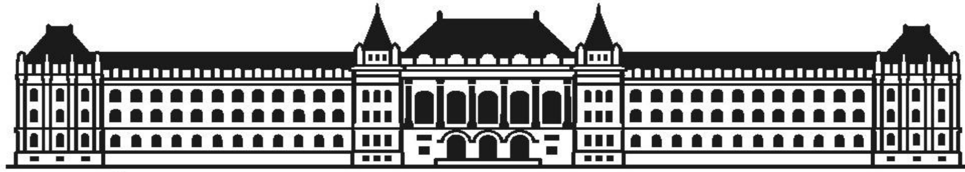


Budapesti Műszaki és Gazdaságtudományi Egyetem

Távközlési és Médiainformatikai Tanszék

TDK Dolgozat

Hálózati szolgáltatások automatikus konfigurálása SDN technológiák alkalmazásával

Készítette: Németh Balázs, mérnök-informatikus, BSc., 4. évfolyam

Konzulensek: Dr. Sonkoly Balázs, Budapesti Műszaki és Gazdaságtudományi Egyetem, Távközlési és Médiainformatikai Tanszék

Dr. Kern András, Ericsson Magyarország Kft.

Budapest, 2013. október 24.

Tartalomjegyzék

1 Bevezetés 5

2 Software Defined Networking 8

2.1 OpenFlow: terjedő szabvány 8
2.2 Deklaratív szemlélet 12
2.2.1 Az OCaml nyelv 12
2.3 Frenetic bevezető 14
2.4 OpenFlow nehézségek elemzése 15
2.4.1 Modulok közötti interakció 16
2.4.2 Alacsony szintű interfész 18
2.4.3 Két szintű program futás 19

3 Frenetic alkalmazási lehetőségei 20

3.1 Szükséges könyvtárak 21
3.2 Tesztkörnyezet: Mininet 22
3.3 FreneticDSL bemutatása 23
3.3.1 Forgalom továbbító program írása 25
3.3.2 Tűzfal funkció készítése 28
3.3.3 Forgalom figyelés implementálása 31

4 Frenetic fejlesztői elemzése 33

4.1 Modulok felépítése 34
4.2 Egy összetett példa elemzése 36

5 Kiterjesztések a Frenetic rendszerhez 40

5.1 Folyamatok közötti kommunikáció 40
5.2 Router modul 41
5.2.1 Útvonalválasztó tábla 42
5.2.2 MAC címek kezelése 44
5.2.3 A router működése 46
5.2.4 Konfiguráció frissítése 48
5.3 ARP modul 50
5.4 DHCP szerver modul 51

6 Tesztelés 54

6.1 Fordítási és konfigurálási idő 56
6.2 Javított stream tisztítási folyamat. 58
6.3 Hálózat késleltetése újrakonfigurálás közben 60
6.4 A javított stream tisztítás hatékonysága 61
6.5 Együttműködés a TCP protokollal 64

7 Összefoglalás 68

8 Hivatkozások 69

Absztrakt

Napjainkban az SDN (Software Defined Networking) megoldások rugalmasságuknak és költséghatékonyságuknak köszönhetően rohamosan terjednek. A koncepció lényege, hogy segítségével szétválaszthatjuk az adat és vezérlési síkot a hálózati eszközökben, így a hálózatot egy központi, tetszőleges logikát megvalósító kontrollerből irányíthatjuk. A kontroller és a hálózati kapcsolók közötti kommunikációt szolgálja az OpenFlow protokoll, mely segítségével az eszközök működése irányítható és felülírható. Az OpenFlow kontroller platformok számos különböző implementációja van jelen (NOX, Beacon, POX stb.), melyek révén új hálózati alkalmazásunk programozási nyelve szinte szabadon megválasztható (C/C++, Java, Python stb.).

Projektünk célja egy olyan SDN architektúra létrehozása, melyben lehetőségünk van leírni különböző típusú szolgáltatásokat, funkciókat és akár ezek közötti relációkat. Mindezt a hálózatot felépítő eszközök és azok szoftvereinek ismerete nélkül, sőt akár a hálózat szerkezete felőli részletes tájékozottság nélkülözésével. Vagyis a jelenleginél eggyel magasabb absztrakciós szinten szeretnénk irányelveket megfogalmazni a hálózatunkra vonatkoztatva, melyet a rendszerünk értelmez és lefordít alacsony szintű szabályokra.

Ilyen magas szintű szolgáltatás leírásnak a beviteléhez a jelenlegi SDN megoldások, illetve az OpenFlow még nem nyújt lehetőséget, mivel a hardveres, valamint szoftveres eszközök túl mélyreható ismeretét és programozását követeli meg a felhasználóktól. Egyelőre SDN szolgáltatások implementálásakor foglalkoznunk kell a konkurens modulok egymásra hatásával, csomagfejlécekre való illeszkedési szabályok meghatározásával, melyek átlapolódása esetén döntenünk kell a bejegyzések prioritásáról, továbbá az elhelyezett csomagtovábbító bejegyzések működést befolyásoló információt rejthetnek el a központi kontroller elől.

Ezen problémák feloldására jelentek meg kutatás alatt lévő megoldások a közelmúltban, melyek felhasználásával, illetve továbbfejlesztésével érhetnénk el célkitűzésünket. A Princeton Egyetemen fejlesztett, jelenleg még kezdeti fázisban lévő, nyílt forráskódú Frenetic kontroller platform teszi lehetővé OpenFlow hálózatok deklaratív programozását egy külön erre a célra kialakított programozási nyelv segítségével. A Frenetic nyelvet alkalmazva hálózatok programozására, néhány fent említett probléma megoldódik.

Jelen dolgozat célja ennek a szoftvernek a kibővítése a projektet támogató modulokkal, melyek segítségével megvizsgálhatjuk a Freneticet előnyök, hátrányok, hiányosságok, skálázhatóság és teljesítmény szempontjából. További cél, hogy javaslatot tegyünk a szoftver lehetőségeinek kibővítésére a dinamikusság és futás közbeni újrakonfigurálás területén. Ehhez a kontroller állapotának egy frissítési algoritmusát valósítjuk meg, és értékeljük ki szimulált hálózati környezetben.

1 Bevezetés

Manapság a hálózati szolgáltatásokkal szemben temérdek elvárásunk van, mivel szerteágazó területeken kerül központi szerepkörbe számítógépek közötti kommunikáció. Itt gondolhatunk a rohamosan fejlődő mobilpiacra, ahol lassan minden eszköz, a nap 24 órájában kíván közel folyamatos internetkapcsolatot fenntartani. Vagy gondolhatunk a lakosság által alapszolgáltatásként igényelt otthoni internetkapcsolatra, ahol az előfizetők a magas rendelkezésre állást, állandó, magas adatátviteli sebességet követelnek meg internetszolgáltatóiktól. De gondolhatunk egy kisebb helyi hálózat által elvárt flexibilis és redundáns hálózati infrastruktúrára, ahol a sűrűn igényelt változtatások mellett is elvárt az alacsony költségű, könnyen kivitelezhető üzemeltetés. Mindezen kritériumok céges, illetve ipari környezetben még magasabb teljesítményjelzőkkel jelennek meg.

A hálózati szolgáltatások statikusságának feloldására ma is vannak megoldások, melyek részben, egységesítés nélkül, szolgáltatás specifikusan oldanak meg bizonyos problémákat a hálózat szintű konfigurálhatósággal. Vegyünk például egy OSI referenciamodell szerinti, második rétegbeli hálózatban működtetett VLAN szolgáltatást. A VLAN konfiguráció egy hálózaton belül kapcsolónként egyedi. Ahhoz, hogy egy hálózati adminisztrátornak minél kevesebbet kelljen a kapcsolókon egyenként felkonfigurálnia, a Cisco létrehozta a VTP (VLAN Trunking Protocol) szolgáltatást. Melynek segítségével egy VTP szerveren megadhatjuk a hálózat VLAN konfigurációjának azon részét, amely az egész hálózaton azonos. Azonban a hálózat üzemeltetőjének ekkor még mindegyik kapcsolóval külön kell foglalkoznia[1], hogy a VLAN szolgáltatás megfelelően működjön. Hasonlóan hozhatunk példát a referenciamodell szerinti harmadik rétegből, ahol az útvonalválasztó protokollok elosztott működésükből következően néha kénytelenek az optimálisnál rosszabb csomagtovábbítási döntéseket hozni.

Az újrakonfigurálás a kezdeti konfiguráció megadásához hasonló komplexitású, így az üzembe helyezésre adott, egyszerűsítő megoldásokhoz hasonlóan, az újrakonfigurálást segítő eszközök sem teljesek és nem egységesek.

Az SDN új szemlélete ezeket a problémákat kívánja feloldani úgy, hogy egy új réteget vezet be az eddigi hálózati architektúra fölé. Az új réteg segítségével elfedhetjük a hardverek

[1] A kapcsoló portjainak VLAN-hoz rendelését mindenképpen egyedien kell elvégezni minden kapcsolóra.

és szolgáltatások különböző interfészeit, valamint kiváló rugalmasságot biztosíthatunk a hálózatnak.

Ma még csak az adatközpontokban nevezhetjük éles környezetben használtnak az SDN által meghatározott elveket és módszereket. Egy adatközpontban kulcsfontosságú az erőforrások lehető legjobb kihasználtsága, melyet dinamikusan felépített, logikailag különálló, de azonos fizikai topológián futó hálózatok létrehozásával valósítanak meg. Ezzel a módszerrel, SDN technológiák segítségével 20-30%-kal jobb erőforrás kihasználás érhető el az SDNCentral, internetes szakmai blog szerint [8].

Az SDN széleskörű elterjedésére számíthatunk a rengeteg előttünk álló lehetőség miatt. Internetszolgáltatóknál (ISPs) például megvalósítható lenne egy olyan lehetőség az előfizetők felé, hogy szükségleteik szerint dinamikusan tudják változtatni maguknak a rendelkezésükre álló sávszélességet. Amely egy költséghatékony megoldás a felhasználónak, és erőforrás takarékos az operátornak.

Az ISP-khez hasonló követelmények támasztódnak a telefonhálózat szolgáltatókkal szemben is. Szóval egy nem túl merész jövendöléssel mondhatjuk, hogy hamarosan a teljes telekommunikációs ipari szektorban meghatározó szerepe lesz az SDN technológiáknak. Ezt alátámasztandó, szintén az SDNCentralra hivatkozhatunk, ahol több telekommunikációs szférában érdekelt multinacionális cég jelenít meg cikkeket és vesz részt témába vágó konferenciákon.

Az SDN megoldásokkal kecsegtet a nehezen kezelhető hálózati szolgáltatások konfigurálásából fakadó nehézségekre. Azonban a mai állapotban ez még nincs kész a megfelelő szinten, mert a hálózati működés szoftveres felüldefiniálásához meglehetősen részletekbe menően kell programoznunk. A mai eszközöket alkalmazva, egy hálózati funkció készítőjének a szolgáltatás működése szempontjából rengeteg irreleváns dologgal kell foglalkoznia. Ezek elfedéséhez, és a programozás egy magasabb absztrakciós szintre emeléséhez szükségünk lenne egy olyan rendszerre, mely segítségével egyszerűen írhatunk le szolgáltatásokat, és automatikusan konfigurálhatjuk fel azokat.

Egy ilyen rendszer segítségével az SDN által bevezetett szoftveres réteget magasabb absztrakciós szintre emelhetnénk, melyben szolgáltatások szintjén programozhatnánk a hálózatot, és egy-egy szolgáltatás, mint elemi építőkocka jelenhetne meg. Ezzel egységesítve a szolgáltatások konfigurálását segítő környezetet, ahol egész szolgáltatás könyvtárakról beszélhetnénk, melyek elemeit felhasználva, tetszőlegesen definiálhatnánk relációkat a

különböző hálózati funkciók között.(service chaining [2]). Egy szoftverfejlesztési párhuzammal élve, mondhatnánk azt, hogy egy szolgáltatás analóg lehetne az objektum orientált paradigma egy osztályával. Szolgáltatás osztályokat példányosíthatnánk, beállíthatnánk egyes paramétereit, hívhatnánk rajta értelmezett műveleteket, hierarchiába szervezhetnénk más objektumokkal és definiálhatnánk több szolgáltatásobjektum együttműködésének módját.

Egy ilyen magas absztrakciós szintű architektúra elegáns megoldása lenne a mai, és remélhetőleg közeljövőbeli, modern hálózati szolgáltatások által minden felállított követelmény teljesítésére.

A következő fejezetben röviden ismertetem az SDN koncepcióit, illetve egy alkalmazását az OpenFlow szabvány keretein belül. Majd rátérek a deklaratív szemléletű programozás általános bemutatására, melyhez a dolgozatban használt nyelvet, az OCaml-t hozom jelen esetben fontos példaként. Utána a Frenetic rendszer alkalmazhatóságát mutatom be és elemzem hálózat adminisztrátori és programozói szemmel. A dolgozat csúcspontjaként részletesen bemutatom a Frenetic rendszerhez általam implementált kiterjesztéseket. Végül a saját moduljaim és a Frenetic rendszer teljesítménybeli elemzéséhez végzek el néhány tesztet, melyek kiértékelése lesz a dolgozat zárszava.

2 Software Defined Networking

Az SDN (Software Defined Networking) vagy szoftveres hálózat definiálás egy újfajta megközelítése minden hálózatokkal kapcsolatos tevékenységnek. A hálózati infrastruktúra központi szoftverből való irányítása révén csökkenthetőek az összeállítási és üzemeltetési költségek, valamint támogatja a hálózatok testreszabhatóságát, és optimalizálási lehetőségeket biztosít.

Az új nézet fő gondolata, egy már más területekről korábban is ismert elv újrafelfedezése hálózati perspektívából: adat és vezérlési sík szeparálása az eszközökben. A hálózat irányítását központosítva, az adminisztrátoroknak nem kell közvetlenül minden eszköz parancssori interfészén elvégeznie a konfigurációs tevékenységeket, ehelyett a teljes hálózat logikáját megvalósító, úgynevezett kontroller beállítása elég a hálózat üzemeltetéséhez.

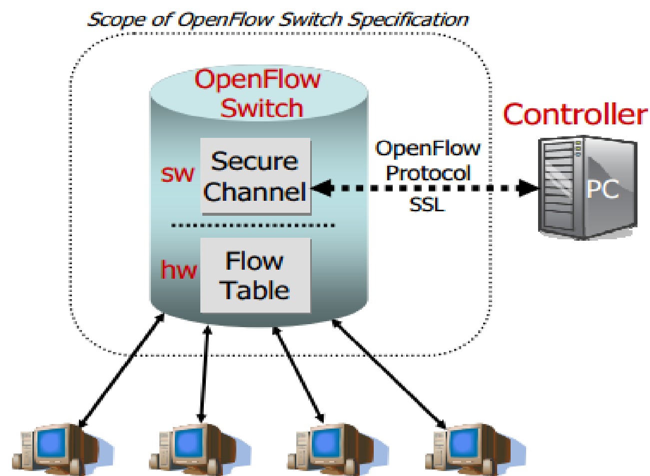
Az SDN szemlélete ma az OpenFlow szabvány révén terjed ipari és akadémiai környezetben is. Ezért ebben a fejezetben részletesebben belemegyünk ezen kommunikációs protokoll ismertetésébe, mely alapjául szolgál a legtöbb SDN megoldásnak.

2.1 OpenFlow: terjedő szabvány

Az OpenFlow nyílt szabvány, a hozzá szükséges, minimális infrastruktúra többlet segítségével lehetőséget nyújt a hálózatok működésének felülírására, módosítására. Erre akadémiai környezetben azért lehet nagy szükség, mert segítségével könnyen tudjuk saját ötleteinket, hálózati algoritmusainkat kipróbálni akár nagyméretű hálózatokon is. A rendszert a Stanford egyetemen fejlesztették ki, gyorsan bővül az irodalma, lelkes fejlesztői közönségre találva gyorsan terjed szerteágazó területeken. Ez a néhány éves múltra visszatekintő protokoll az iparban is teret nyer, hálózati eszköz gyártó cégek, termékeikben támogatást nyújtanak az OpenFlow-val való együttműködésre. Segítségével például lehetőségünk van akár a maitól fundamentálisan eltérő architektúrák kialakítására is. Sokféle alkalmazási területre ad példát az OpenFlow whitepaper [2], melyben a rendszer kialakításának további előnyeit is olvashatjuk, és megismerhetjük az alapokat.

A hálózati működés felülírásának alapelve, hogy a hálózati eszközök egy funkcióban felettük álló, központi kontrollertől kapják a működésükhöz szükséges információt. Így a logika nagy része a kontrollerben kerül megvalósításra, a hálózati eszközökkel szemben kisebb

igényt támasztunk a konvencionális hálózatokhoz képest. Az OpenFlow-képes hálózati eszközök elterjedt elnevezése a switch, holott ez nem feltétlenül egy adatkapcsolati rétegben működő hardver, mert a kontrollerből tetszőleges funkcionalitású eszközként üzemeltethetjük.



1. ábra: Az OpenFlow whitepaperből származó ábra, mely ábrázolja az OpenFlow switch, kontroller és hostok kapcsolatát.

Az 1. ábrán a whitepaperből származó ábrát láthatjuk, mely a fent leírtakat szemlélteti. A switchekben flow táblák helyezkednek el, melyek meghatározzák a switch működését. Minden switchet egy OpenFlow csatorna csatlakoztat a kontrollerhez, melyen az OpenFlow protokollnak megfelelő üzenetváltások történnek.

Mivel a Frenetic rendszer csak az OpenFlow protokoll 1.0.0-ás verziójával foglalkozik, így jelen dolgozatban csak erre térünk ki. A protokollnak egyébként megjelent több újabb verziója is melyekben, már támogatnak switchenként több flow táblát is, melyek bonyolultabb továbbítási szabályrendszerek bevitelét teszik lehetővé. Későbbi verziókban jelent meg IPv6 és QoS támogatás is, de ezek ismeretére dolgozatom során nem volt szükség, így csak az OpenFlow 1.0.0-ás verzióban szereplő alapelveket mutatom be röviden a továbbiakban.

Egy switch flow táblája flow bejegyzésekből épül fel. Itt érdemes megjegyezni, hogy a flow bejegyzést nem szabad összekeverni az SDN terminológián kívül használt flow fogalmával. Ott ugyanis általában két IP címmel rendelkező hálózati elem közötti adatforgalom egy részhalmazaként szokták definiálni kontextusonként eltérő szűkítésekkel. OpenFlow-ban flow bejegyzés alatt azt a szabályt értjük, amely valamilyen illeszkedési struktúrával határozza

meg, hogy mely csomagokra fogja alkalmazni a flow bejegyzés utasítás részét. Tehát összefoglalva és pontosítva egy flow táblát alkotó flow bejegyzések három részből épülnek fel:

- Match Fields: tartalmazza a portot, amelyen a csomag beérkezett, és információkat a csomag fejléceiből. Mindegyik mező lehet egy meghatározott érték, vagy tetszőleges, ha az adott mezőt nem szeretnénk használni a match-elés során. A használható illeszkedési struktúra mezőit a 2. ábrán láthatjuk.
- Counters: több fajta számláló definiált, melyek akkor növekednek, ha a flow-ra illeszkedő csomag került feldolgozásra (pl. bejövő csomagok száma). Statisztikák készítésénél vagy monitorozási feladatok elvégzésnél nagy segítségére lehet a protokoll felhasználóinak.
- Instructions: a végrehajtandó akciók listája, a flow bejegyzésre való illeszkedés esetén. Kérhetjük expliciten egy csomag továbbítását, flood-olását vagy utasíthatjuk a switchet, hogy a kontrollerhez továbbítsa a csomagot, amennyiben szükséges a kontrollerben implementált logika a továbbítási döntés meghozásához. Csomag eldobására nincs külön üzenet, ezt üres akció lista elhelyezésével érhetjük el.

Ha egy switch egyik flow bejegyzésével sem talált illeszkedést, akkor a csomagot felküldi a kontrollernek, így a kontroller oldali logika dönthet a csomag sorsáról.

Ingress Port	Metadata	Ether src	Ether dst	Ether type	VLAN id	VLAN priority	MPLS label	MPLS traffic class	IPv4 src	IPv4 dst	IPv4 proto / ARP opcode	IPv4 ToS bits	TCP/ UDP / SCTP src port ICMP Type	TCP/ UDP / SCTP dst port ICMP Code

2. ábra: Csomag mezők, amelyeket a flow match field-jében használhatunk. (forrás: OpenFlow specifikáció 1.0.0)

A flow táblába elhelyezkedő match struktúrák között előfordulhat átlapolódás. Átlapolódás alatt azt értjük, hogy létezik olyan csomag, amely több flow bejegyzésre is illeszkedik. Ekkor az OpenFlow specifikáció szerint, azonos bejegyzés prioritás esetén, a switch szabadon dönthet, hogy melyik bejegyzés alapján dolgozza fel a bejövő forgalmat. A működés többértelműségének elkerülése érdekében minden flow-hoz definiálnunk kell egy prioritás értéket is. Vagyis átlapolódás esetén a magasabb prioritású bejegyzéshez tartozó utasítás halmaz fog végrehatódni.

A kontroller és a switchek közti csatornákon keresztül értesül a kontroller egy eszköz csatlakozásáról, és további eseményeiről. Továbbá ezen a csatornán keresztül kérdezhetjük le a switchek állapotát, és flow bejegyzéseket helyezhetünk el a tábláikban. Szóval tulajdonképpen az OpenFlow a switchek és a kontroller közötti kommunikációs protokoll. Az üzeneteknek három típusuk van:

- Controller-to-Switch: kontroller által kezdeményezett üzenetek, melyekre az üzenettípustól függően a switchnek válaszolnia kell. Ilyen lehet például egy Read-State üzenet, mellyel statisztikákat kérhetünk a switchtől, vagy Modify-State üzenet, mellyel flow bejegyzéseket helyezhetünk el vagy módosíthatunk.
- Asynchronous: a switchtől érkező üzenetek típusa. Ha a switch nem talált illeszkedést egyik bejegyzéssel sem, akkor egy Packet-in üzenetben küldi fel a kontrollernek a csomagot. Ha a flow-knak lejár a beállított érvényességük (hard timeout) vagy ha egy ideig nem illeszkedett rájuk csomag (idle timeout), törlődnek a flow táblából. Ekkor egy Flow-Removed üzenet érkezik a kontrollerhez. Asynchronous típusú üzenetben érkezik továbbá, a kontroller által lekért statisztika.
- Symmetric: kérés nélkül mehetnek mindkét irányban, például a kontroller – switch kapcsolat létrejöttekor egy Hello üzenet formájában.

Összességében egy OpenFlow-képes switchnek táblabejegyzésekben kell tudnia keresni, valamint csomagokat kell tudnia továbbítania kijelölt forgalmi portokon, vagy a kontroller felé. Egy elterjedt, nyílt forráskódú implementáció az Open vSwitch.

A kontroller egy általános célú PC-n futtatva alkalmas virtuális, vagy akár valós OpenFlow hálózat vezérlésére egy TCP socketen keresztül csatlakoztatva a switchekhez. A kontrollereknek többféle implementációja terjedt el, mint például a NOX, melyet C/C++ nyelven tudunk programozni, vagy a Floodlight, melyet Java nyelven írtak és fejlesztenek, továbbá a POX kontroller, melyre Pythonban tudunk programokat írni. Ezek működését nem részletezem, mert munkám során egy speciális kontroller implementációt használtam, melynek részleteire az alábbiakban térek ki. Példát egy OpenFlow-t alkalmazó programra a Frenetic ismertetésénél találhatunk.

2.2 Deklaratív szemlélet

Hálózati programok tervezésekor az OpenFlow kontrollerek által biztosított procedurális szemléletű programozás nem igazán kézen fekvő, ha nem szeretnénk részletekbe menően foglalkozni az üzenetváltási folyamatokkal. Egy hálózati szolgáltatás leírását kényelmetlen és hosszadalmas utasítások sorozataként definiálni, ugyanis ilyenkor sokkal inkább szeretnénk foglalkozni azzal, hogy mi legyen az üzemállapota a hálózatnak, és nem abban a folyamatban gondolkodni, hogy hogyan juttathatunk el a kívánt állapotba.

A deklaratív nyelvek pontosan ebben szeretnék segíteni a programozók munkáját a szoftverfejlesztésben már évtizedek óta. A deklaratív programozási paradigma lehetővé teszi, hogy kijelentő módban gondolkozzunk egy program működésének leírásakor. A deklaratív szemléletben majdhogynem jelmondatnak nevezhetnénk, a „WHAT rather than HOW” kifejezést, amely jól megfogalmazza a szem előtt tartandó gondolkodásmódot.

A szemlélet átültetésére a hálózati programozás világába napjainkban van tervezési és kísérletezési stádiumban. Ha ezen az úton sikerülne a bevezetőben vízionált, szolgáltatásokkal építőelemként operáló, magas absztrakciós szintű rendszert létrehozni, akkor folytatva a már említett szoftverfejlesztési analógiát: az OpenFlow kontrollereken írt programok lehetnének a hálózatprogramozás Assembly programjai, míg a jelen dolgozatban vizsgált Frenetic rendszer lehetne a „hálózati C fordítónk”, továbbá a szolgáltatás szintű program pedig a C# vagy Java nyelvű szoftver[2].

Mivel a Frenetic rendszer OCaml nyelven készült, ezért ennek a nyelvnek egy minimális ismertetése szükséges a későbbiekhez, így erre ebben a fejezetben fog sor kerülni.

2.2.1 Az OCaml nyelv

Az ML programozási nyelvcsalád általános célú funkcionális programozási nyelveket foglal magába. Az 1970-es években fejlesztették ki az edinburgh-i egyetemen Robin Milner vezetésével. Az ML rövidítés eredetileg *meta-language*-ből (azaz köztes nyelv) jön, mivel kiváló lehetőséget biztosít szintakszis és fordítási funkciók definiálására [4]. A nyelvek

[2] Az analógia felső elemének lehet, hogy jobban illene a deklaratív szemléletű, logikai programozási nyelv, a Prolog, mely könyvtárainak egy jelentős részét C nyelven írták.

kategorizálásához ragaszkodva nevezhetnénk nem tiszta funkcionális programozási nyelvnek, mivel megengedi, hogy a függvényeknek legyenek mellékhatásaik (*side-effect*), ellenben a tisztán funkcionális nyelvekkel, mint például a Haskell.

A nyelvcsalád további jellegzetességeihez tartozik a *garbage collection* (foglalt memóriaterületek automatikus felszabadítása), statikus típusosság, típus következtetés (*type inference*), függvény polimorfizmus, minta illesztés és kivétel kezelés. A nyelvcsalád két fő alcsoportja a Standard ML (SML) és a Caml (eredetileg Categorical Abstract Machine Language), melynek a dolgozat szempontjából fontos, jelenlegi fő implementációja az OCaml (Objective Caml).

Az OCaml nyelv [5] legfontosabb jellemzője az objektumok használata a nyelvcsalád többi tagjával ellentétben. Így egy meglehetősen rugalmas nyelv tárul elénk, mely támogat imperatív, funkcionális és objektum orientált nyelvekben megszokott lehetőségeket. A változók típusait nem kell definiálnunk, ezt az OCaml fordító az inicializáláskor kikövetkezteti a változónak adott értékből (*type inference*), de ennek ellenére szigorúan típusos nyelv. Azaz automatikus típuskonverziót semmikor sem végez, vagyis mindenhol expliciten konvertáló függvényeket kell hívnunk, ha típusok között szeretnénk adatot átadni. A fentiek miatt talán egy kicsit nehezen tanulható a nyelv, mert eleinte sok probléma adódhat abból, hogy a kód alapján nem látjuk a fordító által kikövetkeztetett típust. De mivel a típus eltérésből adódó hibák fordítási időben kiderülnek, sok futási idejű hiba keresésére szükséges időtől kíméli meg programozóját az OCaml.

További szoftver fejlesztési hatékonyságot növelő eszköz a kódértelmező interaktív módja, mely segítségével „read-eval-print” (beolvas-kiértékel-kiír) módban van lehetőségünk kommunikálni a fordítóval parancssori interfészen keresztül. Ez lehetővé teszi programrészletek, metódusok tesztelését az egész kód fordítása nélkül, továbbá nem kell változó értékeit kiírató, függvényeket írnunk a kódrészletbe a helyes működés felől való meggyőződéshez, mert kiértékelés után a kapott eredményt interaktív módban minden hívás után megkapjuk a kimenetre.

Hibakeresés esetére érdemes megemlíteni az *ocamldebug* nevű programot, melyet Emacs integrált fejlesztői környezetben futtatva a program végrehajtás menete nyomon követhető, forráskódot megjelenítő funkcióval együtt.

Továbbá nyelvcsaládjához híven az OCaml beépített lehetőséget biztosít más nyelvek definiálásához. Az *.mly* és *.mll* kiterjesztésű fájlok segítségével generálhatunk szöveg elemzőt

(lexer) reguláris kifejezések halmazából, melyben minden beolvasott kifejezéshez rendelhetünk végrehajtandó akciókat. Az OCaml forrásfájlok kiterjesztése *.ml*, melyhez minden esetben egy, a fájlnévvel azonos nevű OCaml modul (más objektum orientált nyelvekben osztály) kerül létrehozásra. Minden forrásfájlhoz létrehozhatunk egy *.mli* kiterjesztésű fájlt (interfészfájl), mellyel definiálhatjuk, hogy mely tagmodulok és tagfüggvények legyenek kívülről elérhetőek. Ha egy modulhoz nem definiálunk interfészfájlt, akkor a fordító generál egyet, melyben a modul minden részét publikussá teszi.

Most, hogy minden szükséges eszközt megtárgyaltunk a Frenetic rendszer részletekbe menő megismeréséhez, rátérek néhány konkrét példa bemutatására, melyek segítségével illusztrálhatom a magas absztrakciós szintű hálózatprogramozó rendszerek szükségességét.

2.3 Frenetic bevezető

A Frenetic egy deklaratív szemléletű nyelv OpenFlow hálózatok programozására, melynek segítségével a ma megszokott OpenFlow kontroller platformok által nyújtott lehetőségeknél egy szinttel feljebb lépve definiálhatjuk a hálózat működését. Az absztrakciós szint növelésének és a deklaratív szemléletnek köszönhetően a nyelv sok technikai részlettel való törődéstől szabadítja meg a hálózat programozóját. Lehetőséget biztosít a program modulokba foglalására, mely segítségével a hálózat működése anélkül fejleszthető vagy módosítható, hogy a teljes kódot át kellene látnunk. Ehelyett elég csak a modult úgy módosítani, hogy a teljes program által elvárt interfészt változatlanul hagyjuk.

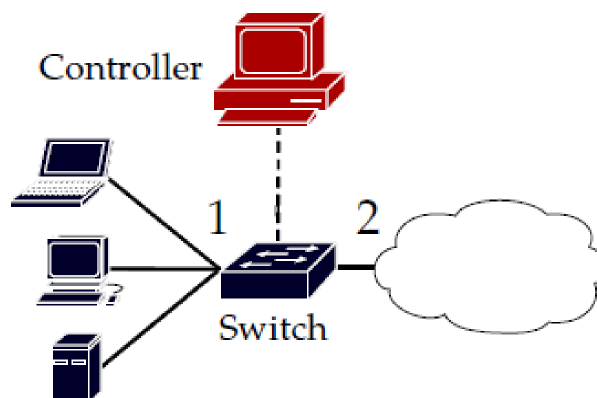
További célja az is, hogy a topológiától amennyire lehet, független legyen a hálózati működés leírása. Ehhez példának hozhatunk egy tűzfal funkciót leíró modult, mely a hálózatban tartózkodó hosztok adott halmazai között csak meghatározott portokon engedi meg a kommunikációt. Ésszerű elvárás, hogy ez az irányelv független legyen a hálózatot reprezentáló gráf szerkezetétől, feltéve, hogy a hálózat lehetővé teszi a minden hoszt pár közötti teljes kapcsolatot.

2.4 OpenFlow nehézségek elemzése

A Frenetic fejlesztői rendszerük létjogosultságát a mai OpenFlow platformok programozásában jelen lévő három fő technikai nehézséggel indokolják, melyeket más problémák mellett a Frenetic kiküszöbölni készül. A fejlesztők több publikációjukban [6][7][8] is összefoglalják ezeket a nehézségeket példákkal illusztrálva, melyekre én is prezentálok egy rövid betekintést a téma könnyebb megérthetősége érdekében. Tehát az alábbi három példa a Frenetic nyelv készítőitől származik az ábrával és a példa kódokkal együtt, melyekben ők a NOX kontroller platform egyik nyelvét használták. Az ésszerű változó elnevezések és a kódértelmezés miatt a jelentés intuitívan érthető a szintaxis pontos ismerete nélkül. A NOX kontrollert egyébként C++, illetve Python nyelven is programozhatjuk [9], a példák az utóbbi nyelven vannak. Nem csak a NOX, hanem általában OpenFlow platformok programozása során a három probléma:

- konkurens modulok interakciói a flow táblában,
- alacsony szintű programozói interfész,
- program két szintű futtatása

Vegyük a következő példákat a nehézségek bemutatására. Legyen adott az alábbi topológia a 3. ábrán egy kapcsolóval, melynek 1-es portjára néhány belső hoszt van csatlakoztatva, 2-es portjára pedig egy nagyobb hálózat (pl.: Internet). A switchhez szaggatott vonallal csatlakoztatott számítógép ikon az OpenFlow kontrollert reprezentálja.



3. ábra: Topológia az OpenFlow nehézségeket illusztráló példákhoz [6]

A 4. ábrán látható a switch alap, kapcsolatteremtő programja, mely tulajdonképpen egy ismétlő funkciót valósít meg. A *repeater* függvény bemeneti paramétere a switchet azonosító változó, melynek összetettebb topológián lenne jelentősége. A *pat1*, illetve *pat2* változók a telepítendő flow bejegyzések illeszkedési struktúráját definiálják, a match struktúra nem megadott mezői tetszőlegesek lehetnek az illeszkedéshez. A *DEFAULT* értékek a bejegyzés prioritására vonatkozik, ennek csak átlapolódó match struktúra esetén van jelentősége. A *None* paraméter érték a flow-k *timeout*-jára (érvényességére) vonatkozik, mely jelen esetben permanens. A flow bejegyzést telepítő függvény utolsó paramétere az akciók listája, mely jelen esetben az illeszkedő csomagok kiküldése az eszköz 2-es, illetve 1-es portján.

A *switch_join* metódus a kapcsoló kontrollerhez való csatlakozása által kiváltott eseménykor hívódik meg, mely a törzsében megadott kód lefuttatásával válaszol, vagyis felkonfigurálja a switchet a repeater funkcióval.

```
def repeater(s):
  pat1 = {IN_PORT:1}
  pat2 = {IN_PORT:2}
  install(s,pat1,DEFAULT,None,[output(2)])
  install(s,pat2,DEFAULT,None,[output(1)])

def switch_join(s):
  repeater(s)
```

4. ábra: Repeater funkciót megvalósító kódrészlet [6]

2.4.1 Modulok közötti interakció

Tegyük fel, hogy ki szertnénk bővíteni a programot egy külső webszerverektől érkező forgalom megfigyelő funkcióval. Ehhez telepítenünk kell egy olyan flow bejegyzést, melyre minden ilyen forgalomhoz tartozó csomag illeszkedik, hogy periodikusan le tudjuk kérdezni az OpenFlow által biztosított, bejegyzésekhez rendelt számlálók értékeit. Ezt az alábbi 5. ábrán látható példakóddal valósíthatjuk meg.

```
def monitor(s):
  pat = {IN_PORT:2,TP_SRC:80}
  install(s, pat, DEFAULT, None, [])
  query_stats(s, pat)

def stats_in(s, xid, pat, ps, bs):
  print bs
  sleep(30)
  query_stats(s, pat)
```

5. ábra: Webforgalom figyelő kódrészlet [6]

A *pat* struktúrába most felvettük a 80-as TCP portra vonatkozó megkötést, mely a webszerverektől jövő forgalomra szűr. A telepített flow bejegyzésben az üres akció lista a csomag eldobását jelenti. A *query_stats* eljárás lekérdezi a *pat* illeszkedési struktúrával rendelkező flow bejegyzések számlálóit az *s* switchtől. Ez az eljáráshívás aszinkron, vagyis a switchtől kapott statisztikát nem várja meg a program végrehajtás, hogy visszatérési értékben adja vissza, hanem azonnal a következő utasítás elvégzésére lép tovább. Így visszatérési értékében a kérést azonosító számot ad vissza.

A beérkező számláló értékeket a *stats_in* függvény kezeli, mely a statisztika kérésre adott válasz megérkezése által kiváltott esemény hatására hívódik meg. Az eljárás kiírja a kapott statisztikát, majd fél perc múlva új kérést indít ugyanarra a flow bejegyzésre.

A célunk tehát az lenne, hogy a monitorozási és továbbítási funkciókat egyszerre futtathassuk a switchen, ezt ideális esetben úgy tehetnénk meg, hogy egymás után meghívjuk a két függvényt a switch csatlakozásakor. Azonban sajnos ez nem egy működőképes megoldás a két modul által telepített flow bejegyzések közti interakció miatt. Ugyanis a *repeater* által telepített *pat2* illeszkedési struktúrájú flow átlapolódik a *monitor* által telepített bejegyzéssel, prioritás értéküket pedig nem definiáltuk, ezért a switch szabadon választhat, hogy az illeszkedő csomagokat melyik flow bejegyzés alapján dolgozza fel. Azonban ez mindkét választás esetén hibás működés lenne, mert a csomag vagy eldobásra kerül, vagy nem növeli a megfigyelő bejegyzés számlálóit.

Ilyen esetben a jelenlegi OpenFlow megoldások mellett kézzel kell előállítanunk a mindkét funkciót megvalósító, elvárt működést. Az így kapott NOX platformbeli kódot a 6. ábrán láthatjuk.

```
def repeater_monitor(s):
  pat1 = {IN_PORT:1}
  pat2 = {IN_PORT:2}
  pat2web = {IN_PORT:2, TP_SRC:80}
  install(s, pat1, [output(2)], DEFAULT)
  install(s, pat2, [output(1)], DEFAULT)
  install(s, pat2web, [output(1)], HIGH)
  query_stats(s, pat2web)
```

6. ábra: Monitorozási és továbbítási funkciót megvalósító kód [6]

A harmadiknak telepített flow bejegyzés prioritása *HIGH*, így a bejövő webforgalomra ez a szabály fog illeszkedni (számlálás, továbbítás), más bejövő forgalomra pedig a *pat2* match struktúrájú bejegyzés, mely az 1-es portra továbbít. Általánosságban az ilyen kód előállítása nehézkes, és a programozó különös figyelmét igényli, annak ellenére, hogy ez nem tartozik a funkciók közvetlen megvalósításához.

2.4.2 Alacsony szintű interfész

Következő nehézség szemléltetéséhez tegyük fel, hogy a forgalom figyelőt úgy szeretnénk kibővíteni, hogy egy adott belső hosztra (a switch 1-es portján, 10.0.0.9-es IP címmel) érkező webforgalom kivételével figyelje a hálózati tevékenységet. Mivel az OpenFlow protokoll nem követeli meg a switchektől, hogy képesek legyenek match struktúrák kivonására vagy negálására, így ez a nehézség abból adódik, hogy a switchek megvalósítását részletekbe menően kell ismernünk a programozásukhoz, vagyis a programozói interfész túl alacsony szintű. Ennek a problémának a megoldására is átlapoló szabályokat kell írnunk, melyek közötti többértelműséget prioritások meghatározásával tudjuk feloldani. A kapcsolódó példakódot a 7. ábrán találjuk.

```
def repeater_monitor_noserver(s):
  pat1 = {IN_PORT:1}
  pat2 = {IN_PORT:2}
  pat2web = {IN_PORT:2, TP_SRC:80}
  pat2srv = {IN_PORT:2, NW_DST:10.0.0.9, TP_SRC:80}
  install(s, pat1, DEFAULT, None, [output(2)])
  install(s, pat2, DEFAULT, None, [output(1)])
  install(s, pat2web, MEDIUM, None, [output(1)])
  install(s, pat2srv, HIGH, None, [output(1)])
  query_stats(s, pat2web)
```

7. ábra: Továbbítás, webforgalom figyelés adott hoszt kivételével [6]

A kódrészletben utoljára telepített flow bejegyzés prioritása most a legmagasabb, mely elvégzi a figyelni nem kívánt forgalom továbbítást anélkül, hogy webforgalmat számláló szabályra illeszkedne a csomag.

2.4.3 Két szintű program futás

A harmadik probléma az, hogy egy SDN program egyik része a kontrolleren fut, mely dönt a flow szabályok elhelyezéséről, eltávolításáról, másik része pedig a switchen, azaz a program két konkurens szinten fut. Így a kontroller a forgalom csak azon részét látja, amelyre nem volt illeszkedő flow bejegyzés. Azonban a switchen nem lekezelt csomagok által kiváltott események a kontroller oldali program kódrészeinek futását vonják maguk után. Tehát a két program kölcsönösen hat egymásra aszinkron üzenetekkel, melynek menedzselése nehéz feladat lehet. Azonban ez az elrendezés elengedhetetlen a forgalom hatékony feldolgozásához, mert a teljes központosítás nem lenne skálázható.

A Frenetic fejlesztői erre is hoznak konkrét példát, mely megtekinthető a hivatkozásoknál található dokumentumokban [6], de ebben a dolgozatban ezt a példát inkább nem mutatnám be, mert az előzőekhez képest nem szolgál különösebb új tanulsággal.

Általánosságban ezek a problémák intuitívan és átláthatóan kifejezhetőek lennének logikai operátorokkal és halmazműveletekkel. Az SDN programok méretének növekedésével a könnyű átláthatóságra és helyes működés egyszerű belátására egyre nagyobb szükség van, így egy ilyen magasabb absztrakciós szintű architektúra fejlesztése időszerű.

3 Frenetic alkalmazási lehetőségei

Mára a Frenetic több implementációja is elkészült, megbízhatóságot és funkcionalitást egyre növelve jelentek meg a teljesen elölről kezdett, más-más programozási nyelvekben megvalósított változatai.

A kezdeti verzió a Python nyelven implementált Frenetic [6][7], melyben már kiküszöbölésre kerültek az előző fejezetben elemzett OpenFlow/NOX nehézségek. Ennek megvalósítására bevezetésre került a „kontroller lát minden csomagot" absztrakció, a magas szintű csomagfejléc-mintakezelés és a modulok együttműködésének egyszerű leírása a funkciók kompozíciója segítségével. Ezen elveket a későbbi verziók is megtartották, így kifejtésükre az általam vizsgált implementáció ismertetésénél térek ki.

A következő verzió a nyelvet és a fordítót dolgozza jobban ki, ehhez a fejlesztők saját algebrai kalkulust is készítettek, mely segítségével formálisan leírható a fordítási algoritmus, és annak a szemantikához viszonyított, rendszer szintű helyessége bizonyítható. A nyelvet átnevezték NetCore-ra, melyet a Haskell funkcionális nyelven implementáltak, belevéve *wildcard* illeszkedési struktúrákat használó fordítást, és néhány heurisztikus optimalizációt a generált flow táblák méretének kordában tartásához (a pythonos implementáció csak pontos illeszkedési struktúrákat használt). Az algoritmus részletes, formális leírása megtalálható a [10] hivatkozásban.

Jelen dolgozatban vizsgált megvalósítás OCaml nyelven készül, egy GitHubon tárolt nyílt forráskódú projekt formájában [11]. Néhány külsős fejlesztővel együttműködésben a Princeton és a Cornell egyetem munkatársai fejlesztik a rendszert, melyben Frenetic néven a futtatói környezetre hivatkoznak, NetCore vagy FreneticDSL (Domain Specific Language) néven pedig a magas absztrakciós szintű nyelvre, és a hozzá tartozó fordítóra. Ehhez implementációs részleteket közlő publikáció még nem jelent meg a dolgozat írásának ideje alatt, így vizsgálatom és fejlesztésem során a forráskódra, illetve a korábbi publikációkra támaszkodtam.

Ismereteim szerint az ocaml-es verzióról szóló első publikáció a 2013. júniusi *ACM SIGPLAN Conference on Programming Language Design and Implementation* konferencián jelent meg [12], mely munkám érdemi részének befejezése után jutott el hozzám. Ezen publikáció írói prezentálják az első formális OpenFlow modellt, mely segítségével a Coq

formális bizonyítás kezelő rendszerben [13] leírva gép-ellenőrzött bizonyítást adnak a Frenetic rendszer helyességére. A formális bizonyítás a tanulmányozásával nem foglalkoztam.

A publikációban (és a forráskód egyik megjegyzésében) találhatunk utalást arra, hogy tervezik a rendszert dinamikus konfigurálhatósággal is kiegészíteni, melyet jelenleg még nem támogat a Frenetic rendszer szinten. Az ebből fakadó problémákat és korlátokat a következő fejezetekben tárgyaljuk.

3.1 Szükséges könyvtárak

Az OCaml-ben implementált Frenetic forráskódját GitHubon hosztolt projektként érhetjük el [11], melynek vizsgálatom során az 1.0.2-es stabil verzióját használtam. Más verzióra való hivatkozás nélküli állításaim a Frenetic 1.0.2-re vonatkoznak. A fejlesztők egyébként azóta is napi aktivitással dolgoznak a kódon, melynek eredményei a projekt verziókövető tárhelyén nyíltan nyomon követhető.

Mint már korábban említettem, a Frenetic az OpenFlow 1.0.0-ás verziót használja, melyet a projekt keretein belül megalkotott, OCaml-ben megírt, az 1.0.0-át majdnem teljes egészében megvalósító OpenFlow könyvtár támogat. A segédkönyvtár tartalmaz egyébként néhány kísérleti fejlesztést az OpenFlow 1.3 támogatására is, melyek a Frenetic által jelenleg még nincsenek használatban. A fejlesztők implementáltak OCaml-ben további két segédkönyvtárat: hálózati csomagok konvertálására bináris struktúra és OCaml struktúrák között, illetve egy alap funkciókkal rendelkező kontroller platform hívásait megvalósító könyvtárat, melyet Ox kontrollernek neveztek el.

A fenti három könyvtár szintén elérhető a verziókövető rendszerben említésük sorrendjében *ocaml-openflow*, *ocaml-packet*, illetve *ox* néven [14]. Sajnos egyik segédkönyvtár sem teljes, a csomagkezelő könyvtárban például kizárólag TCP, illetve ICMP transzport rétegbeli protokollok akadálytalan használatát teszi lehetővé, valamint az IP és ARP fejlécek konstruálását segíti. Ha valamilyen más fajta csomagot szeretnénk alkalmazni jelen rendszer támogatásával, akkor kényszermegoldáshoz kell folyamodnunk, melynek egyik lehetséges útjára a későbbiekben térek ki.

A rendszer telepítéséhez és fordításához szükséges néhány további könyvtár, melyek implementálása nem a Frenetic fejlesztőihez köthető. Ilyenek a *Cstruct*, illetve *Lwt* könyvtárak.

Cstruct egy könyvtár és szintaxis kiterjesztés az OCaml nyelvhez, melynek segítségével egyszerűen el lehet érni C nyelvhez hasonló struktúrákat közvetlen az OCaml forráskódból. Ennek a könyvtárnak a függvényeit alkalmazza például az ocaml-packet segédkönyvtár bináris struktúrák előállítására. A Cstruct Frenetic által használt 0.7.0-ás verziója szintén elérhető GitHub-on [15].

Az Lwt könyvtár (Lightweight Threads Library) [16] egy kooperatív szálkezelő funkciókat nyújtó kiterjesztés az OCaml nyelvhez. A függvénygyűjtemény lényege, hogy minden esetben a változó értékének visszaadása helyett egy szállal térnek vissza a metódusok. Így a függvényhívások gyakorlatilag aszinkron módon hatódnak végre. A futásuk befejeződésével a függvény által visszaadott szál értéke tartalmazza a hívás visszatérési értékét. A könyvtár biztosít számos OCaml szintaxis kiterjesztést, illetve operátort, melyek használatával a szálak egymás után láncolása, vagy párhuzamos indítása gyorsan leírható, és könnyen átlátható.

Szerencsére a rendszer beüzemeléséhez nem kell az összes említett könyvárat egyenként beszereznünk, mert a fejlesztők készítettek egy VirtualBox virtuális gépet Ubuntu 13.04 operációs rendszerrel, melyre minden szükséges könyvtárat feltelepítettek a Mininet 2.0-val [3] és a Frenetic 1.0.1-es verziójával együtt. Habár az 1.0.1-ás verzió nem mutat sok különbséget a Frenetic 1.0.2-höz képest, érdemes a frissítést elvégezni.

Fontos megjegyezni, hogy a Frenetic jelenlegi, stabilnak nevezett verziója, még nem került semmiféle módon forgalomba. Jelen dolgozatban vizsgált verziót kizárólag a rendszer fejlesztői, illetve a munka iránt érdeklődők tudták csak használni a nyílt forráskódnak köszönhetően. Továbbá a fentiekben említett gép-ellenőrzött bizonyítás nem biztosítja a szoftver hibátlan működését, hanem csak az algoritmus elvi működésének helyességét igazolja, így az implementáció még nem esett át mindenféle szükséges tesztelésen. Mindezek ellenére egy ígéretes kezdeményezésnek indul a területen, így fejlesztésének figyelemmel követése és elemzése megéri a belefektetett munkát.

3.2 Tesztkörnyezet: Mininet

A Mininet 2.0 [3] egy Pythonban implementált, Linux rendszer alatt futó, nyílt forráskódú hálózat szimulációs eszköz, mely segítségével egy asztali PC teljesítményű

számítógépen akár százas nagyságrendű hosztokból felépített, szimulált hálózat kezelésére is alkalmas. A Mininet használtam a Frenetic kontroller által irányított hálózat szimulálására, illetve a dolgozat végén bemutatott méréseket is ebben a környezetben végeztem. Ezért szeretnék ebben a fejezetben betekintést nyújtani a Mininet alkalmazási lehetőségeire.

A szimulált hálózati környezetben minden hosztot és switchet egy-egy shell folyamat reprezentál, melyekhez a Mininet konzolján keresztül férhetünk hozzá a shell parancsokat a node nevével prefixálva. A node-ok nincsenek egymástól teljesen szeparálva, mindegyik az operációs rendszer fájljait és folyamatait látja, de hálózati interfészeiket, és a hálózati működéssel kapcsolatos beállításaikat külön kezelhetjük. Így például mindegyik hosztnak van külön ARP tárolója, és külön példányban futtathatnak DHCP klienst, mely a későbbiekben fontos lesz jelen dolgozatban bemutatott munka szempontjából.

Mininet API (Application Programming Interface) segítségével összeállíthatunk tetszőleges topológiát, és lehetővé teszi tesztek automatizálását. Az API részletezésére nem térnék ki, hiszen az a dolgozat szempontjából nem releváns, de a hivatkozások között megtalálható a rá vonatkozó referencia [4].

A Mininet segítségével szimulálhatunk konvencionális hálózati működést, de kiváló lehetőséget nyújt OpenFlow-val való együttműködésre, mert a hálózatot indíthatjuk úgy, hogy csatlakozzon egy távoli kontrollerhez a megadott TCP socketen keresztül. Használt switchei OpenFlow képesek, a kontroller alkalmazásból fel tudjuk konfigurálni a flow tábláikat, amely a *dpctl dump-flows* parancs segítségével a Mininet konzolból is ellenőrizhető.

A Frenetic rendszer elemzéséhez, illetve a megvalósított kiterjesztések teszteléséhez a Mininet 2.0-ás verziót használtam, az összeállítás együttes működésére a későbbiekben láthatunk példát.

3.3 FreneticDSL bemutatása

A preparált Frenetic virtuális géphez készült egy OpenFlow gyorstalpaló az Ox platformra, illetve egy kidolgozott Frenetic feladatsor (tutorial), mely bemutatja a rendszer nyújtotta alap lehetőségeket [17]. A tutorial két lépcsős szerkeze jól tükrözi a rendszer architektúráját is, hiszem a Frenetic nyelv nem csak absztrakciós szintben áll feljebb a hagyományos OpenFlow kontrollereknél, hanem architektúrában is. A Frenetic kontroller nem

lecserélni kívánja az eddigi kontroller implementációkat, hanem egy felettük lévő szinten, de velük együtt dolgozni.

Ebben a fejezetben a turorial általam átdolgozott változatát vesszük végig, fókuszba helyezve a Frenetic nyújtotta lehetőségek és szemlélet példaorientált illusztrációját. Vagyis az ebben a fejezetben prezentált ábrák és példakódok nagy része a saját termékem[3]. A használt topológia, illetve a tutorial alapú bemutatás ötlete származik a Frenetic fejlesztőitől. Természetesen a kutatás korai stádiumában, a nyelv megismeréséhez szükséges volt számos, a tutorialban bemutatotthoz hasonló alkalmazás készítése, melyeket a későbbiekben a Frenetic képességeinek feltárására használhattam fel. A Frenetic tutorial Ox platformra vonatkozó, első felét itt kihagynám, mivel hasonló jellegű feladatokat láthattunk az OpenFlow/NOX nehézségek elemzésénél.

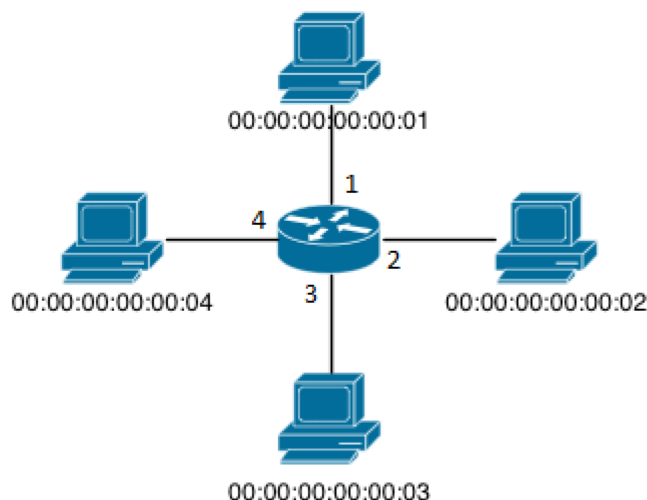
A hálózati működés leírását úgynevezett irányelvekkel (policy), tudjuk definiálni, melyeket párhuzamosan vagy szekvenciálisan összekapcsolva összetettebb irányelveket kaphatunk. A policy-kat el tudjuk helyezni a csomagfolyam útjába, így a kontroller által lefordított irányelvek minden csomagra kiértékelődnek. Ezt neveztük a „kontroller lát minden csomagot" absztrakciónak. Fontos megjegyezni, hogy ez valójában nem így történik, hiszem az OpenFlow szabályokra lefordított irányelveknek a hálózati forgalom a switchekben kerül feldolgozásra.

A párhuzamos és szekvenciális irányelv összekapcsoló operátorok biztosítják, hogy a programot modulárisan kezelhessük. Szétválaszthatjuk az egyes funkciókat külön policy-ban megírva, melyeket akár külön fájlba mentés után az operátorok segítségével egyszerűen összekapcsolhatunk.

Irányelvek leírásánál használhatunk a csomagfejlécekre vonatkozóan definiálható predikátumokat, melyek között logikai és halmaz műveleteket alkalmazva az illeszkedési feltételek magas szinten kezelhetők.

A következő néhány példában vegyük az 8. ábrán látható hálózati topológiát, melyben egy OpenFlow switch köt össze négy PC-t. A hálózatot Mininet segítségével hozhatjuk létre, legegyszerűbben az ábrán látható indulási paraméterek megadásával.

[3] A topológiát bemutató ábra, és így maga a hálózati topológia a tutorialból van. [17]




```
sudo mn --controller=remote --topo=single,4 --mac
```

8. ábra: A feladatokban használt topológia és a hozzá tartozó Mininet indítási parancs [17]

A *topo* indulási paraméterrel jelezhetjük, hogy milyen jellegű topológiát szeretnénk: egy switchet, négy darab hoszttal. A *mac* kapcsoló az alapértelmezett, véletlenszerűen generált hoszt MAC címek helyett, emberi szemmel is kényelmesen olvasható értékekkel indítja a hosztokat. Mininetben a hosztokra a generált neveik alapján hivatkozhatunk *h1, h2, h3* és *h4*-ként, melyek ilyen sorrendben kapják meg a növekvő MAC címeket is, és ilyen sorrendben kapcsolódnak a switch növekvő azonosítókkal rendelkező portjaihoz. A hosztokhoz IP címek is generálódnak automatikusan, melyek a MAC címekhez hasonlóan, sorrendben rendelődnek hozzá a hosztokhoz, alapbeállításként a 10.0.0.0/8-as IP címtartományból. A *controller* paraméterrel jelen esetben azt jelezzük a Mininetnek, hogy távoli (remote) kontrollerhez kell csatlakoznia, melyet alapbeállításként a loopback címen (127.0.0.1) keres, a 6633-as porton.

3.3.1 Forgalom továbbító program írása

Egy egyszerű csomagtovábbító policy az 9. ábrán látható, mely minden switch pár között kapcsolatot létesít.

```
let forwarding =
  if dlTyp = arp then
    all
  else if dlDst = 00:00:00:00:00:01 then
    fwd(1)
  else if dlDst = 00:00:00:00:00:02 then
    fwd(2)
  else if dlDst = 00:00:00:00:00:03 then
    fwd(3)
  else if dlDst = 00:00:00:00:00:04 then
    fwd(4)
  else drop in

monitorTable(1, forwarding)
```

9. ábra: Csomagtovábbítást megvalósító Frenetic kódrészlet

FreneticDSL-ben, sok más deklaratív nyelvhez (pl.: OCaml, Haskell) hasonlóan a *let VAR = ... in* szerkezettel zárhatunk közre kifejezéseket, melyekre a kód későbbi részein, *VAR* néven hivatkozhatunk. Van beépített *if-then-else* szerkezet, mellyel az akciók kiértékeléséhez feltételeket rendelhetünk.

A *dlTyp* predikátummal az Ethernet fejléc típusmezőjére fogalmazhatunk meg feltételt. Az *arp* beépített nyelvi konstans, melynek értéke 0x0806, vagyis az ARP keretet azonosító 8 bites érték. A 9. ábra szerint ARP csomag esetén az *all* akciót kell az irányelvnek jelentenie, amely a csomag floodolását (azaz kiküldését minden porton, a beérkező kivételével) vonja maga után. A *dlDst* predikátum az Ethernet keret cél MAC címére vonatkozik, amely ha az adott étéket veszi fel, a csomag az *fwd* paraméterében megadott azonosítójú switch porton kerül továbbításra. Amennyiben a felsorolt négy MAC cím közül a csomag cél MAC címe egyikre sem illeszkedik, a csomag eldobásra kerül.

A *monitorTable* nyelvi elem segítségével megnézhetjük, hogy a második paraméterként megadott policy, milyen flow bejegyzésekre fordult az első paraméterként, azonosítójával megadott switch flow táblájában. A kimeneten a flow szabályok prioritása csökkenő sorrendben jelenik meg. A *monitorTable* hívása egyébként nem kötelező, ha nem vagyunk kíváncsiak a generált flow szabályokra, a policy-t egy külön sorban is odaírhatjuk a fájl végére.

Miután elindítottuk a Mininetet a 8. ábrán látható paranccsal, kipróbálhatjuk a csomagtovábbítási irányelvünket. A példakódot elmentve a *mytutorial1.nc* fájlba, és az állományt a Freneticnek indulási paraméterként átadva, a rendszer lefordítja a kódot, és

felkonfigurálja a csatlakozó switchet. A *monitorTable* kimenetét és a policy fájl fordítását a 10. ábrán láthatjuk.

```
frenetic@frenetic:~/OpenFlow_cft/src/frenetic-frenetic.1.0.2/examples$ frenetic
mytutorial1.nc
Flow table at switch 1 is:
 {dlTyp = arp} => [Output AllPorts]
 {dlDst = 00:00:00:00:00:01} => [Output 1]
 {dlDst = 00:00:00:00:00:02} => [Output 2]
 {dlDst = 00:00:00:00:00:03} => [Output 3]
 {dlDst = 00:00:00:00:00:04} => [Output 4]
 {*} => []
```

10. ábra: A monitorTable kimenete a Frenetic sztenderd outputján.

Mininetben a beépített *pingall* paranccsal végrehajthatunk egy teljes kapcsolat tesztet, mely a hosztok automatikusan kiosztott IP címeit felhasználva, minden hosztról kiküld egy ICMP *ping request* csomagot minden másik hosztnak. A Mininet indítását és a sikeres kapcsolatteszt konzolra kapott kimenetét a 11. ábrán láthatjuk.

```
frenetic@frenetic:~$ sudo mn --controller=remote --topo=single,4 --mac
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1) (h4, s1)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
*** Starting 1 switches
s1
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (0/12 lost)
mininet> 
```

11. ábra: Mininet indítása és a kapcsolatteszt kimenete

A hálózatunk működésének vizsgálatát a *Wireshark* protokoll analizáló programmal is megfigyelhetjük. A Mininet által létrehozott virtuális interfészeket is felhasználhatjuk

csomagok elkapására. A 12. ábrán a switch 1-es azonosítójú interfészén elkapott csomagokat láthatjuk.

No.	Source	Destination	Protocol	Info
1	00:00:00_00:00:01	Broadcast	ARP	Who has 10.0.0.2? Tell 10.0.0.1
2	00:00:00_00:00:02	00:00:00_00:00:01	ARP	10.0.0.2 is at 00:00:00:00:00:02
3	10.0.0.1	10.0.0.2	ICMP	Echo (ping) request id=0x0c8d, seq=1/256, ttl=64
4	10.0.0.2	10.0.0.1	ICMP	Echo (ping) reply id=0x0c8d, seq=1/256, ttl=64
5	00:00:00_00:00:01	Broadcast	ARP	Who has 10.0.0.3? Tell 10.0.0.1
6	00:00:00_00:00:03	00:00:00_00:00:01	ARP	10.0.0.3 is at 00:00:00:00:00:03
7	10.0.0.1	10.0.0.3	ICMP	Echo (ping) request id=0x0c8e, seq=1/256, ttl=64
8	10.0.0.3	10.0.0.1	ICMP	Echo (ping) reply id=0x0c8e, seq=1/256, ttl=64
9	00:00:00_00:00:01	Broadcast	ARP	Who has 10.0.0.4? Tell 10.0.0.1
10	00:00:00_00:00:04	00:00:00_00:00:01	ARP	10.0.0.4 is at 00:00:00:00:00:04
11	10.0.0.1	10.0.0.4	ICMP	Echo (ping) request id=0x0c8f, seq=1/256, ttl=64
12	10.0.0.4	10.0.0.1	ICMP	Echo (ping) reply id=0x0c8f, seq=1/256, ttl=64

12. ábra: A switch 1-es interfészén,
Wiresharkkal elkapott csomagok

Mivel a hosztok ARP cache-se kezdetben üres, a *h1* hosztnak minden másik hoszt eléréséhez először ARP kérést kell indítania, hogy feloldja az IP címet, és megszerezze a hozzá tartozó MAC címet, amely alapján a továbbítási policy el tudja irányítani a csomagot.

Több switchből álló topológia esetén szükségünk lenne a Frenetic által biztosított *switch = <ID>* predikátumra, mely segítségével a továbbítási viselkedést egy adott switchhez (vagy logikai operátorok segítségével switchek egy halmazához) tudjuk rendelni. Erre vonatkozó példától most eltekintenék, mivel a *switch* predikátumon kívül, más mondanivalója nem lenne az eddigiekhez képest. De ha az olvasó ezt mégis szükségesnek érzi, az előző példák alapjául szolgáló Frenetic Tutorialban [17] megtalálható a továbbítási irányelv egy két szintű fatopológiára.

3.3.2 Tűzfal funkció készítése

Játsszon a *h1* hoszt webszerver funkciót! Implementáljunk Freneticben egy olyan tűzfal irányelvet, hogy a szerverrel csak HTTP protokollal (vagyis 80-as TCP porton keresztül) lehessen kommunikálni, a másik három hoszt között pedig csak ICMP csomagokat engedjünk forgalmazni. Ahogy már korábban említésre került, a Frenetic lehetőséget nyújt a policy-k külön fájlba való szeparálására. A *mytutorial2.nc* fájl tartalma a 13. ábrán látható.

```
include "mytutorial1.nc"

let firewall =
  if dlTyp = arp ||
     ((dlDst = 00:00:00:00:00:01 && tcpDstPort = 80) ||
      (dlSrc = 00:00:00:00:00:01 && tcpSrcPort = 80)) ||
     (!dlDst = 00:00:00:00:00:01 && nwProto = icmp)
  then pass
  else drop in

monitorTable(1, firewall; forwarding)
```

13. ábra: Külön fájlban megírt tűzfal policy

Az *include* kulcsszó után, idézőjelek között megadott fájlból beolvashatjuk a benne tárolt policy-kat, melyekre ebben a fájlban hivatkozni szeretnénk. Ilyenkor természetesen a *montitorTable* sort letörölhetjük a *mytutorial1.nc* fájlból.

Az irányelveket egymás mögé kapcsolhatjuk a szekvenciális operátorral, melyre a pontosvessző karakterrel hivatkozhatunk. Ilyenkor gondolhatunk úgy a forgalomra, hogy minden csomagra kiértékelődik a szekvenciális operátor bal oldalán található policy, majd a jobboldali. Ezért miután az *if-then-else* szerkezet feltételében megfogalmaztuk a megfelelő predikátumot, annak teljesülése esetén a csomag továbbítását szeretnénk ítéletként kapni, melyet jelen esetben a *pass* akcióval jelzünk. A *pass* kulcsszó tehát annyit tesz, hogy a kapott csomagot módosítás nélkül továbbadja a következő policynak.

A predikátumot a teljes forgalom három részhalmazát kijelölő részre bonthatjuk, melyen részhalmazok továbbítását engedélyezni szeretnénk a hálózatban:

- Minden ARP csomag, melyek lebonyolítják az IP címfeloldást
- A szerverhez irányuló HTTP forgalom, vagy a szervertől indított http forgalom
- Minden ICMP csomag, melynek nem a szerver a forrása (így a szerverhez irányuló pingek sem lesznek sikeresek, mert a szervertől érkező válaszcsomag eldobásra kerül)

Az így generált flow tábla már lényegesen bonyolultabb, melyet kézzel megírni már meglehetősen vesződséges lett volna a prioritás értékek és a megfelelő, átlapolódó illeszkedési struktúrák meghatározása miatt. Az így kapott *monitorTable* kimenetet a 14. ábrán láthatjuk.

Ehelyett tisztán megfogalmazható, könnyen érthető formában írhattuk le a kívánt viselkedést a FreneticDSL segítségével.

```
frenetic@frenetic:~/OpenFlow_cft/src/frenetic-frenetic.1.0.2/examples$ frenetic mytutorial2.nc
Flow table at switch 1 is:
 {dlTyp = arp} => [Output AllPorts]
 {dlDst = 00:00:00:00:00:01, dlTyp = arp} => [Output 1]
 {dlDst = 00:00:00:00:00:02, dlTyp = arp} => [Output 2]
 {dlDst = 00:00:00:00:00:03, dlTyp = arp} => [Output 3]
 {dlDst = 00:00:00:00:00:04, dlTyp = arp} => [Output 4]
 {dlDst = 00:00:00:00:00:01, dlTyp = ip, nwProto = tcp, tpDst = 80} => [Output 1]
 {dlSrc = 00:00:00:00:00:01, dlDst = 00:00:00:00:00:01, dlTyp = ip, nwProto = tcp, tpSrc = 80}
=> [Output 1]
 {dlSrc = 00:00:00:00:00:01, dlDst = 00:00:00:00:00:02, dlTyp = ip, nwProto = tcp, tpSrc = 80}
=> [Output 2]
 {dlSrc = 00:00:00:00:00:01, dlDst = 00:00:00:00:00:03, dlTyp = ip, nwProto = tcp, tpSrc = 80}
=> [Output 3]
 {dlSrc = 00:00:00:00:00:01, dlDst = 00:00:00:00:00:04, dlTyp = ip, nwProto = tcp, tpSrc = 80}
=> [Output 4]
 {dlSrc = 00:00:00:00:00:01, dlTyp = ip, nwProto = tcp, tpSrc = 80} => []
 {dlDst = 00:00:00:00:00:01} => []
 {dlDst = 00:00:00:00:00:01, dlTyp = ip, nwProto = icmp} => [Output 1]
 {dlDst = 00:00:00:00:00:02, dlTyp = ip, nwProto = icmp} => [Output 2]
 {dlDst = 00:00:00:00:00:03, dlTyp = ip, nwProto = icmp} => [Output 3]
 {dlDst = 00:00:00:00:00:04, dlTyp = ip, nwProto = icmp} => [Output 4]
 {*} => []
```

14. ábra: Tűzfal és továbbítási irányelvekből generált flow tábla

A helyes működés ellenőrzéséhez végezzünk el Mininetben egy *pingall*-t, és indítsunk egy egyszerű HTTP szervert üzemeltető Python szkriptet, mely a futtatáskor aktuális könyvtár fájljait szolgálja ki a klienseknek. A 15. ábrán látható a generált ICMP forgalom által és a *test.txt* fájl webszervertől való lekérése által produkált kimenet. A teszt eredménye összhangban van az implementálni kívánt működéssel.

```
mininet> h1 python -m SimpleHTTPServer 80 &
mininet> h2 curl 10.0.0.1/test.txt
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    40  100    40    0     0   8810      0 --:--:-- --:--:-- --:--:-- 10000
This text is only for testing purposes!
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X X
h2 -> X h3 h4
h3 -> X h2 h4
h4 -> X h2 h3
*** Results: 50% dropped (6/12 lost)
mininet> █
```

15. ábra: Forgalom lebonyolítása a 80-as porton, és ICMP csomagok küldése

3.3.3 Forgalom figyelés implementálása

Utolsó példában bővítsük ki a programot webforgalom figyelő funkcióval, melyet a párhuzamosítás operátor segítségével tehetünk meg egyszerűen. Ehhez hozzunk létre egy új fájlt *mytutorial3.nc* néven, melynek tartalmát az 16. ábrán láthatjuk. A tűzfalat megvalósító fájlban hagyjuk benn a parancsot, amely megnyitja a *mytutorial1.nc* fájlt, de a szükségtelen *monitorTable* eljárást töröljük ki.

```
include "mytutorial2.nc"

let monitoring =
  if tcpSrcPort = 80 ||
     tcpDstPort = 80
  then monitorLoad(2, "HTTP traffic") in

monitorTable(1, firewall; (forwarding + monitoring))
```

16. ábra: Monitorozási funkció hozzáadása az eddigiekhez

A *monitorLoad* első paraméterében megadhatjuk, hogy hány másodpercenként szeretnénk összegezve megkapni a forgalom nagyságát. Második paramétere pedig, hogy milyen címkével szeretnénk a kapott eredményeket ellátni. A *monitorLoad* eljáráshoz rendelt predikátummal definiálhatjuk, hogy milyen jellegű forgalomra szeretnénk a mérést végezni. Bármely webszerverhez, vagy attól érkező forgalmat a 80-as forrás vagy cél portra való szűréssel kaphatjuk meg. A forgalom figyelés kimenetét a kontroller oldali konzolon, a Frenetic sztenderd kimenetén láthatjuk, ahogy ezt az 17. ábra is mutatja.

```
[HTTP traffic] 0 packets and 0 bytes in the last 2.000000 seconds.
[HTTP traffic] 0 packets and 0 bytes in the last 2.000000 seconds.
[HTTP traffic] 74 packets and 5863 bytes in the last 2.000000 seconds.
[HTTP traffic] 46 packets and 3662 bytes in the last 2.000000 seconds.
[HTTP traffic] 2 packets and 148 bytes in the last 2.000000 seconds.
[HTTP traffic] 22 packets and 1757 bytes in the last 2.000000 seconds.
[HTTP traffic] 0 packets and 0 bytes in the last 2.000000 seconds.
[HTTP traffic] 0 packets and 0 bytes in the last 2.000000 seconds.
```

17. ábra: A monitorLoad HTTP forgalmat figyelő kimenete

A párhuzamosítás operátor az összeadásjel. Mindkét operátor balasszociatív, és azonos prioritású, de mivel a forgalom figyelő policy-val csak a tűzfal által megszűrt forgalmat szeretnénk nézni, szükséges a 16. ábrán látható zárójelezés.

A párhuzamosítás operátor megvalósítása csomag duplikálással történik. Felfoghatjuk úgy, hogy amikor a csomag elér egy olyan helyre ebben a policy *pipeline*-ban (csővezetékben), ahol kettő vagy több policy-nak megfelelően, párhuzamosan kell továbbítási döntést hozni, akkor a csomag megsokszorozódik, és minden csomag pontosan egy irányelv alapján lesz feldolgozva a párhuzamosak közül. Ennek segítségével olyan policy-k is végrehajthatók párhuzamosan, amelyek különböző továbbítási döntést hoznak.

4 Frenetic fejlesztői elemzése

A policy összekapcsoló operátorokat az előző fejezetben felsorolt jó tulajdonságiak ellenére óvatosan kell kezelni, mert könnyen létrehozhatunk olyan irányelv csővezetéket, melyben szokatlan viselkedést észlelhetünk. Például két szekvenciálisan összekapcsolt policy továbbítási döntései közül mindig a korábbi döntés érvényesül, ami a későbbi policy működését elrontja. Tehát attól, hogy az irányelveket magas szintű kompozícióval tudjuk összekapcsolni, a megfelelő operátor kiválasztását alapos átgondolásnak kell megelőznie.

Az előző példákból jól láthatjuk, hogy a *firewall* és a *monitoring* irányelvek függetlenek a hálózat szerkezetétől. Vagyis ha a fenti működést egy másik hálózatra szeretnénk átültetni, kizárólag a *forwarding* policy-t kellene lecserélnünk, a *mytutorial1.nc* fájlban kellene csak megírnunk a topológia specifikus hálózati policy-t.

A FreneticDSL meglehetősen megkönnyíti egy hálózat programozójának a munkáját, azzal, hogy sok alacsony szintű dolgot rejt el az OpenFlow switchek viselkedéséből. Tehát kijelenthetjük, hogy a kitűzött cél elérése felé jó úton halad a nyelv fejlesztése. Azonban ez nem teljesen a dolgozat bevezetőjében vízionált rendszer rétegében mozog, hanem talán eggyel alatta, mert nem nyújt lehetőséget a szolgáltatások objektumként való kezelésére. De a Frenetic által nyújtott interfész, és fordítási mechanizmus hasznos lehet a szolgáltatás szintű absztrakciós réteg implementálására. A FreneticDSL teljes szintaxisát és a predikátumok teljes listáját a project GitHub oldalán megtalálhatjuk Frenetic Manual néven [18].

Jelenleg a Frenetic rendszerhez nincs nyilvánosan elérhető, komolyabb dokumentáció. Így a mélyebb megismerés érdekében a forráskódba kellett beleásnom magam. Ebben a fejezetben tárgyalt megállapítások mindegyike a saját termékem. A nyílt forráskódjának köszönhetően lehetőségünk van saját modulok fejlesztésére, melyeket a *.nc* fájlokból beolvashatóvá tudunk tenni, és így a policy fájlok kifejezőképességét tetszőlegesen ki tudjuk terjeszteni.

4.1 Modulok felépítése

Jelen dolgozatban vizsgált Frenetic 1.0.2 verzióban egyelőre csak egy ilyen stabilan működő modul található, mely tanuló switch algoritmust valósít meg.[4] Ezt policy fájlban a *learn* kulcsszóval érhetjük el, melyet *switch* predikátumokkal switchek egy halmazához rendelve, azokon futtathatjuk a tanuló switch algoritmust. A *learn* kulcsszóra tekinthetünk úgy, mint egy tetszőleges policy fájlban definiált irányelvre vagy beépített policy-ra (mint például a *drop* vagy a *pass*): kompozícióba helyezhetjük más policy-kkal, minden helyen szerepelhet, ahol az eddigi statikus policy-k.

A modulok segítségével dinamikusan működő irányelveket tudunk létrehozni, melyek továbbítási döntései a hálózat működése közben változhatnak. Ehhez minden modul számon tart egy kívülről közvetlenül nem elérhető adatfolyamot (data stream), melyet Freneticben a korábban már említett Lwt könyvtár Lwt_stream [19] modulja (vagy alkönyvtára) által biztosított adattípus felhasználásával valósítottak meg. A *stream* adattípus tulajdonképpen egy lista, melynek létrehozásakor kapunk egy függvényt, melynek hívásával a paraméterként átadott adatot beleteszi az adatfolyamba. Az lwt_streambeli, adatfolyamot példányosító függvény OCaml-ben megszokott jelölés szerinti szignatúráját a 18. ábrán láthatjuk.

```
val create : unit -> 'a t * ('a option -> unit)
```

18. ábra: Adatfolyamot létrehozó függvény az Lwt_stream könyvtárból

A *unit* az OCaml-beli egységtípus, melyet kódban egy üres, kerek zárójelpárral írunk. Mivel az OCaml funkcionális nyelv, minden függvénynek vissza kell térnie valamilyen értékkel. Ahol más nyelvekben egy függvény visszatérési értéke *void*, ezt OCaml-ben egységelemet visszaadó függvénynek feleltethetjük meg. A *create* füsggvény egy N-essel (tuple) tér vissza, mely elemeinek típusait a * operátor választja el. Szokványosan OCaml-ben *'a*-val jelezzük a tetszőleges típus helyét. Így az N-es első eleme egy tetszőleges típusú adatfolyam, második eleme pedig egy függvény, amellyel a streambe adatelemet tudunk elhelyezni (ezt nevezzük az adatfolyam *push* függvényének).

[4] Van kezdetleges NAT és állapotteljes tűzfal (stateful firewall) funkciókat megvalósító modul is, de ezek működése tesztjeim szerint még nem teljes.

A *t* típus OCaml-ben konvencionálisan mindig az adott OCaml modul típusát jelenti, jelen esetben a tetszőleges típusú adatot tároló adatfolyam típusa a *'a t*. A *'a option* jelentése olyan típus, mely kétféle értéket vehet fel: *Some 'a*, illetve *None*. Ha az adatfolyamba *None* értéket teszünk, az a stream befejezését jelenti. Egy adatfolyam elemeinek típusa tetszőleges lehet, de minden elem típusának azonosnak kell lennie, mely gyakorlatilag (*type inference* miatt) az első elhelyezett elem típusa.

Az OCaml szintaxis mélyebb ismertetése jelen dolgozatnak nem célja.

A policy fájlokban leírt irányelvek beolvasás után, a Frenetic belső struktúráiban *policy típus*ként tárolódnak. Így például egy policy fájlban leírt *if-then-else* struktúra programozói szemmel egy *ITE (predicate, policy, policy)* értékű policy típus, ahol az első paraméter a beolvasott predikátum (mely a policy típushoz hasonlóan egy predikátum típus), a második paraméter a predikátum teljesülése esetén érvényes irányelv, a harmadik paraméter pedig az „else ágon" teljesülő irányelv. Itt megjegyzendő, hogy emiatt a belső struktúra miatt az „else ág" megadása kötelező.[5] Hasonlóan, minden policy fájlból beolvasható kódegységnek megvan a rendszer belső struktúrában tárolt alakja, melyek részletes bemutatása az elvégzett munka ismertetése szempontjából nem releváns[6].

A modulok által nyilvántartott adatfolyam típusa a policy (vagyis, ahol a 18. ábrán *'a* állt, oda jön a policy típus), ezért a későbbiekben erre policy streamként is fogunk hivatkozni. Az OCaml-ben megírt, dinamikus policy-t vezérlő Frenetic modulokkal szemben elvárás, hogy legyen egy olyan függvényük, amely visszaadja a modul policy streamjét, becsomagolva *NetCore_Stream.t* típusba.

A NetCore_Stream tulajdonképpen egy becsomagoló (warpper) modul az Lwt_stream modulhoz, mely segítségével az adatfolyamhoz kapcsolhatjuk a Frenetic belső eseménykezelőit (például azt, amelyik figyeli a stream változását és fordítást kezdeményez). További előny a NetCore_Stream becsomagoló használatából az, hogy a FreneticDSL kódban minden helyen állhat dinamikus policy, ahol statikus policy is. A Frenetic készítői implementáltak a két adatfolyam típus között konvertáló függvényt. Egyetlen lényeges különbség köztük, hogy ha egy modulnak csak a NetCore_Streamjét érjük el, akkor nem tudunk a policy streambe adatot küldeni, mert nincs *push* függvényünk az Lwt_streamhez.[7] Ez

---

[5] Természetesen az „else ágon" *pass* policy megadásával gyakorlatilag az ág elhagyását érjük el.

[6] Frenetic forráskódjának NetCore_Types.ml fájljában viszonylag könnyen olvasható formában megtalálható az irányelvek leírásának teljes belső szintaxisa.

[7] A push függvényt a két streamtípus közötti konverzióval sem kaphatjuk vissza.

biztonsági szempontból egy jó döntés lehet, de legnagyobb hátulütőjére ezen fejezet végén, a példánál kitérünk. A két adatfolyam megkülönböztetése egyszerű: a modulon belül mindig az Lwt_streammel foglalkozunk, modulon kívül pedig mindig a NetCore_Streammel.

A modul policy streamjét visszaadó a függvény segítségével tudjuk összerendelni a modul által megvalósított funkciót valamilyen kifejezéssel, amelyet a policy fájlból beolvasva indíthatjuk a kívánt dinamikus működést. A már említett tanuló switch algoritmus policy fájlból, *learn* kulcsszóval való példányosításakor a *NetCore_MacLearning.make ()* függvény hívódik meg, mely visszatér a kezdeti üres policy streammel.

Tegyük fel, hogy egy switchen kizárólag egy tanuló switch algoritmust futtatunk, melyet egy policy fájlból példányosítottunk, és FreneticDSL-lel fordítottunk le flow táblára. Ilyenkor kezdetben, a switch flow táblájában kizárólag egy flow bejegyzés található, amely minden beérkező csomagot a kontrollernek továbbít. A tanuló switchet implementáló modul megkapja a beérkező csomagot, valamilyen belső struktúrába eltárolja, hogy melyik porton találhatja meg azt a hosztot, amelytől a csomagot kapta. Elhelyezi a policy streamjében a továbbítási irányelvet a megtanult hosztra, majd minden porton kiküldi a csomagot, flow bejegyzés elhelyezése nélkül. Modulok használatánál nem kell expliciten meghívnunk a flow bejegyzést telepítő eljárást, mert ha a policy folyamba elhelyezzük a bejegyzést, akkor a Frenetic észleli a változást, és újrafordítja az policy streamet. A Frenetic mindig csak az utoljára elhelyezett irányelvet veszi figyelembe a fordításnál.

További elvárás a modulokkal szemben, hogy legyen egy csomagkezelő függvényük, mely akkor hívódik meg, ha egy switch a kontrollerhez továbbít egy csomagot, mert neki nem volt ráilleszkedő bejegyzése, vagy egyik flow szabály kifejezetten erre utasította. Az ilyen függvények szignatúrája kötött, bemeneti paraméterei: a küldő switch azonosítója, a switch azon portjának azonosítója, amelyen a csomag érkezett és maga a csomag. A csomagkezelő függvény visszatérési értékének pedig egy csomagtovábbítási műveletnek kell lennie.

4.2 Egy összetett példa elemzése

A modulok, modul adatfolyamok és a fordítás működésének teljes megértése és magyarázatának kiteljesítése érdekében vegyünk egy példát, melyben nem annyira a Frenetic

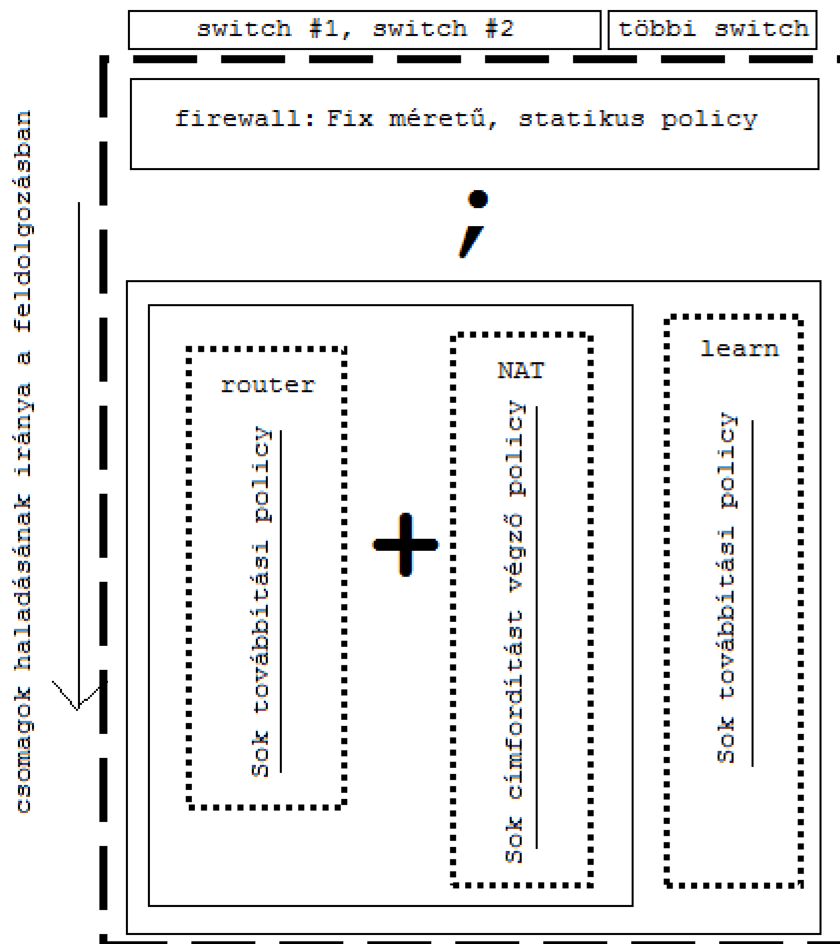
kód jelentése a fontos, hanem a szintaxis és a belső modulkezelés közötti kapcsolat összefüggéseinek illusztrálása.

```
let firewall =
  if srcIP = 10.0.0.1 then drop
  else pass in

let dummy =
  if switch = 1 || switch = 2 then
    router + NAT (publicIP = 10.0.0.2)
  else learn in

firewall; dummy
```

19. ábra: Példakód a modulok közötti kapcsolatok szemléltetéséhez

Legyen a 19. ábrán látható kód egy policy fájl teljes tartalma, melyet a FreneticDSL-lel szeretnénk lefordítani. A *router* modul ismertetésére később részletesen kitérünk, a példában csak annyi a szerepe, hogy továbbítási policy-ját adatfolyamban tárolja. A *NAT* modul egy kezdetleges verziója már használható a Freneticben, címfordítást megvalósító irányelveit policy streamben tárolja, pontos működése most ennek sem lényeges. A fájl beolvasása, és a modulok adatfolyamainak létrehozása után a Frenetic egy nagy, konstans szerkezetű policy folyammá olvasztja össze a struktúrát, melyet nevezhetünk fő adatfolyamnak is.

A 20. ábrán blokk diagrammal ábrázolva láthatjuk, hogy a példakódunkat hogyan ábrázolja a Frenetic belső szerkezetében.



20. ábra: A policy-k belső tárolásának szerkezete

A *firewall* policy minden switchre érvényes, mivel a hozzá rendelt feltétel nem tartalmaz *switch* predikátumot. A feltétel nem teljesülése esetén, a csomagok továbbadásra kerülnek a policy csővezetékben (pipeline) a következő policynak, ahol switch azonosítók szerint kerülnek szétválasztásra.

Az 1-es és 2-es switchen router és NAT funkciót futtatunk párhuzamosan, vagyis a feldolgozás alatt álló csomag duplikálódik, és mindkét policy kap egy-egy csomagot további feldolgozásra. A többi switchen a már említett tanuló switch algoritmus fut. Mindhárom modul használ belső szerkezetében Lwt_streamet, melyet a modul példányosításakor NetCore_Streamre konvertálva ad vissza.

A pontozott vonallal keretezett adatfolyamok mérete futás közben változik. Ha valamelyik modul beérkező csomagot kap, aminek feldolgozása után új policy-t helyez el a policy streamben, majd hoz valamilyen továbbítási döntést. Ha bármelyik modul végez ilyen

műveletet, a rendszer érzékeli a változtatást és újrafordítja a hálózat teljes irányelvét. Erre azért van szükség, mert például a router modul adatfolyamába elhelyezett policy, a flow táblában változáshoz vezethet a NAT funkcióval való párhuzamos szemantika megvalósítása miatt több helyen; és a flow táblában alacsonyabb prioritással elhelyezkedő szabályok prioritásait is lehet, hogy módosítani kell. Így minden switchhez új flow tábla generálódik a frissített irányelvhez illeszkedve.

A 20. ábrán szaggatott vonallal keretezett rész is tulajdonképpen egy adatfolyam, mert fordításnál ennek típusa NetCore_Stream (vagyis a függvény, amely a fájl egészét fordítja NetCore_Stream típussal tér vissza). Ahogy már korábban említettem, ehhez a típushoz nem tartozik *push* függvény, nincs mögötte egy modul, amely a belső Lwt_stream struktúra módosításán keresztül tetszőlegesen formálhatja az adatfolyam felépítését. Így a beolvasott policy fájlból képzett belső szerkezet (a kékkel keretezett rész) állandó, viselkedését csak közvetetten és korlátozottan tudjuk módosítani a modulok (learn, router, NAT) adatfolyamainak változtatásán keresztül.[8]

Tehát a Frenetic jelenlegi verziójában csak korlátozott lehetőség van a hálózat magas szintű, dinamikus működésének biztosítására. A korlátozottság nem csak a fő policy stream szerkezetének állandóságából fakad, hiszen arra sincs lehetőség, hogy egy modul működését befolyásoljuk futás közben. Például, ha egy adott pillanatban, valami külső esemény hatására szeretnénk a tanuló switch által megjegyzett switch port – MAC cím összerendeléseket elfelejtetni, akkor azt szintén csak a rendszer újraindításával tehetjük meg. Tehát a dinamikus viselkedés a modulok szintjén is korlátozott.

Ahhoz, hogy a Freneticet felhasználhassuk a szolgáltatás szintű hálózatprogramozó rendszer támogatásához, olyan modulokat lenne szükséges készíteni, amelyek lehetővé teszik a működés közbeni konfiguráció módosítást. Egy fundamentális, dinamikusan konfigurálható kapcsolatteremtő szolgáltatáshoz szükségünk lenne egy router modulra.

A dolgozat hátralevő részében a router modul modulszintű dinamikus viselkedésének megvalósítását szeretném bemutatni, az ehhez szükséges eszközöket és javasolt módszereket teszteljük és értékeljük ki.

---

[8] A policy fájl szerkezetének manipulálása, és a rendszer újraindítása után természetesen megváltoztatható a fő adatfolyam szerkezete is, de ez nem tűnik egy kivitelezhető megoldásnak nagy gyakorisággal változó környezetben.

5 Kiterjesztések a Frenetic rendszerhez

A Frenetic rendszer legfőbb hiányossága tehát, a dinamikus viselkedés területén fedezhető fel, amely alkalmazási lehetőségeit lényegesen lecsökkenti. Ennek kijavítására első lépésként a modul szintű dinamikus viselkedést tűztük ki célul. Vagyis azt szeretnénk elérni, hogy a rendszer teljes újraindítása nélkül, egy modul működését befolyásolni tudjuk valamilyen külső esemény hatására. Ennek illusztrálására előrevetítenék egy példát.

Tegyük fel, hogy egy router[9] funkcionalitást megvalósító modult példányosítottunk egy hálózati szolgáltatásokat magas szinten leíró rendszer által generált policy fájlból. A routing tábla kezdeti felkonfigurálása megtörtént valamilyen módon. Azonban ekkor, ha valami változás történik a hálózat topológiájában, amelynek következményeként meg kellene változtatni a routing tábla bejegyzéseit (valamelyik linken küldött forgalmat másfelé kellene küldeni), ezt csak a rendszer újraindításával tudnánk megtenni.

A teljes rendszer újraindításának kikerüléséhez tehát első sorban szükségünk van folyamatok közötti kommunikációra.

5.1 Folyamatok közötti kommunikáció

Az OCaml által biztosított Unix modul segítségével tudunk Unix domain socket alapú kommunikációt megvalósítani egy operációs rendszer két független folyamata között. A Unix domain socket vagy IPC (Inter Process Communication) socket által biztosított API (Application Programming Interface) hasonló a TCP socketéhez, de a kommunikáció hálózati protokollt nem használ, az információcsere a Linux kernelen keresztül zajlik. A két folyamat közötti kommunikáció lényegében egy mindkét programból elérhető fájlon keresztül valósul meg, így a Unix socket címe a fájlrendszer egy adott végpontja.

A Unix socketet megvalósító kód nem az én munkám terméke, egy kollégám, Vaszkun Gábor készítette, akivel közösen dolgoztunk a Frenetic megismerésén és kiterjesztésén. A megvalósítás leírása Gábor szakdolgozatának részét fogja képezni, így hivatalos

[9] Az ISO OSI referenciamodell szerinti L3 (hálózati) rétegbeli útvonalválasztó eszköz.

dokumentáció, vagy leírás még nem készült róla. A program használatát csak felhasználói oldalról mutatom be dolgozatomban.

Nevezzük a Unix socket szerver oldalának a Frenetic rendszer forráskódjának azon kiegészítését, amely létrehozza a socketet, és fogadja a külső programtól érkező adatokat. A kliens csatlakozik ehhez a sockethez, és a biztosított CLI-n (Command Line Interface) keresztül adott parancsok hatására JSON [20] struktúrák beolvasására képes a fájlrendszerből, melyeket a socketen keresztül el tud juttatni a szervernek. A szerver a kapott struktúrát beolvassa, és lehetőséget nyújt arra, hogy a Frenetic moduljainak függvényeit hívjuk.

A Unix socket segítségével megvalósul az az elvárás, hogy külső események bekövetkezésekor tudjunk kommunikálni a Frenetic moduljaival. Visszatérve a router modulos példánkhoz, most már lehetőségünk van a modulnak információt átadni a routing tábla futás közbeni újrakonfigurálásához.

Tehát a socket szerver által jól meghatározott formátumnak megfelelő JSON fájlokat szerkeszthetjük, majd a parancssori interfész segítségével meghatározhatjuk a fájlokban megadott konfiguráció elküldésének pillanatát.

5.2 Router modul

Jelenleg a Frenetic által biztosított policy-k segítségével csak tanuló switcheken keresztül tudunk egy több switchből álló topológiában teljes kapcsolatot létesíteni anélkül, hogy az irányelvek definiálása közben ne kelljen foglalkoznunk a hoszt – switch port összerendelésekkel. Ez meglehetősen nagy hiányosság, mivel így nem tudunk a hosztok között az OSI referenciamodell szerinti harmadik rétegben kapcsolatot létesíteni. Vagyis a hálózatunkban minden hosztnak azonos alhálózatból kell IP címet kapnia, arról nem is beszélve, hogy a hosztok címzése így nem hierarchikus. Ezért szükséges egy alap router funkciókat implementáló Frenetic modul, mely képes a hálózati forgalmat változtatható konfigurációval az IP címek alapján irányítani. Egy router modul nélkülözhetetlen eleme lenne a szolgáltatásokkal operáló, magas absztrakciós szintű hálózat programozási nyelvnek.

Ennek megvalósításához OCaml-ben kell elkészítettem egy olyan Frenetic modult, amely megfelel az előző fejezetben bemutatott modul felépítésnek. Vagyis belső szerkezetében nyilván tart egy Lwt_stream típusú modult, melybe dinamikusan helyezhetünk el új továbbítási

irányelveket. Továbbá rendelkezik egy olyan fő függvénnyel, amely a bejövő csomagok eseménykezelőjeként funkcionál, vagyis minden bemeneti switch – port – csomag hármashoz rendel egy irányelvet, amihez igazodva a rendszernek kezelnie kell a hasonló csomagokat. Utolsó fő követelményünk pedig, hogy rendelkezzen egy olyan függvénnyel, amellyel a Frenetic rendszer példányosítani tudja a modult, ha egy policy fájlban a modulhoz rendelt karaktersorozatot találja.

5.2.1 Útvonalválasztó tábla

A routerhez első sorban implementáljunk egy útvonalválasztó táblát megvalósító almodult, amely képes eltárolni a routerekhez rendelt táblabejegyzéseket, és megvalósítja a rajtuk értelmezett műveleteket. Ehhez az OCaml nyelv lehetőséget nyújt az objektum orientált nyelvekben „osztály”-nak nevezett egység definiálásával. Egyébként az OCaml nyelv ezt modulnak hívja, ami nem összekeverendő a Frenetic modul fogalmával.

Mivel egy Frenetic modul több switchet is kiszolgálhat egyszerre (mint például a 20. ábrán a *learn* modul), az útvonalválasztó tábla implementálásánál szem előtt kell tartanunk, hogy minden táblabejegyzés mellé el kell tárolnunk a switch azonosítóját, amelyikre vonatkozik. Tehát egy bejegyzést az alábbi öt adatmező együttesével határozhatjuk meg:

- A cél alhálózat IP címe (pl.: 192.168.0.0).
- A cél alhálózathoz tartozó alhálózati maszk (ez egy 192.168.0.0/N-es alhálózat esetén olyan IP cím, amelynek első N bitje 1 értékű, a többi 0).
- A switch azonosítója, amelyre az útvonalválasztó táblabejegyzés vonatkozik.
- A switchport azonosítója, amelyen az illeszkedő csomagok továbbításra fognak kerülni.
- Valamint az útvonalválasztásról további döntéseket hozó „next hop” (vagyis következő) router IP címe.

A hálózati címek, illetve maszkok mindegyike egy 32 bites szám, mivel a Frenetic nem biztosít külön struktúrát ennek a gyakran előforduló számpárnak. Emiatt az IP címeket minden esetben pontos címeknek kell kezelnünk, és nem szabad egy címre egymagában alhálózati címként néznünk akkor sem, ha 0-ás bitekre végződik. Ez első hallásra talán nem jelent nagy akadályt, de emiatt nem tudunk a Frenetic jelenlegi állapotában IP címek segítségével alhálózatokra irányelveket megfogalmazni, ami komoly hiányosság. A switch azonosítója egy

64 bites szám, ez megegyezik azzal az azonosítóval, amelyre a policy fájlok *switch* predikátumában hivatkoztunk. A switchport azonosítója egy 16 bites szám, ez pedig azzal az értékkel egyezik meg, amelyre a továbbítási irányelvek írásánál az *fwd* policy paraméterében adtam meg. A „next hop" router mező értelmezésére a későbbiekben térek ki.

Az öt mezőből álló táblabejegyzést OCaml-ben definiálhatjuk egy különálló struktúra típusként, melyből egy listát alkotva kész az útvonalválasztó tábla[10]. A táblán értelmezett műveletek mindegyikét egy külön függvénnyel valósítottam meg. Első sorban szükségünk van táblabejegyzés hozzáadó és törlő függvényekre, melyek a kényelmesség kedvéért karakterláncként (*string*) várják bemeneti paramétereiket, elvégzik a szükséges konverziókat, valamint kívülről hívhatóak, hogy a Unix socketen keresztül adatot tudjunk juttatni a routing táblába.

Az útvonalválasztó táblán értelmezett legfontosabb funkció, hogy el tudja végezni a routerek útvonalválasztó működését leíró szabványokban definiált „longest prefix match"-et. Ez a függvény bemeneti paraméterként kap egy switch azonosítót és egy IP címet, melyek alapján eldönti, hogy melyik az a táblabejegyzés, amely a leghosszabb prefixszel illeszkedik a kapott IP címre. Ehhez végigmegy a modul routing táblájában és kiválasztja azokat a bejegyzéseket, amelyek az adott switchre vonatkoznak, majd mindegyikkel végrehajtja az alábbi lépéseket:

- Kezdetben jegyezzük meg a -1-et mint leghosszabb prefix illeszkedést.
- A táblabejegyzés struktúra cél IP címének veszi a bitenkénti konjunkcióját (ÉS-elését)[11] az alhálózati maszkkal.
- A paraméterként kapott IP címnek is veszi a bitenkénti konjunkcióját a táblabejegyzés alhálózati maszkjával.
- Amennyiben az ÉS-elésekből kapott két szám azonos, az aktuális táblabejegyzés illeszkedik a kapott IP címre, az alhálózati maszk hosszával azonos számú biten.
    - Ha ez hosszabb, mint az eddigi leghosszabb illeszkedés, akkor jegyezzük meg az aktuális táblabejegyzést, és lépjünk a következő bejegyzésre.

---

[10] A tábla megvalósítása hatékonyabb lenne, ha egy hash tábla segítségével szétdarabolnánk switchenként az útvonalválasztó táblát. Azaz a hash táblában switch azonosító lenne a kulcs, az érték pedig az adott switch routing táblája.

[11] A logikai AND művelet: *a* AND *b* = 1, akkor és csak akkor, ha *a* és *b* is 1.

    - Ha nem hosszabb, akkor lépjünk a következő táblabejegyzésre.
- Ha az ÉS-elésekből kapott két szám nem azonos, akkor az aktuális táblabejegyzés nem illeszkedik a kapott IP címre, lépjünk tovább a következő táblabejegyzésre.
- Amennyiben a tábla végére értünk, az utoljára megjegyzett táblabejegyzésre volt a leghosszabb prefix illeszkedés.

5.2.2 MAC címek kezelése

A csomagtovábbító irányelv összeállítása előtt szükségünk van a cél eszköz MAC címére, amely azonosítja az alhálózatban. Ehhez létre kell hoznunk egy ARP (Address Resolution Protocol) tárolót (ARP cache), amely IP cím – MAC cím összerendeléseket tartalmaz. Kezdetben ezt valósítsuk meg statikusan felkonfigurálhatóra kívülről hívható függvények segítségével, melyet majd a későbbi funkciók bevezetésével kiterjesztünk. A router modulban az ARP tárolót az OCaml által biztosított hash tábla segítségével valósítjuk meg, melyben switch azonosító – IP cím párosokhoz, mint kulcshoz rendelünk MAC címet.

A FreneticDSL-ben létezik olyan policy, amely támogatja csomagok MAC címeinek átírását. De sajnos ezt csak egy komoly korláttal tudjuk megtenni: tudnunk kell, hogy mi volt a csomag eddigi MAC címe. A policy fájlból beolvasható szintaxis *dlDst <eddigi MAC-cím>* → *<új MAC cím>* alakú, vagyis a régi és az új MAC cím helyére is pontos 48 bites MAC címet kell írnunk. Annak illusztrálására, hogy ez miért okoz gondot a router modul implementációjával kapcsolatban, vegyünk egy példát.

Legyen egy *R* router egyik interfészére csatlakoztatva a *H* hoszt, a többire pedig tetszőleges más eszközök (más routerek vagy hosztok). A valódi routerek minden interfészének különböző MAC címe van. Amikor egy bejövő csomag érkezik *R*-hez, amely a *H* hoszt IP címével van megcímezve célként, *R*-nek ki kell keresnie az ARP tárolójából a *H* IP címéhez rendelt MAC címet, majd át kell írnia a bejövő csomag MAC címét a *H* hoszt MAC címére. Ezt a MAC cím átírást egy policy-val szeretnénk megvalósítani. De mivel a bejövő csomag módosítandó cél MAC címe a router bármelyik interfészének (kivétel azét, amelyre *H* van csatlakoztatva) MAC címét felveheti, annyi *H*-hoz továbbító policy-t kellene felvennünk, ahány interfészre érkezhet a csomag. Ez a gyakorlatban nem kivitelezhető megoldás, mert a

router szerepét játszó OpenFlow switch flow táblája túl gyorsan növekedne a routerre kapcsolt hosztok számával.

A FreneticDSL-ben hasonló policy-kkal, és hasonló korlátokkal tudjuk módosítani a továbbítandó csomagok forrás MAC címét, valamint forrás és cél IP címét, illetve transzport rétegbeli portját, továbbá VLAN azonosítóját. Dokumentáció hiányában nem találtam ezen implementációs döntésre magyarázatot, de a néhány példakódban és a rendszer forráskódjában ehhez igazodva használják a felsorolt funkciókat. Ez mindenképpen a kezdeti Frenetic rendszer rossz tulajdonságai közé sorolható, mert kényelmetlen és rosszul skálázódó megoldás, melyre egyelőre nincs a rendszerben alternatíva.

A Frenetic további elemzése, és a router modul folytatása érdekében kényszermegoldást kellett találnom a problémára: legyen az összes router, összes interfészének MAC címe azonos. Ez a konvenció azért elfogadható, mert az esetek nagy részében transzparensen képes biztosítani az L3-beli kapcsolatot a hosztok között. Így ha érkezik a routerhez egy bejövő csomag, akkor az biztosan erre a közös MAC címre fog érkezni, amelyre már elhelyezhetjük a Frenetic által biztosított cél MAC cím módosító policy-t a router modul policy streamjében. Ezzel a konvencióval a routerek abban az esetben biztosan nem fognak működni, amikor routereket tanuló switchcsel szeretnénk összekötni. Ugyanis ekkor a tanuló algoritmus megtanulja a közös MAC címet úgy, mintha az csak egy adott porton helyezkedne el. Vagyis ebben az alhálózatban a MAC cím már nem azonosítja a router interfészeket egyértelműen, így a kommunikáció nem lesz sikeres a routerek között. Egy ilyen topológiának gyakorlati megfontolásokból fontos szerepe lehet, de ettől most tekintsünk el.

További kérdés a MAC címek kezelésével kapcsolatban a router által fogadott, és indított ARP kérések, illetve válaszok megvalósítása. A router modul implementálásának első fázisában tekinthetünk minden hosztra és routerre úgy, hogy ARP tábláik teljesen fel vannak töltve. Vagyis, hogy miden IP cím – MAC cím összerendelést ismernek, amelyre szükségük lehet. Ezt azért tehetjük meg, mert a kezdeti konfiguráció megadásakor a hosztok ARP tábláit fel tudjuk tölteni a teszteléshez használt Mininet szkriptből, valamint a routerek ARP tábláit a router modul általam megírt, ARP tároló felkonfiguráló függvényeivel.

5.2.3 A router működése

A MAC címekre, illetve az ARP kezelésre vonatkozó szükséges konvenciók bevezetése után vissza kell térnünk az útvonalválasztó tábla bejegyzéseinek „next hop" adatmezőjére. Valódi routerekben ez a következő router azon távoli interfészének IP címe, amely azzal a porttal van azonos alhálózatban, amelyen a csomag továbbításra kerül. Így ilyenkor ebben az esetben is történhet ARP kérés – válasz a távoli IP cím feloldására. Mivel konvenciónk szerint minden router interfész MAC címe azonos a „next hop"-ra nincs szükség ebben a formájában. A router modul a „next hop" címet arra használja, hogy a routing tábla bejegyzések célját meg tudja különböztetni aszerint, hogy az illeszkedő csomagot egy másik routerhez kell irányítani, vagy közvetlenül a hoszthoz.[12] Tehát ha a „next hop" cím 0.0.0.0, akkor a hoszt „on link", vagyis közvetlenül csatlakoztatott, ellenkező esetben pedig még legalább egy router áll a hoszt előtt a csomag útjába.

Tehát összességében elmondhatjuk, hogy a routerek interfészeinek valójában nincs saját IP címük. Azonban amikor hosztok egy csoportját kötjük a router egy interfészére, esetleg tanuló switchen keresztül, akkor a hosztok konfigurációjában szerepelnie kell az alapértelmezett átjáró (default gateway) IP címének, ahova az alhálózaton kívülre irányuló forgalmat kell küldeni. Ez általánosan elfogadott konvenció szerint, az adott alhálózat legmagasabb kiosztható IP címe szokott lenni. Tehát a hosztokat felkonfigurálva egy alapértelmezett átjáró IP címmel, és a címhez tartozó ARP bejegyzéssel, a hosztok által generált forgalom teljes része a közös MAC címre címezve megy a routerekhez.

Itt érdemes azt is megemlíteni, hogy ha meg szeretnénk valósítani, hogy a router minden interfésze saját konfigurációval rendelkezzen, ezt szintén kontroller oldali adminisztrációval kell megtennünk. Mivel erre a jelen körülmények között nem volt szükség, ennek megvalósítását működése során a modul nem használja. A router modulban van az ARP tárolóhoz hasonlóan egy hash tábla, amelynek kulcsa switch azonosító – switchport azonosító adatpáros, melyekhez IP címet és alhálózati maszkot rendelünk. A router interfészeinek felkonfigurálására az útvonalválasztó táblához hasonló módon a Unix socketen keresztül van lehetőség, melyhez a szükséges függvényt elkészítettem, de végül nem használtam.

[12] Valódi routerben is a „next hop" látja el ezt a funkciót, csak ott a következő router címének hiánya jelenti, hogy a hoszt „on link". Azonban az ilyen információt a valódi routerek automatikusan helyezik továbbítási tábláikban.

Felmerülhet a kérdés, hogy egy ilyen megszorításokkal implementált router hogy tud együttműködni más fizikai vagy szoftveres eszközökkel. Itt gondolhatunk hardveres switchre, routerre, vagy akár a Freneticben létező tanuló switchre.

Például ha egy router modult futtató OpenFlow switchet össze szeretnénk kötni egy valódi routerrel, probléma lehet abból, hogy ennek az interfésze minden bizonnyal nem a Frenetic routerek közös MAC címét fogja használni. Az együttműködési probléma megoldásához mindenképpen implementálnunk kellene az interfész konfigurációk kezelését, valamint azt is, hogy az útvonalválasztó tábla „next hop” IP címe valóban a következő router interfészére mutasson. Ekkor a Frenetic router indítana ARP kérést a valódi router címének feloldásához, és hasonlóan, visszafelé jövő forgalom esetén, a valódi router ARP kérésére a Frenetic router tudna ARP választ küldeni a port konfigurációja alapján.

Azonban a *learn* kulcsszóval, Frenetic policy fájlokban példányosítható, tanuló switch modullal az együttműködés jelen állapotában is megoldott. Azaz a router egy interfészére tanuló switchet köthetünk, amelyen keresztül több hoszt kommunikálhat az alhálózaton kívülre, a router adott interfészén. Mivel a router modul a MAC cím módosító Frenetic policy körülményessége miatt nem változtatja meg a routertől küldött csomagok forrás MAC címét, a tanuló switch nem tudja megtanulni, hogy melyik interfészén található a router. Ugyanis a tanuló algoritmusban a switch csak a forrás MAC cím alapján tudja a hosztokat interfészekhez rendelni. Ez azonban nem rontja el a kommunikációt. Amikor a hosztok csomagot szeretnének küldeni a routeren keresztül, akkor a tanuló switch minden alkalommal úgy kezeli a csomagot, mintha még nem látott volna a Frenetic routerek közös MAC címére címzett csomagot. Tehát a csomagok minden esetben megtalálják a router interfészét, de sok felesleges forgalom is generálódik a floodolt üzenetek miatt, szükségtelenül terhelve a hálózatot. Ennek a teljesítménybeli problémának a kijavítását akkor tudnánk megtenni, ha a forrás MAC cím módosító policy nem igényelné a csomag eddigi forrás MAC címének értékét.

Mint már korábban említettem az *Útvonalválasztó tábla* című fejezetben, a Frenetic nem nyújt támogatást alhálózati címek kezelésére, vagyis a rendszerben az IP cím típus mindig egy pontosan meghatározott IP címet jelent. Az útvonalválasztó tábla implementálásánál az ebből adódó problémát meg tudtuk kerülni azzal, hogy bevezettünk egy struktúrát, ami a maszkot és az IP címet közösen tárolja.

Az OpenFlow szabvány szerint a switchek flow tábláiban lehetnek csak részlegesen illeszkedő IP címek, melyek segítségével tudnánk alhálózatokra vonatkozóan tömören

megfogalmazni továbbítási szabályokat. De mivel a flow táblákat csak a Frenetic irányelv leíró nyelvén keresztül tudjuk változtatni, a router modul által generált továbbítási szabályokat csak pontos IP címekre tudjuk megfogalmazni. Tehát például, ha a 10.0.0.0/8-as alhálózatba szeretnénk adatot továbbítani a 10.0.0.1-es és 10.0.0.2-es hosztoktól, akkor a routerben mindkét hosztra külön továbbítási szabályt kell írnunk az OpenFlow switch flow táblájában. A Frenetic további fejlesztésében erre mindenképp figyelmet kellene fordítani, hogy legyen lehetőség „wildcard" IP (illetve MAC és egyéb) címek megadására a policy fájlban.

A működés bemutatásának zárásául, nézzük végig nagy vonalakban, a router modul üzemi stádiumait, és néhány helyen pontosítsuk a korábbiakat:

- Policy fájlból a *router* kulcsszóval tudunk policy streamet létrehozni a router modulban, melyet *switch* predikátumokkal tudunk switchek egy csoportjához rendelni.
- A Unix socketen keresztül eljuttatjuk az útvonalválasztó tábla konfigurációját a Frenetichez.
- Kezdetben a routert megvalósító OpenFlow switch flow táblájában csak egy kontrollernek továbbító flow bejegyzés található.
- A routerhez beérkező csomag eljut a kontroller csomagkezelő függvényéhez, amely a routing táblán elvégzett „longest prefix match" alapján eldönti, hogy merre kell továbbítani a csomagot.
- A kontroller az új csomagtovábbítási irányelvet elhelyezi a router modul policy streajében az adott hosztra.
- A Frenetic rendszer észleli a változást a hálózat működését leíró irányelvekben, mely hatására a teljes policy streamet lefordítja flow bejegyzésekre és újratelepíti a switchekbe.

5.2.4 Konfiguráció frissítése

A router modul tervezésénél kikötöttük, hogy az útvonalválasztó tábla futás közben konfigurálható legyen a Unix socketen keresztül. A routing tábla bejegyzés hozzáadó és törlő függvényeinek a segítségével megoldott, hogy adatot tudjunk juttatni a modulba újraindítás nélkül. Azonban eddig azzal nem foglalkoztunk, hogy mi legyen a konfiguráció frissítése előtt elhelyezett továbbítási szabályokkal a policy streamben. Az útvonalválasztó tábla

bejegyzéseinek megváltoztatásával a policy stream nem változik meg automatikusan. Vagyis lehet a streamben olyan továbbítási szabály, amelyhez tartozó táblabejegyzést már kitöröltünk, vagy egy újonnan elhelyezett táblabejegyzést figyelembe véve a policy stream nem megfelelő továbbítási döntést hozna.

A menet közbeni konfiguráció helyes kivitelezéséhez tehát szükséges, hogy megvalósításra kerüljön valami eljárás, amely fenntartja a konzisztenciát az útvonalválasztó tábla és a policy stream között. Ehhez két módszer került bele a router modul implementációjába.

Elsőnek vegyük az egyszerűbb esetet, ahol minden routing tábla változás hatására visszahelyezzük a kezdeti üres policy streamet, amelyben csak a kontrollerhez továbbító szabály van. Ekkor nyilván konzisztens lesz a switchek flow táblája az útvonalválasztó táblával, mert a konfiguráció frissítés után a switch minden továbbítási döntés előtt konzultálni fog a kontrollerrel. A központi logika pedig már az útvonalválasztó tábla új állapota alapján fogja megalkotni a továbbítási policy-t, mely a streambe helyezés után, a switch flow táblájába is bekerül. Ez a megoldás talán elsőre nem tűnik túl hatékonynak, értékelésére a következő fejezetben térünk ki.

A konzisztencia más fajta fenntartására próbáljuk meg csak azokat a továbbítási irányelveket kitörölni a policy streamből, amelyek hibás továbbítási döntést hoznának az útvonalválasztó tábla új állapotához képest. Ehhez az útvonalválasztó tábla bejegyzéseihez felvettem egy azonosítószámot, amellyel egyszerűen tudjuk egyértelműen azonosítani a bejegyzéseket. Továbbá létrehoztam a router modulban egy *matchedIPs* nevű listát, amely eltárolja, hogy mely switcheken, mely IP címekre, mely azonosítójú útvonalválasztó táblabejegyzés miatt helyeztünk el csomagtovábbítási policy-kat.

A táblabejegyzés törlő függvénnyel megadhatjuk, hogy melyik alhálózatra vonatkozó bejegyzést szeretnénk kivenni az útvonalválasztó táblából. A törlő függvény összegyűjti a törlendő bejegyzések azonosítóit a *todelroute* listába. Azonban annak jelzésére, hogy mikor szükséges a policy stream tisztítási eljárását végrehajtani, egy új eljárást kell felvennünk, amivel a konfiguráció írója utasíthatja a rendszert arra, hogy végezzen konzisztencia vizsgálatot a routing tábla és a policy stream között. Erre azért van szükség, mert konfiguráció megadásakor, és frissítésekor is állhat egymagában a bejegyzés törlő, vagy elhelyező függvény. Így nem tudjuk a kontroller belső állapota alapján megítélni, hogy a költséges stream tisztítási művelet végrehajtása szükséges-e. Megvalósíthatnánk akár egy olyan mohó megoldást, ahol

minden routing tábla konfiguráló függvény hívása esetén lefuttatjuk a tisztítást, de meglehetősen „brute force” megoldás. A konfiguráció írója a Unix socket bemeneteként szolgáló JSON fájl szerkesztésekor a *make-state-consistent* logikai változó beállításával jelezheti, hogy szeretné-e a policy stream tisztítását végrehajtó eljárást futtani.

A policy stream tisztítási folyamat végiglépked a router modul által nyilvántartott, összes továbbítási irányelven, mindegyikre megvizsgálva, hogy szükséges-e a törlése a policy streamből. Jelöljük a policy streamet *S*-sel, és ennek egy elemét (egy egzakt IP címre vonatkozó továbbítási irányelvet) *P*-vel. Ekkor a vizsgálat lépései:

- Az aktuális *P*-ből megkapjuk, hogy melyik switchre és melyik cél IP címre vonatkozó továbbítási irányelvnél tartunk az ellenőrzésben.
- A *todelroute* listában megvannak, hogy mely útvonalválasztó táblabejegyzések kerültek törlésre a konfiguráció frissítés során.
- A *matchedIPs* listában megvan, hogy mely *P*-k, mely routing táblabejegyzések miatt kerültek elhelyezésre a policy streamben.
- *S*-en végighaladva, minden *P*-re megnézzük, hogy töröltük-e azt a routing táblabejegyzést, amely miatt *P* elhelyezésre került *S*-ben.
    - Ha igen, akkor *P*-t töröljük *S*-ből, és lépjünk a következő továbbítási irányelvre.
    - Ha nem, akkor *P* marad a policy streamben, és lépjünk a következő továbbítási bejegyzésre
- Mindezt folytassuk, amíg a végére nem érünk a policy streamnek.

5.3 ARP modul

A router modul, és az azt használó hálózat konfigurációjának elkészítését meglehetősen problémássá teszi, hogy minden routernek és minden hosztnak fel kell töltenünk az ARP tárolóját minden IP cím – MAC cím párosítással, amelyre szüksége lehet. Ennek megoldásához szükséges implementálnunk, hogy a routerek képesek legyenek válaszolni egy ARP kérésre, illetve, hogy tudjanak ARP kérést indítani egy hoszt IP címének feloldására.

Természetesen a modul implementálásával azt szeretnénk elérni, hogy a router működése változatlan legyen, vagyis az ARP modul használata esetén ne kelljen feltölteni a

router ARP tárolóját, és ne kelljen a hosztoknak statikusan megadni az alapértelmezett átjáró MAC címét.

Szerencsére a Frenetic által használt, OCaml-ben implementált *packet* könyvtár támogatást nyújt ARP üzenetek összeállítására. Azaz lehetőségünk van a szükséges adatokból egy konstruktor segítségével előállítani ARP kérést, vagy ARP választ. Ezen csomagok felépítését most nem részletezném.

Az ARP modulnak összesen három függvénye van:

- egy ARP kérést előállító függvény,
- egy csomagkezelő függvény, amely a beérkező kérésekre ARP választ állít össze,
- valamint egy modul létrehozó függvény, amely visszatér egy minden csomagot a kontrollerhez továbbító, konstans policy-vel.

Továbbélve a router MAC címekkel kapcsolatban tett konvenciójával, az ARP modul válaszában mindig egy adott konstans MAC címmel tér vissza, amelyet az ARP modul példányosító függvénye paraméterül kap.

A modul policy fájlból való beolvashatóságát egyelőre nem tettem lehetővé, de ezzel a forráskód könnyen kibővíthető a routernél használt módon. A router kezdeti irányelvéhez a Frenetic által biztosított párhuzamosítás operátor (+) segítségével kapcsolhatjuk hozzá az ARP modult példányosító függvény visszatérési értékét. Így minden router képes válaszolni a hozzájuk érkező ARP kérésekre.

5.4 DHCP szerver modul

Következő kényelmetlenég, amelyet ki szeretnénk küszöbölni a router modul használatával kapcsolatban, az a hosztok felkonfigurálása, mely lehetővé teszi számukra az L3-beli kommunikációt a routereken keresztül. Az SDN biztosította lehetőségeket kihasználva, DHCP szervert egyszerűen tudunk egy kontroller alkalmazásként készíteni, amely minden switchhez beérkező DHCP felderítő (discovery) üzenetet lekezel az adatsík további terhelése nélkül.

A DHCP szerver felkonfigurálására szintén a Unix socketen keresztül teremtettem lehetőséget. A socket által használt JSON fájlokban, jól meghatározott struktúrában megadhatjuk, hogy az adott switch melyik portjához, milyen alhálózatot rendeljünk, és hogy mi legyen az alhálózat alapértelmezett átjárójának IP címe. Az alhálózatot itt is két IP címmel (az alhálózati IP címmel, és a hozzátartozó maszkkal) adhatjuk meg, melyből a DHCP szerver kiválaszthatja az azon a switchporton lévő hosztok IP címeit.

A JSON fájl feldolgozásakor meghívódik a megfelelő függvény, amely a DHCP modul hash tábláját feltölti a megadott konfigurációval. A hash táblában switch azonosító – switchport azonosító párokhoz, mint kulcshoz rendelünk alhálózat struktúrákat, amely tartalmazza az IP címet, maszkot, és az alapértelmezett átjáró IP címét.

A DHCP működés rövid ismertetése érdekében nézzük át a szerver és a kliens által küldött üzeneteket:

- A DHCP kliens (hoszt) periodikusan küld felderítő (discovery) üzeneteket broadcast IP és MAC címre küldve, melyben jelzi az igényét a dinamikus felkonfigurálásra.
- Egy vagy több DHCP szerver ezt megkapja, és küld egy ajánlott (offer) IP konfigurációt a kliensnek.
- A DHCP kliens csak annak a szervernek küld kérést (request), amelytől el szeretné fogadni a konfigurációt.
- A szerver erre egy nyugtával (acknowledge) válaszol, mely csak a DHCP üzenettípust azonosító bájtban tér el a DHCP offertől. Ezzel jelezve, hogy a konfiguráció még mindig szabad, és a kliens használhatja.

A modulhoz beérkező DHCP kérések *A* forrás MAC címe alapján állítjuk elő a hoszt adott alhálózatbeli IP címét. Az alhálózat IP címe legyen *I*, maszkja pedig *M*. Ekkor a hoszt *H* IP címe a 21. ábrán látható:

$$H = (I \,\&\, M) + (A \; mod \; \overline{M})$$

21. ábra: A DHCP szerver IP kiosztása MAC cím alapján

Ahol „*&*” bitenkénti ÉS-elést, „*mod*” maradékos osztást, a felülvonás pedig bitenkénti negálást jelent.

Az egyszerű DHCP szerver implementálásával az okozta a legtöbb problémát, hogy a Frenetic által használ *packet* könyvtár nem ismeri a DHCP protokollt, sőt az UDP fejléc[13] elkészítésében sem tudtam hasznát venni. Így a szerver által kiküldendő két csomagot binárisan építettem fel. Azaz a csomag megfelelő mezőibe beillesztettem a DHCP modulom által küldendő konfigurációt. Ezek a mezők, a teljesség igénye nélkül: cél MAC, IP ellenőrző összeg, DHCP szerver IP címe, a hosztnak kiosztott IP cím és maszk, alapértelmezett átjáró címe, valamint a DHCP üzenettípus.

A DHCP modul fájlból beolvashatóságát megvalósítottam, így a DHCP funkciót igénylő switchek működését leíró irányelvek mellé a párhuzamosítás operátorral kapcsolhatjuk a DHCP policy-t. Ennek szintaktikája *DHCP (IP)*, ahol *IP*, a DHCP szerver IP címe, amelyre a hosztok a DHCP kéréseket (request) küldhetik.

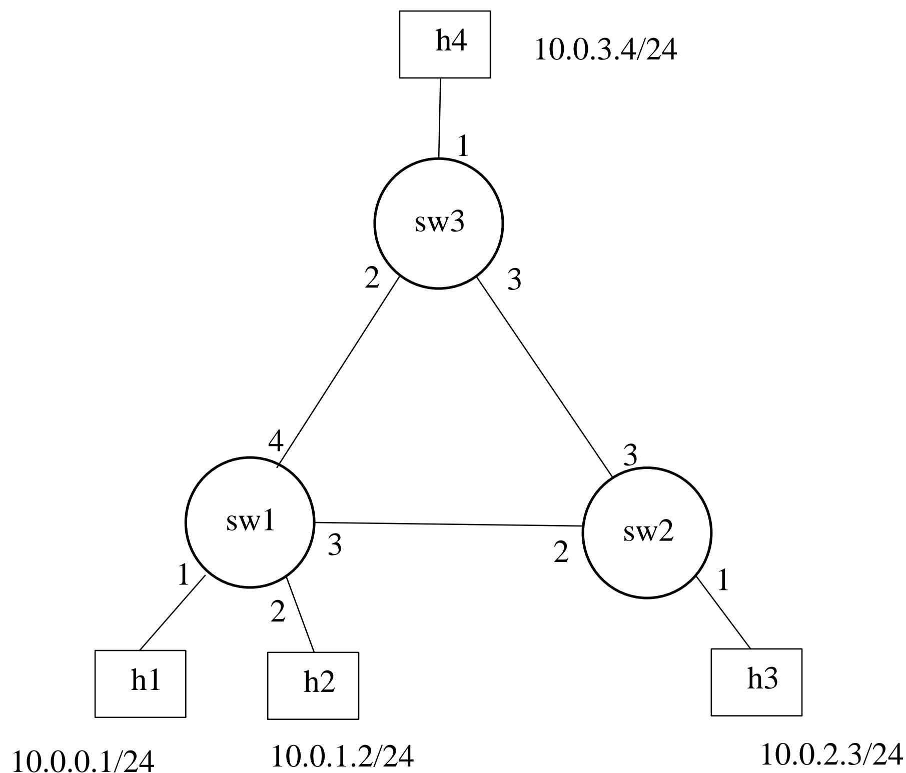
[13] A DHCP protokoll az UDP transzport rétegbeli protokoll felett működik.

6 Tesztelés

A Frenetic elemzésének kiteljesítéséhez, és az implementált modulok értékeléséhez szükséges néhány mérést elvégeznünk a rendszeren. Ehhez az alábbi kérdésekre keressük a válaszokat:

- Mennyi időbe tart a rendszernek lefordítani az irányelveket, és ez hogyan skálázódik a policy stream növekedésével? Mi az a maximális méret, amit még kezelni tud?
- Mennyi időt töltünk a policy stream javított tisztítási folyamatával?
- Hogy befolyásolja a hálózat késleltetését a kétféle konfiguráció frissítési eljárás?
- Hol nyerünk a javított stream tisztítási algoritmussal?
- Hogy reagál a TCP protokoll az újrakonfigurálásra?

A tesztek elvégzéséhez egy Mininetben kialakított hálózati topológiát használtam, melyet a Mininet API felhasználásával elkészített Python szkript formájában tudunk elindítani. A hálózat egy egyszerű háromszög topológia négy hoszttal, melyet a 22. ábrán láthatunk.



22. ábra: A teszteléshez használt Mininet topológia.

A 22. ábrán egyúttal láthatjuk a hosztok, illetve switchek neveit, melyek alapján a későbbiekben hivatkozni fogunk rájuk. Az ábrán a switchek port azonosítóit is feltűntettem, melyek szintén állandóak lesznek a tesztekben. Az *sw1*, *sw2*, *sw3* switchek azonosítói rendre 1, 2, 3, a hosztok MAC címei a Mininet által automatikusan hozzájuk rendelt értékek, melyek csupa 0-ák, csak az utolsó hexadecimális számjegyeikben térnek el, amelyek a hoszt nevében a „h” utáni számmal azonosak. Mindegyik link maximális adatátviteli sebessége 10Mbit/s, és késleltetéseik 5 ms-osak az egyik irányba, tehát az RTT-hez (Round Trip Time) mindegyik 10 ms-ot ad hozzá.

A switchek mindegyikét egy router modul szolgálja ki, amely policy streamjében mindháromra vonatkozóan megtalálhatóak a továbbítási irányelvek. Mindegyik routeren fut az ARP modul, és a hosztok által indított DHCP szerver felderítő üzenetek egyenesen a kontrollerhez kerülnek továbbításra, amint elérték az egyik switchet. A DHCP szerver a 22. ábrán látható alhálózatokat rendeli a switchek egyes portjaihoz, melyekből a hosztok MAC címeik alapján, a kiosztott IP címek a DHCP szerver működésének megfelelőek.

A hálózat működését leíró JSON fájl tartalma a 23. ábrán látható. A Unix socket szerver oldalán a policy JSON tömb elemei összefűzésre kerülnek, melyet a Frenetic rendszer megkap lefordításra és a switchek felkonfigurálására.

```
{
        "policy": ["let pol = ",
                   "if switch = 1 || switch = 2 || switch = 3 then router",
                   "else pass in",
                   "DHCP ( 10.10.10.10 ) + pol"]
}
```

23. ábra: A hálózatot irányelvét deifniáló JSON fájl

A routerek felkonfigurálását JSON fájlban írhatjuk le, melyet a rendszer a Unix socketen keresztül kap meg. A kezdeti konfiguráció szerint a hosztok a legkevesebb routert érintő útvonalon érik el egymást. Egy külön fájlban készítettem egy konfiguráció frissítést, mely úgy őrzi meg a hosztok közötti teljes kapcsolatot, hogy az *sw1* és *sw2* routerek között linket nem használja. A router modulok, illetve DHCP szerver felkonfigurálását elvégző JSON fájlt nem mutatnám be, terjedelmességére és irrelevanciájára való tekintettel.

A tesztelések során a címkiosztást és az irányelveket is beleértve, ezt a konfigurációt használtam. Ahol nem emelek ki a teszt szükségelte, külön módosítást, ott ez a konfiguráció érvényes.

6.1 Fordítási és konfigurálási idő

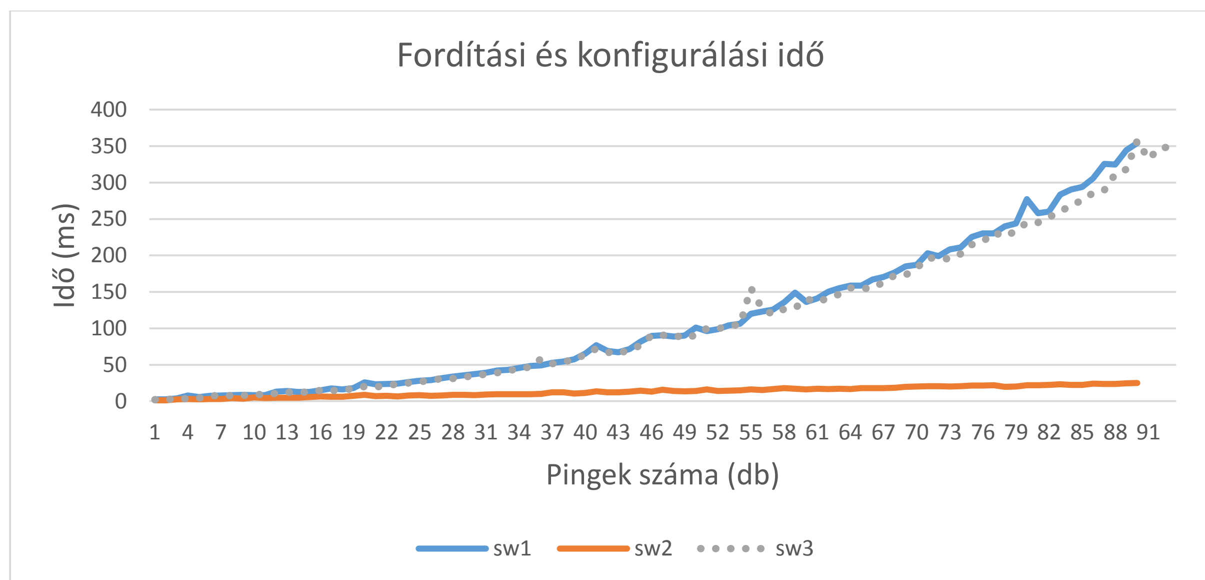
A fordítási és switch konfigurálási idő együttesét a *NetCore_Controller.ml* fájlban található *configure_switch* függvény futási idejével mértem. A függvény egy switch azonosítót és a hálózat teljes, *NetCore_Stream* típusú irányelvét kapja paraméterül, melyből kiválasztja az adott switchre vonatkozó irányelveket, flow táblát generál belőlük, melyet elküld a switchnek. A függvény futási idejét az OCaml által biztosított Unix modul *gettimeofday* hívásaival mértem, mely dokumentációja szerint [21] mikroszekundum pontosságú.

A folyton növekvő policy stream létrehozásához kihasználhatjuk, hogy a router csak pontos IP címekre tud továbbítási bejegyzéseket elhelyezni. A hálózaton konfiguráció frissítést is elvégezve (*sw1*-*sw2* link nincs) *h1*-ről küldjünk ping üzeneteket a 10.0.2.0/24-es alhálózatba,

különböző, növekvő IP címekkel. A *h3* hoszt IP címét állítsuk át 10.0.2.253-ra[14], azaz a legmagasabb, még nem kiosztott címre (10.0.2.254 az alapértelmezett átjáró az alhálózatban), így egyik PING REQUEST üzenetre sem kapunk választ, tehát a visszafelé útra nem fog elhelyeződni továbbítási szabály.

Tehát minden küldött ping üzenetre elhelyeződik egy továbbítási policy az *sw1*-ben, és az *sw3*-ban. Az *sw2*-re vonatkozó policy-k pedig nem változnak, mert *sw2* az útvonalválasztó táblája alapján látja, hogy a cél alhálózat hozzá közvetlenül csatlakozik, így megpróbálja a cél IP címet feloldani az ARP protokoll segítségével. De mivel az adott IP című hoszt nem létezik az alhálózaton, választ nem kap senkitől, a csomag eldobódik, a policy streamben pedig nem történik változás.

A mérést egy shell szkripttel végeztem, mely paraméterül várja, hogy a 10.0.2.0-ás IP címtől kezdve, hány IP címre küldjön pinget. A 24. ábrán láthatjuk az eredményt 160 címig történő mérésre.



24. ábra: Fordítás és konfigurálás együttes ideje a policy stream mérete függvényében.

Sajnos a fordítás és konfigurálási algoritmus nem túl jól skálázódik, 90 db switchenkénti bejegyzés esetén már több mint 300 ms-be tart. Sőt az összességében körülbelül 180 db policy-t tartalmazó adatfolyam esetén a Frenetic rendszer összeomlott. Vagyis egyszerűen befejezte a végrehajtást hibaüzenet nélkül, erőforrásait (pl. a Unix socketet)

[14] A DHCP szerver működésével együtt, közvetetten a MAC címen keresztül tudjuk befolyásolni az IP címet. A beállított MAC cím 00:00:00:00:00:FD, amely a 10.0.2.253-as IP címet eredményez az adott alhálózatban.

felszabadítatlanul hagyva. A mérést többször is elvégeztem, és mindegyik esetben a 100. körüli ping üzenet után (kb. 200 policy) történt meg az összeomlás.

Annak ellenére, hogy a *sw2*-re vonatkozó policy-k száma nem nő, flow táblájának előállítása mégis növekszik a mérettel. Ez végeredményben természetes, hiszen többől kell kiválasztani azt a konstans darabszámú irányelvet.

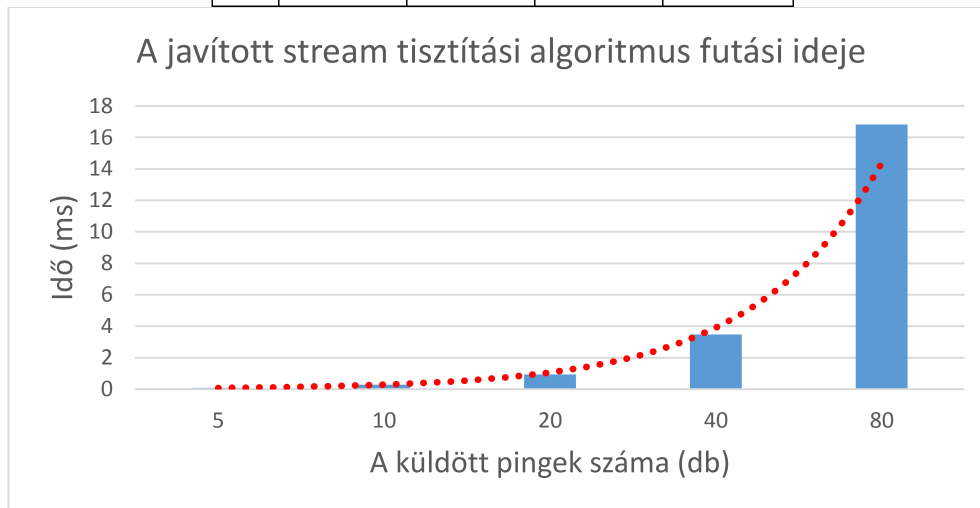
6.2 Javított stream tisztítási folyamat.

Kíváncsiak lehetünk arra is, hogy vajon mennyi időbe tart az a policy stream tisztítási algoritmus, amikor csak azokat a továbbítási irányelveket töröljük ki a router modul streamjéből, amely ellent mondana a frissített konfigurációnak. Ugyanis ez az egyszerű, mindent törlő tisztításhoz képest többletet jelent időben, amely befolyásolhatja a hálózat kiesésének idejét egy újrakonfigurálás alkalmával.

Az előző mérésben használt ping üzeneteket küldő shell szkriptet alkalmazva töltöttem fel a policy streamet, szintén a *h1* hosztról, a 10.0.2.0/24-es alhálózatba, az *sw3* switchen keresztül küldött ping üzenetekkel. De ebben az esetben a szkript paramétere nem volt állandó, amellyel meghatározhatjuk, hogy hány ping üzenet generálja a policy streamet.

A *gettimeofday* hívásokat ezúttal egy saját kódrészletem köré kellett elhelyezni, amely végiglépked a policy stream teljes egészén, és egyenként megvizsgálja a benn lévő szabályokat, hogy okoznának-e inkonzisztenciát a törölt, illetve hozzáadandó útvonalválasztó táblabejegyzésekkel. Az elvégzett mérések eredményeit a 25. ábrán láthatjuk.

N	1. mérés (ms)	2. mérés (ms)	3. mérés (ms)	átlag (ms)
5	0.0879	0.0851	0.0920	0.088333
10	0.2720	0.2980	0.2689	0.279633
20	0.8621	1.0400	0.9351	0.945733
40	3.5150	4.0979	2.8281	3.480333
80	16.5760	17.5600	16.3580	16.83133



25. ábra: A javított policy stream tisztítási algoritmus futási ideje táblázatban és grafikonon.

A táblázat N oszlopában látható, hogy milyen paramétert kapott a shell szkript. A három mérés szórása a táblázat adatai alapján láthatóan viszonylag kicsi, így az átlagolással nem viszünk nagy hibát az eredménybe.

Az algoritmus futása alkalmával pontosan egyszer megy végig a policy streamen, így futása lineáris lépésszámú a policy stream méretére nézve, és konstans módosítandó táblabejegyzést feltételezve. A grafikonon látható szaggatott fonal egy exponenciális görbe, ami körülbelül illeszkedik a mérés eredményeire. Mivel a vízszintes tengely exponenciális skálázású, mondhatjuk, hogy a mérési eredmény alátámasztja a lépésszámra vonatkozó számításainkat.

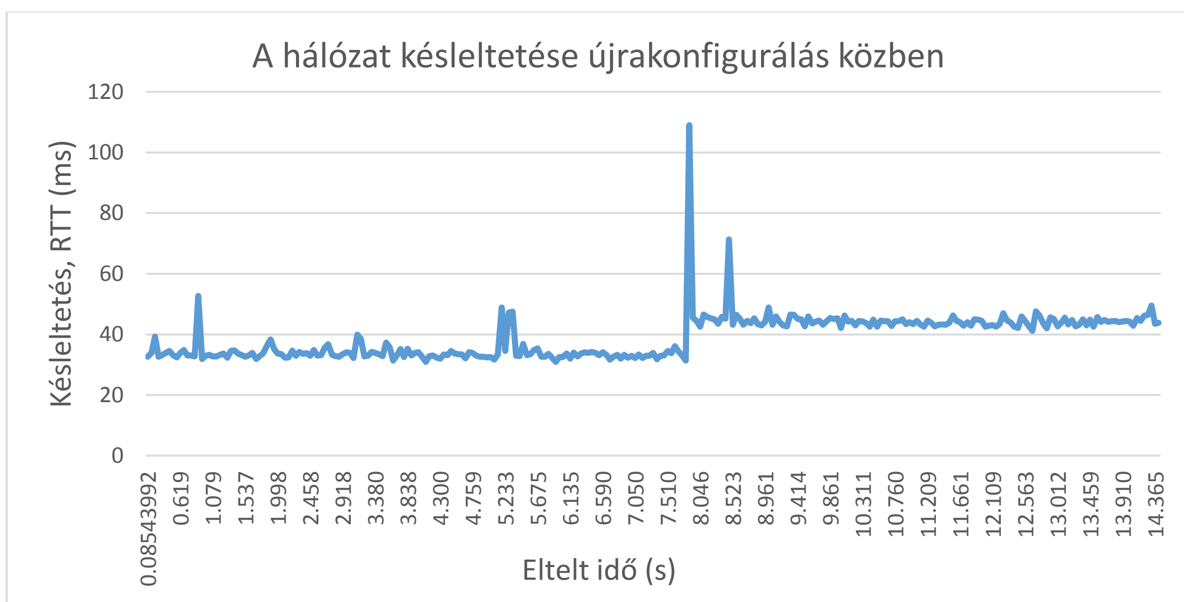
Mivel 100 db körüli pinggel generált policy stremet a rendszer már nem tud kezelni, nagyobb értékekre nem tudtam megmérni az algoritmus futási idejét. Így ezek alapján kijelenthetjük, hogy az algoritmus megvalósításával nem vittünk szűk keresztmetszetet a rendszerbe.

6.3 Hálózat késleltetése újrakonfigurálás közben

Ha egy futó hálózathoz új konfiguráció érkezik, akkor a Frenetic rendszernek a módosításokhoz igazodva el kell végeznie a megfelelő policy stream módosításokat, valamint le kell fordítania a FreneticDSL nyelvű kódot OpenFlow táblabejegyzésekre, és fel kell konfigurálnia a switchek flow tábláit.

Az eddig használt topológiánkon végeztem mérést a késleltetés ingadozás mértékének meghatározására. A topológia kezdeti konfigurációján, amikor még minden linket használatban tartunk, *h1*-ről indítottam egy ping folyamatot, mely 50 ms-onként küld egy PING REQUEST csomagot a *h3* hoszt 10.0.2.3-as IP címére. Az érkező válaszok alapján folyamat minden küldött csomagra meg tudja határozni késleltetést. A hálózaton eközben háttérforgalom nem ment, így az eredmények optimális, laboratóriumi körülmények között születtek.

A ping folyamat indítása után néhány másodperccel küldjük el a kontrollernek a konfiguráció frissítést, amelyben a korábban leírtak alapján, úgy helyezzük el az útvonalválasztó táblabejegyzéseket, hogy a *sw1-sw2* linket ne használják. Ekkor az ezen a linken küldött forgalom átirányul a *sw3*-at is érintő útvonalra. A mérés kapott eredményét a 26. ábrán láthatjuk.



26. ábra: A hálózat késleltetése konfiguráció frissítés közben

A mérést többször is elvégeztem a teljes törlést és a javított tisztítást alkalmazó konfiguráció frissítési eljárásokkal is, de az eredmény mindegyik esetben nagyon hasonló jellegű volt. Ennek az az oka, hogy ebben az esetben a policy stream meglehetősen kicsi, mert gyakorlatilag egy darab ping üzenet által generált méretű. Ilyen esetben pedig még a javított algoritmus futási ideje is közel elhanyagolható a hálózat késleltetéséhez képes, ezért kaphattuk a két algoritmussal közel azonos eredményt.

A 26. ábrán jól látható, hogy a konfiguráció frissítés előtt 30-35 ms a késleltetés, ami közelítőleg a *h1-sw1-sw2-h3* úton lévő három link késleltetésének összege. A többi mérés során is az újrakonfigurálás pillanatában mért késleltetés 90-140 ms között mozgott, ahogy ezt az ábrán is láthatjuk. A frissítés után a forgalom a *h1-sw1-sw3-sw2-h3* útvonalon halad, amely 4 link RTT-jának az összege. A mérésekből átlagolás nem végeztem, a bemutatott eredmény egy konkrét esetből kapott minta.

6.4 A javított stream tisztítás hatékonysága

Az útvonalválasztó tábla egy feltöltött állapotában, amikor még forgalmazás nem történt a hálózaton, a switchek flow táblái, illetve a modulok policy streamjei még üresek. Ezek feltöltését a routing tábla alapján, a beérkező forgalom váltja ki, amely csak kezdetileg okoz késleltetést, későbbi forgalom továbbításához a switcheknek már nem szükséges konzultálnia a kontrollerrel. Vajon mekkora késleltetés különbségeket eredményez ez a működés? Hogyan befolyásolja a kétféle frissítési algoritmus ezeket a késleltetési értékeket?

Vegyünk egy ping üzenet küldési sorozatot az alábbi módon, amikor minden alkalommal kettő, egy másodperc különbséggel küldött ping csomagot teszünk a hálózatra: *h1→h2; h1→h3; h1→h4; h2→h3; h2→h4; h3→h4*, azaz egy minimális szükséges sorozatot, hogy minden routing táblabejegyzést használjunk legalább egyszer. Mivel ez a mérés ilyen formájában nehezen automatizálható, kézzel végeztem el. Azonban ekkor az ARP bejegyzések időnkénti kiürülése miatt bizonytalan lenne, hogy szükséges-e a feloldás vagy nem. Ennek kiküszöbölése érdekében az ARP modult használaton kívülre helyeztem úgy, hogy minden router és hoszt ARP tábláját feltöltöttem minden számára szükséges IP cím – MAC cím párosítással. Így a késleltetéseket nem fogja befolyásolni, hogy a csomag kiküldése előtt szükséges-e ARP címfeloldást végezni vagy nem.

A 27. ábrán látható módon három esetben végeztem el a méréseket. A táblázatokban minden méréshez két sor tartozik, vagyis mindegyik esetben háromszor végeztem el a méréseket, melyek átlagát a táblázatok utolsó soraiban láthatjuk. Mindegyik táblázat adatai miliszekundumban értendők.

Az 1. esetben a *h1→h2* üzenetküldésnél az *sw1* flow táblájában még egyik célállomásra sincs elhelyezve flow szabály, így a REQUSET és a REPLY üzenetek is kiváltanak egy bejövő csomag eseményt a kontrollerben, melyek mindegyikére policy stream kibővítéssel és fordításával válaszol a Frenetic. Amire a második ping üzenet is kiküldésre kerül, a switch már le tudja kezelni a forgalmat a kontroller beavatkozása nélkül is. Ehhez hasonlóan a *h1→h3* üzenet váltásnál, a *h3*-nak célzott forgalom igényli, míg a már a *h1*-nek célzott csomagok nem igénylik a kontroller beavatkozását a továbbírásba.

A 2. esetben a policy streamből nem kerülnek törlésre a *sw1* azon továbbítási irányelvei, amelyek a *h1*-hez, *h2*-höz és *h4*-hez küldenek csomagokat, hiszen a javított esetben csak azok törlődnek, amelyek az *sw1-sw2*-n való forgalmazásra utasították volna a switchet. Ezért az átlagos első, illetve második pingek késleltetései közel azonosak. Így például a *h1→h3* pingelésnél is csak összesen három esetben kell a switchnek a kontrollerhez fordulnia (odafelé *sw1*-ben, *sw3*-ban, visszafelé csak *sw2*-ben, mert *sw3*-ban maradt a bejegyzés *h1* célú továbbításra)

A 3. esetben a policy streamből minden bejegyzést törlésre kerül, és visszaállítottuk a kezdeti, minden forgalmat a kontrollernek továbbító policy-t. A *h1→h3* mérés azért ilyen kiemelkedően magas, mert a forgalom a *h1-sw1-sw3-sw2-h3* útvonalon halad, ahol sokszor kell a kontrollerhez fordulni továbbítási döntés meghozásával.

1. eset	Kezdeti flow elhelyezés (ms)					
	h1 -> h2	h1 -> h3	h1 -> h4	h2 -> h3	h2 -> h4	h3 -> h4
1. ping	89.0	92.4	92.9	38.9	34.0	38.0
2. ping	21.9	39.9	32.4	34.5	33.8	36.5
1. ping	87.6	87.0	104.0	52.0	39.5	52.4
2. ping	24.1	36.1	33.6	38.0	34.4	44.3
1. ping	87.0	94.1	94.1	60.0	34.2	40.0
2. ping	22.3	35.7	31.1	37.7	34.0	33.0
1. ping átl.	87.9	91.2	97.0	50.3	35.9	43.5
2. ping átl.	22.8	37.2	32.4	36.7	34.1	37.9

2. eset	Flow elhelyezés javított újrakonfigurálás után (ms)					
	h1 -> h2	h1 -> h3	h1 -> h4	h2 -> h3	h2 -> h4	h3 -> h4
1. ping	22.9	elveszett	34.9	50.8	33.0	33.2
2. ping	23.9	43.5	32.4	44.3	32.9	32.3
1. ping	23.4	46.6	32.8	47.4	33.6	33.7
2. ping	27.9	44.0	33.0	45.9	33.1	32.9
1. ping	22.6	49.7	34.1	46.6	34.0	33.9
2. ping	23.4	48.2	34.9	46.1	35.3	33.8
1. ping átl.	23.0	(48.2)	33.9	48.3	33.5	33.6
2. ping átl.	25.1	45.2	33.4	45.4	33.8	33.0

3. eset	Flow elhelyezés teljes törlés után (ms)					
	h1 -> h2	h1 -> h3	h1 -> h4	h2 -> h3	h2 -> h4	h3 -> h4
1. ping	23.3	171.0	57.7	67.1	33.0	elveszett
2. ping	25.6	43.6	33.6	44.3	32.4	31.2
1. ping	25.6	162.0	60.3	71.3	34.3	35.5
2. ping	24.2	43.1	32.5	47.7	33.7	33.7
1. ping	23.6	146.0	50.3	76.7	35.4	elveszett
2. ping	21.9	46.3	33.4	45.0	38.9	33.7
1. ping átl.	24.2	159.7	56.1	71.7	34.2	(35.5)
2. ping átl.	23.9	44.3	33.2	45.7	35.0	32.9

27. ábra: A kontrollerhez felküldött csomagok (1.), illetve switchen feldolgozott (2.) csomagok késleltetése

Az adatok alapján jól látható, hogy legtöbb esetben nyereségesen jövünk ki a javított tisztítási algoritmus használatából, mert gyorsabb konvergenciát eredményez a kontroller router moduljának belső állapota és a switchek flow táblái között.

6.5 Együttműködés a TCP protokollal

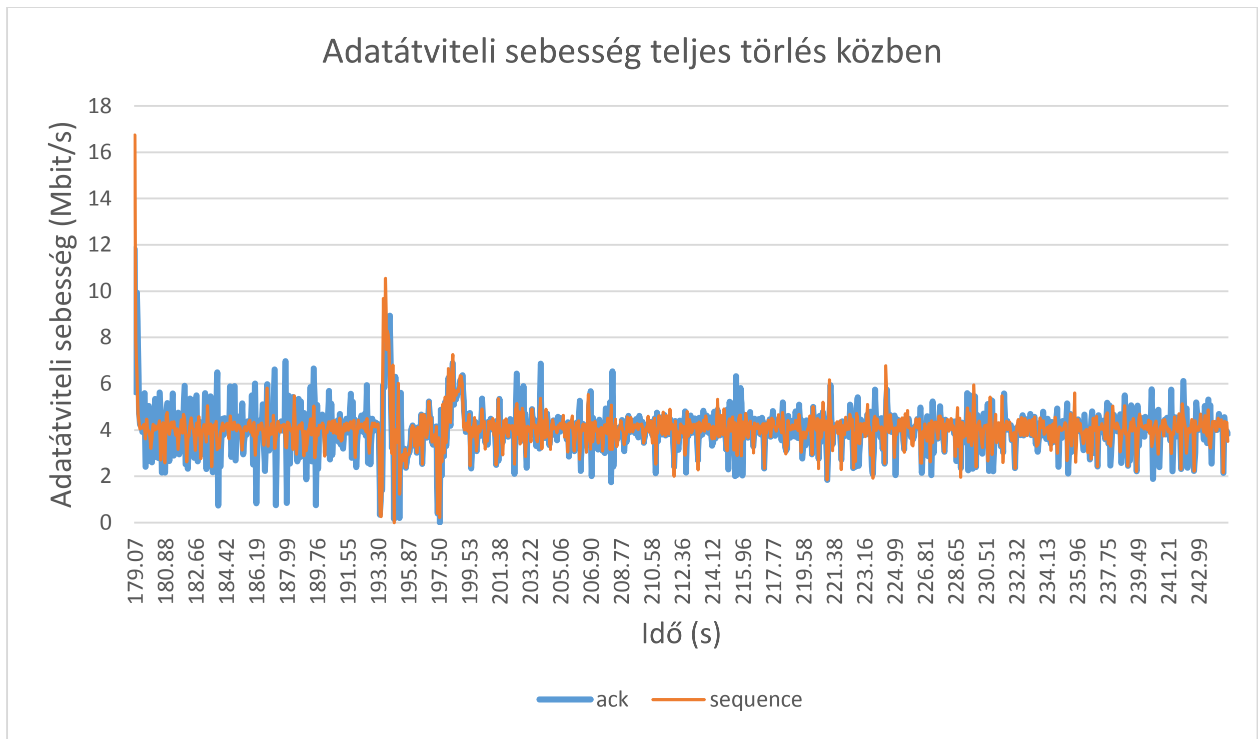
Felmerülhet a jogos kérdés, hogy az újrakonfigurálásnál mért késleltetések hogyan befolyásolnak egy folyamatban lévő TCP kapcsolat felett folyó adatátvitelt. Ennek vizsgálatához nézzük meg a TCP kapcsolat adatátviteli sebességét, illetve a torlódási ablak értékét az idő függvényében, miközben a hálózaton elvégezzük a szokásos konfiguráció frissítést.

Ehhez a *tcp_probe* Linux kernel modult használtam, amely egy fájlba minden TCP-vel kapcsolatos eseményt naplóz. A mérés elvégzéséhez a *h1* hoszton indítottam a már említett *SimpleHTTPServer,* beépített Python szkript segítségével egy HTTP szervert, amelyről a *h3*-ról indított kéréssel letöltöttem egy 31.5 Mbyte-os szövegfájlt. Az átvitel ideje alatt végrehajtottam a hálózaton az eddigiekkel azonos konfiguráció frissítést.

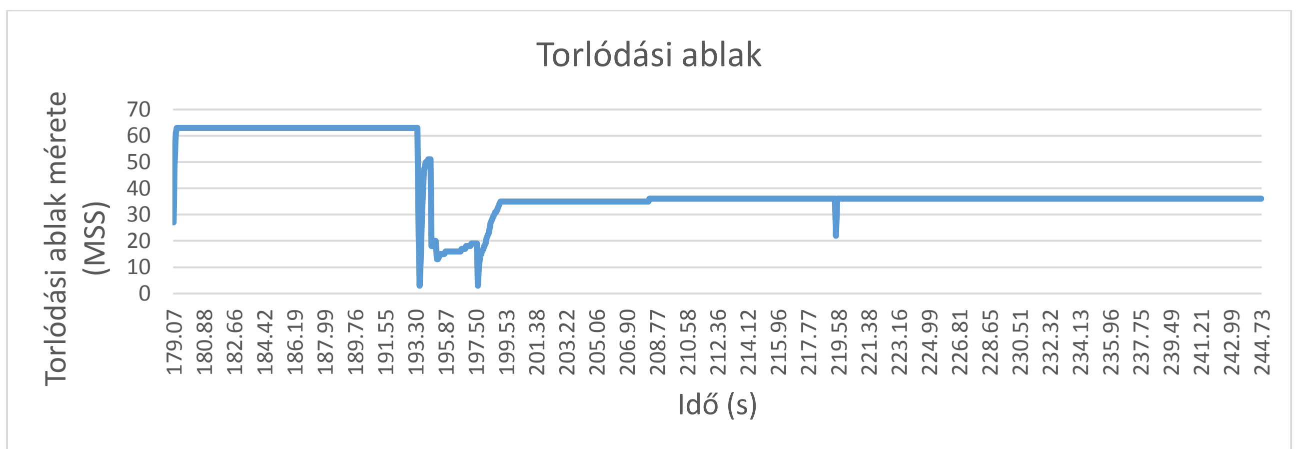
A *tcp_probe* kimenetének elemzéséhez készítettem egy Python szkriptet, amely a nyugták alapján körülbelül 50 ms-onként átviteli sebességet számol. Végiglépkedve a TCP nyugtákra szűrt naplófájlon, az időbélyegek alapján számolja a csomagok közötti különbségeket, és amikor először olyan két csomaghoz ér, amelyek között legalább 50 ms a különbség, akkor erre az intervallumra a TCP *sequence* és *ack number* alapján is kiszámolja a bájtkülönbségeket. Az 50 ms-nál sűrűbb mérésekkel gyakran előfordult, hogy nem érkezett egyáltalán TCP nyugta az adott intervallumban, így a tapasztalatok alapján ez az intervallum megfelelőnek tűnik.

A teljes törléssel elvégzett frissítés után a TCP nehezebben állítja helyre a kapcsolatot, mint az optimalizált tisztítás. Hiszen a javított esetben a *sw1* és *sw2* nem veszíti el az „onlink" információkat, amely az előző mérések alapján néhány 10 ms nyereséget jelent a teljes törléshez képest. Úgy tűnik, hogy ez a különbség a TCP működésében hatványozottan megmutatja magát. A teljes törléssel és a javított tisztítással kapott eredmények adatátviteli sebességre vonatkozó grafikonjait rendre a 28. és 29. ábrán[15] láthatjuk.

[15] A torlódási ablakot MSS (Maximum Segment Size)-ban mérjük, mely a csomagok maximális mérete bájtban.



28. ábra: Adatátviteli sebesség teljes törlés mellett



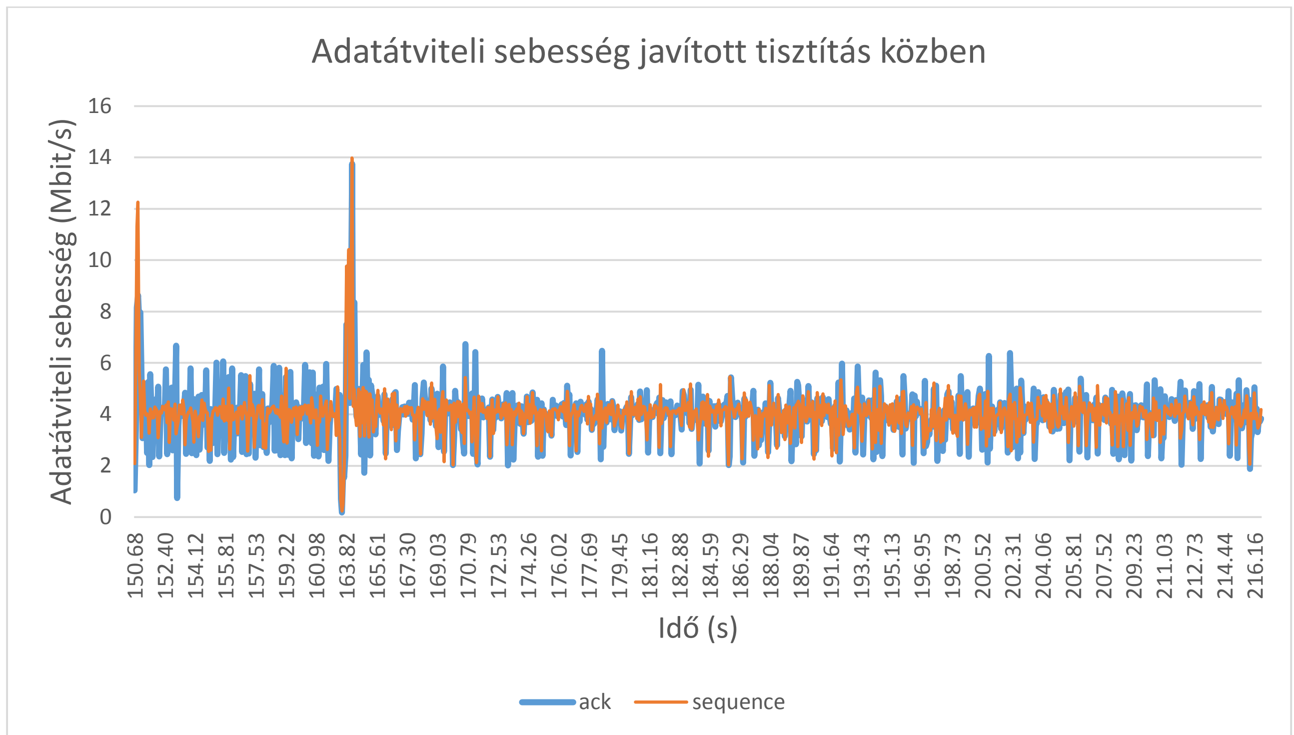
29. ábra: Torlódási ablak teljes törlés mellett

A 28. ábrán a *sequence* alapján számolt, és az *ack* alapján számolt átviteli sebességek láthatóak egymásra illesztve. A két mérési mód alapján megközelítőleg ugyanazokat az értékeket kaptuk. Sajnos a Mininetben 10 Mbit/s-re lekorlátozott linkeken csak az ábrán látható 4-6 Mbit/s-ot tudták teljesíteni[16]. Ennek oka valószínűleg a fogadó oldali ablak (receiver

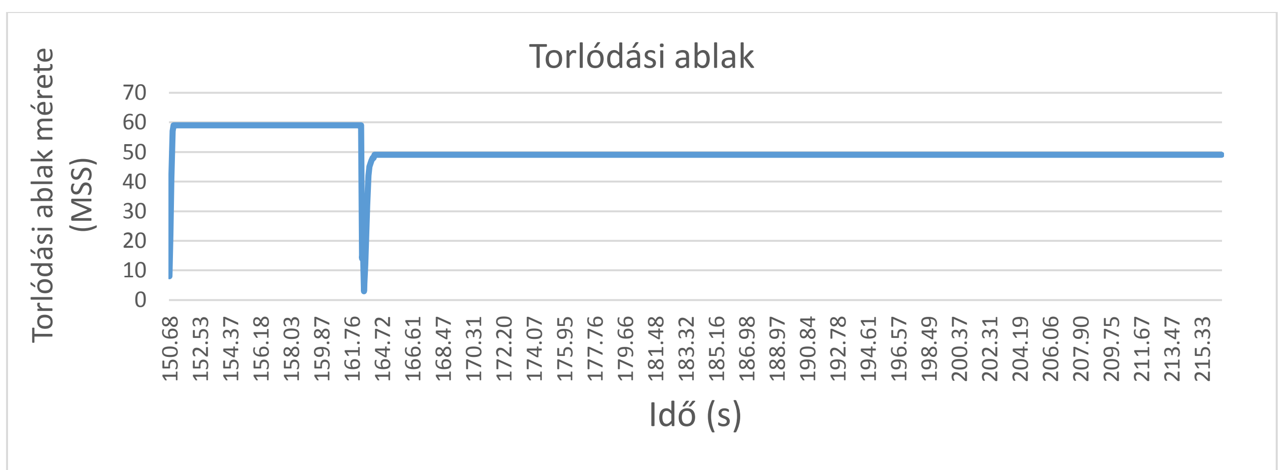
[16] A mérést egyébként elvégeztem a Wireshark protokoll analizálóval is, és az eredmények nagyon hasonlóak lettek.

window) korlátozott mérete lehet. Erre az alapján következtethetünk, hogy a torlódási ablak sem növekszik tovább egy bizonyos értéknél. A 28. és 29. ábra alapján is azt mondhatjuk, hogy az átvitel stabilizálódása 5-6 másodperc után történik csak meg.

A javított tisztítási algoritmust alkalmazva, a mérést azonos körülmények között megismételve, a kapott eredményeket a 30. és 31. ábrán láthatjuk.



30. ábra: Adatátviteli sebesség javított tisztítás közben



31. ábra: Torlódási ablak javított tisztítás mellett.

A kapcsolat helyreállása jelentősen gyorsabban végbemegy ebben az esetben, mindössze 1-2 másodpercet vesz igénybe. De az előző mérésnél említett problémák problémák az adatátviteli sebesség átlagos értékével, hasonlóan itt is fenn állnak. A méréseket természetesen mindkét esetben többször is elvégeztem, de az ábrákon bemutatott statisztikák egy-egy mérés eredményei. A mérések alkalmaival az értékek kissé ingadoztak, de következtetéseinket csak alátámasztják.

Összességében tehát a tisztítási algoritmusra érdemes időt szánni, további lépésszám csökkentést a policy stream valamilyen típusú rendezésével lehetne elérni, amely későbbi munkák során megfontolandó.

7 Összefoglalás

Dolgozatom elején áttekintettem a mai SDN technológiák nyújtotta lehetőségeket, és a velük kapcsolatos problémákat. Bevezetést tartottam a deklaratív programozási paradigma alap gondolataiba az OCaml nyelv bemutatásán keresztül. Majd megvizsgáltuk, hogy a deklaratív programozási szemlélet milyen előnyökkel járhatna a hálózatprogramozásban. Ennek illusztrálására bemutattam néhány példát, melyek a mai rendszerek nagyobb méreteiben komoly problémákat jelenthetnek.

Bemutattam a Frenetic rendszer, új szemléletű hálózatprogramozást megvalósító működését, mely a mai hálózatokkal kapcsolatos számos problémára készül megoldást nyújtani. A rendszer egyik fő hiányosságának azt találtam, hogy jelenlegi verziójában csekély lehetőséget nyújt a dinamikus konfigurálásra.

A Frenetic rendszer fejlesztői által megkezdett ösvényt követve, szükség lenne egy szolgáltatás szintű hálózatprogramozó keretrendszerre, mely a Frenetickel együttműködve szolgálhatná a hálózat üzemeltetőit. Ehhez a Frenetic felhasználási területeinek felderítésére volt szükség.

A Frenetic rendszert főleg későbbi felhasználásra való alkalmasság szempontjából vizsgáltam. A rendszer dinamikusság területén mutatott hiányosságai minél hamarabbi pótlásra szorulnak, ezért célul tűztem ki egy olyan Frenetic modul megvalósítását, amely ezen a területen jobban teljesít.

Az általam implementált router modul támogathatná a Freneticre támaszkodó, szolgáltatás szintű hálózatprogramozási keretrendszer dinamikusan konfigurálható, kapcsolatteremtő szolgáltatását. A router modul alkalmazhatóságának kiterjesztéséhez készítettem egy alap ARP funkciókat ellátó modult, illetve a hálózat hosztjainak IP címkiosztását megvalósító DHCP modult.

A Frenetic elemzésében és teljesítmény tesztelésében kapott eredményeim alapján azt láthatjuk, hogy a Frenetic rendszer jó úton halad afelé, hogy egy egyszerűen kezelhető, megbízhatóan működő, modern hálózatüzemeltetői keretrendszer létrehozását előreléptesse.

8 Hivatkozások

[1] Roy Chua, Matt Palmer, Craig Matsumoto: *SDN Central – The independent community & #1 resource for SDN and NFV*. http://www.sdncentral.com/sdn-use-cases/ (letöltés: 2013. október 24.)

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner: *OpenFlow whitepaper*. http://archive.openflow.org/documents/openflow-wp-latest.pdf (letölrés: 2013. október 24.)

[3] Bob Lantz, Brandon Heller, Nikhil Handigol, Vimal Jeyakumar: *Mininet – An Instant Virtual Network on your Laptop (or other PC)* http://mininet.org/

[4] Csáji Balázs Csanád, Bárány Vince, Bui Thi Lanh: *Az ML nyelvcsalád* http://nyelvek.inf.elte.hu/leirasok/ML_nyelvcsalad/index.php?chapter=1 (letöltés 2013. október 24.)

[5] Objective Caml hivatalos weblapja: http://ocaml.org/ (2013. október 24.)

[6] Walter Robert J. Harrison: Frenetic: *A Network Programming Language*, Princeton University, 2011. május

[7] Nate Foster, Rob Harrison, Matthew L. Meola, Michael J. Freedman, Jennifer Rexford, David Walker: *Frenetic: A High-Level Language for OpenFlow Networks*, 2010 november

[8] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto , Jennifer Rexford, Alec Story, David Walker: *Frenetic: A Network Programming Language*, 2011. szeptember

[9] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker: *NOX: Towards an operating system for networks.* 2008. július

[10] Nate Foster, Rob Harrison, Christopher Monsanto, David Walker: *A Compiler and Run-time System for Network Programming Languages*. 2012. január

[11] Arjun Guha, Mark ReitBlatt, Nate Foster, Cole Schlesinger: *Frenetic 1.0.2 forráskód* https://github.com/frenetic-lang/frenetic/tree/frenetic.1.0.2 (letöltés: 2013. július)

[12] Arjun Guha, Mark ReitBlatt, Nate Foster: *Machine-Verified Network Controllers*, 2013. június

[13] Action for Technological Development: *The Coq Proof Assistant* http://coq.inria.fr/ (letöltés: 2013. október 24.)

[14] Arjun Guha, Mark ReitBlatt, Emin Gün Sirer, David Walker*: The Frenetic Project* minden kapcsolódó forrása, https://github.com/frenetic-lang (letöltés: 2013. október 24.)

[15] Richard Mortier, Anil Madhavapeddy: *Cstruct*, https://github.com/avsm/ocaml-cstruct (letöltés: 2013. október 24.)

[16] OCSIGEN: *Lightweight Thread Library*, http://ocsigen.org/lwt/manual/manual (letöltés: 2013. október 24.)

[17] Arjun Guha: *Frenetic Tutorial*, https://github.com/frenetic-lang/frenetic/wiki/Frenetic-Tutorial (letöltés: 2013. október 24.)

[18] Mark ReitBlatt: *Frenetic Manual*, https://github.com/frenetic-lang/frenetic/wiki/A-NCManual (letöltés: 2013. október 24.)

[19] OCSIGEN: *Lwt_stream* Manual Page, http://ocsigen.org/lwt/api/Lwt_stream (letöltés: 2013. október 24.)

[20] IETF RFC 4627 *The application/json Media Type for JavaScript Object Notation*, http://json.org/, (2006. július)

[21] INRIA: OCaml *Unix modul* referenciája, http://caml.inria.fr/pub/docs/manual-ocaml-4.00/libref/Unix.html (letöltés: 2013. október 24.)