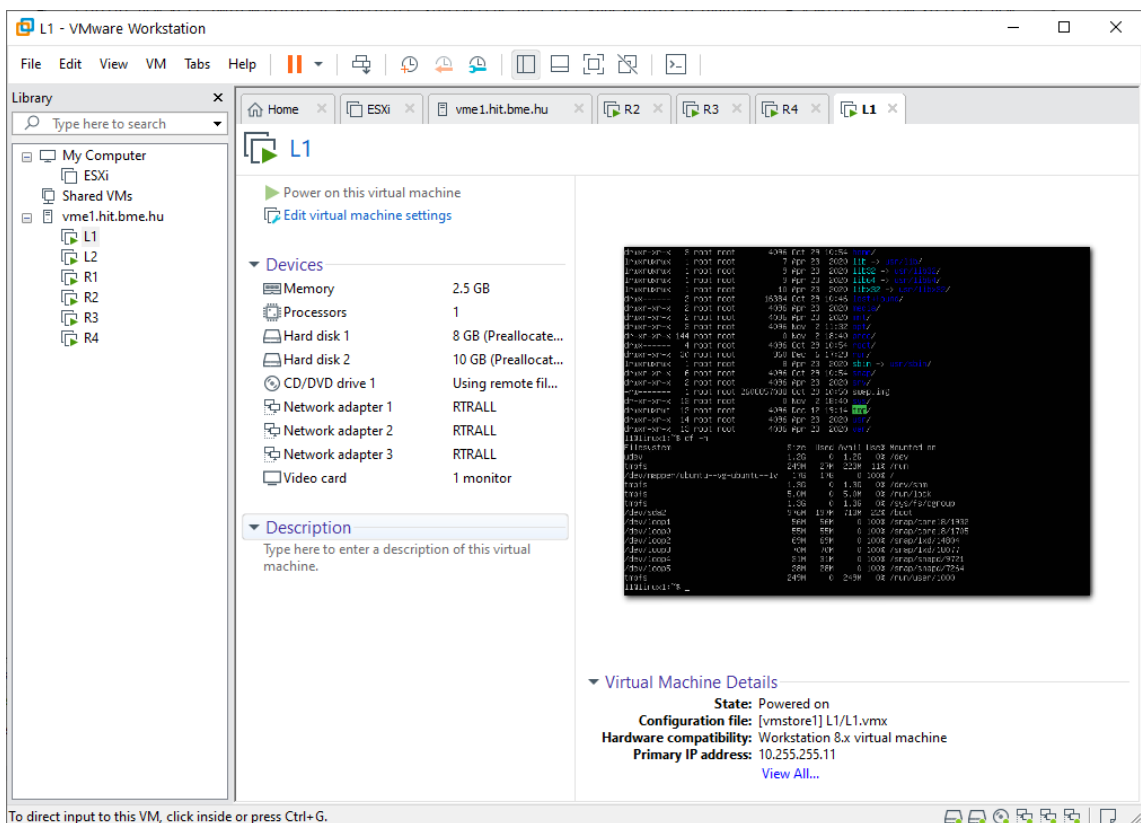
Az ESXi-hez kétféleképpen tudtam hozzáférni a biztonság miatt VPN tunnel kialakítása után. Volt egy web alapú GUI (graphical user interface) (4.8. ábra), ahol http-n keresztül tudtam megnyitni a terminálokat, valamint vmrc protokollon keresztül külön ablakban, vagy a gépemen lévő VMware Workstation felületén keresztül. Maga a VPN először az egyetemi VPN szerverein keresztül működött, majd az egyszerűség miatt egy saját OpenVPN access-t kaptam. Ezen keresztül biztonságosabb volt a hozzáférés, valamint több portot lehetett az ESXi-ből nyitni kifelé. A fejlesztés során közvetlen SSH, vagy FTP hozzáférésre nem volt lehetőségem. A GUI limitáltan engedélyezte a Copy+Paste funkciót kívülről, belülről csak körülményesen lehetett kimenteni adatokat, emiatt a terminál képei is elsősorban képként jelennek meg.

A hálózati logikai kialakításból már egyértelműen látszódik, hogy milyen logikai egységeknek kell megjelenüniük az ESXi-ben. Két darab Linux VM állt rendelkezésre, azon belül pedig teljes szabadságom volt a virtualizálási technológia kiválasztására és a VM-ek számának, módjának és hálózati megoldásának a kialakítására.

Mivel a korábban említett szempontok miatt nem Virtuális gépekben gondolkodtam a containeres megoldásokat kellett megterveznem. Az egyes containerek felhúzása és elindí-

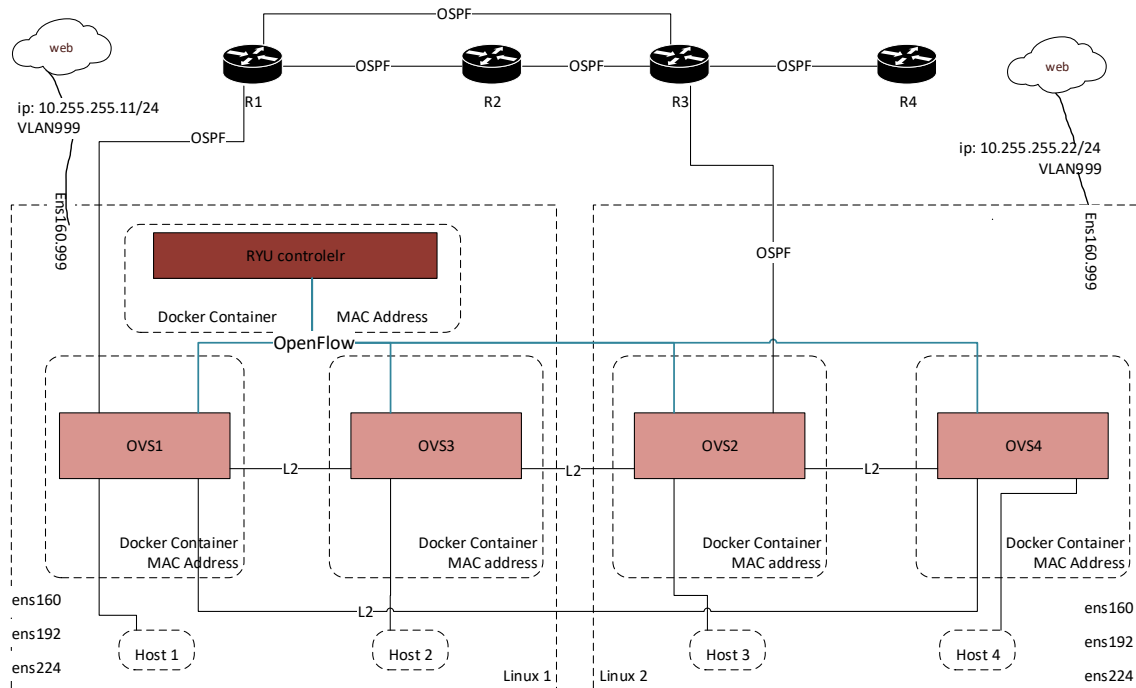


4.7. ábra. ESXi HTML alapú grafikus user interfésze



4.8. ábra. VMware Workstation GUI

tása nem jelent különösebb nehézséget, hasonló módon működik, mintha VM-eket indítanánk. Annyi előnyünk származhat belőle, hogy az egyes alkalmazásoknak találhatunk már meglévő Image-eket a Docker Hub-ról, így a Ryu kontrollert és az Open vSwitch-et is meglévő elemekből használtam fel, nem kellett egyenként telepítenem (Ryu Image: osrg/ryu, OVS image: socketplane/openvswitch).



4.9. ábra. Docker Containerok, logikai kapcsolatok és elválasztás

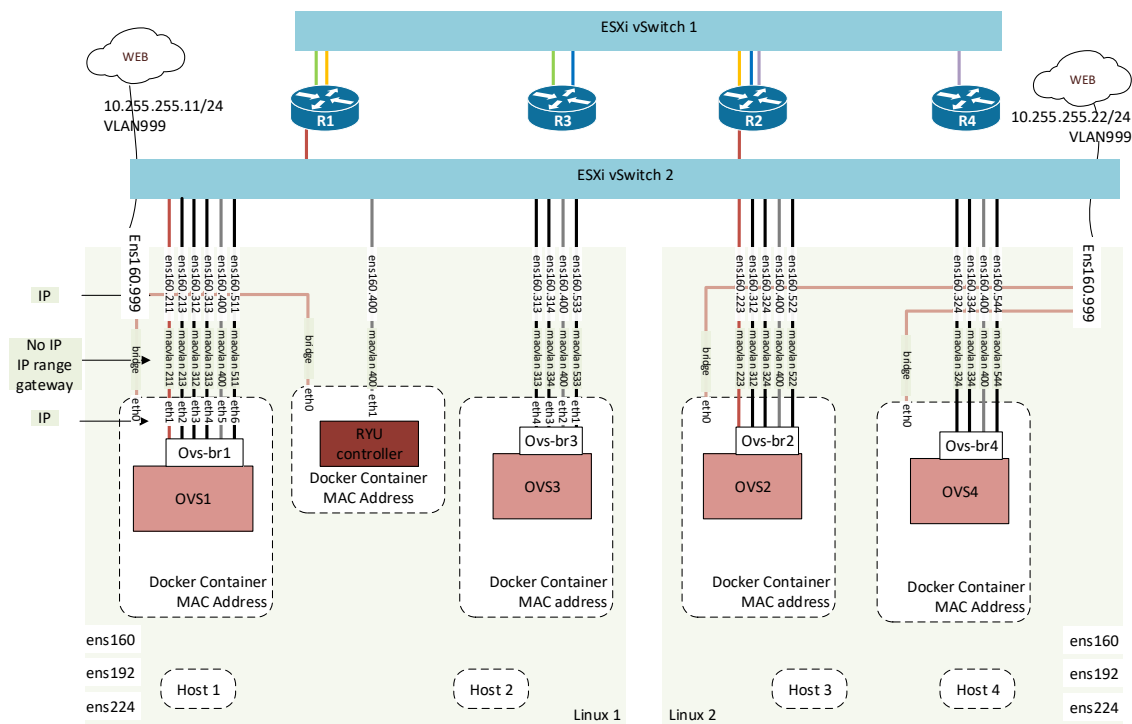
A 4.1.1. alszakaszban, ahálózati tervekben lévő ábrákban már jelölve vannak, hogy milyen virtuális elem van egy-egy konténerben. Kék keretben látható a 4 Open vSwitch és a Ryu controller is. A hostokat névterekkel terveztem szétválasztani.

```
$ sudo apt-get update
$
$ sudo apt-get install \
  apt-transport-https \
  ca-certificates \
  curl \
  gnupg-agent \
  software-properties-common
$
$ # Add 'Docker's official GPG key:
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
$
$ # Verify that you now have the key with the fingerprint 9DC8 5822 9FC7 DD38 854A E2D8 8D81
   803C 0EBF CD88, by searching for the last 8 characters of the fingerprint.
$
$ sudo apt-key fingerprint 0EBFCD88
$ # pub rsa4096 2017-02-22 [SCEA]
$ # 9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88
$ # uid [ unknown] Docker Release (CE deb) <docker@docker.com>
$ # sub rsa4096 2017-02-22 [S]
$
$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release
  -cs) stable"
$
$ # Install Docker Engine
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

4.2. lista. Docker telepítése

#### 4.1.3.1. Docker hálózati logika kialakítása

Érdekes szakmai feladatot és kihívást jelentett a Dockeren belül az úgynevezett Underlay hálózat kialakítása. Mivel a feladat a lehető legnagyobb, fizikainak látszó logikai szétválasztás, ezért az egyes containerekhez is MAC címet kellett rendelni (4.9. ábra). Ezt a macvlan interfészekkel lehetett megvalósítani.



4.10. ábra. Interfészek kialakítása és összekötése a Linuxban, Dockerben és az OVS-ben

A macvlanról volt már szó. A lényege az, hogy az egyes interfészekhez MAC címet rendel és VLAN taggel látja el a kimenő csomagokat, hogy ezáltal a switch, amire kapcsolódik külön "fizikai" egységként lássa, függetlenül attól, hogy az egy virtuális gépen vagy containerben van.

Maga a macvlan tervezése egyszerű volt, mert minden VLAN-hoz kellett rendelni egy-egy macvlant, hogy ezáltal valósulhasson meg a VLAN szerinti szétválasztás. Amit nehezebb megérteni, hogy miként vannak az interfészek, mi mivel van összekötve és mi hogyan csatlakozik.

A 4.10. ábra segítségével könnyen megérthetjük, hogy vannak ezek az interfészek összekötve. Kívül a host Linux *ens160* interfésze van csatlakoztatva az ESXi trunk switchéhez. Ez VLAN szerint szét van szedve subinterfészekre, hogy egyenként csatlakozzanak a Docker macvlan-jaira. A container belül csak *eth1*, *eth2*, stb. interfészeket lát, az OVS pedig ezekre kapcsolódik rá a kialakított bridge interfészekeken keresztül.

Látható a Linux1 OVS1-en még egy docker bridge interfész. Ez a containernek internet hozzáférését biztosította. Ez nem lett ábrázolva az összes containerben.

A konfigurálásra nézzünk egy példát (Linux 1, OVS1, Ryu és macvlan211-en keresztül):

```
$ sudo docker network create -d macvlan \
  --subnet=10.0.11.0/30 \
  --parent=ens160.211 \
  macvlan_211
```

```

$
$ # OVS Docker container létrehozása:
$ sudo docker run --itd --name=ovs1 --cap-add NET_ADMIN socketplane/openvswitch
$
$ docker run --itd --name vController osrg/ryu /bin/bash
$
$ sudo docker network connect --ip 10.0.11.2 macvlan_211 ovs1
$
$ # Verify:
$ sudo docker network ls
$ sudo docker network inspect macvlan_211
$ sudo docker container inspect macvlan_211

```

#### 4.3. lista. Docker hálózat konfigurálása

Ha már fel van telepítve a Docker a Linuxra, akkor – ahogy a kódban is látszik – először létrehoztam az interfészeket, majd elindítottam `docker run`-al a containert, csatlakoztattam a létrehozott networkot, majd ellenőriztem, hogy minden működik-e. Ezt minden egyes interfésszel és containerrel valamint OVS-el megtettem.

##### 4.1.4. SDN konfigurálás

Amint készen vannak a containerek lehet konfigurálni az egyes nodeokat. A Ryu-ban különösebb teendő nincs, mivel az csak a Python modulokat futtatja, majd az elkészült programot kell feltölteni. Magához a kontrollerhez 6633-as porton lehet hozzáférni.

Az Open vSwitchek konfigurálása már körülményesebb. Létre kell hozni bridzseket (*ovs-brx* – x: OVS azonosítója), majd ezen a bridge-n portokat. Ezeket a portokat hozzáillesztem a container portjaihoz. Végül megadom a kontroller elérhetőségét az OVS-eknek.

Példa az OVS1 *ovs-br1* konfigurálása által:

```

$ # Alias, hogy ne kelljen mindig beírni a 'sudo docker exec ovs' parancsot. c_ovs mint container ovs
$ alias c_ovs1='sudo docker exec ovs1'
$
$ # OVS bridge létrehozása és up állapotba állítása
$ c_ovs1 ovs-vsctl add-br ovs-br1
$ c_ovs1 ifconfig ovs-br1 up
$
$ # Verify:
$ c_ovs1 ovs-vsctl show
$
$ # OVS portok hozzáadása a containerek portjaihoz
$ c_ovs1 ovs-vsctl add-port ovs-br1 eth1
$ c_ovs1 ovs-vsctl add-port ovs-br1 eth2
$ c_ovs1 ovs-vsctl add-port ovs-br1 eth3
$ c_ovs1 ovs-vsctl add-port ovs-br1 eth4
$ c_ovs1 ovs-vsctl add-port ovs-br1 eth5
$ c_ovs1 ovs-vsctl add-port ovs-br1 eth6
$
$ # Kontroller hozzárendelése:
$ c_ovs1 ovs-vsctl set-controller ovs-br1 tcp:172.16.0.10

```

#### 4.4. lista. OVS config

##### 4.1.5. Névterek létrehozása virtuális host-ként

A hálózati elemeket már létrehoztam és konfiguráltam. Felépült az SDN és a legacy hálózat is. Már csak a hostok hiányoznak. Azt a már említett módon névterekkel oldottam meg.

A névterek a Linux networking funkciójának beépített eszköze. Az `ip-netns`[41] szétválasztja és izolálja a Linux különböző hálózati részeit. Mindegyik névtér rendelkezik saját

interfészekkel, különálló és független routing és forwarding táblával. Külön dolgozzák fel az egyes csomagokat és protollokat. Gyakorlatilag olyan, mintha egy különálló hálózati eszközről, hostról beszelnénk. Létezik egy úgynevezett root névtér is, ami a Linux alapértelmezett hálózati egysége. Ha nem hozunk létre új egységeket, ezt használjuk.

Maga a névterek kialakításának vizsgálatára az `ip a` és az `ip route` parancsok paraméterezésével és meghívásával lehet.

Ez az alkalmazás szempontjából elégséges a hostok kialakítására. A hálózat tesztelése és monitorozására elég, ha ezek a hostok pingelhetők és a feldolgozott adatsomagokat vizsgálhatjuk a hostokon, nem kell egy teljes értékű és különálló Linuxot használni.

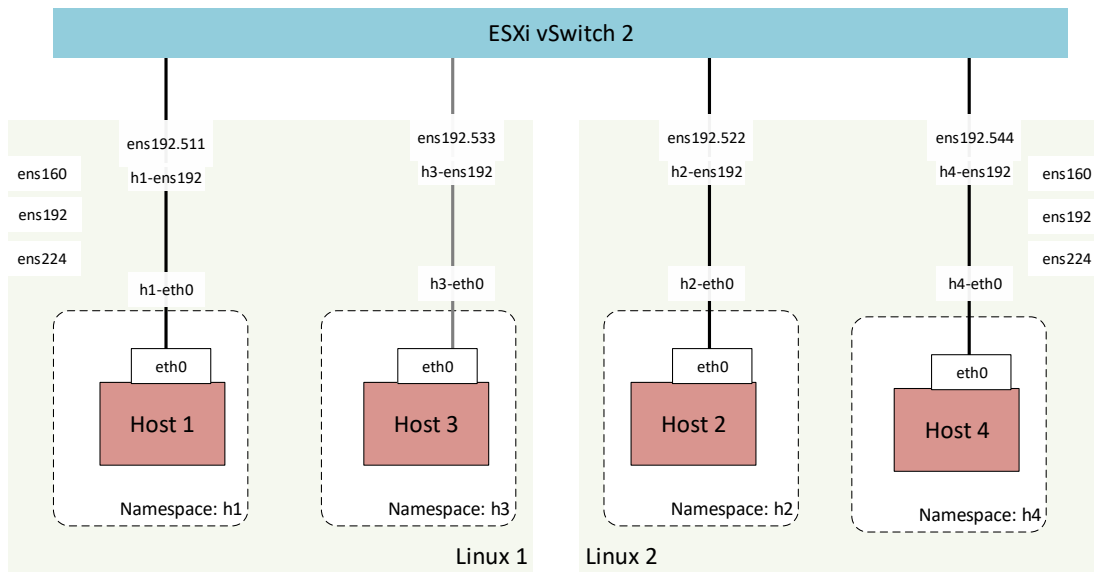
Névterek kialakítása a Linux 1-en:

```
$ # Új névtér létrehozása:
$ ip netns add h1
$ ip netns add h3
$
$ # Validálás
$ ls /var/run/netns
$
$ # Névtér törlése:
$ ip netns del h1
$
$ # Névtér vizsgálata és kezelése:
$ ip netns exec h1 ip link
$ ip link
$
$ # Csatlakoztatás interfészhez
$ ip link add h1-eth0 type veth peer name h1-ens192
$ ip link set h1-eth0 netns h1
$
$ # Ha OVS bridge-hez kötjük:
$ # ovs-vsctl add-port ovs-br1 h1-ens192
$ # De ez esetben a host interfészéhez kötjük
$
$ # Az interfészeket up állapotba állítjuk
$ ip link set h1-ens192 up
$ ip netns exec h1 ip link set dev lo up
$ ip netns exec h1 ip link set dev h1-eth0
$
$ ip netns exec h1 ip address add 192.168.1.3/29 dev h1-eth0
$
$ # Linux vlan konfigurálás az ens192 interfészre
$ sudo modprobe 8021q
$ sudo vconfig add ens192 511
$ # Ha ovs bridge lenne:
$ # sudo ip addr add 192.168.1.3/29 dev ens192.511
$ sudo ip link set up ens192.511
$
$ # h1-eth0 - h1-ens192 bridge csatlakoztatása a host vlan interfészéhez
$ ip link set h1-ens192 netns ens192.511
```

#### 4.5. lista. Névterek konfigurálása

Először létre kell hozni az egyes névtereket (`ip netns add *névtér név*`), majd ellenőrizni kell, hogy létre lettek-e hozva. Ha sikerült, az `ip link add *interface 1* type veth peer name *interface 2*` paranccsal létre kell hozni egy linket, ami még nincs hozzákötve közvetlenül egyetlen interfészhez sem. Ezután az `ip link set *interface 1* netns *névtér név*` paranccsal a létrehozott névtérben lévő interfészt hozzáadjuk a névtérhez. Ekkor már nem lehet látni a root névtérben (tehát ha egyszerűen kiadjuk az `ip link` parancsot). Ezután csak a névtéren belül lehet konfigurálni a linkeket az `ip link exec *névtér név* *ip parancsok*`





4.11. ábra. A Linuxokban kialakított névterek architektúrája

parancs beírása után. Up státuszba állítottam, hozzákötöttem a létrehozott linkre, majd konfiguráltam a Linux vlan interfészét, hogy azt a link másik végéhez kössem.

#### 4.1.6. Bash script

Mivel többször újra kellett telepíteni a két Linux VM-et, a teljes konfigurálást megírtam Bash scriptben. Ebbe elhelyeztem biztonsági ellenőrzéseket is (például, ha egy csomag telepítve volt nem telepítette újra), illetve ki lehetett választani a konfigurálandó Linux interfészeket (alapértelmezett *ens160*), valamint magát a Linuxot (Linux1 vagy Linux2).

```
$ ./linux_config.sh --help
: Helps my diploma project to execute configs easier
usage: COMMAND

Commands:
  linux1 [interface]    configure my Linux 1
                        Details: ...
  linux2 [interface]    configure my Linux 2

Parameters: port (default ens160)

Options:
  -h, --help            display this help message.
```

4.6. lista. Bash script használata

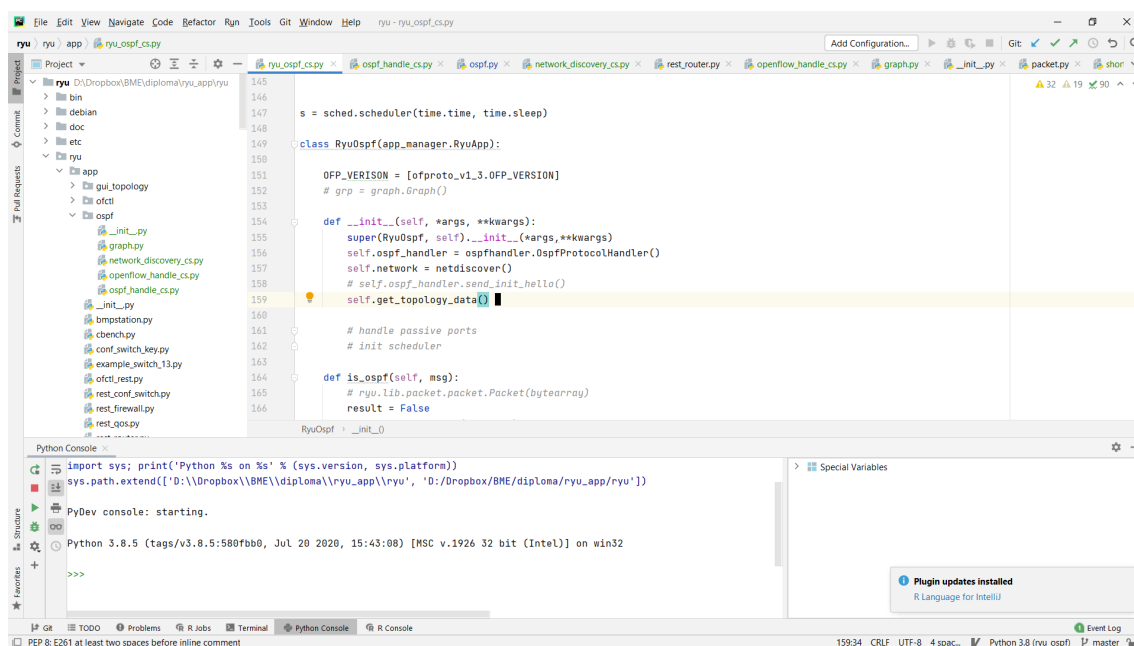
## 5. fejezet

# Hibrid SDN programozás

Amint elkészült az SDN hálózat és felkonfiguráltuk a routereket is nekiállhatunk az SDN programozásának. A diplomamunka során az elsődleges cél az volt, hogy a legacy oldalról minden egyes OVS úgy tűnjön, mintha egy teljes értékű router lenne a helyén. Ezáltal a router ugyanúgy beveszi az OF switchek által hirdetett networköket az LSDB adatbázisába. Természetesen az útvonalválasztást, a csomagküldéseket és a kommunikációt a virtuális hálózatok világában a hálózati egységek és üzenetek kezelését nem az sdn node-ok intézik, hanem csak a központi kontroller utasításait hajtják végre. Természetesen az egyes Flow táblák el tudják menteni a már ismerős csomagokat, de az új és ismeretlen döntési helyzeteket a központi logika intézi.

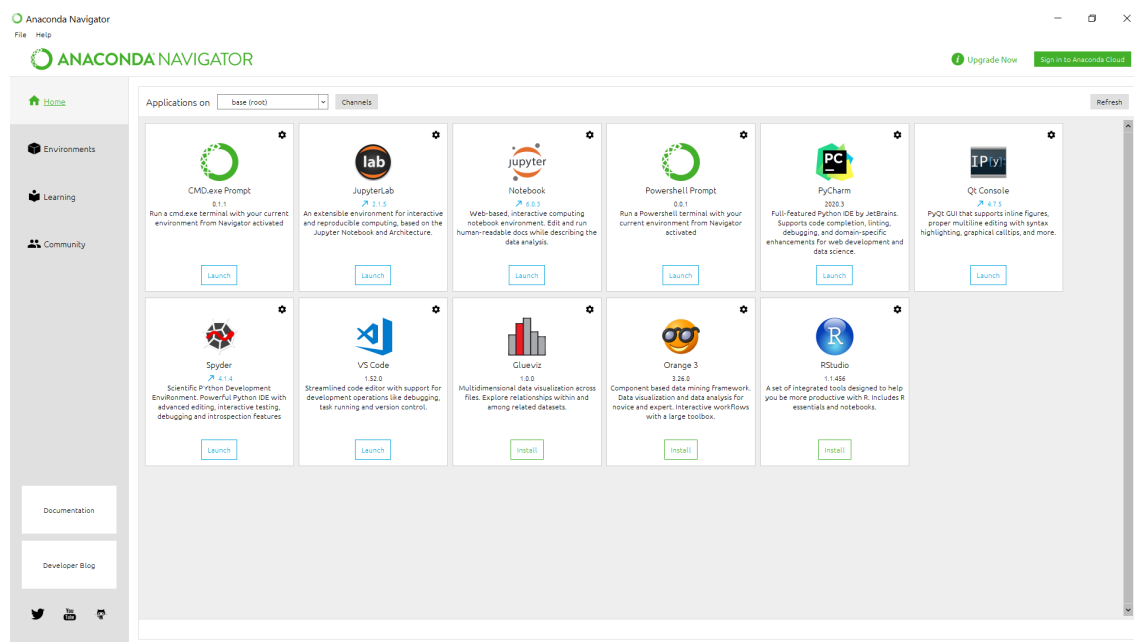
A program megírásakor cél volt, hogy egy kész környezetben (environment) már csak a programot kelljen betölteni a northbound interfészen keresztül, ne kelljen a kernel vagy architektúráis szinten módosítani a kontrollert, se a megvalósított környezetet.

Korábban megvizsgáltuk a Ryu programszerkezetét, logikáját és megnéztük, hogy milyen lehetőségeket tudunk kiemelni az SDN frameworkből (3.4.2. alszakasz). A Ryu egy python alapú, moduláris kontroller, többféle modul áll rendelkezésre különböző alkalmazások írására, amiből a fejlesztés során többet fel is lehetett használni. Cél volt az is, hogy ezekből minél több be legyen emelve az alkalmazásba.



5.1. ábra. PyCharm fejlesztői környezet

Magának a fejlesztésnek python egyik legnépszerűbb környezetében, a PyCharm-ban álltam neki, amit az Anaconda python segítő és kiegészítőket tartalmazó toolon keresztül telepítettem. Mivel nem volt korábbi ismeretem pythonban, ezért internetes kurzusok segítségével tanultam meg a nyelv elemeit.



5.2. ábra. Anaconda Navigator GUI

## 5.1. Statikus kapcsolat kialakítása, próbaprogramok futtatása

Az teljes funkcionalitás programozása előtt az SDN megismerése és megértése miatt ki lett alakítva egy stub szerű default route-al ellátott hálózat is. Itt az egyes nodeok az SDN-en belül switch funkcióval működtek, illetve hogyha ismeretlen IP címet kaptak, akkor automatikusan egy default route felé küldték a csomagokat. Az általam telepített portokat és létrehozott IP-ket beégettem a kódba, így nem volt szükség topology discovery-re. Itt a flow táblák közvetlen konfigurálásához használtam az `ovs-ofctl` parancsot is.

## 5.2. RyuOspf alkalmazás programozása

Az *RyuOspf* alkalmazásnak két fő résznek kell lennie: egyrészt a belső SDN nodeok kezelése és feltérképezése (ha új node jön be legyen illesztve a topology-ba, legyen a routingolás, OF topology lekérdezés, flow tábla kezelés), illetve a külső legacy network-ből jövő OSPF protokoll feldolgozása, beillesztése a hálózati képbe és ezek kezelése úgy, hogy a külső routereknek ne tűnjön fel, hogy teljesen más típusú hálózat van a szomszédságban.

Az virtuális hálózat világában az OpenFlow-nak köszönhetően a kontroller ismeri az SDN nodeokat. Ezt függvényekkel és eventekkel ki lehet nyerni ezt a térképet a Ryuból. A bejövő OSPF egységeket pedig az protokoll üzenetei által lehet felvenni a saját adatbázisba, az RID és bejövő LSA-k által el lehet menteni a szükséges és érdekes adatokat, hogy aztán összegezve a belső gráfot Dijkstra számítással ki lehessen számolni a legrövidebb utat, majd ebből következtetve az egyes OVS switcheknek le lehet küldeni a next-hop adatokat.

### 5.2.1. RyuOspf main főosztály

A RyuOspf main osztály elsődleges funkciója a Ryu és az OSPF northbound alkalmazás különböző funkcióinak kezelése, illetve a csomagok szűrését első körben itt kezeli a program. Ez fogja el az eseményeket, mint például a `EventOFPacketIn` bejövő OF üzenet eseményt, vagy új OVS-kor csatlakozott `EventSwitchEnter` eseményt.

Az osztály a `ryu-manager.RyuApp` osztály leszármazottja. Ezen keresztül kap hozzáférést a kontrollerhez és sdn specifikus függvényeket, ezáltal kapcsolódhat rá a Ryu által generált eseményekre.

A kód elején ki lehet választani az OpenFlow verziót. Én az egyszerűség és széleskörű támogatottság miatt az 1.3-as verziót használok. Több újdonságot tartalmaz az 1.0-hoz képest, viszont könnyebben érthető, mint a legújabb 1.5-ös.

Az inicializálásnál kell a változókat konstruálni:

**OFP\_VERSION** Ki lehet választani az OpenFlow verziót. Az őosztályból van leszármaztatva. Én a legújabb, 1.3-as OF-et használok.

**ospf\_handler** `OspfPacketHandler()` példánya, ez dolgozza fel a beérkező és kimenő OSPF üzenetet

**network** `netdiscover()` példánya, ez tárolja majd a hálózatot (LSDB)

**of\_handler** Az OpenFlow-val kapcsolatos funckiókat ennek az osztálynak a példánya kezeli.

Magának az osztálynak a motorja három esemény. A kontroller által generált eseményekre a `@set_ev_cls(<event type>, <action>)` dekorátorral lehet feliratkozni. Lehetőség önálló eseményeket is létrehozni a programban meglévő python modulokkal.

Függvények összefoglalva:

Függvény név	Esemény	Paraméterek	Részletek
<code>packet_in_handler()</code>	<code>ofp_event.EventOFPacketIn</code>	<code>self, ev</code>	Az OpenFlow bejövő üzeneteit kezeli
<code>get_topology_data()</code>	<code>event.EventSwitchEnter</code>	<code>self, ev</code>	Amennyiben változás lesz a hálózatban, ez a függvény hívódik meg. Ez hívja meg a hálózatkezelő osztályt.
<code>counter()</code>	<code>sched.scheduler</code>	<code>self, sleep</code>	Python beépített osztály Hello Update kezelésére. Megadott időnként hívódik újra.

### 5.1. táblázat. A RyuOspf függvényei

Az első event az OpenFlow által generált `Packet In`, ami minden OVS-ekből felküldött üzeneteket tartalmazza. Ezeket a kontroller egy FIFO pipeline sorban `ev` változó adatstruktúrájában adja tovább az eseményre feliratkozó függvénynek. Erre az eventre a `packet_in_handler` függvény csatlakozik rá, hogy feldolgozza a bejövő csomagokat.

Maga a függvény a beérkező üzenet feldolgozása során két fő típusú csomagra számít. Az egyik az OSPF csomag, a már említett módon a Hello, DBD, LSR, LSU és LSack. A másik egy ismeretlen nodeból érkező ETH csomag. Az első esetben átadjuk a kezelést az `ospf_handler` osztálynak, hogy aztán ő eldöntse, milyen típust hogy kezel. Az ismeretlen SDN nodeból érkező ETH package pedig hozzáadja a topológiához a csomópontot, vagy ha már fel van véve, megkeresi a hálózatban a címet (source vagy destination), hogy frissítse, ha nem találja, akkor elárasztja a hálózaton a beérkezett üzenetet.

A csomagot a már említett módon egy `ev` nevű változóban kapja meg a függvény, ami esetünkben egy `ryu.controller.ofp_event.EventOFPMsgBase(msg)` leszármazott, pontosabban egy `EventOFPPacketIn` példány. Az attribútumai a 5.2. táblázatban látható.

Attribútumok	Leírás
<code>msg</code>	An object which describes the corresponding OpenFlow message.
<code>msg.datapath</code>	A <code>ryu.controller.controller.Datapath</code> instance which describes an OpenFlow switch from which we received this OpenFlow message.
<code>timestamp</code>	Timestamp when Datapath instance generated this event.

## 5.2. táblázat. Az `ofp_event` attribútumai

Bontsuk ki a `datapath` elemet, szükségünk lesz még a funkcióira.

Attribútumok	Leírás
<code>id</code>	64-bit OpenFlow Datapath ID. Only available for <code>ryu.controller.handler.MAIN_DISPATCHER</code> phase.
<code>ofproto</code>	A module which exports OpenFlow definitions, mainly constants appeared in the specification, for the negotiated OpenFlow version. For example, <code>ryu.ofproto.ofproto_v1_0</code> for OpenFlow 1.0.
<code>ofproto_parser</code>	A module which exports OpenFlow wire message encoder and decoder for the negotiated OpenFlow version. For example, <code>ryu.ofproto.ofproto_v1_0_parser</code> for OpenFlow 1.0.
<code>ofproto_parser.OFPxxxx(datapath,...)</code>	A callable to prepare an OpenFlow message for the given switch. It can be sent with <code>Datapath.send_msg</code> later. <code>xxxx</code> is a name of the message. For example <code>OFPFlowMod</code> for flow-mod message. Arguments depend on the message.
<code>set_xid(self, msg)</code>	Generate an OpenFlow XID and put it in <code>msg.xid</code> .
<code>send_msg(self, msg)</code>	Queue an OpenFlow message to send to the corresponding switch. If <code>msg.xid</code> is None, <code>set_xid</code> is automatically called on the message before queueing.
<code>send_packet_out</code>	deprecated
<code>send_flow_mod</code>	deprecated
<code>send_flow_del</code>	deprecated
<code>send_delete_all_flows</code>	deprecated
<code>send_barrier</code>	Queue an OpenFlow barrier message to send to the switch.
<code>send_nxt_set_flow_format</code>	deprecated
<code>is_reserved_port</code>	deprecated

## 5.3. táblázat. Az `ryu.controller.controller.Datapath` elemei

És nézzük meg az OpenFlow PacketIn csomagot az OpenFlow dokumentációból:

```
struct ofp_packet_in {
    struct ofp_header header;
    uint32_t buffer_id; /* ID assigned by datapath. */
    uint16_t total_len; /* Full length of frame. */
    uint8_t reason; /* Reason packet is being sent (one of OFPR_*) */
    uint8_t table_id; /* ID of the table that was looked up */
    uint64_t cookie; /* Cookie of the flow entry that was looked up. */
    struct ofp_match match; /* Packet metadata. Variable size. */
    /* The variable size and padded match is always followed by:
     * - Exactly 2 all-zero padding bytes, then
     * - An Ethernet frame whose length is inferred from header.length.
    */
}
```

```

* The padding bytes preceding the Ethernet frame ensure that the IP
* header (if any) following the Ethernet header is 32-bit aligned.
*/
uint8_t pad[2]; /* Align to 64 bit + 16 bit */
uint8_t data[0]; /* Ethernet frame */
};
OFP_ASSERT(sizeof(struct ofp_packet_in) == 32);

```

### 5.1. lista. Packet In struktúrája és elemei

A `set_ev_cls(ev_cls, dispatcher)` dekorátor második paramétere lehet `HANDSHAKE_DISPATCHER`, ami SDN Hello üzeneteket küld és fogad, `CONFIG_DISPATCHER`, ami feature-request üzeneteket küld és fogad, az általunk használt `MAIN_DISPATCHER`, ami switch-feature set-config üzeneteket küld és fogad, illetve a `DEAD_DISPATCHER`, ami a peerek lecsatlakozásáért felelős.

Ezután nézzünk rá az általam megírt kódra:

```

class RyuOspf(app_manager.RyuApp):

    def __init__(self, *args, **kwargs)...

    @set_ev_cls(ofp_event.EventOFPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):

        # Open OpenFlow packe
        # First: decode incoming packet
        msg = ev.msg
        dp = msg.datapath
        dpid = dp.id
        ofproto = dp.ofproto

        # Parse packet and check what its type is:
        pkt = packet.Packet(array.array('B', msg.data))
        for p in pkt:
            if p.protocol_name == 'ospf':
                pck_type = ospfhandler.ospf_packet_in(p, self.network) # TODO How to check packet
type

            if p.protocol_name == 'eth':
                src = p.src
                dst = p.dst
                dpid = dp.id
                self.mac_to_port.setdefault(dpid, {})

                ## Shortest Path forwarding with OpenFlow on Ryu by castroflaviojr
                if src not in self.net: #Learn it
                    self.net.add_node(src) # Add a node to the graph
                    self.net.add_edge(src,dpid) # Add a link from the node to it's edge switch
                    self.net.add_edge(dpid,src,{ 'port':msg.in_port}) # Add link from switch to node and
make sure you are identifying the output port.
                    if dst in self.net:
                        path=nx.shortest_path(self.net,src,dst) # get shortest path
                        next=path[path.index(dpid)+1] #get next hop
                        out_port=self.net[dpid][next]['port'] get output port
                    else:
                        out_port = ofproto.OFPP_FLOOD
                ## Shortest Path forwarding with OpenFlow on Ryu by castroflaviojr

                self.network.handle_switich_msg(src, dst, self.network, dpid)
                ospf_pkt = pkt.get_protocol(ospf.ospf)
                eth = pkt.get_protocol(ethernet.ethernet)

                # if ospf.
                # osppkt.get_protocols(ospf.ospf)

```

```
self.check_ospf(msg)
```

### 5.2. lista. RyuOspf main osztály

Amennyiben nem OSPF üzenet jön, hanem csak egy L2-es csomag, akkor az alábbi választások állnak elő. Ha a source MAC ismeretlen, meg kell tanulni. Ha a destination MAC ismeretlen, el kell árasztani a hálózaton. Ha a destination MAC ismert, meg kell keresni a shortest path-ot, onnan ki kell nézni a next-hop-ot, majd ki kell küldeni FlowMod üzenettel.

A másik meghatározó esemény, ha frissül az OVS-ek listája. Ebben az esetben a network objektum által hívom meg a megfelelő funkciót. A harmadik esemény meg egy általam generált időzítő, ami 10 másodpercenként meghívja az `ospfhandler` objektum `send_hello()` függvényét

```
s = sched.scheduler(time.time, time.sleep)

class RyuOspf(app_manager.RyuApp):

    ...

    # When a new OpenvSwitch Enters
    @set_ev_cls(event.EventSwitchEnter)
    def get_topology_data(self, ev):
        self.network.build_network()

    def counter(self, sleep):
        s.enter(sleep, 1, ospfhandler.send_hello())
```

### 5.3. lista. RyuOspf main osztály egyéb eseményei

#### 5.2.2. OpenFlow handle modul

Az egyszerűség és modularitás kedvéért létrehoztam egy OpenFlow-t kezelő osztályt is. A már korábban is említett (2.2.2) három fő OpenFlow típus, ami kell nekünk: az aszinkron `PacketIn`, illetve a `Controller-to-Switch Modify-State FlowMod` és `PacketOut` üzenetek. Az elsőt már a főosztályban kezeltük, ezért azzal nem kell foglalkozni. Az osztályban a `Control-to-Switch` elemek kezelése valósul meg.

A `Modify-State` üzenetek `OFPPFlowMod` csomagjával az egyes OVS-ek `FlowTable` elemeit tudjuk módosítani, törölni vagy újat hozzáadni. Ezt abban esetben kell megtennünk, amikor újonnan csatlakozott a hálózathoz egy OSPF router, vagy új OVS lett felvéve a hálózatba, esetleg módosult a topológia. Mindkét esetben `Next-hop` utat akarunk hozzáadni a táblához Dijkstra számolás után. Az osztály elérése: `ryu.ofproto.ofproto_v1_3_parser.OFPPFlowMod`.

Mielőtt az egyes csomagküldést megnézzük vizsgáljuk meg a `FlowMod` elemeit, illetve hogy nekem milyen konfigurálás kell:

Használni kell az `OFPIstructionActions` függvényt, ahol első paraméterként elfogadjuk az `action`-t, amit a függvény hívásakor kaptunk. A kapott `action` állítja be, hogy milyen porton menjen ki a csomag (`packet_out`) és hogy `matchelés`nél milyen feladatot hajtson végre az OVS. A `OFPMatch` feltételt pedig szabadon lehet eldönteni. Én csak a IPv4 címet állítottam be `match` feltételnek.

`OFPMatch` csomagban lehet meghatározni a szükséges `match` feltételeket, tehát hogy milyen feltételekkel válassza ki az adott `FlowTable` entry-t. Esetünkben mivel routing protokollhoz hasonlónak kell lennie elég a `Destination` cím alapján döntenie (`ipv4_dst`).

Attribútumok	Default	Leírás
cookie	0	Opaque controller-issued identifier
cookie_mask	0	Mask used to restrict the cookie bits that must match when the command is OFPFC_MODIFY* or OFPFC_DELETE*
table_id	0	ID of the table to put the flow in
command	OFPFC_ADD	One of the following values: OFPFC_ADD, OFPFC_MODIFY, OFPFC_MODIFY_STRICT, OFPFC_DELETE, OFPFC_DELETE_STRICT
idle_timeout	0	Idle time before discarding (seconds)
hard_timeout	0	Max time before discarding (seconds)
priority	default	Priority level of flow entry
buffer_id	msg.packet-in	Buffered packet to apply to (or OFP_NO_BUFFER)
out_port	na	For OFPFC_DELETE* commands, require matching entries to include this as an output port
out_group	na	For OFPFC_DELETE* commands, require matching entries to include this as an output group
flags	OFPFF_SEND_FLOW_REM	Bitmap of the following flags: OFPFF_SEND_FLOW_REM, OFPFF_CHECK_OVERLAP, OFPFF_RESET_COUNTS, OFPFF_NO_PKT_COUNTS, OFPFF_NO_BYT_COUNTS
match		Instance of OFPMatch
instructions		list of OFPInstruction* instance

#### 5.4. táblázat. FlowMod változói

Az osztály másik függvénye a `send_packet_out()`. Itt a `OFPPacketOut()` függvény paramétereit kell beállítani, hogy aztán ki lehessen küldeni az OVS-eknek a kiküldendő csomag továbbításának adatait.

Attribútumok	Értékek	Leírás
buffer_id	dp.ofproto	ID assigned by datapath (OFP_NO_BUFFER if none)
in_port		Packet's input port or OFPP_CONTROLLER
actions	OFPActionOutput	list of OpenFlow action class
data	msg	Packet data of a binary type value or an instances of packet.Packet.

#### 5.5. táblázat. A Packet-Out attribútumai

A kódban a paraméterek beírása után látható, hogy a `send_msg(msg)` függvény küldi ki a szükséges üzenetet a switcheknek.

```
OFP_NO_BUFFER = 0xffffffff
```

```
class OpenFlowHandle(app_manager.RyuApp):
```

```

    def __init__(self, *args, **kwargs):
        super(OpenFlowHandle, self).__init__(*args, **kwargs)

    def send_flow_mod(self, datapath, actions, ipv4_dst, buffer_id = OFP_NO_BUFFER):
        ofp = datapath.ofproto
        ofp_parser = datapath.ofproto_parser

        cookie = cookie_mask = 0
        idle_timeout = hard_timeout = 0
        priority = ofp.OFP_DEFAULT_PRIORITY
        command = ofp.OFPFC_ADD
        flags = ofp.OFPFF_SEND_FLOW_REM
```



```

inst = [ofp_parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS,
                                         actions)]

match = ofp_parser.OFPMatch(ipv4_dst = ipv4_dst)

mod = ofp_parser.OFPFlowMod(
    datapath = datapath, match = match, cookie = cookie, command = command,
    idle_timeout = idle_timeout, hard_timeout = hard_timeout, buffer_id = buffer_id,
    priority = priority, flags = flags, instruction = inst)

datapath.send_msg(mod)

def send_packet_out(self, datapath, buffer_id, in_port,
                    out_port, msg_ser):
    ofp = datapath.ofproto
    ofp_parser = datapath.ofproto_parser

    actions = [ofp_parser.OFPActionOutput(out_port, 0)]
    req = ofp_parser.OFPPacketOut(datapath, buffer_id,
                                   in_port, actions, msg_ser)

    datapath.send_msg(req)

```

**5.4. lista.** OpenFlowHandle osztály

### 5.2.3. OSPF handle modul

Az OSPF modulnak kívülről nézve teljes értékű OSPF protokollt kell mutatnia. Az LSDB-nek valós 1-es típusú router LSA-kat kell tárolnia és a fontosabb adataival kell feltölteni az adatbázist. Magát az adatbázist a hálózati térképet tartalmazó **NetworkTopology** modul tartalmazza, az egyes node-okhoz lesz hozzá rendelve az IP, a mask, a state és az RID is.

Mielőtt belenézünk a kódba vizsgáljuk meg, mik kellenek az OSPF üzenetekhez. Ehhez vizsgáljuk meg a legfontosabb csomagok tartalmát és headerjeit. Az adatbázis szempontjából a legfontosabb a router LSA-k, ezeknek a headerjét használja fel a többi csomag is és ezeket tárolja az LSDB.

Az üzenetek összeállítása szempontjából fontos eltárolnunk az adatbázisban az RID-t (Advertising Router), rendelni kell az egyes csomópontokhoz LSID-t, valamint az egyes linkeknek az Link ID-t (1 - szomszéd router, 3 - hirdetett network), IP-t és a Subnet Maskot. Nem eltárolandó, de érdekes lesz még a linkek száma, ki kell majd számolni az LSA hosszát, a Type értéke 1 lesz (point-to-point), a többi lehet általánosan 0 vagy alapértelmezett érték.

Érdemes megnézni még a Hello csomagot, mivel ez gyakran kerül kiküldésre Tartozik hozzá két számláló, a RouterDeadInterval (40 lesz) és a HelloInterval (10 lesz), valamint a DR és a BDR routerek információt jelölő flag is, amit ez esetben 0-ra kell állítani. A többi adat az LSA-kból kinyerhető, vagy default értéken tartható.

Magának a modulnak a feladata a beérkező csomag azonosítása, feldolgozása és a protolloknak megfelelő csomagok kiküldése. A feldolgozás az OSPFProtocolHandler osztály **ospf\_packet\_in()** függvényében kezdődik meg, ahol tovább bontja az OSPF csomagot a program és eldönti, hogy kell feldolgozni.

```

class OspfProtocolHandler:
    ...
    def ospf_packet_in(self, ospf_msg, network):
        if(ospf_msg.get_protocol(ospf.OSPFHello)):
            self.hello_in(ospf_msg, network)
        elif(ospf_msg.get_protocol(ospf.OSPFDBDesc)):
            self.dbd_in(ospf_msg, network)

```

```

0                               1                               2                               3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|               LS age               |      Options      |      1      |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|               Link State ID               |               |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|               Advertising Router               |               |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|               LS sequence number               |               |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|      LS checksum      |      length      |               |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|      0      |V|E|B|      0      |      # links      |               |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|               Link ID               |               |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|               Link Data               |               |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|      Type      |      # TOS      |      metric      |               |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|               ...               |               |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|      TOS      |      0      |      TOS metric      |               |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|               Link ID               |               |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|               Link Data               |               |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|               ...               |               |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|               |               |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

```
elif(ospf_msg.get_protocol(ospf.OSPF_LSReq)):
self.lsreq_in(ospf_msg, network)
elif(ospf_msg.get_protocol(ospf.OSPF_LSUpd)):
self.lsu_in(ospf_msg, network)
elif(ospf_msg.get_protocol(ospf.OSPF_LSAck)):
self.lsakc_in(ospf_msg, network)
else:
print("It wasn't an OSPF packet")

return ospf_msg.get_protocol(ospf.ospf)
...
```

Megnézve a már korábban megvizsgált szomszédsági protokollt a 2.1.2.1. alaszakaszban láthatjuk, mit kell kezdeni az egyes beérkező csomagokkal. Először a Hello üzenet, majd a DBD, LSReq, LSU és LSack üzenetek jelennek meg a kommunikációban. Az OSPF linkek állapotokat a NetworkDiscovery modul tartalmazza.

0										1										2										3																			
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1																		
Version #										1										Packet length																													
										Router ID																																							
										Area ID																																							
Checksum																				AuType																													
										Authentication																																							
										Authentication																																							
										Network Mask																																							
HelloInterval																				Options										Rtr Pri																			
										RouterDeadInterval																																							
										Designated Router																																							
										Backup Designated Router																																							
										Neighbor																																							
										...																																							

5.6. lista. Hello csomag formátum

csomagot kell küldenie. A negyedik és ötödik esemény pedig az időzített, ciklikus frissítés küldése és fogadása. Tehát **Packet in** esetben, ezáltal ha beérkező Hello-t kapunk, akkor válaszul minden esetben a jelenleg feltérképezett hálózatot tartalmazó Hello üzenetet kell küldeni.

DBD üzenet csak inicializáláskor jelenik meg, amikor a Router **ExStart** és **Exchange** státuszban van. A csomag küldését először a belépő router kezdi (esetünkben úgy vesszük, hogy ez a kontroller), majd - mivel úgy terveztem a hálózatot - a legacy csomópont átveszi a master szerepet és az irányítást is. Ez esetben már csak válaszolni kell a DBD-re. A DBD cserénél ellenőrizni kell, hogy a benne található LSA header-ek megegyeznek-e az eltárolt LSDB (`NetworkDiscovery.get_network_summary()`) tagjaival. Amennyiben különbség van, LSR csomaggal LSA frissítést kell kérni.

Az LSR, LSU és LSack összefüggő csomagok. Az LSR a kapcsolat kiépítésénél a Loading state-ben lesz kiküldve abban az esetben, ha a DBD entry-k nem egyeznek meg az eltárolt hálózati adatokkal. Amennyiben különbségek vannak, LSR-el LSA update üzeneteket kérünk. LSR fogadás esetén LSU küldésre van szükség. Az LSU küldés lehet válasz az LSR-re, tehát egy LSR Packet In esemény is meghívhatja, valamint akkor is kimehet, amikor változás történik a hálózatban. LSU küldés esetében az LSDB-ből ki kell keresni a kapott LSA header alapján a részletes adatokat, hogy össze lehessen állítani az LSA-kat, amit bele lehet illeszteni a kiküldendő csomagba. LSU-ra a válasz, ha megérkezett egy LSack üzenet.

A Ryu már tartalmazza az OSPF csomagok legfontosabb részeit, így a le lett kódolva a csomagstruktúra, a parse és serialize függvények. Én ezeket felhasználva hoztam létre

általános kezelésre szolgáló `OSPFHandler` osztályt, valamint csomagonként `Hello`, `DBD`, `LSR`, `LSU` és `LSAck` osztályokat a `Ryu-s` osztályok leszármazottjaiként. Ezáltal elég csak a függvényeket megírni. Mindegyiknél van `packet_in()` függvény, ami az adott csomag beérkezését dolgozza fel a fent említett módon, illetve a `send_packet()` pedig az adott típus küldését oldja meg szükség esetén megadott paraméterekkel.

#### 5.2.4. NetworkDiscovery modul

A Ryu OpenFlow által felépített hálózati térkép mellett az alkalmazás számára szükség van egy saját adatbázisra is, ami az OSPF szomszédok adatait is tartalmazni tudja. Szükség van az OSPF által használt Djisktra shortest path algoritmusra is, amivel a next-hop node-ot is ki lehet számolni. Itt tárolódnak az LSA-val kapcsolatos adatok is, mint például RID-k, a mask, hirdetett hálózatok és minden szükséges adat.

Tárolandó változók:

- node ID = `nx.Graph()`
  - RID - LSA-ba is
  - mask - LSA-ba is
  - Neighbor state - OSPF számításához
- linkek

Ehhez a modulhoz több segítség is adódott. A Ryuhoz található a Git-en egy hasonló hálózati felderítéssel foglalkozó projekt. Ezt a kódot a <https://github.com/castroflavio/ryu> oldalon érhetjük el, illetve volt egy dokumentáció[35] is hozzá (sajnos a magyarázó oldal azóta törölve lett). Másrészt a Pythonhoz van egy *Netorkx*[42] nevű, kifejezetten hálózati adatbázis (gráf) tárolással, kezeléssel, feldolgozással foglalkozó csomag. Ez szabadon bővíthető és használható bármilyen alkalmazásra.

A Network Topology célja, hogy az SDN-ben felderített networkon kívül eltárolja az OSPF nodeokat is. Ez funkcionál tulajdonképpen LSDB-kén is, így LSA entry-ként tárolja a csúcsoakat, ezáltal az egyes csomópontokhoz RID-t, hirdetett hálózati adatokat és egyéb LSA-hoz tartozó adatokat el kell tárolnia. Mivel egy single area hálózatról van szó, elég csak 1-es típusú LSA-ról értekezni.

A modulnak tudnia kell node-ot hozzáadni, elvenni és módosítani a hálózatban, változás esetén legrövidebb utat kell számolnia és frissítenie kell a Flow táblákat és OSPF Update üzenetet kell küldenie, kérésre át kell tudnia adnia specifikus adatokat például DBD vagy Hello csomag összeállításához, adott esetben OpenFlow FlowMod vagy PacketOut üzenethez.

A `EventSwitchEnter` esemény meghívja a `build_network(self, event)` függvényt. Ez kezdi meg a hálózati topológia felderítését a `get_switch` és `get_link` függvények meghívásával. Ez egy tömböt ad vissza az SDN switchekkel és a linkekkel. Ezt hozzá tudjuk adni a hálózathoz. Tekintettel arra, hogy itt még nem tudunk konkrétabb információkat a csomópontokról, ezért azt később, a csomag küldésekor, Packet In üzenet feldolgozása közben tudjuk hozzáadni.

Új csomópont és link hozzáadáskor az `add_node()` és `add_edge()` függvény hívódik meg, ami az elem hozzáadása után rögtön futtat egy Djisktra-t és a frissített adatok alapján módosítja az összes OVS flow tábláját és OSPF LSU-update üzenetet küld ki a legacy routereknek. Maga az adatbázis egy gráf mátrix, sok paraméterrel és adattal. Az út számításánál figyelembe kell venni azt is, hogy a csomagok ne ragadjanak be egy hálózatban lévő hurokba. Elem

Attribútumok	Értékek	Példa
dpid	Datapath ID	"1"
mfr_desc	Manufacturer description	"Nicira, Inc.",
hw_desc	Hardware description	"Open vSwitch",
sw_desc	Software description	"2.3.90",
serial_num	Serial number	"None",
dp_desc	Human readable description of datapath	"None"

### 5.6. táblázat. A topology API-ből (get\_switch) lekérhető adatok

hozzáadása a `net.add_node(node, rid = rid, mask = mask, state = state)` függvénnyel lehetséges, majd a Dijkstra-t az összes source és destination node megnevezésével a `shortest_path(net, source = src, target = dst, method='dijkstra')` függvénnyel lehet számítani, hogy aztán az OpenFlow Handle osztályban megírt `send_flow_mod()` függvénnyel módosítsuk az OVS-ek tábláját. Az SDN-ben mindegyik linket 1 költségűnek tekintünk. A FlowMod kiküldése előtt létre kell hozni `ryu.lib.packet.openflow.openflow(msg)` OpenFlow csomagot datapath-nak.

Csomópont módosító függvényt is kell, hogy változtatni lehessenek az adatokat a hálózatban, illetve azért, hogy a `build_network()` függvény által felderített hálózat után fel lehessen venni a csomópontok részletes adatait is a portokon bejövő csomagok segítségével. A nodeok módosítását a node attribútumainak beírásával egyszerűen új értéket adhatunk a default, vagy előző érték helyett.

Szükséges még lekérdezni a csomópontok adatait `get_node_info()` függvénnyel a paraméterben megadott node egy dictionary tömb visszatérő értékeként lehet lekérdezni az információkat. Ezt lehet felhasználni LSA, Hello, vagy OSPF csomag összeállítására. A `get_network_summary()` függvény a hálózat összegzését adja vissza egy struktúrában (LSA header információkat).

Bár a kiegészítő modulnak köszönhetően nem kell megírni a Dijkstra algoritmust, de a megértéséhez érdemes egy pillantást vetni rá. Az legrövidebb út számítás egy csomópont kiválasztásával kezdődik, aminek 0 értéket adunk, az összes többi végtelen. Megvizsgáljuk a szomszédos csomópontokat, majd kiszámoljuk a távolságukat. Ha minden egyes szomszéd távolságát kiszámítottuk megjelöljük a vizsgált csomópontokat. Majd ezután a jelöletlen csomópontok fele a legkisebb távolságú csomópont felől haladunk tovább. Újra megvizsgáljuk a szomszédokat, újra kiszámoljuk a távolságot, megjelöljük és a legkisebb úton haladunk tovább, amíg a teljes gráfot fel nem térképeztük.

```
# Python program for Dijkstra's single
# source shortest path algorithm. The program is
# for adjacency matrix representation of the graph
# source: https://www.geeksforgeeks.org/python-program-for-dijkstras-shortest-path-algorithm-greedy-algo-7/

# Library for INT_MAX
import sys

class Graph():
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                       for row in range(vertices)]

    def minDistance(self, dist, sptSet):
        min = sys.maxsize
        for v in range(self.V):
```

```

    if dist[v] < min and sptSet[v] == False:
        min = dist[v]
        min_index = v
    return min_index

def dijkstra(self, src):
    dist = [sys.maxsize] * self.V
    dist[src] = 0
    sptSet = [False] * self.V

    for cout in range(self.V):
        u = self.minDistance(dist, sptSet)
        sptSet[u] = True
        for v in range(self.V):
            if self.graph[u][v] > 0 and \
               sptSet[v] == False and \
               dist[v] > dist[u] + self.graph[u][v]:
                dist[v] = dist[u] + self.graph[u][v]

```

**5.8. lista.** Djisktra mintakód

### 5.2.5. RyuOspf alkalmazás futtatása

A programot a `/ryu/app` könyvtárba másolás után lehet indítani. Mielőtt tényleg lefuttatjuk az alkalmazást szükséges pár egyéb funkciót is indítani, vagy csomagot telepíteni. Így például szükségünk lesz a `sudo pip install networkx` csomagra a NetworkX futtatására, illetve a topológia lekérdezés miatt indítani kell a ryuból egy `ryu-manager sp.py --observe-links` modult. Tehát összegezve

```

$ pip install networkx
$ ryu-manager sp.py --observe-links
$ ryu-manager --observe-links RyuOspf.py

```

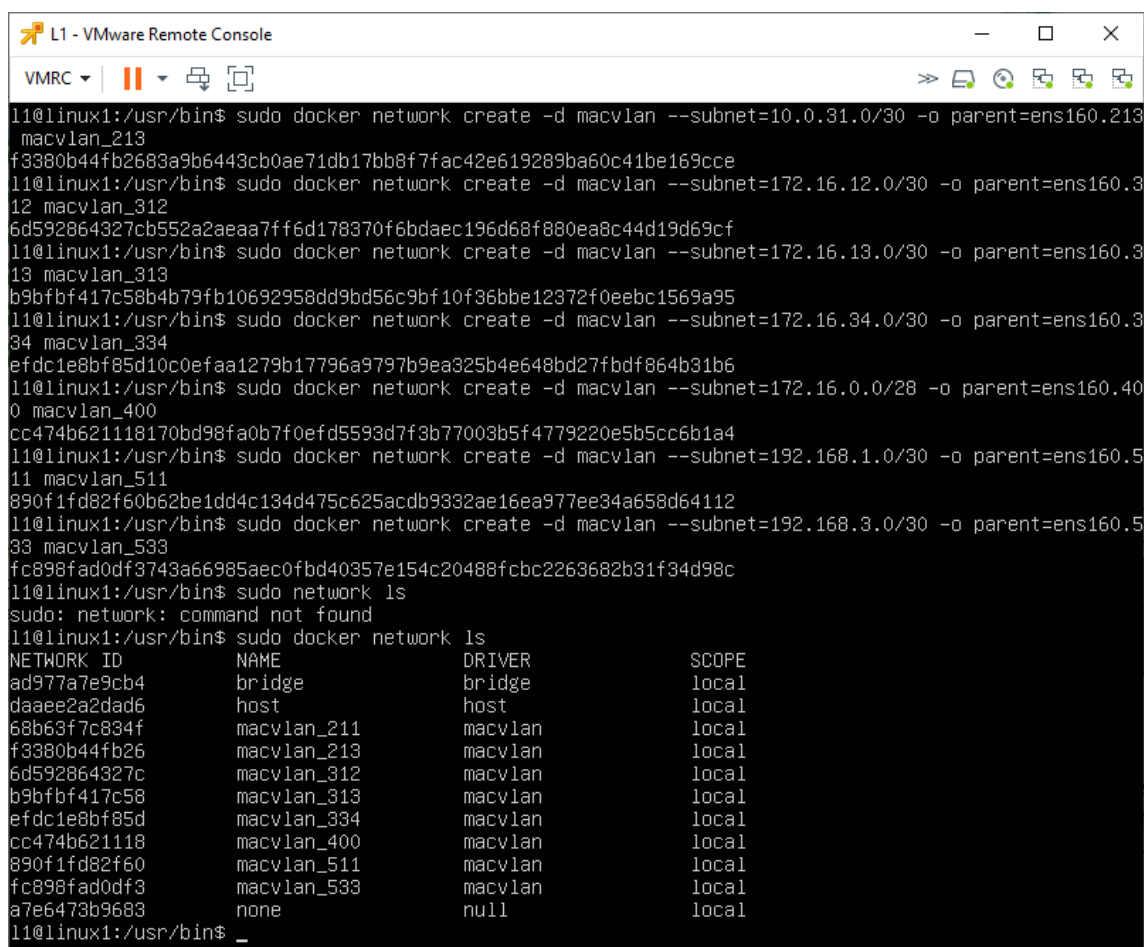
**5.9. lista.** RyuOspf futtatása

## 6. fejezet

# Eredmények

Sajnos az eredmények kiértékelésére és feldolgozására az environment kiépítésének bonyolalmai miatt már nem maradt idő. Mindig, amikor elkészült egy hálózati rész pingeléssel ellenőriztem a kapcsolatot, amivel megbizonyosodtam, hogy ténylegesen létrejött a kapcsolat. A jelenlegi állapotban még nem kommunikáltak egymással a hostok, így azok között nem működött a ping. Illetve nagyon korlátozott lehetőségeim voltak a környezetből adatokat kinyerni, ezért amiket nem mentettem el képként az adott állapotban utólag már nem volt lehetőségem ellenőrizni.

Az alábbi ábrákon az environment felépülését látjuk, valamint a már felépült kapcsolatokat.



```
L1 - VMware Remote Console
VMRC
l1@linux1:/usr/bin$ sudo docker network create -d macvlan --subnet=10.0.31.0/30 -o parent=ens160.213 macvlan_213
f3380b44fb2683a9b6443cb0ae71db17bb8f7fac42e619289ba60c41be169cce
l1@linux1:/usr/bin$ sudo docker network create -d macvlan --subnet=172.16.12.0/30 -o parent=ens160.312 macvlan_312
6d592864327cb552a2aeaa7ff6d178370f6bdaec196d68f880ea8c44d19d69cf
l1@linux1:/usr/bin$ sudo docker network create -d macvlan --subnet=172.16.13.0/30 -o parent=ens160.313 macvlan_313
b9bfbf417c58b4b79fb10692958dd9bd56c9bf10f36bbe12372f0eebc1569a95
l1@linux1:/usr/bin$ sudo docker network create -d macvlan --subnet=172.16.34.0/30 -o parent=ens160.34 macvlan_334
efdc1e8bf85d10c0efaa1279b17796a9797b9ea325b4e648bd27bfdf864b31b6
l1@linux1:/usr/bin$ sudo docker network create -d macvlan --subnet=172.16.0.0/28 -o parent=ens160.400 macvlan_400
cc474b621118170bd98fa0b7f0efd5593d7f3b77003b5f4779220e5b5cc6b1a4
l1@linux1:/usr/bin$ sudo docker network create -d macvlan --subnet=192.168.1.0/30 -o parent=ens160.511 macvlan_511
890f1fd82f60b62be1dd4c134d475c625acdb9332ae16ea977ee34a658d64112
l1@linux1:/usr/bin$ sudo docker network create -d macvlan --subnet=192.168.3.0/30 -o parent=ens160.533 macvlan_533
fc898fad0df3743a66985aec0fbd40357e154c20488fcbc2263682b31f34d98c
l1@linux1:/usr/bin$ sudo network ls
sudo: network: command not found
l1@linux1:/usr/bin$ sudo docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
ad977a7e9cb4        bridge             bridge             local
daaee2a2dad6        host               host               local
68b63f7c834f        macvlan_211       macvlan            local
f3380b44fb26        macvlan_213       macvlan            local
6d592864327c        macvlan_312       macvlan            local
b9bfbf417c58        macvlan_313       macvlan            local
efdc1e8bf85d        macvlan_334       macvlan            local
cc474b621118        macvlan_400       macvlan            local
890f1fd82f60        macvlan_511       macvlan            local
fc898fad0df3        macvlan_533       macvlan            local
a7e6473b9683        none              null               local
l1@linux1:/usr/bin$ _
```

6.1. ábra. Macvlan interfészek a Linux1-ben

Először a Docker, containerek és a macvlan felépítésével foglalkoztam és azokat ellenőriztem a megadott parancsokkal. Hiba esetén javítottam azokat a megfelelő módon. Majd amikor maga a konténeres környezet felépült az SDN hálózati megoldásokat ellenőriztem. Megvizsgáltam az OVS bridzseket, van-e kapcsolat egymás között, illetve látják-e a kontrollert. A kapcsolat felépült, de mivel program nem határozta meg az OVS-ek működését nem volt kapcsolat. Amikor elindítottam egy példakódot (például switchy.py), akkor látható volt L2-es kapcsolat.

```

Selecting previously unselected package iputils-ping.
(Reading database ... 5622 files and directories currently installed.)
Preparing to unpack .../iputils-ping_3%3a20190709-3_amd64.deb ...
Unpacking iputils-ping (3:20190709-3) ...
Setting up iputils-ping (3:20190709-3) ...
l1@linux1:/usr/bin$ sudo docker exec ovs3 ping 172.16.13.2
PING 172.16.13.2 (172.16.13.2) 56(84) bytes of data.
64 bytes from 172.16.13.2: icmp_seq=1 ttl=64 time=0.118 ms
64 bytes from 172.16.13.2: icmp_seq=2 ttl=64 time=0.044 ms
64 bytes from 172.16.13.2: icmp_seq=3 ttl=64 time=0.047 ms
64 bytes from 172.16.13.2: icmp_seq=4 ttl=64 time=0.048 ms
^C
l1@linux1:/usr/bin$ sudo docker exec ovs3 ping 172.16.0.2
PING 172.16.0.2 (172.16.0.2) 56(84) bytes of data.
64 bytes from 172.16.0.2: icmp_seq=1 ttl=64 time=0.128 ms
64 bytes from 172.16.0.2: icmp_seq=2 ttl=64 time=0.078 ms
64 bytes from 172.16.0.2: icmp_seq=3 ttl=64 time=0.062 ms
^C
l1@linux1:/usr/bin$ sudo docker exec ovs3 ping 172.16.0.5
^C
l1@linux1:/usr/bin$ sudo docker exec ovs3 ping 172.16.0.10
PING 172.16.0.10 (172.16.0.10) 56(84) bytes of data.
64 bytes from 172.16.0.10: icmp_seq=1 ttl=64 time=0.104 ms
64 bytes from 172.16.0.10: icmp_seq=2 ttl=64 time=0.027 ms
64 bytes from 172.16.0.10: icmp_seq=3 ttl=64 time=0.037 ms
64 bytes from 172.16.0.10: icmp_seq=4 ttl=64 time=0.028 ms
64 bytes from 172.16.0.10: icmp_seq=5 ttl=64 time=0.031 ms
64 bytes from 172.16.0.10: icmp_seq=6 ttl=64 time=0.059 ms
64 bytes from 172.16.0.10: icmp_seq=7 ttl=64 time=0.033 ms
^C
l1@linux1:/usr/bin$ sudo docker exec ovs-vstcl add-br1
Error: No such container: ovs-vstcl
l1@linux1:/usr/bin$ sudo docker exec ovs1 ovs-vstcl add-br1
ovs-vstcl: unknown command 'add-br1'; use --help for help
l1@linux1:/usr/bin$ sudo docker exec ovs1 ovs-vstcl add-br ovs-br1
ovs-vstcl: unix:/var/run/openvswitch/db.sock: database connection failed (No such file or directory)
l1@linux1:/usr/bin$ sudo docker exec ovs1 ovs-vstcl add-br ovs-br1

```

6.2. ábra. Ping az OVS3-ból kifelé

```

l1@linux1:/usr/bin$ sudo docker exec ovs1 ovs-vstcl show
96bcafea-96b7-4bf9-b667-5b67d421fe97
Bridge ovs-br1
  Port eth6
    Interface eth6
  Port eth1
    Interface eth1
  Port eth4
    Interface eth4
  Port eth2
    Interface eth2
  Port eth5
    Interface eth5
  Port eth3
    Interface eth3
  Port ovs-br1
    Interface ovs-br1
      type: internal
  ovs_version: "2.13.0"
l1@linux1:/usr/bin$ _

```

6.3. ábra. OVS1 interfészei



Ellenőrizni kellett a routerek által felépített beállításokat és kapcsolatokat is. Itt a különböző **show** parancsok segítettek a konfiguráció ellenőrzésében, a ping, illetve traceroute pedig a kapcsolatok vizsgálatában.

Magasabb szinten a csomagok vizsgálatát tcpdump és Wireshark programokkal lehet. Ezek elkapják a Linux interfészein keresztül haladó csomagokat és akár ki is tudják bontani azokat, hogy részletesebben megvizsgáljuk. Különböző toolokkal akár a hálózat sebességét is ellenőrizhetjük.

## 7. fejezet

# Összegzés

A SND és hálózatok témája a Diplomamunka előtt nekem ismeretlen fogalmak voltak. Ezáltal nehéz volt elindulni a megvalósításban. Nem volt tapasztalatom sem Linux-ok terén, sem hálózatkonfigurálásban sem. A feladat során tanultam meg a python programozási nyelvet, a virtualizációs technológiákat. Nagyon sok alap tudást vehettem a diplomamunka feladat írása közben.

A munka megalkotása során figyeltem arra, hogy az iparban jelenleg is használható megoldásokat válasszak és később használható tudást szerezzek azon keresztül. Nagyon érdekes volt egy containeres rendszert felépíteni és azt vizsgálni. A Linuxos ismeretek is hasznos tudást nyújtanak a későbbiekben. Érdekes volt még a több szintű virtualizáció próbálgatása és a megoldás keresése. Amikor elkészült a környezet elmélyedhettem az SDN világába és elég mélyen megismerhettem az OpenFlow működését. A protokoll megismerésével szintén egy hasznos gondolkodásmód tárult elém.

Nem rögtön kaptam meg a fejlesztő környezetet, így nem azon kezdtem a fejlesztéseket, hanem saját, Windows alapú VM-eken. Itt több lehetőségem volt és könnyebb volt alkalmazásokat megvalósítani, mint a céleszközön, ezért tanulásra kifejezetten hasznos volt, ezt később tesztelési célból továbbra is tudtam használni, mivel az ESXi korlátozott hozzáférést adott. Mivel nehézkes volt a fájlcsere is, megismerkedhettem a Git verziókövető alkalmazással is, ami szintén egy gyakran használt tool az iparban. Mivel erőforrás is korlátozottan állt rendelkezésre, így figyelnem kellett, hogy ne pazaroljam ezeket. Ez szintén egy hasznos készség lehet később.

Problémaként felmerült – ami a jelenlegi állapotban is akadályozza a munkát – ,hogy az ESXi belső switch-e valamiért letiltotta a VM-eket a külső internetről. Ezáltal nem tudtam letölteni csomagokat, a programomat sem tudom felmásolni a szerverre, így kipróbálni sem volt tőlem független okok miatt.

Az esetlegesen felmerülő problémák miatt sokszor saját környezetben próbálkoztam felépíteni a környezetet, ami ugyan úgy működik, mint ESXi hypervisor-on. Bár segítség volt a megírt bash script, de nem tudtam CISCO IOS image-eket szerezni és a script paramétereit is át kellett írni úgy, hogy teljes értékűen működjön.

Maga a téma egy érdekes problémára ad megoldást. Nem teljeskörű és mindenre kiterjedő válaszokat és megoldásokat nyújt, inkább csak egy gondolkodást indít, hogy miképpen lehet egy hasonló problémakört megoldani.

### 7.1. Tapasztalatok

Mivel kezdő voltam az SDN, virtualizáció, telekommunikáció, Linux és a dolgozatban felvázolt témákban rengeteg új dolgot volt lehetőségem tanulni és tapasztalni. Bár a Hibrid SDN funkcionalitását nem tudtam ellenőrizni, de a különálló SDN és legacy csomópontok

közötti kommunikáció felépítésére és vizsgálására volt lehetőségem. A Docker egy speciális alkalmazásával is tanulhattam az iparban is érdekes megoldásokat. A hibrid SDN paradigma is érdekes gondolkodásmódot adott és jó kiindulási lehetőség, hogy hogyan álljak neki egy új technológiai probléma megoldásának.

## 7.2. Fejlesztési lehetőségek

Legfontosabb továbblépés maga a környezet tényleges felélesztése és a problémák teljeskörű megoldása lehet. Mivel mindig banális dolgokon csúszott a projekt, ezért a megoldásuk sem lehetetlen, csak körülményes és időigényes a problémák megtalálása.

Továbbá nem teljeskörű OSPF lett implementálva a Ryu Northbound interfészére. Lehetőség lehetne arra, hogy akár egy SDN node legyen master egység, be lehet vezetni multiarea hálózatokat, DR és BDR network adjacency megoldásokat és egyéb OSPF protokollokat.

Tovább lehet fejleszteni még a hálózati topológia kezelő modult, hogy időnként újraellenőrizze a hálózatot, timert lehet állítani az egyes tagokra, jelenleg nincs megvalósítva egy node törlése, a loopok sincsenek még kiküszöbölve a hálózatban.

# Köszönetnyilvánítás

Köszönöm a konzulensem, Dr. Zsóka Zoltán segítségét, akinek a szakmai felkészültsége és támogatása nélkül ez a dokumentum nem jöhetett volna létre. Különösen köszönöm, hogy akkor is foglalkozott a témával, amikor nem volt kötelessége, munkaidőn kívül vagy passzív félévemben. Köszönöm a családom támogatását, akik segítették az egyetemi pályafutásomat. Családomból külön köszönöm Szabó Gergely segítségét, aki szintén villamosmérnökként szakmai ötleteket és tanácsokat tudott adni. Köszönöm Vadas Evelinnek a diplomamunka során nyújtott támogatását, valamint Hári Veronikának, Bartal Grétinek és Kanyó Zitának. Szakmai segítséget nyújtott még Dulfiqar A Alwahab, Csernok Gábor, illetve Sipos Péter.

# Irodalomjegyzék

- [1] Megyeri Csaba. Hibrid sdn hálózati megoldások vizsgálata, 2017.
- [2] Yingya Guo, Zhiliang Wang, Xia Yin, Xingang Shi, and Jianping Wu. Traffic engineering in sdn/ospf hybrid network. *2014 IEEE 22nd International Conference on Network Protocols*, pages 563–568, 2014.
- [3] Wendell Odom with Cisco Press. *CCNA 200-301 Official Cert Guide*. 2020.
- [4] J. Moy from Network Working Group with. Rfc2328: Ospf version 2. <https://tools.ietf.org/html/rfc2328>, April 1998.
- [5] Open Network Fondation. Software-defined networking, 2014.
- [6] Dr. Nick Feamster. Software defined networking by the university of chicago. <https://www.coursera.org/learn/sdn/home/welcome>, 2014.
- [7] Nick Feamster, Jennifer Rexford, and Ellen Zegura. *The Road to SDN*. 2013.
- [8] Peterson, Cascone, O'Connor, and Vachuska. Software-defined networks: A systems approach. <https://sdn.systemsapproach.org/>, 2020.
- [9] Thomas D. Nadeau and Ken Gray. *SDN: Software Defined Networks Chapter 4. SDN Controllers*. 2013.
- [10] MurphyMc. Welcome to the nox-classic wiki. <https://github.com/noxrepo/nox-classic/wiki>, 2012.
- [11] McCauley. Pox documentation. <https://noxrepo.github.io/pox-doc/html/>, 2015.
- [12] ryu developement team. ryu documentation, Sep 24, 2020.
- [13] Atlassian. Project floodlight: Floodlight controller documentation. <https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview>, 2016.
- [14] OPenDaylight Project. Opendaylight project documentation. <https://docs.opendaylight.org/en/latest/index.html>, 2016-2018.
- [15] ONF employees. Onos documentation. <https://wiki.onosproject.org/display/ONOS/ONOS+Documentation>, 2020.
- [16] Open Networking Foundation. Openflow switch specification version 1.0-1.5, 2011-2015.
- [17] Open Networking Foundation. *ONF SDN Evolution*. 201.

- [18] János Farkas, Stephen Haddock, and Panagiotis Saltsidis. Software defined networking supported by ieee 802.1q. 2014.
- [19] Stefano Salsano, Pier Luigi Ventre, Francesco Lombardo, Giuseppe Siracusano, Matteo Gerola, Elio Salvadori, Michele Santuari, Mauro Campanella, and Luca Prete. Hybrid ip/sdn networking: open implementation and experiment management tools. 2014.
- [20] VMware. Vmware vsphere documentation. <https://docs.vmware.com/en/VMware-vSphere/index.html>, 2020.
- [21] Esxi magyar leírás. [http://servira.com/fogalomtar/200.vmware\\_esx\\_\\_\\_esxi](http://servira.com/fogalomtar/200.vmware_esx___esxi), 2020.
- [22] GNU Project. Gnu. <https://www.gnu.org/>, 2020.
- [23] Canonical. Ubuntu server guide. <https://ubuntu.com/server/docs>, 2020.
- [24] Oracle Corporation. Oracle vm virtualbox user misc, 2004-2020.
- [25] RedHat. Kvm documentation. <https://www.linux-kvm.org/page/Documents>, year.
- [26] Docker Inc. Docker documentation. <https://docs.docker.com/>, 2013-2020.
- [27] Wikipedia. Promiscuous mode. [https://en.wikipedia.org/wiki/Promiscuous\\_mode](https://en.wikipedia.org/wiki/Promiscuous_mode), 2020.
- [28] Mininet documentation. <https://github.com/mininet/mininet/wiki/Documentation>, 2019.
- [29] Linux Foundation Collaborative Projects. Open vswitch documentation. <https://docs.openvswitch.org/en/latest/>, 2016.
- [30] Openvswitch vs openflow: What are they, what's their relationship? <http://www.fiber-optic-transceiver-module.com/openvswitch-vs-openflow-what-are-they-whats-their-relationship.html>, 2018.
- [31] Linux Foundation. Open vswitch misc ovs-vsctl. <http://www.openvswitch.org/support/dist-docs/ovs-vsctl.8.txt>, year.
- [32] RYU project team. *RYU SDN Framework*. 2014.
- [33] Sridhar Rao. Sdn series part four: Ryu, a rich-featured open source sdn controller supported by ntt labs. <https://thenewstack.io/sdn-series-part-iv-ryu-a-rich-featured-open-source-sdn-controller-supported-by-ntt> 23 Dec 2014 1:13pm.
- [34] Sriram Natarajan with RYU Code Contributor. Ryu controller tutorial. <http://sdnhub.org/tutorials/ryu/>, 2014.
- [35] Castro Flaviojr. Shortest path forwarding with openflow on ryu. <http://sdn-lab.com/2014/12/25/shortest-path-forwarding-with-openflow-on-ryu/>, December 25, 2014.
- [36] Marcel Caria, Tamal Das, Admela Jukan, and Marco Hoffmann. Divide and conquer: Partitioning ospf networks with sdn. 21 October 2014.

- [37] Dr Andrew W. Moore from University of Cambridge. Pee-wee ospf protocol details. <https://www.cl.cam.ac.uk/teaching/1011/P33/documentation/pwospf/index.html>, November 2010.
- [38] Wireshark user's guide. [https://www.wireshark.org/docs/wsug\\_html\\_chunked/](https://www.wireshark.org/docs/wsug_html_chunked/), 2020.
- [39] tcpdump and libcap documentation. <https://www.tcpdump.org/index.html#documentation>, 2020.
- [40] Aaron Phillips. How to run a remote packet capture with wireshark and tcpdump. <https://www.comparitech.com/net-admin/tcpdump-capture-wireshark/>, 2020.
- [41] Michael Kerrisk. ip-netns(8) — linux misc page. <https://man7.org/linux/man-pages/man8/ip-netns.8.html>, 2020.
- [42] Aric Hagberg, Dan Schult, and Pieter Swart. Networkx reference, Aug 22, 2020.