

TDK-dolgozat

Sárközi Gergely

Mélyen programozható és önkonfiguráló hálózatok terheléelosztása dinamikus útvonalválasztással

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

INFORMÁCIÓS RENDSZEREK TANSZÉK



Szerző:

Sárközi Gergely

programtervező informatikus MSc

1. évfolyam

Témavezető:

Dr. Vörös Péter

egyetemi adjunktus

Budapest, 2023. 12. 03.

Tartalomjegyzék

1. Bevezetés	1
1.1. Programozható hálózatok	1
1.2. Terheléselosztás	1
1.3. Önkonfiguráló hálózatok	2
1.4. Saját megközelítés	3
1.5. Dolgozat szerkezete	3
2. Kapcsolódó munkák	5
3. Modell áttekintés	7
3.1. MAC tanulás és alapvető csomagtovábbítás	9
3.2. Lehetséges terheléselosztási megközelítések	12
3.3. Hozzhoz csatlakozó portok felismerése	13
3.4. Folyamok azonosítása	14
3.5. Folyamhoz tartozó útvonal tanulás alapjai	15
3.5.1. Tanulás megkezdésének okai, tanulás szükségességének esetei . .	15
3.5.2. Tanulási folyamat	17
3.5.3. Levél switch értesítése	21
3.6. Túlterheltség érzékelése és útvonal váltás	23
3.7. Nem használt útvonalak elfelejtése	27
3.8. Tanulási folyamat okozta hálózati terhelés	32
4. Megvalósítás	34
4.1. Vezérlősíkk	34
4.2. Folyamok azonosítása	35
4.3. Útvonal és folyamfüggetlen bejegyzések tárolása	35
4.3.1. Lehetséges perzisztencia megközelítések	36
4.3.2. Útvonal bejegyzések törlése	39

4.4. Túlterheltség megállapítása, kezelése	41
5. Kiértékelés	43
5.1. Szimulációkhoz használt topológia	44
5.2. Útvonalak sávszélességének összegét meg nem haladó küldési ráta	44
5.3. Hálózat sávszélességét meghaladó küldési ráta	52
6. Összefoglalás	58
6.1. Fejlesztési lehetőségek	59
6.1.1. Link meghibásodás, megváltozott hálózati topológia kezelése . . .	59
6.1.2. Gyakrabban útvonal váltások engedélyezése	59
A. Dolgozathoz felhasznált mesterséges intelligencia alapú eszközök	60
Irodalomjegyzék	61

Kivonat

Hálózati switchek egységnyi idő alatt csak bizonyos csomagmennyiség továbbítására képesek. Ennek a rátának a túllépése esetén a switchek a csomagok puffereelésére kényszerülnek, ezzel késleltetést vezetve be a hálózati kommunikációba. A pufferek (sorok) megtelítése esetén csomagvesztés lép fel, ami a TCP folyamatokat a küldési rátájuk csökkentésére, azaz az adatátvitel lassítására kényszeríti. Az ilyen szituációk elkerülését segíti elő többek között a terheléelosztás. A hálózati forgalom elosztása több lehetséges útvonal között a switchek egyenletesebb kihasználtságához (terheléséhez), azaz a késleltetés, illetve a csomagvesztés csökkenéséhez vezet.

A dolgozatban egy hatékony, manuális konfigurációt nem igénylő megoldást mutatunk be a központi vezérlő nélküli, ismeretlen topológiájú hálózatok terheléelosztására az adatkapcsolati réteg szintjén. Algoritmusunk kontrollált elárasztás segítségével minden egyes folyamhoz meghatároz egy útvonalat. Amennyiben egy ilyen útvonalon megnövekedett késleltetés, azaz túlterhelés lép fel, akkor a megoldásunk dinamikusan másik lehetséges, kevésbé terhelt útvonalakat próbál keresni és próbál folyamatokat átterelni rájuk, amíg a túlterhelés meg nem szűnik.

A megoldásunkat a P4 adatsík programozási nyelven megvalósítottuk és szimulációk segítségével összehasonlítottuk a következő korábbi munkákkal: ECMP, ARP-Path és TCP-Path. A szimulációkban megvizsgált hálózati körülmények során az algoritmusunk 12% és 48% közötti mértékben rövidítette le az adatátvitelhez szükséges időt. Továbbá a mi megoldásunk, a korábbi központi vezérlő nélkül működő munkákkal ellentétben, nem csak bizonyos (például ARP, TCP) protokollok használata esetén képes terheléelosztásra, hanem bármely egy- vagy kétirányú, Ethernet alapú adatátvitelt támogatja.

1. fejezet

Bevezetés

1.1. Programozható hálózatok

Az elmúlt időszakban a szoftver által definiált hálózatok (SDN) [1] jelentősége lényegesen megnőtt. Az ilyen hálózatok a csomagtovábbítás (adatsík) és a hálózat vezérlés (vezérlősík) szétválasztásán alapulnak. Ennek az új megközelítésnek köszönhetően nagy sebességű és rugalmas hálózatok hozhatók létre egyszerűen.

Egyes esetekben az adat- és a vezérlősík közötti kommunikáció okozta késleltetés, vagy az adatsíkról a vezérlősíkra átküldendő adat nagy mennyisége arra adott okot, hogy az adatsík szerepe megnőjön, azaz egyes vezérlési funkciók is közvetlenül az adatsíkban legyenek megvalósítva. Továbbá az adatsíkon sokszor csak bizonyos, a hardver által is támogatott protokollok feldolgozására volt lehetőség [2]. Ezeknek a problémáknak a megoldására jelentek meg az adatsík programozási nyelvek, amelyekkel a teljes csomagfeldolgozási folyamat egy magas absztrakciós szinten megadható.

Jelenleg a protokollfüggetlen P4 [3] adatsík programozási nyelv a legelterjedtebb, aminek segítségével újabb hálózati protokollok könnyen implementálhatók. A vezérlősík és az adatsík kommunikációja standardizálva van, és a P4 nyelven megírt programokat számos céleszközzre le lehet fordítani például a T4P4S [4] fordítónak köszönhetően.

1.2. Terheléselosztás

Amennyiben egyes switchek túl sok csomag továbbításáért felelnek, akkor a hozzájuk tartozó queue (kiküldendő csomagok sora) telítődhet, ezzel extra késleltetést vezetve be a hálózatba. Ha a switch sorok megtelnek, akkor csomagvesztés is bekövetkezik, amire

reagálva a TCP folyamatok (kapcsolatok) csökkentik a küldési rátájukat. A csökkentett küldési ráta eredményeként pedig ugyan annyi adatot több időbe telik átküldeni a hálózaton. Tehát ha egy switch túl sok csomag továbbításáért felel, akkor az adatátvitelhez szükséges idő megnő, illetve rövid üzenetek is később érnek csak célba.

Az ilyen problémák kezelésére többféle megközelítés létezik. Az Active Queue Management (AQM) megoldások célja, hogy a switch sorok ne teljenek be, hanem a TCP folyamatok küldési rátája még a telítődés előtt csökkenjen [5]. Egy másik, akár párhuzamosan is használható megközelítés a terheléelosztás. Ennek célja, hogy a hálózat erőforrásai egyenletesen legyenek kihasználva, azaz ne legyenek túl sok, illetve túl kevés csomag továbbításáért felelő switchek a hálózatban. Az egyenletes kihasználás következményeként torlódás kisebb eséllyel lép fel, a hálózati késleltetés csökken, a maximális sávszélesség pedig nő.

Nem minden hálózat alkalmaz terheléelosztást. Például a (Rapid) Spanning Tree Protocol (STP) [6] a redundáns linkeket felismeri és kikapcsolja, azaz a hálózati topológiát egy fává alakítja, ezzel egyszerűen megvalósítható csomagtovábbítást tesz lehetővé. Viszont egy fában nincsen lehetőség terheléelosztásra, hiszen két hoszt közötti csomagküldésre csak egy lehetséges útvonal létezik.

A legalapvetőbb terheléelosztási módszer az Equal-Cost Multi-Path (ECMP) [7], amely több azonosan jónak ítélt útvonal közül választ egyet oly módon, hogy az azonos folyamba tartozó csomagok azonos útvonalon haladjanak. Az algoritmus minden útvonalra egyenlő számú folyamatot próbál osztani a folyamatok sávszélesség igényét figyelmen kívül hagyva, így a véletlennek köszönhetően sokszor torlódás tud fellépni átlagos folyamszámok mellett [8]. Továbbá a két hoszt közötti lehetséges útvonalak megállapítása nem az ECMP része: ezt a feladatot általában a vezérlőszík látja el.

Manapság számos, különböző felhasználási területekre tervezett terheléelosztási algoritmus létezik, amelyek a lehetséges útvonalak megállapítására is képesek. Ezekről bővebben a Kapcsolódó munkák fejezetben van szó.

1.3. Önkonfiguráló hálózatok

Önkonfigurálónak hívjuk azokat a hálózatokat, amik külső konfiguráció nélkül is képesek csomagtovábbításra egy számukra eddig ismeretlen topológiában. A napjainkban használt legtöbb hálózat ilyen: a switchek megtanulják, hogy az egyes MAC címek melyik portjuk felé találhatók, ezáltal mind unicast, mind broadcast csomagok továbbításá-

ra képesek. Az önkonfiguráláson alapuló csomagtovábbítást *Transparent Bridging*-nek is szokás nevezni [9].

Amennyiben a hálózati topológia előre ismert és rögzítve van, akkor van lehetőség önkonfiguráló technológiák helyett központi konfigurálást alkalmazni: egy központi vezérlő átadhatja a switcheknek a csomagtovábbításhoz szükséges szabálykészleteket, amiket egyébként maguktól kéne megtanulniuk. Az ilyen típusú hálózatok korlátozott elterjedése miatt a dolgozatban az önkonfiguráló hálózatokra fogunk fókuszálni.

1.4. Saját megközelítés

A dolgozatban egy kontrollált elárasztáson alapuló algoritmust ismertetünk, amelynek segítségével alacsony késleltetésű és terheltségű útvonalak határozhatók meg bármilyen (ismeretlen) hálózati topológiában mindennemű központi (globális) vezérlő nélkül. Garantáljuk továbbá az útvonalak szimmetriáját (egy hoszt pár közötti kommunikáció mindkét irányban azonos útvonalon történjen). Túlterheltség bekövetkezése esetén egyesével egy újabb (kevésbé terhelt) útvonalra irányítunk folyamatokat, amíg a túlterheltség meg nem szűnik. A megoldás a következő szélsőséges esetekben is képes teljes funkcionalitásra:

- Csomagvesztés történik, így a saját vezérlési csomagjaink is elveszhetnek (eldobásra kerülhetnek) a hálózatban.
- Az adott folyamhoz tartozó (UDP) kommunikáció egyirányú, azaz egy hoszt pár közötti kommunikáció során csak az egyik hoszt küld csomagokat.

Bemutatjuk továbbá egy implementációját az algoritmusnak, amit a P4 adatsík programozási nyelvben valósítottunk meg, switchenkénti lokális vezérlők segítségével.

1.5. Dolgozat szerkezete

A továbbiakban a dolgozat az alábbi módon épül fel: a Kapcsolódó munkák fejezetben összefoglaljuk a legjelentősebb terheléselosztási megoldásokat, kitérve azok alkalmazhatóságára önkonfiguráló, illetve programozható hálózatokban. A Modell áttekintés fejezetben az általunk megvalósított terheléselosztási algoritmus működéséhez szükséges feltételeket, valamint magának az algoritmusnak a működését ismertetjük egy magas szinten. Az implementációs részleteket a Megvalósítás fejezetben fejtjük ki. A Kiértékelés fejezetben összehasonlítjuk az általunk bemutatott terheléselosztási megoldást a legfőbb

kapcsolódó munkákkal, különböző hálózati topológiákat és körülményeket megvizsgálva. Végül az Összefoglalás fejezetben összegezzük a dolgozatunk legfőbb eredményeit és algoritmusunk továbbfejlesztési lehetőségeit.

2. fejezet

Kapcsolódó munkák

Számos megoldás született terheléelosztásra, ezek közül van programozható hálózatokra tervezett algoritmus, illetve önkonfiguráló hálózatot megvalósító is.

A szerzők [10]-ben egy alternatív útvonalra váltás segítségével megvalósított torlódás elkerülési megoldást mutattak be. Egy központi vezérlőszék egy fő és egy alternatív útvonalat határoz meg. A vezérlőszék a teljes hálózat topológiáját, állapotát ismeri, ez alapján tud optimális útvonalakat kiszámítani. Amennyiben bármelyik switch a hálózatban érzékeli, hogy túl sok időt tölt el a csomag feldolgozásával (ami a switch vagy a kimeneti port túlterheltségére utal), akkor az egyik folyamat az alternatív útvonalra irányítja át innentől. Amennyiben az adott switch nem ismer alternatív útvonalat (mert például a topológiából adódóan nincs is), akkor szól az őt megelőző switchnek, hogy az útvonal váltást ejtse meg, vagy ő is szóljon az őt megelőző switchnek. A szerzők az algoritmust az adatsíkban (a P4 programozási nyelvben) valósították meg, így az nagyon gyorsan képes észlelni a torlódást és reagálni rá. Viszont a megoldás egy központi vezérlőt igényel, így önkonfiguráló hálózatokban nem alkalmazható.

Az ARP-Path [11] ezzel szemben kifejezetten önkonfiguráló hálózatokhoz illő útvonalválasztási algoritmus. A switchek az *ARP request* és *ARP reply* csomagok segítségével az összes lehetséges útvonalat kipróbálják az ARP forrás és cél hosztok között, de csak a leggyorsabbat tanulják (jegyzik) meg. Mivel a túlterhelt útvonalakon a hálózati késleltetés magasabb, ezért ez az algoritmus terheléelosztási megoldásként is működik. Viszont ennek a megoldásnak a hátránya, hogy csak MAC szintű terheléelosztásra képes (például folyamszintű terheléelosztás helyett), valamint létrejöhetnek aszimmetrikus útvonalak amennyiben két fél egyszerre küld ARP csomagokat. Ezen felül már megtanult útvonalak megváltozott hálózati terhelés esetén nincsenek újratanulva, így a megközelítés terhelés-

elosztási képessége is korlátozott.

[12]-ben az előbb ismertetett ARP-Path algoritmust általánosítják a szerzők All-Path néven, illetve részletesebben megvizsgálják a megközelítést, például a teljesítményét vagy a kompatibilitást más típusú switchekkel.

A szerzők a [13] cikkben szintén az ARP-Path algoritmussal foglalkoznak: megvizsgálják egy központi vezérlő és az ARP-Path algoritmus párhuzamos használatát, valamint a P4 programozási nyelv okozta nehézségeket is ismertetik.

[14]-ben a szerzők TCP-Path néven létrehozta az ARP-Path mintájára egy olyan algoritmust, ami az ARP-Path-tal ellentétben nem hozza meg a párok között határoz meg egy útvonalat, hanem minden TCP kapcsolathoz új útvonalat tanul meg, ezzel hatékonyabb terheléelosztást érve el. Ennek az algoritmus változatnak a megvalósíthatósága P4 programozható hálózatokban viszont nem lett megvizsgálva, illetve a tanulási folyamat a hálózati késleltetést csak a TCP kliensől a TCP szerverig vezető útvonalon méri, a másik irányban fellépő hálózati terhelést figyelmen kívül hagyva.

A CONGA [15] egy modern terheléelosztás megközelítés, amely adatközpontokban való alkalmazásra van tervezve. Folyamszintű terheléelosztás helyett a szerzők a folyamatokat *flowlet*-ekre (szünetek által elválasztott csomag löketekre) bontják, és ezen a kisebb egységen végzik a terheléelosztást, ezzel nagyobb hatékonyságot érve el.

[16]-ben a CONGA egy továbbfejlesztése olvasható HULA néven. Az algoritmus memóriaigényét lényegesen csökkentették a szerzők, ezzel a skálázhatóságot javítva. Továbbá bemutatták, hogy ez az új változat programozható hálózatokban is megvalósítható, nem szükséges hozzá egyedi hardver.

A Spanning Tree Protocol (STP) [6] lecserélését célzó TRILL [17] és Shortest Path Bridging (SPB) [18] algoritmusok megtanulják a hosztok közötti útvonalat vagy útvonalakat és több lehetséges útvonal esetén alkalmazzák a Bevezetés fejezetben ismertetett ECMP terheléelosztást. A folyamatok útvonalának megállapításához a folyamat azonosítóját hashelik, és a kapott értékhez tartozó útvonalon küldik tovább a folyamat csomagjait. Tehát a folyamatok sávszélesség igényét és a switchek terheltségét nem veszik figyelembe, valamint hash ütközések is felléphetnek (a hashelés nem feltétlenül egyenletesen osztja szét a folyamatokat, ha a folyamatok száma nem elég nagy), ezzel az algoritmusok terheléelosztási képessége korlátozott.

3. fejezet

Modell áttekintés

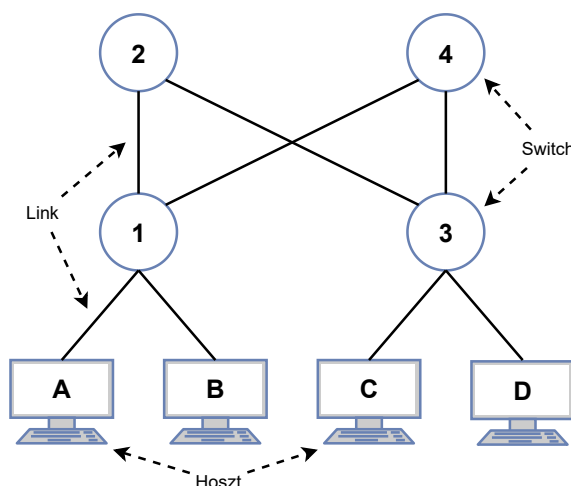
Feladatunk egyrészt egy ismeretlen topológiájú hálózatban unicast, illetve multi-cast/broadcast Ethernet csomagok kézbesítése. Nem áll rendelkezésünkre egy központi vezérlő (amely a teljes hálózatot ismerné, amely az összes switch vezérlésére képes lenne) és a hálózati eszközök nincsenek külsőleg konfigurálva: kezdetben nem tudnak semmit a hálózatról.

Törekszünk az eldobott (nem továbbított) és a duplikált (többször továbbított) csomagok minimalizálására. Célunk továbbá a továbbítást szimmetrikus útvonalakon megtenni, azaz törekszünk arra, hogy két fél közötti kommunikáció azonos útvonalon történjen, attól függetlenül, hogy a csomagot a két fél közül ki küldte, ezzel különféle lehetséges hálózati problémákat elkerülve [19].

Feladatunk továbbá a terheléelosztás megvalósítása is. A túlterheltség, azaz a torlódás a csomagtovábbítás megghiúsulását eredményezheti (hiszen torlódás során csomagdobás következhet be), valamint a hálózati késleltetést is megnövelheti. Ezeknek az elkerülése az elsődleges célja a bemutatott terheléelosztásnak.

A hálózat az általunk programozható switchekből és hosztokból épül fel. A hosztok lehetnek például számítógépek, vagy akár más hálózatokkal összeköttetést kialakító routerek is. A hosztok teljesen szabványos hosztok, azaz nem várjuk el tőlük, hogy bármilyen újfajta protokollt támogassanak. A switchek mindannyian azonos szoftvert futtatnak, de teljesítményük (maximális sávszélességük) eltérhet. A switchek és a hosztok között linkek húzódnak. Minden hoszt pontosan egy switchhez kapcsolódik, de a switchek akár-hány másik switchhez csatlakozhatnak. Az említett hálózati elemek a 3.1. ábrán vannak szemléltetve.

Minden, ezen megkötéseknek eleget tevő hálózat támogatva van, azaz a hálózati to-



3.1. ábra. Hálózatok alapvető építőelemei.

pológiák köröket is tartalmazhatnak. A körök más szemszögből azt jelentik, hogy két hálózati eszköz között több lehetséges útvonal található, ami a terheléselosztás szempontjából elengedhetetlen. Tehát a körök nemhogy meg vannak engedve, de a dolgozatban bemutatott megoldás elsősorban a köröket tartalmazó hálózatokra lett tervezve.

A switchek egységnyi idő alatt csak megadott mennyiségű csomag továbbítására képesek, ennek túllépése esetén következik be torlódás. Ez általában úgy van megvalósítva, hogy egy switch minden portjához tartozik egy kiküldendő csomagokat tartalmazó sor. Ez a sor egy adott kapacitással rendelkezik, és ha betelik, akkor csomagvesztés lép fel. A switch a sorból bizonyos időközönként képes kiküldeni egy-egy csomagot, azaz a sor mérete a hálózati késleltetést is befolyásolja.

A megoldásunkban egy saját Ethernet protokollt, azon belül pedig többféle csomag típust definiálunk. A programozható switchek ennek a protokollnak a segítségével tudnak kommunikálni egymással, így tudják a terheléselosztást megvalósítani. A korábban említett, hálózatban fellépő csomagvesztés a saját protokollunkba tartozó csomagokat is érinthet, azaz a switchek közötti kommunikáció során is elveszhetnek csomagok.

A switchek az OSI referencia modell második szintjén helyezkednek el, így saját MAC címmel nem rendelkeznek. Ezért a saját protokollunk használata során az Ethernet forrás, illetve cél MAC címet egy egyedi MAC címre állítjuk be, aminek köszönhetően a hosztok eldobják az ilyen típusú csomagokat, azaz a switchek nem zavarják a hosztok működését.

3.1. MAC tanulás és alapvető csomagtovábbítás

A megoldásunk kétféle tanulást és csomagtovábbítást végez. Egyrészt szükség van kifejezetten MAC címek tanulására, hogy broadcast (vagy ismeretlen, korábban elő nem forduló folyamatokhoz tartozó) csomagokat továbbítani lehessen. A dolgot ez hívjuk alapvető csomagtovábbításnak. A másik tanulási módszer kifejezetten egy-egy folyamathoz tanul útvonalat, erről később (a 3.5. szakaszban) lesz szó.

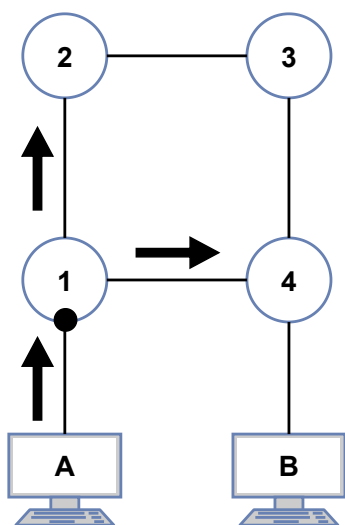
MAC tanulás alatt egy switchnek azon portjának a feljegyzését értjük, amelyik porton át érhető el egy adott MAC címmel rendelkező hoszt. A tanulás menete az alábbi: amennyiben egy switch egy ismeretlen forrás MAC címmel rendelkező csomagot kap, akkor elmenti a MAC címet és a bemeneti portot. Amennyiben már korábban hozzá lett rendelve egy port az adott MAC címhez, akkor nem történik tanulás.

Az alapvető csomagtovábbítás során nem vesszük figyelembe, hogy az adott csomag melyik folyamba tartozik, csupán az Ethernet fejlécek alapján továbbítjuk a csomagot. Amennyiben egy ismert cél MAC címmel rendelkező unicast csomagot kell továbbítani, akkor egyszerűen a már korábban feljegyzett porton küldjük ki azt.

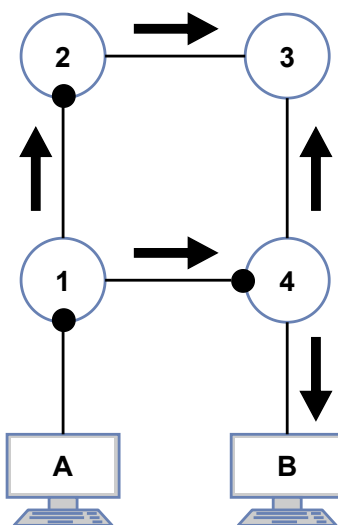
Hogyha broadcast vagy multicast a csomag, vagy a cél MAC címet korábban nem tanultuk meg, akkor a bemeneti portot is ellenőrizni kell. Ha a bemeneti port egyezik a (korábban) feljegyzett porttal, akkor a csomagot a bementi port kivételével minden porton kiküldjük (split horizon csomagtovábbítás). Erre azért van szükség, mert nem tudjuk, hogy pontosan melyik irányba továbbítsuk a csomagot. A cél az, hogy lehető legtöbb esetben sikeresen célba juttassunk csomagokat, így az egyetlen biztonságos megoldás a csomag minden irányba való kiküldése. Csak így tudjuk garantálni, hogy a csomag eljut a megfelelő hoszthoz. Split horizon csomagtovábbításra a 3.2. és a 3.3. ábrán látható példa.

Amennyiben viszont a bemeneti port nem az a port, amit korábban feljegyeztünk a forrás MAC címhez, akkor (feltéve, hogy nincs csomagvesztés) már megérkezett, vagy meg fog érkezni egy megegyező csomag a megtanult porton keresztül is. Ez a korábban bemutatott split horizon csomagtovábbítás következménye: egyes esetekben a csomagot minden lehetséges útvonalon továbbítjuk, és ha ezek az útvonalak egy switchben összeérnek, akkor az a switch a csomagot többször is meg fogja kapni. Emiatt a helytelen porton beérkező broadcast/multicast, illetve korábban meg nem tanult cél MAC címmel rendelkező unicast csomagokat el kell dobni. Ezzel nemcsak a duplikált csomagok hosztokhoz való továbbítását előzzük meg, hanem a broadcast vihar jelenséget is: ha a csomag eldobása helyett itt is split horizon módon továbbítanánk azt, akkor a hálózatban a csomag a

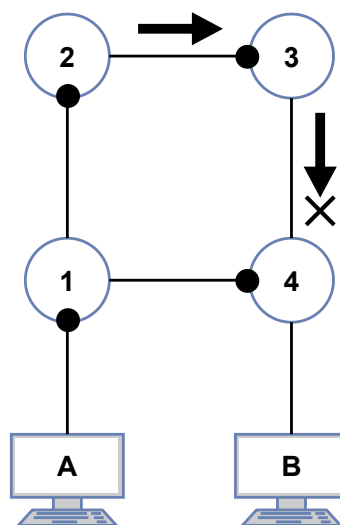
végtelenségig bent maradhatna, a switchek a végtelenségig továbbíthatnák azt egymásnak és a csomagok száma folyamatosan nőhetne. Helytelen porton keresztül fogadott csomagra a 3.4. ábrán látható egy példa.



3.2. ábra. Split horizon csomagtovábbítás: a bemeneti port elmentésre kerül, a többi porton továbbításra kerül a csomag. A nyilak csomagokat, a linkekhez tartozó pöttyök pedig a portok elmentését jelölik.



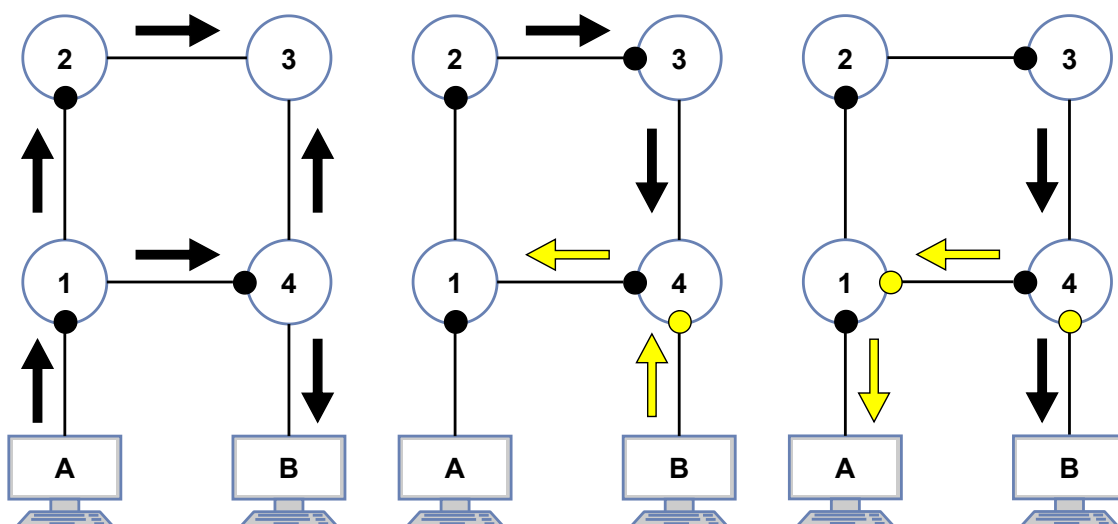
3.3. ábra. További példák a split horizon csomagtovábbítás működésére. A 3-as switchbe kétszer fog megérkezni a csomag. Feltételezzük, hogy a 2-es switchből induló csomag érkezik meg előbb.



3.4. ábra. A 3-as switch által továbbított csomagot a 4-es switch eldobja helytelen bemeneti port miatt. Ismételt split horizon csomagtovábbítás esetén a csomag újra eljutna az 1-es switchhez, azaz körbe érne.

A tanulás és a csomagtovábbítás algoritmusának következményeként, a hálózat indulásakor felléphet, hogy pár csomag többször érkezik meg a célhoz, azaz duplikált csomagok lépnek fel. Önmagában sem az ismert, sem az ismeretlen cél unicast MAC címhez tartozó csomagtovábbítás nem okozna ilyen problémát. Amennyiben viszont egy switch (az ismeretlen cél MAC cím miatt) több portján is kiküld egy csomagot, és egyes fogadó switchek már ismerik a cél MAC címet (mert nekik már volt lehetőségük megtanulni azt, azaz mert hozzájuk már eljutott egy csomag amit az adott hoszt küldött), ők mind továbbítani fogják a csomagot a cél hoszthoz. A duplikált csomagok nem jelentenek problémát, hiszen ez a viselkedés meg van engedve és mivel csak nagyon rövid ideig következhet be ez az eset (mert ha pár switch már megtanulta a cél MAC címhez tartozó portot, akkor hamarosan a többi, csomagtovábbításban érintett switch is meg fogja). A többször fogadott csomagok speciális esetét a 3.5. ábra szemlélteti.

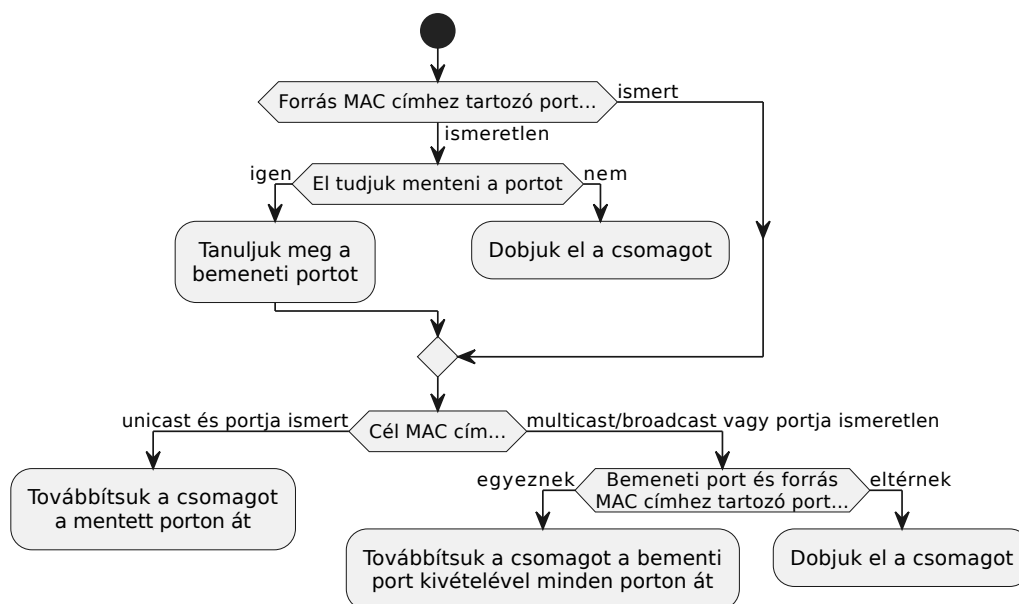
A MAC címekhez tartozó portok elmentése nem minden esetben lehetséges az adattárolási megoldástól függően. A dolgozatban egy olyan perzisztencia megközelítést mu-



3.5. ábra. Hoszthoz duplikált unicast csomag megérkezésének folyamata. A fekete nyilak például egy kérést tartalmazó csomagot, a sárga nyilak pedig a kérésre valamilyen választ reprezentálnak. A switch portokhoz tartozó pöttyök egy-egy megtanult MAC címet jelölnek.

tatunk be a Megvalósítás fejezetben, amely például hash ütközések és a P4 programozási nyelv limitációi miatt egyes kivételes esetekben, ideiglenesen, nem képes adatmentésre. Ha az alapvető csomagtovábbítás során a bemeneti port elmentése sikertelen, akkor a csomagot el kell dobni. Ezzel tudjuk garantálni, hogy a MAC címekhez tartozó megtanult portok egy összefüggő körmentes gráfot alkossanak (azaz egy fát, aminek a gyökere az adott MAC címmel rendelkező hoszt). Ha a csomag nem kerülne eldobásra, akkor egy hálózati hurok következtében egy switch a saját maga által továbbított csomagból tudna tanulni, ezzel egy kört tartalmazó útvonalat mentve el. A MAC tanulási folyamatnak és az alapvető csomagtovábbításnak a teljes algoritmusát a 3.6 ábrán van szemléltetve egy folyamatábra segítségével.

A MAC tanulási folyamat során elképzelhető, hogy aszimmetrikus útvonal kerül megtanulásra két hoszt között. Ez akkor történhet, hogyha a két hoszt egyszerre küldi ki az első csomagját. Ez csak nagyon ritkán fordulhat elő, és a következő szakaszokban bemutatott útvonal tanulás esetén (ami unicast csomagokra nézve lecseréli a MAC tanulás által meghatározott útvonalat) ilyen asszimmetria már nem léphet fel.



3.6. ábra. MAC tanulás és alapvető csomagtovábbítás algoritmus

3.2. Lehetséges terheléelosztási megközelítések

Hogy a folyamatokat a legkevésbé zavarjuk, azaz hogy a folyamatok hálózati késleltetése konstans maradjon, és a folyamathoz tartozó csomagok a kiküldésükkel azonos sorrendben érkezzenek meg, mi a lehető legkevesebbszer szeretnénk egy folyamatot egy új, korábbiútól eltérő útvonalra irányítani. Ezért a terheléelosztást úgy valósítjuk meg, hogy egy közel optimális útvonalat tanulunk meg, majd ha ezen a megtanult útvonalon torlódás lép fel, akkor egy újabb, kevésbé túlterhelt útvonalon keresztül továbbítjuk a folyamatot onnantól kezdve. Az útvonalváltásnak a megvalósítására több megközelítés lett azonosítva.

Egyrészt a tanulási folyamat során el lehetne menteni alternatív útvonalakat, amikre később lehet váltani. Ennek a megközelítésnek előnye, hogy az útvonalváltás akár nagyon gyorsan megtörténhet, viszont cserébe több memóriát igényel, hiszen ezeket az alternatív útvonalakat valahol tárolni kell. Továbbá a váltás pillanatában meg kéne tudni állapítani, hogy melyik alternatív útvonal a legjobb. Ennek megállapítása viszont csak az alternatív útvonalak folyamatos ellenőrzésével lehetséges: hiszen a hálózat terheltsége az útvonalak felfedezése óta megváltozhatott, így a korábban lementett terheltségről szóló adatok már nem feltétlenül aktuálisak.

Mi inkább egy másik megközelítést választottunk: alternatív útvonalakat csak akkor kutatunk fel, amikor szükség lenne rájuk (túlterheltség esetén), a hálózat pillanatnyi terheltségét ezáltal figyelembe véve. Azonban egy másik meghatározó kérdés is felmerült:

az útvonalak felderítése, tanulása során a switchek egyenrangú felek legyenek-e, vagy szükség van egy vezérlő kijelölésére, aki képes az útvonal tanulást koordinálni?

Megbízható útvonal tanulás teljesen egyenrangú switchekkel nem tűnt megvalósíthatónak. Például ha egy adott switch túl van terhelve és ezt jelzi az útvonalon az őt megelőző switchnek, akkor ennek a switchnek egy olyan új útvonalat kéne keresnie, amely kikerüli a túlterhelt switchet. Ehhez viszont a switchnek meg kéne tudni állapítania, hogy létezik-e rajta keresztül egy ilyen új útvonal, vagy pedig az alternatív útvonal keresését egy másik (az útvonal szempontjából az őt megelőző) switchre kéne bízni. Ennek a lokális döntésnek a meghozatala nem tűnt hatékonyan kivitelezhetőnek, ezért egy másfajta megközelítés mellett döntöttünk: a tanulás megkezdéséért mindig egy olyan switch felel, amelyik közvetlenül egy hoszthoz van csatlakoztatva. Ezek a switchek minden lehetséges útvonalnak részei, hiszen a hosztok csak rajtuk keresztül csatlakoznak a hálózathoz, így tőlük indulva biztosan fellelhető az összes lehetséges útvonal.

Tehát ha egy hálózatban egy adott folyamhoz egy switch (új) útvonalat szeretne igényelni, akkor ezt az egyik hoszthoz közvetlenül csatlakozó switchnek kell jeleznie. A két ilyen switch közül mégpedig annak kell küldenie a megfelelő igénylő csomagot, amelyik a kisebb numerikus értékű MAC című hoszthoz tartozik a cél és a forrás MAC címek közül. Ennek a definíciónak köszönhetően egy ismeretlen topológiában is könnyen megadható egy adott folyamhoz egy egyértelmű vezérlő.

3.3. Hoszthoz csatlakozó portok felismerése

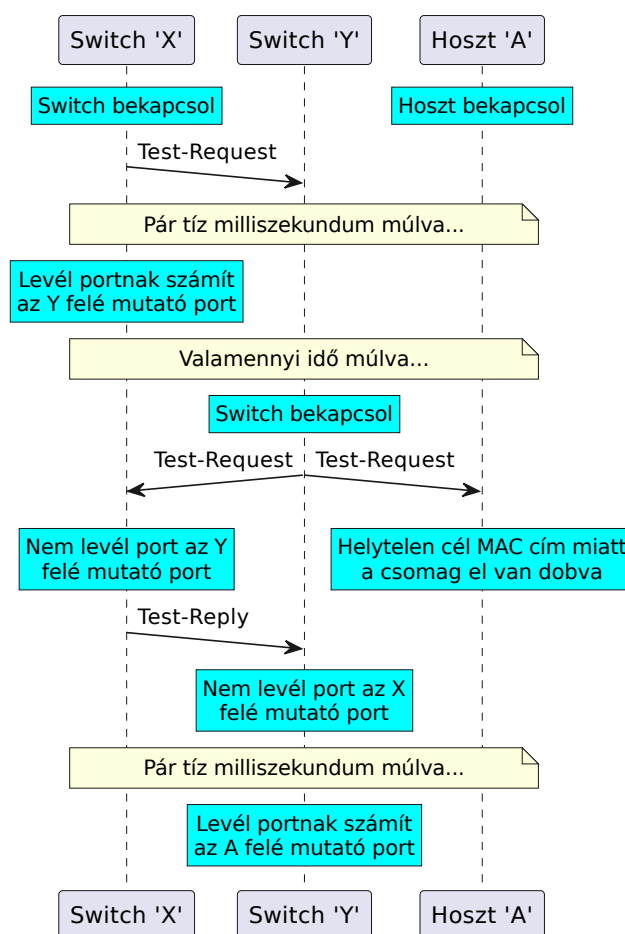
Levél portnak hívjuk a switchek azon portjait, amik közvetlenül egy hoszthoz csatlakoznak. Továbbá egy switchet egy adott MAC címre nézve levél switchnek hívunk, ha az adott MAC címhez tartozó hoszthoz közvetlenül csatlakozik.

Annak megállapítása érdekében, hogy a switchek mely portjai csatlakoznak közvetlenül egy hoszthoz, az alábbi eljárást hajtjuk végre.

- Indulásukkor a switchek az összes portjukon kiküldenek egy (egyedi, általunk definiált) *Test-Request* csomagot.
- Amennyiben egy switch egy *Test-Request* csomagot kap, akkor a bemeneti porton válaszként kiküld egy *Test-Reply* csomagot.
- Egy *Test-Request* vagy egy *Test-Reply* csomag fogadásakor a switch elmenti, hogy a bemeneti port nem egy levél port. A kapott csomag pedig eldobásra kerül.

- Egy port akkor számít levél portnak, hogyha a porton nem érkezett egy *Test* csomag sem, pedig a switch indulása óta eltelt elegendő idő ahhoz, hogy a kiküldött *Test-Request* csomagra válasz érkezzen.

Ennek az algoritmusnak köszönhetően egymáshoz képest jelentős késleltetéssel elindított switchek esetén is helyesen ismerjük fel a hosztokhoz csatlakozó portokat. Azokat a portokat is levél portnak nyilvánítjuk, amikhez nincsen semmi csatlakoztatva, de ez nem jelent problémát a terheléelosztási megoldásunk számára. Az algoritmus működésére példát a 3.7. ábrán láthatunk.



3.7. ábra. Példa levél portok megállapítására. A használt topológiában az 'X' switch és az 'A' hoszt közvetlenül az 'Y' switchhez csatlakoznak, és nincsen más link a hálózatban.

3.4. Folyamok azonosítása

A terheléelosztási megoldásunk lényege, hogy a különböző folyamatok egymástól eltérő útvonalakat tudjanak használni. Ennek érdekében szükség van a folyamatok egymástól való megkülönböztetésére, aminek a menetét ez a szakasz ismerteti.

Egy folyamat az alábbi fejléc értékek azonosítanak. Újabb protokollok támogatása érdekében további fejlécekkel könnyen kibővíthető az algoritmus. Amennyiben egy csomag nem tartalmaz egy adott fejléct, akkor a megfelelő értéket egy konstanssal kell helyettesíteni, például nullával. Így legrosszabb esetben (azaz egy nem implementált kapcsolatorientált protokoll esetén) más folyamba sorolható csomagokat egy közös folyamba sorolunk, ezzel csökkentve a terheléelosztás lehetséges hatékonyságát, de a csomagok továbbra is a küldésükkel azonos sorrendben fognak megérkezni.

- Forrás, illetve cél MAC cím
- Ethernet protokoll azonosító (*ether type*, például ARP vagy IPv4)
- Forrás, illetve cél IP cím
- IP protokoll azonosító (például UDP vagy TCP)
- Forrás, illetve cél UDP/TCP port

Amennyiben az IPv4 protokoll helyett az IPv6 protokoll van alkalmazva, akkor az IP protokoll azonosító, illetve a UDP/TCP portok vizsgálatára nincsen szükség: elég az IPv6 csomagok által használt folyam címke (*flow label*) [20] kiolvasása, ami magába foglalja az UDP/TCP fejlécekből kinyerhető adatokat. Ezáltal új, UDP-re vagy TCP-re alternatívát nyújtó szállítói réteg protokollok automatikusan támogatva vannak.

3.5. Folyamhoz tartozó útvonal tanulás alapjai

A korábban bemutatott MAC cím alapú csomagtovábbítás elsősorban a multicast, illetve a broadcast csomagok célba juttatására van használva. Unicast csomagok esetén szükség van arra, hogy a más-más folyamba tartozó csomagokat más-más útvonalon továbbítsuk, hiszen így érhető el terheléelosztás.

Broadcast, illetve multicast csomagok esetén nem végzünk terheléelosztást: a csomagokat minden switchhez továbbítani kell, hogy ők aztán az összes hozzájuk csatlakozott hoszthoz eljuttathassák azokat.

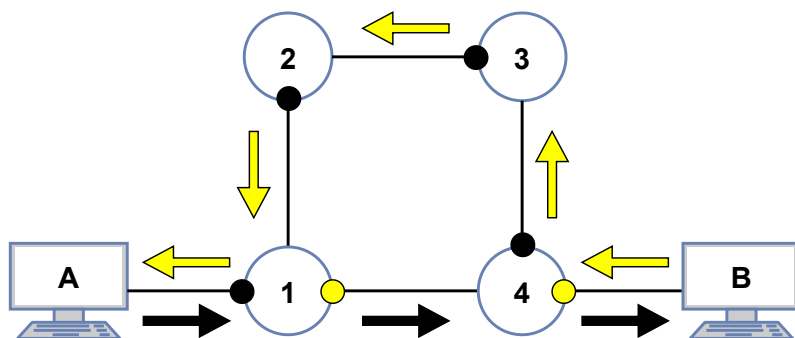
3.5.1. Tanulás megkezdésének okai, tanulás szükségességének esetei

Ahogy korábban említésre került, csak a folyam (forrás és a cél MAC címek közül) kisebb MAC címéhez tartozó levél switch képes egy tanulási folyamatot elindítani. Viszont bármelyik switch érzékelheti azt, hogy szükség lenne egy tanulási folyamatra, és küld-

het egy tanulás igénylési csomagot a megfelelő levél switch felé. Egy tanulási folyamat elindítására az alábbi esetekben van szükség:

- A switchnek egy adott folyamhoz (a feldolgozás alatt álló csomag folyamához) tartozó útvonal bejegyzése hiányos: nincs elmentve, hogy a folyamhoz tartozó két hoszt közül az egyiket (vagy mindkettőt) melyik porton keresztül kéne elérni. Minden folyamhoz a többi folyamtól különböző port páros van elmentve a switchekben, azaz például egy újabb folyam megjelenésekor az itt bemutatott okból egy levél switch biztosan fog egy tanulási folyamatot igényelni.
- A switchnek egy adott folyamhoz tartozó elmentett portja alapján a pillanatnyi csomag rossz porton keresztül lett fogadva. Ez az eset az aszimmetrikus útvonalak felismeréséért felel. Bár az algoritmus nem hozna létre aszimmetrikus útvonalakat, a switchek közötti kommunikációra használt (dolgozatban definiált) csomagok elvesztése eredményezheti ezt, ahogy ez majd a későbbiekben részletezve lesz. Új útvonal tanulásának következményeként előfordulhat, hogy pár csomag még a régi útvonalat követve érkezik a switchbe, ezért útvonalváltást követően életbe lép egy türelmi időszak, hogy tényleg csak aszimmetrikus útvonalak esetén legyen egy újabb tanulási folyamat igényelve. Amennyiben minden switch összehasonlítja a bemeneti portot és az elmentett (ellenkező iránybeli csomagtovábbításra használt) portot, akkor garantáltan érzékelve lesznek az aszimmetrikus útvonalak: ha egy útvonal aszimmetrikus, akkor a csomagok az egyik hosztól a másik hosztig eltérő útvonalon vannak továbbítva, mint a fordított irány esetén. Azaz két eltérő útvonal létezik, ezek viszont garantáltan találkoznak valahol, hiszen azonos hosztokból indulnak ki és érkeznak be. Ezekben a találkozási pontokban az említett port összehasonlítás garantáltan felismeri az ilyen aszimmetrikus útvonalakat. Az előbbiekre a 3.8. ábrán láthatunk példát.
- További, kiegészítő esetek (például túlterheltség esetén új útvonalra váltás) később lesznek ismertetve: először a tanulási algoritmus alapjai kerülnek bemutatásra.

Amennyiben ezekben az esetekben egy tanulási folyamatot indítunk, illetve a tanulási folyamat tényleg képes egy nem túlterhelt útvonalat találni, akkor a céljainkat elértük: minden folyamhoz külön-külön meg fogunk tanulni egy szimmetrikus, nem túlterhelt útvonalat, és az útvonal részét képező összes switch ismerni fogja a portokat, amiken keresztül a kisebb, illetve a nagyobb MAC című hoszthoz csomagokat tud továbbítani.



3.8. ábra. Példa egy aszimmetrikus útvonalra. A fekete nyilak az A hosztból B hosztba forgalmazott csomagokat, a fekete pöttyök pedig az A hoszthoz vezető megtanult portokat jelentik. A sárga jelölések az ellentétes irányt reprezentálják. Az aszimmetrikus útvonalat mind az 1-es, mind a 4-es switch felismeri, hiszen a port, amin keresztül az egyes csomagok fogadva lettek és a csomag forrás MAC-jéhez megtanult port eltér.

3.5.2. Tanulási folyamat

Tegyük fel, hogy a kisebb MAC című hoszthoz tartozó levél switch egy útvonal tanulási folyamatot szeretne elindítani. Ez megtörténhet akkor, hogyha maga a levél switch érzékeli az útvonal tanulás szükségességét, vagy akkor, hogyha egy útvonal tanulási igény eljut hozzá. Annak ismertetése, hogy egy ilyen útvonal tanulást igénylő üzenet hogyan jut el a levél switchhez, a 3.5.3. szakaszban történik.

Ez a szakasz a tanulási folyamat algoritmusának egy egyszerűsített változatát ismerteti. Az algoritmust a következő szakaszokban újabb funkciókkal egészítjük ki, viszont az alábbiakban ismertetett lényegi része változatlan marad.

Csak a kisebb MAC című hoszthoz tartozó levél switch indíthat el egy tanulási folyamatot egy adott folyamathoz, és ő is csak akkor, hogyha az előző tanulási folyamat ehhez a folyamathoz (ha volt már ilyen) elég régen (például pár száz milliszekundummal ezelőtt) történt, azaz biztosan véget ért már. Ez a feltétel egyrészt a párhuzamos tanulási folyamatok kizárásáért felel, hiszen ezek meg tudnák zavarni egymást (például a különféle türelmi időszakok esetében). Másrészt a tanulás során el kell tudni dönteni, hogy egy kapott csomag a pillanatnyi tanulási folyamathoz tartozik (csak későn érkezett meg a többi csomaghoz képest), vagy pedig egy újabb tanulási folyamat része.

A tanulási folyamat első lépéseként a levél switch kiküld egy *Learn* csomagot az összes olyan portján, amely switchhez csatlakozik (azaz amely nem levél port). Ezen felül a folyam útvonalához tartozó memória bejegyzésben beállítja, hogy a kisebb MAC címhez melyik port tartozik. Ezt a portot a switch ismeri (a MAC tanulási folyamatnak köszönhetően), hiszen csak a port ismerete esetén tudja a switch kimondani magáról, hogy

ő egy levél switch.

Amennyiben a switch nem csak a folyam kisebbik MAC címére nézve levél switch, hanem a nagyobbik MAC címre nézve is, akkor a tanulási folyamat egyetlen lépésből áll. Ebben a lépésben nincs is szükség *Learn* csomagok küldésére, csupán az útvonal bejegyzésben be kell állítani, hogy a két MAC címhez mely portok tartoznak, és ezzel kész is az útvonal tanulás.

Hogyha egy switchhez egy *Learn* csomag érkezik, és a switch a folyam nagyobbik MAC címére nézve nem egy levél switch, akkor háromféle eset léphet fel. Az első esetben a switch még soha nem kapott az adott folyamhoz tartozó *Learn* csomagot, a második esetben pedig bár már kapott ilyen csomagot, azóta már eltelt annyi idő, hogy egy újabb tanulási folyamat elindulhatott. Ebben a két esetben a bemeneti portot el kell menteni a folyamhoz tartozó útvonal bejegyzésbe, mint a kisebb MAC címhez tartozó port, illetve a csomagot továbbítani kell a bementi port kivételével az összes porton át. Amennyiben viszont a switch nem sokkal korábban már kapott egy azonos folyamhoz tartozó *Learn* csomagot, akkor ezt a második *Learn* csomagot el kell dobni: ez a duplikált csomag egy lassabb útvonalon érkezett, a switch pedig már továbbította a gyorsabb útvonalhoz tartozó *Learn* csomagot.

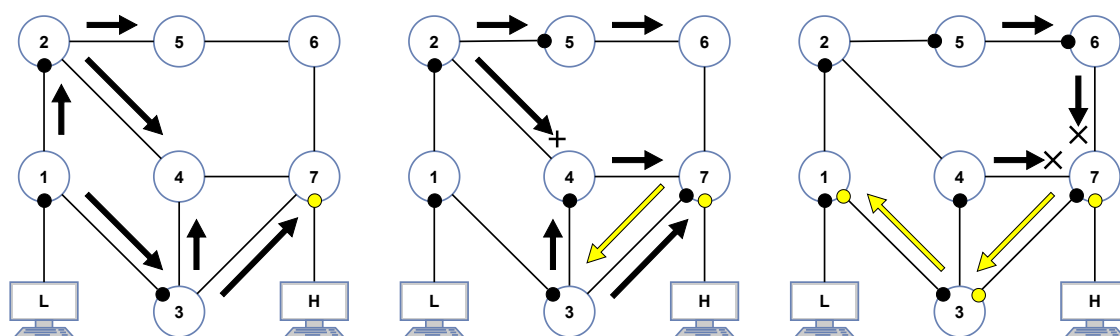
Végül a *Learn* csomag eljut a nagyobbik MAC címhez tartozó levél switchhez. Ilyenkor a switch a korábban ismertetett módon vagy eldobja a csomagot, vagy megtanulja a bemeneti portot, egy kivétellel: a *Learn* csomag split horizon módon való továbbítása helyett csak egyetlen csomagot küld a switch: egy *Confirmation* csomagot a bemeneti porton keresztül. Amikor egy switch egy *Confirmation* csomagot kap, akkor a bemeneti portot megtanulja, mint a nagyobbik MAC címhez tartozó port, és a *Confirmation* csomagot továbbítja a korábban megtanult, kisebbik MAC cím felé mutató porton keresztül. A kisebbik MAC címhez tartozó levél switch már nem továbbítja a *Confirmation* csomagot, hanem eldobja azt, ezzel pedig a tanulási folyamat véget ér.

A *Learn* vagy a *Confirmation* csomagok elvesztése esetén előfordulhat, hogy egy nem optimális útvonal kerül megtanulásra, vagy akár az is, hogy a tanulási folyamat eredményeként nem kerül egy útvonal se megtanulásra. Az utóbbi esetben részben a már korábban megtanult útvonal, ha van ilyen, marad életben, hiszen nincs ami felülírja azt. Ez viszont aszimmetrikus útvonalak kialakulásához vezethet (csak *Confirmation* csomagok elvesztése esetén alakulhat ki aszimmetrikus útvonal). Ha a megtanult útvonal aszimmetrikus, vagy sem korábban, sem pedig a szóban forgó tanulási folyamat során nem sikerült egy útvonalat megtanulni az adott folyamhoz, akkor a Tanulás megkezdésének okai sza-

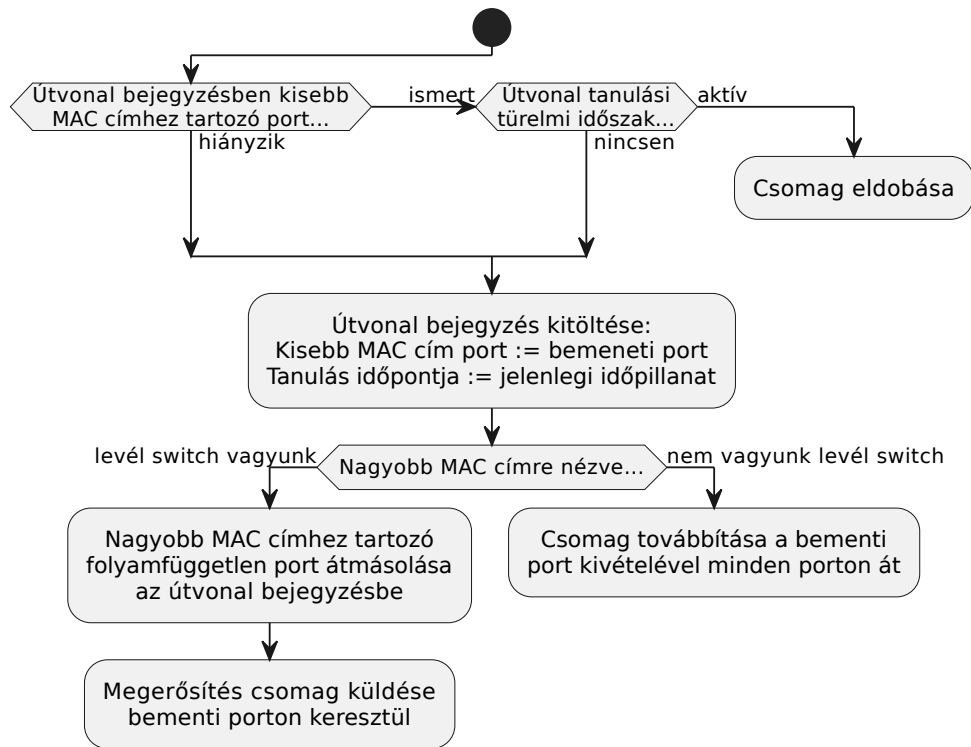
kaszban ismertetett egyik eset életbe lép és legalább az egyik switch egy újabb tanulási folyamatot fog igényelni, ami (amennyiben sikeres) ki fogja javítani a problémát.

Az adattárolás megvalósításától függően, a MAC tanuláshoz hasonlóan az útvonal-bejegyzések megtanulásánál is előfordulhat, hogy ideiglenesen nem sikerül egy új folyamathoz egy útvonal bejegyzést létrehozni (lefoglalni). Ilyenkor a csomagot figyelmen kívül hagyjuk: úgy teszünk, mintha az adott *Learn* csomag meg sem érkezett volna a switch-hez, hanem a csomagvesztés következtében eldobásra került volna. Ennek az indoklása egyezik a MAC tanulásnál ismertetett indoklással: ha a *Learn* csomagokat ilyen esetben is továbbítanánk, akkor nem tudnánk garantálni, hogy az algoritmus egy körmentes útvonalat tanul meg.

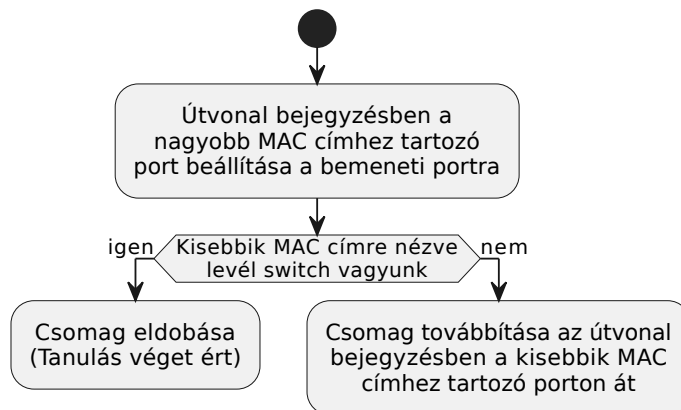
Összefoglalva, a tanulás során először a hálózatot a kisebbik MAC címhez tartozó levél switchtől kezdve elárasztjuk *Learn* csomagokkal. Minden switch akkor továbbítja split horizon módon a kapott *Learn* csomagot (és akkor tanulja meg a bemeneti portot), ha ez az első *Learn* csomag, amit megkapott az adott tanulási folyamat során. Ennek köszönhetően a leggyorsabb útvonalak vannak tovább építve. A nagyobbik MAC címhez tartozó levél switchhez leggyorsabban eljutó *Learn* csomagra a levél switch egy *Confirmation* csomaggal válaszol, ami a korábban megtanult portokon keresztül van továbbküldve, vissza egészen a kisebbik MAC címhez tartozó levél switchig. A switchek a *Confirmation* csomag segítségével tanulják meg a nagyobbik MAC cím felé mutató portjukat. A tanulási algoritmus működés közben a 3.9. ábrán látható. A *Learn* és a *Confirmation* csomagok feldolgozásának algoritmusai pedig a 3.10. és a 3.11. ábrákon is megtekinthetők. A MAC tanulást és a megtanult útvonalat is figyelembe vevő csomagtovábbítási algoritmus a 3.12. ábrán látható.



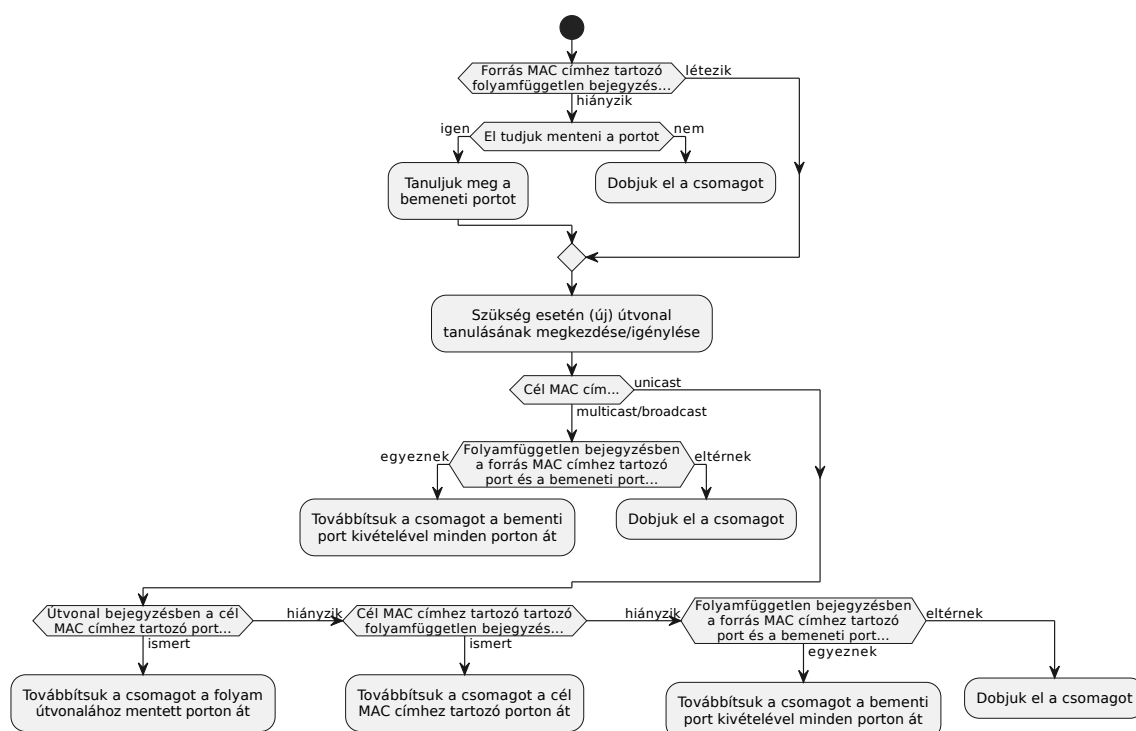
3.9. ábra. Tanulási algoritmus működése körökert tartalmazó hálózatban. Az útvonal az L és a H hosztokhoz tartozó levél switchek között kerül kiépítésre, ahol L a kisebb MAC címmel rendelkező hoszt. A fekete nyilak *Learn*, a sárga nyilak pedig *Confirmation* csomagokat jelölnek. A keresztkben végződő nyilak a csomagok eldobását jelzik. A portokhoz tartozó pöttyök pedig a megtanult portokat szimbolizálják.



3.10. ábra. *Learn* csomagok feldolgozásának algoritmus



3.11. ábra. *Confirmation* csomagok feldolgozásának algoritmus



3.12. ábra. Csomagtovábbítás algoritmus

3.5.3. Levél switch értesítése

A 3.5.1. szakaszban ismertettük, hogy a switchek milyen esetekben érzékelik, hogy útvonal tanulásra van szükség. Azonban mivel csak a kisebb MAC címhez tartozó levél switch képes egy útvonal tanulási folyamatot elindítani, így amennyiben az útvonal tanulást igénylő switch nem ez a levél switch, valahogyan egy értesítést kell küldenie. Ez a szakasz ennek a *Request* csomagnak a kisebb MAC címhez tartozó levél switch felé való küldésével, továbbításával foglalkozik.

A switchek új tanulási folyamatot csak bizonyos időközökkel (ami lehet például a tanulás indítási türelmi időszak tizede) igényelhetnek (azaz két új *Request* csomag kiküldése között valamennyi időnek el kell telnie, akkor is, ha eltérő folyamathoz tartoznának a csomagok). Ha ez a türelmi időszak nem létezne, akkor amíg útvonal tanulásra valamilyen okból szükséges, addig a switchek minden fogadott csomag esetén (annak továbbításával párhuzamosan) egy *Request* csomagot kiküldhetnének. Ezzel a hálózat feleslegesen lenne terhelve: tanulási folyamat csak bizonyos időközönként indítható (párhuzamos tanulási folyamatok elkerülése érdekében), ezért rövid időn belül több *Request* csomag küldése redundáns. Viszont fontos, hogy a switchek addig folyamatosan igényeljenek egy tanulási folyamatot, amíg az tényleg el nem kezdődik: a *Request* csomag elveszhet a hálózatban,

ilyenkor pedig a levél switch soha nem értesül arról, hogy el kéne indítania egy tanulási folyamatot. Továbbá ha a levél switchhez el is jut az igénylés, bizonyos okokból (például a tanulás indítási türelmi időszak miatt) nem feltétlenül fog egy tanulási folyamat elindulni. A *Request* csomag periodikus küldésével garantáljuk, hogy szükség esetén, akár valamilyen késleltetéssel, de biztosan elinduljon egy tanulási folyamat.

Az algoritmusunknak olyan folyamatokat is támogatnia kell, amik csak egy irányban forgalmaznak csomagokat (például a nagyobb MAC címmel rendelkező hoszttól a kisebb MAC címmel rendelkező hoszt felé), ezért egy switch nem tud egy csomagot triviálisan a kisebb MAC címhez tartozó levél switchnek küldeni: a switch nem feltétlenül tudja, hogy melyik portján keresztül érheti el ezt a levél switchet, ha még egy tanulási folyamat sem ment végbe. Ezen okból kifolyólag a MAC tanulás során elmentett folyamfüggetlen bejegyzések sincsenek felhasználva a levél switch értesítése során.

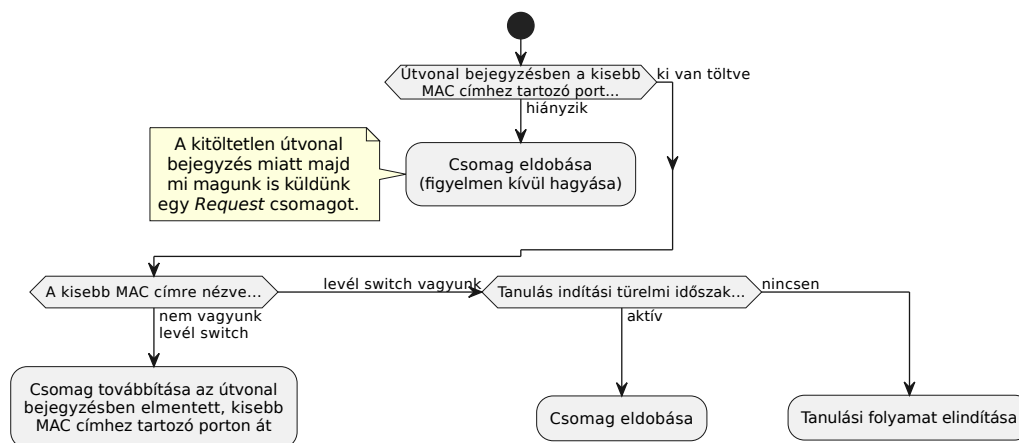
Amennyiben a pillanatnyilag feldolgozás alatt álló csomag Ethernet forrás MAC címe megegyezik a folyam kisebb MAC címével, akkor a kisebb MAC címmel rendelkező levél switch a bemeneti porton keresztül érhető el, a *Request* csomagot ezen a porton keresztül kell küldeni. Hasonlóan egyszerű a helyzet, hogyha a switch az útvonal tanulás során már elmentette a kisebb MAC címhez tartozó portot: ekkor ezen a mentett porton keresztül kell a *Request* csomagot küldeni. Hogyha viszont egyik eset sem áll fenn, akkor a switch a bemeneti port kivétel az összes portján keresztül kénytelen egy *Request* csomagot küldeni. Az ismertetett esetekben biztos, hogy a csomagot fogadó switchek közül az egyik vagy maga a levél switch (aki a tanulási folyamatot el tudja indítani), vagy pedig a pillanatnyi switch-től a levél switchhez közelebb található. A *Request* csomag küldési algoritmus a 3.13. ábrán is megtekinthető.



3.13. ábra. Új *Request* csomagok küldésének algoritmus

Amikor egy nem levél switch megkap egy *Request* csomagot, akkor azt továbbítania kell a kisebb MAC címhez tartozó levél switch felé. Ha a folyamhoz tartozó útvonal bejegyzés nem tartalmazza, hogy melyik porton keresztül érhető el ez a levél switch, akkor

a csomagot viszont el kell dobni. Ez azért nem okoz problémát, mert ilyen esetben majd maga a csomagot eldobó switch fog küldeni egy *Request* csomagot a hiányos útvonal bejegyzése miatt. Ha a csomag eldobása helyett azt split horizon módon az összes, bemeneti porttól eltérő porton továbbítaná a switch, akkor könnyen broadcast vihart okozhatna az algoritmus a hálózatot ilyen csomagokkal: egy három switchből álló kör folyamatosan ilyen csomagokat tudna továbbítani egymásnak. A *Request* csomagok továbbításának algoritmus a 3.14. ábrán látható.



3.14. ábra. *Request* csomagok feldolgozásának algoritmus

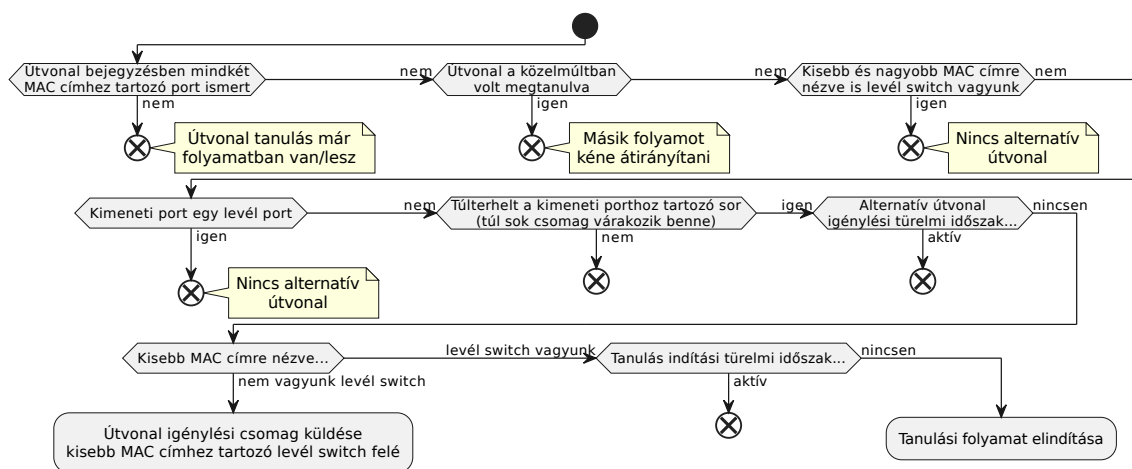
Összefoglalva, ha egy switch egy tanulási folyamatot szeretne igényelni (mert például még nem tanulta meg, hogy a pillanatnyilag feldolgozás alatt álló csomagot hogyan kell továbbítani), akkor a kisebb MAC címhez tartozó levél switchhez kell egy *Request* csomagot eljuttatnia. Az igénylő switch ezt a *Request* csomagot a legrosszabb esetben elküldi több szomszédos switchnek, akik aztán vagy eldobják a csomagot, vagy továbbítják azt a levél switch felé (vagy pedig ők maguk a levél switch). Garantált, hogy a *Request* csomagot fogadó switchek közül legalább az egyik képes továbbítani a csomagot, vagy pedig ő maga is fog előbb-utóbb egy útvonal tanulást igényelni a hiányos útvonal bejegyzése miatt. Így garantált, hogyha bármelyik switch egy útvonal tanulási folyamatot igényel egy aktív (csomagokat forgalmazó) folyamathoz, akkor a tanulási folyamat el lesz indítva, legfeljebb valamilyen késleltetéssel.

3.6. Túlterheltség érzékelése és útvonal váltás

A switchek nem csak a korábban bemutatott esetekben kezdeményezhetik egy újabb tanulási folyamat elindítását. Bár az eddig bemutatott algoritmus felfogható az ARP-Path

[11], illetve a TCP-Path [14] átalakított és protokollfüggetlen változatának, a terhelésselosztási hatékonyság lényegesen javítható.

A korábban (3.5.1. szakaszban) ismertetett eseteken túl akkor is új tanulási folyamatot kezdeményez egy switch, ha érzékeli, hogy túl van terhelve. A túlterheltség a kiküldésre váró csomagok sorának telítettsége alapján van meghatározva: ha a sorban túl sok csomag található (avagy, ezzel ekvivalensen megfogalmazva: ha egy csomag túl sok időn át várakozott a sorban, azaz a csomag késleltetése túl nagyra nőtt), akkor a switch (pontosabban a switchnek az adott sorhoz tartozó portja) túl van terhelve. Az ehhez tartozó tanulási folyamat igénylési algoritmus a 3.15. ábrán tekinthető meg.



3.15. ábra. Túlterhelt switch esetén alternatív útvonal igénylésének algoritmus

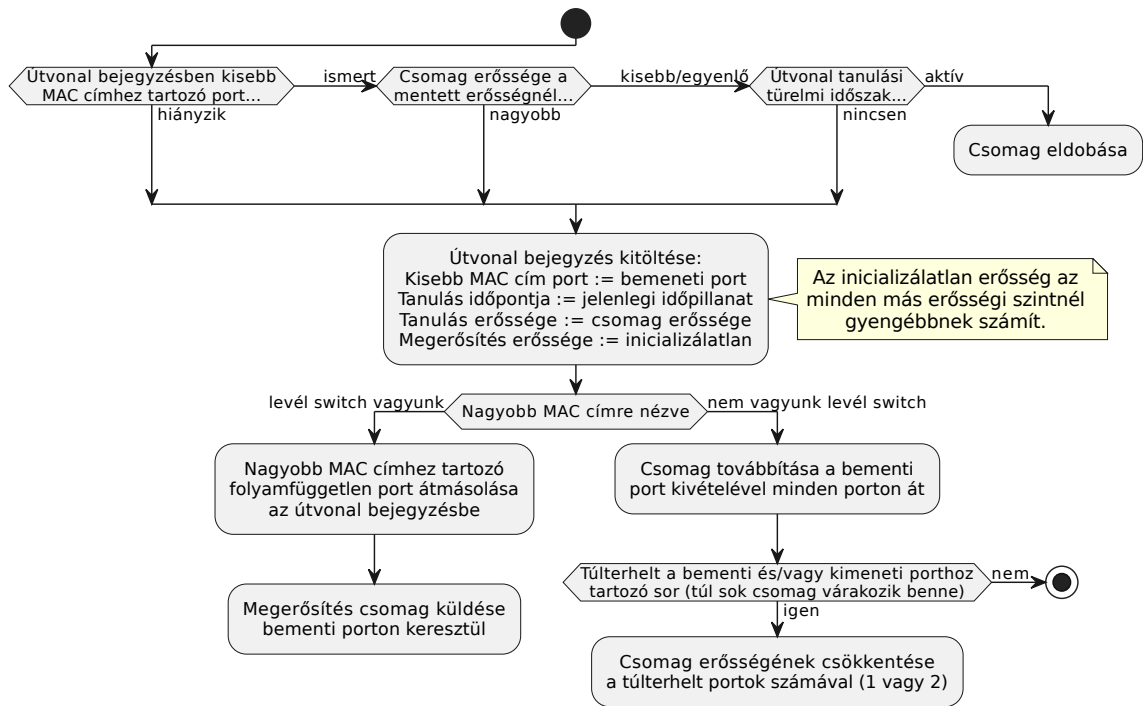
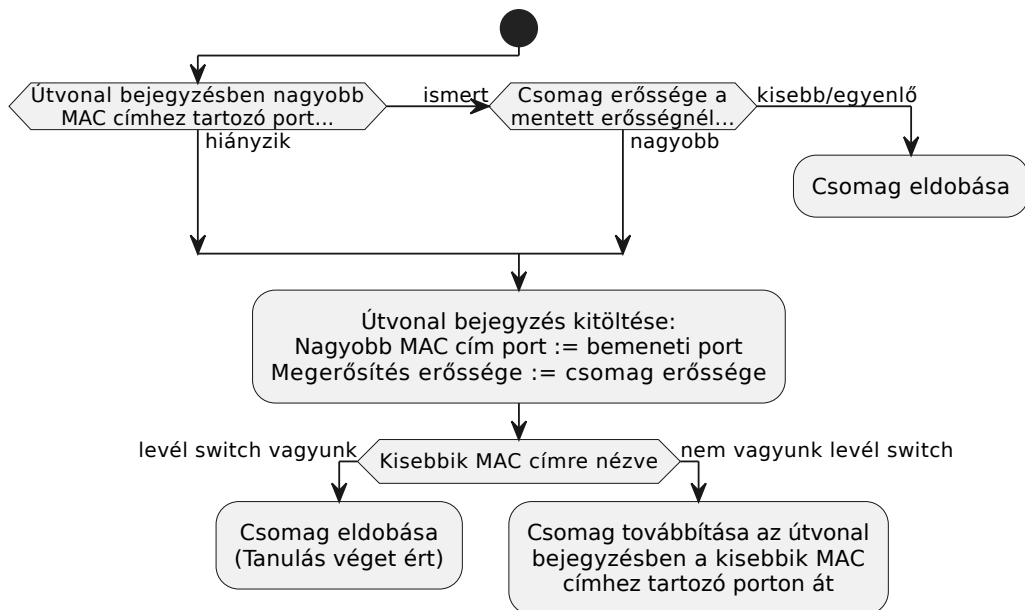
Önmagában egy újabb tanulási folyamat elindítása, egyéb változtatások nélkül azonban még nem elegendő. Bár a túlterhelt switcheken a késleltetés megnő, és az elárastáson alapuló, leggyorsabb útvonalat meghatározó tanulási algoritmus tényleg nagyobb eséllyel fog egy alternatív, nem túlterhelt útvonalat meghatározni az újratanulás során, ez nem minden esetben következik be. A tanulási folyamat során csupán a *Learn* csomagok megérkezési sorrendje számít, azaz a kisebb MAC címhez tartozó levél switchtől a nagyobb MAC címhez tartozó levél switchhez küldött csomagok által elszenvedett késleltetés mértéke határozza meg az útvonalat. Ahogy korábban említésre került, a túlterhelt switchek esetén csupán egy-egy porton nő a késleltetés, azaz a switchen csak az egyik irányban áthaladó csomagok késleltetése nő meg. Amennyiben az a port van túlterhelve, amin keresztül a *Learn* csomag továbbítva van a nagyobb MAC címhez tartozó levél switch felé, akkor a korábban bemutatott tanulási algoritmus tényleg inkább egy alternatív útvonalat fog megtanulni. Viszont hogyha a *Learn* csomag a túlterhelt porton keresztül van fogadva, és egy másik (nem túlterhelt) porton keresztül van továbbítva, akkor csak a hálózatot

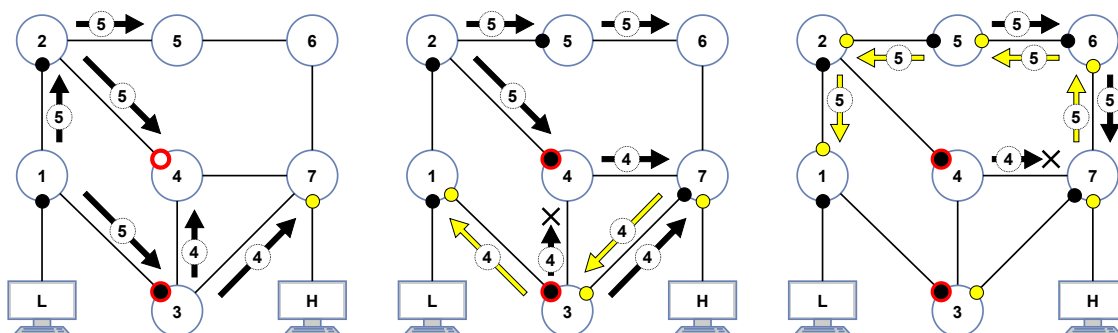
ellenkező irányban bejáró *Confirmation* csomag szenved késleltetést, ahol viszont a késleltetés nem számít, a tanulási folyamatot nem befolyásolja. (A *Learn* csomagra ebben az esetben azért nem hat a késleltetés, mert a switchek általában csak a kiküldendő csomagokat tárolják sorban, a beérkező csomagokat rögtön fel tudják dolgozni.)

Tehát a tanulási folyamat során a *Learn* csomagoknak nem csak a switchek kimeneti portjain fellépő késleltetést kéne figyelembe venniük (ami automatikusan megtörténik, hiszen ezen késleltetés miatt később érnek el a levél switchhez), hanem a bemeneti porthoz tartozó sor telítettsége miatti, a *Learn* csomagra nem ható késleltetést is. Ennek érdekében a *Learn* (és a velük azonos felépítésű *Confirmation*) csomagokat kiegészítettük egy erősség mezővel, ami a *Learn* csomag által bejárt útvonal minőségét jelöli (nagyobb erősség értékek jobb, kevésbé túlterhelt útvonalakat jelölnek).

A tanulási folyamat megkezdésekor mindig maximális erősségű *Learn* csomagokat hoznak létre a levél switchek. Amikor egy switch továbbküld egy *Learn* csomagot, akkor továbbítás előtt megnézi, hogy a *Learn* csomagot fogadó bemeneti, vagy a *Learn* csomaghoz rendelt kimeneti port túl van-e terhelve. Túlterhelt port(ok) esetén a csomag erőssége egy-egy értékkel csökkentésre kerül és így van továbbítva. Továbbá *Learn* csomagok fogadása során már nem csak a csomagok beérkezési sorrendje számít, hanem azok erőssége is: amíg korábban mindig csak az első (leggyorsabb) *Learn* csomag került feldolgozásra (továbbításra), addig most már az elsődleges szempont a csomag erőssége lett, a beérkezési sorrend pedig csak azonos erősségű csomagok esetén számít. A *Learn* csomagok feldolgozásának frissített algoritmus a 3.16. ábrán található.

A *Confirmation* csomagok feldolgozása is megváltozott: korábban a switchek a beérkezett *Confirmation* csomagot mindig feldolgozták (hiszen egy tanulási folyamat során egy switch maximum egy darab ilyen csomagot kaphatott), most már viszont egy switch többször is kaphat *Confirmation* csomagot, és csak akkor továbbítja őket, ha azok erőssége az előzőnél (ha volt ilyen) nagyobb. Több *Confirmation* csomag ugyan annak a tanulási folyamatnak a részeként úgy keletkezhet, hogy a nagyobb MAC címhez tartozó levél switchhez először egy gyengébb, utána pedig egy erősebb *Learn* csomag érkezik. Ilyenkor mindkét *Learn* csomagra válaszolni fog a levél switch egy-egy *Confirmation* csomaggal, hiszen a gyengébb *Learn* csomag fogadásakor még nem tudja a levél switch, hogy később egy erősebb csomag is meg fog érkezni. A *Confirmation* csomagok feldolgozásának frissített algoritmus a 3.17. ábrán tekinthető meg, a 3.18. ábrán pedig a frissített tanulási folyamatra egy példa látható.


3.16. ábra. *Learn* csomagok feldolgozásának (frissített) algoritmus

3.17. ábra. *Confirmation* csomagok feldolgozásának (frissített) algoritmus



3.18. ábra. Frissített tanulási algoritmus működése. A túlterhelt portok piros gyűrűkkel vannak jelölve. A fekete nyilak *Learn*, a sárga nyilak pedig *Confirmation* csomagokat jelölnek, a hozzájuk tartozó szám az erősségüket jelzi. A keresztben végződő nyilak a csomagok eldobását jelzik.

3.7. Nem használt útvonalak elfelejtése

A switchek működésük során számos folyamfüggetlen, illetve (folyamfüggő) útvonal bejegyzést tanulnak meg: a lokális hálózatban akár az összes hoszt MAC címét (és a hozzá tartozó portot) megtanulják, illetve minden egyes, az adott switchen áthaladó folyamhoz megtanulnak egy-egy útvonal bejegyzést. Bár a lokális hálózatban csak véges sok hoszt található, még mindennapi használat során is nagyon sok különböző folyam (eltérő folyam azonosítóval rendelkező csatorna/kapcsolat) van igénybe véve: eltérő IP címekkel való kommunikációk külön-külön folyamoknak számítanak, illetve egy adott szolgáltatáshoz való TCP kapcsolat ismételt megnyitása is sokszor egy eltérő folyamnak számít, hiszen a TCP kapcsolat forrás portja általában egy operációs rendszer által választott véletlenszerű szám, azaz az előző TCP kapcsolat forrás portjától különbözik, ami egy eltérő folyam azonosítót eredményez.

Ha a folyam azonosítóknak csak egy részét, a 32 bites forrás/cél IPv4 cím, illetve a 16 bites forrás/cél TCP/UDP port mezőit vesszük figyelembe, már akkor is nagyon sok eltérő folyamat tudunk megkülönböztetni: $2 * 2^{32} * 2 * 2^{16} = 2^{50} = 1,13 * 10^{15}$ darabot. Ennyi eltérő útvonal bejegyzés párhuzamos tárolására egy switchnek feltehetően nincsen elegendő memóriája. A probléma orvoslása érdekében a már régóta nem használt folyamokhoz tartozó bejegyzéseket a switcheknek ki kéne törölniük, hogy a felszabadult memóriát a ténylegesen aktív, folyamatban lévő kapcsolatokhoz tartozó útvonal bejegyzések tudják felhasználni.

Amennyiben útvonal tanulás (azaz egy útvonal bejegyzés) számára nincs elég memória, az bár ellehetetleníti a terheléelosztás teljes funkcionalitásának igénybevételét, de

csomagvesztést önmagában nem okoz: a csomagokat a 3.1. szakaszban bemutatott alapvető csomagtovábbítás segítségével továbbra is képes a switch továbbítani. Hogyha viszont folyamfüggetlen bejegyzések, azaz MAC tanulás számára sincsen szabad memória a switchben, akkor csomagvesztés fog bekövetkezni, ahogy az az alapvető csomagtovábbítás algoritmusáról (3.6. ábra) leolvasható.

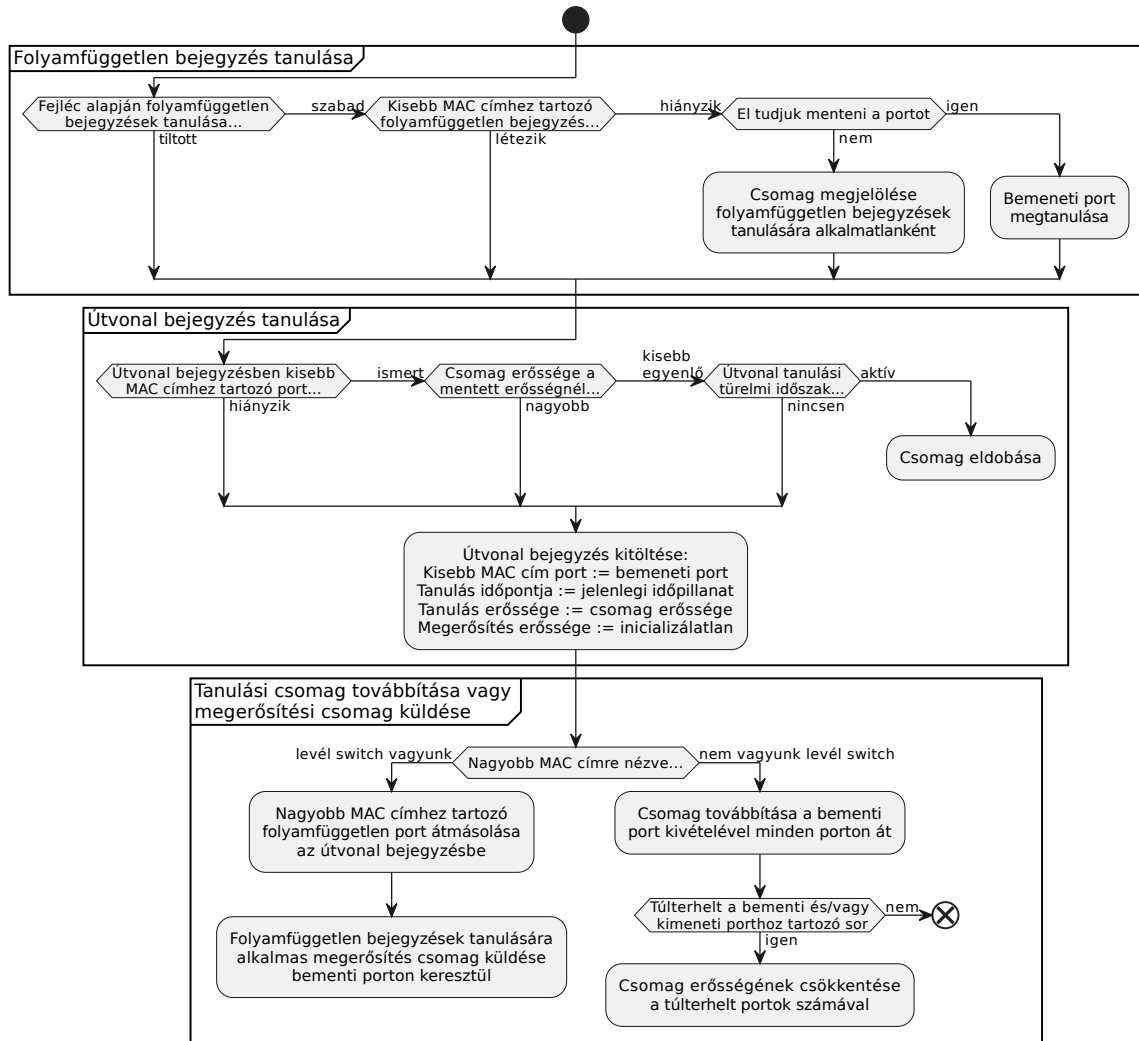
A már korábban megtanult, viszont jelenleg nem használt folyamatokhoz tartozó útvonal bejegyzések törlése során fontos szempont, hogy csomagvesztés akkor se következessen be, ha a folyamat újra használatba lép a törlés után. Bár, ahogy korábban említésre került, útvonalak megtanulása nélkül is lehetséges a csomagvesztés nélküli csomagtovábbítás, a már megtanult útvonal bejegyzések törlése képes csomagvesztést okozni (ha az algoritmus nincsen módosítva). Erre akkor kerülhet sok, ha a szóban forgó folyamat csak egy irányban forgalmaz csomagokat. Ilyenkor a cél MAC címhez vezető folyamfüggetlen bejegyzések nem lesznek megtanulva, hiszen a MAC cím tanulás csak a csomagok forrás MAC címét képes feljegyezni. Az alapvető csomagtovábbítás algoritmusában ebben az esetben split horizon módon továbbítja a csomagokat, azaz amennyiben a bemeneti port egyezik a forrás MAC címhez megtanult porttal, akkor a csomag az összes többi porton keresztül van továbbítva. Hogyha viszont a ki nem törölt útvonal bejegyzések segítségével egy csomag eljut egy olyan switchhez, ami kitörölte az adott útvonal bejegyzést, akkor nem feltétlenül azon a porton érkezett meg a csomag, ami a folyamfüggetlen útvonal bejegyzésben a forrás MAC címhez fel van jegyezve, azaz ilyenkor a csomagot nem lehet split horizon módszer segítségével továbbítani, hanem el kell dobni.

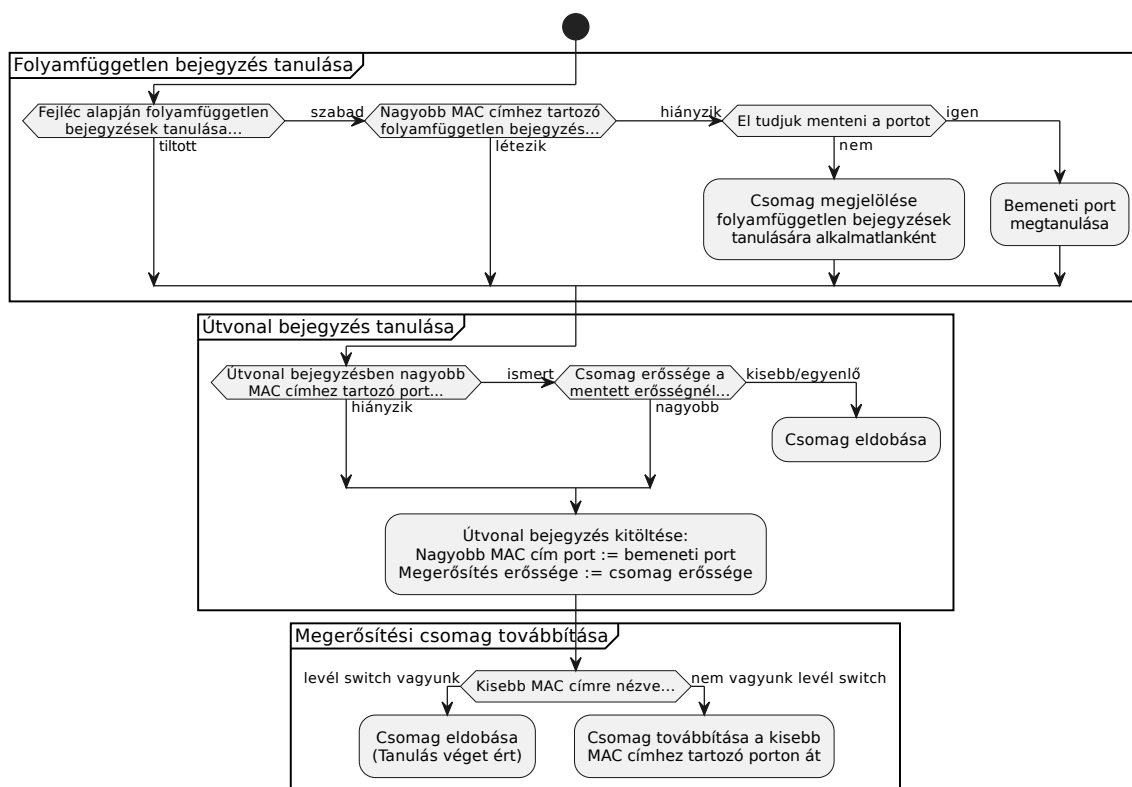
A útvonal bejegyzések felszabadítása (törlése) miatti csomagvesztés megszüntetése érdekében az útvonal tanulási algoritmust egészítettük ki. Az útvonal tanulási folyamat az eddigiekkel azonos módon működik, viszont nem csak útvonal bejegyzések kerülnek megtanulásra, hanem velük párhuzamosan a folyamat két MAC címéhez tartozó folyamfüggetlen bejegyzések is. A megtanult útvonalon elhelyezkedő switchek elmentik az említett folyamfüggetlen bejegyzéseket, így az útvonal bejegyzés kitörlése esetén a csomagok továbbítása megvalósítható lesz a folyamfüggetlen bejegyzések segítségével.

A *Learn* csomagok a kisebb MAC címhez tartozó levél switchtől indulnak ki, így egy *Learn* csomag feldolgozása során a bemeneti port elmenthető a kisebb MAC címhez tartozó folyamfüggetlen bejegyzésbe: erre a mentésre akkor van szükség, ha még nincsen semmilyen port hozzárendelve az adott MAC címhez. A *Learn* csomagok mintájára a *Confirmation* csomagok segítségével a nagyobb MAC címekhez tartozó portok tanulhatóak meg. Azonban a két csomagtípus feldolgozása során is, a MAC tanulás al-

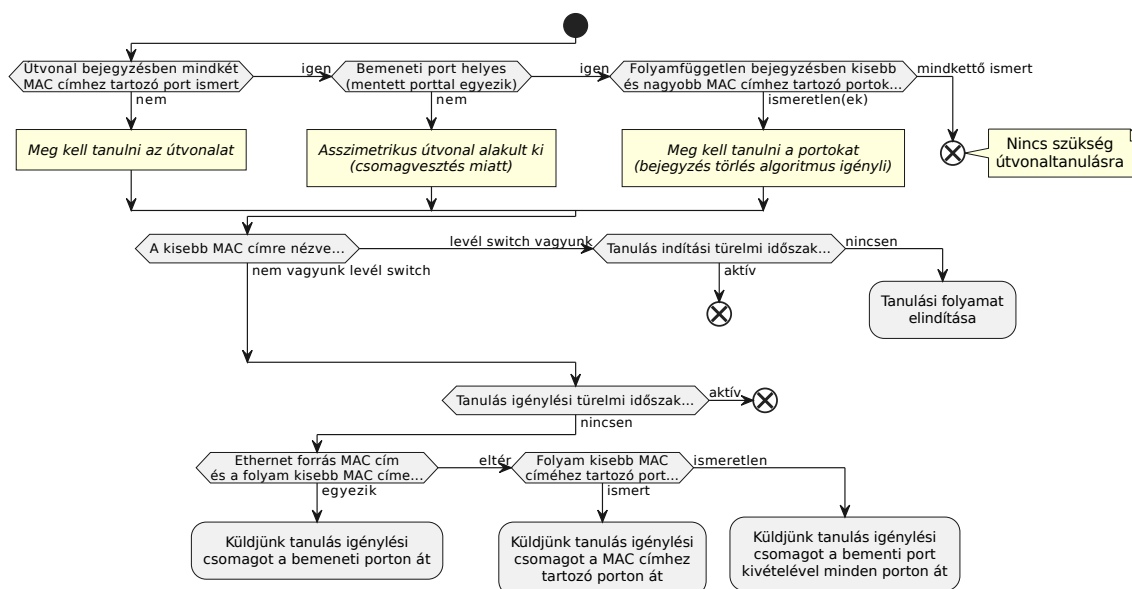
goritmusához hasonlóan, a modell megengedi a megvalósításnak, hogy az adatmentés sikertelen legyen, azaz ideiglenesen a folyamfüggetlen bejegyzés nem feltétlenül hozható létre. Ilyenkor a tanulási folyamat során továbbított *Learn* vagy *Confirmation* csomagból más switcheknek nem szabad folyamfüggetlen bejegyzéseket tanulniuk: egy folyamfüggetlen bejegyzésben elmentett port csak olyan switchre mutathat, amelyben szintén ki van töltve a folyamfüggetlen bejegyzés, hiszen a bejegyzések által felépített gráf körmentessége így van garantálva. Ennek érdekében a két csomagtípusban egy újabb fejléc került bevezetésre, amely tárolja, hogy az adott csomagból szabad-e folyamfüggetlen bejegyzéseket tanulni. A *Learn* és a *Confirmation* csomagok feldolgozásának teljes algoritmusai a 3.19, illetve a 3.20 ábrákon tekinthetők meg.

A megoldás nem garantálja, hogy az útvonal bejegyzések felszabadítása (törlése) ne okozzon csomagvesztést, hiszen egyes *Confirmation* csomagok elvesztése esetén (illetve ha az adattárolás néha ideiglenesen nem lehetséges) a folyamfüggetlen bejegyzések nem lesznek megtanulva. A probléma mérséklése érdekében a tanulási folyamat igénylési algoritmust kibővítettük, hogy hiányzó folyamfüggetlen bejegyzések esetén is egy újabb tanulási folyamat kezdődjön, a hiányzó bejegyzések pótlása érdekében. Ennek következményeként a csomagvesztés esélye elhanyagolható, illetve, ha a csomagvesztés mégis bekövetkezne, akkor a rendszer magától helyreáll: ilyen esetben is egy újabb tanulási folyamat lesz igényelve, azaz egy útvonal meg lesz tanultva és a csomagvesztés meg fog szűnni. A tanulási folyamatok igénylésének teljes algoritmusai a 3.21. ábrán található.


3.19. ábra. *Learn* csomagok feldolgozásának (teljes) algoritmus



3.20. ábra. Request csomagok feldolgozásának (teljes) algoritmus



3.21. ábra. Új útvonal tanulási folyamat igénylésének (teljes) algoritmus

3.8. Tanulási folyamat okozta hálózati terhelés

A tanulási folyamat részeként a megoldásunk, a hosztok között forgalmazott csomagok mellé, többlet csomagokat hoz létre, amelyek segítségével valósítjuk meg az útvonal kiválasztást. Ebben a szakaszban megvizsgáljuk, hogy ezek az általunk létrehozott csomagok legfeljebb mekkora terhelést jelentenek a hálózat számára. Ennek a vizsgálata különösen fontos, hiszen a terheléelosztási megoldásunk egy túlterhelt útvonal érzékelése esetén is egy tanulási folyamatot indít el, tehát egy már leterhelt hálózatot terhelünk tovább, valamilyen mértékben, egy optimálisabb útvonal megtalálásának reményében.

Ebben a szakaszban az alábbi jelöléseket fogjuk alkalmazni:

- N : hálózatban található switchek számának felső becslése
- P : egy adott switch más switcheknek (azaz nem hosztokhoz) csatlakozó portjainak számának felső becslése
- S : megkülönböztethető tanulás erősségi szintek száma (hálózati topológiától független, a switchek szoftverébe ágyazott konstans)

A terhelés becslése érdekében a switchek által végzett csomagtovábbításokat fogjuk megszámolni. Például egy hagyományos csomag továbbítása két hoszt között legfeljebb az összes hálózati switchen keresztül jut célba (feltételezzük, hogy a switchek már megtanulták a csomag továbbításához szükséges útvonalat, azaz nincsen szükség split horizon csomagtovábbításra), hiszen körmentes útvonalakat állapít meg a megoldások és a leghosszabb lehetséges út az összes switchen keresztül vezet. Tehát egy csomag célba juttatásához két hoszt között legfeljebb N csomagtovábbításra van szükség.

Egy tanulási folyamatot elindíthat egy levél switch, vagy más switchek igényelhetik azt. Amennyiben egy másik switch igényli a tanulást, akkor ennek a switchnek egy csomagot kell küldenie a levél switch részére. Legrosszabb esetben (a levél switch kivételével) az összes switch küld egy tanulás igénylési csomagot mielőtt a tanulási folyamat elindulna, illetve a switcheknek a náluk a levél switchtől távolabb elhelyezkedő switchek által létrehozott tanulás igénylési csomagokat is továbbítania kell a levél switch felé. Így a szükséges csomagtovábbítások száma, mielőtt még egy tanulási folyamat elindulna, legfeljebb akár $\frac{(N-1)N}{2} = \frac{1}{2}(N^2 - N)$ is lehet: az első switch egy új csomagot küld, a második switch egy új csomagot küld és az előző switch csomagját továbbítja, a harmadik switch egy új csomagot küld és az előző két switch csomagjait továbbítja, stb.

A tanulási folyamat első részében a hálózat *Learn* csomagokkal van elárasztva a kisebb MAC címhez tartozó levél switchtől indulva. Minden switchhez eljut egy *Learn*

csomag, a switchek (kivéve a nagyobb MAC címhez tartozó levél switchet) pedig az első ilyen csomagot minden portjukon (kivéve a portot, amin keresztül a csomag fogadva lett) továbbítják. Amennyiben egy switch egy nagyobb erősségű *Learn* csomagot kap, mint korábban, akkor az újabb csomagot is minden portján továbbítja. Így egy switch legfeljebb $(P - 1)S$ csomagot továbbít. Ehhez az szükséges, hogy a switchekhez a legkisebb erősségű *Learn* csomagok jussanak el leghamarabb, ami valószínűtlen, hiszen az erősség az útvonal gyorsaságára utal, azaz jellemzően a legnagyobb erősségű csomagok jutnak el először a switchekhez, de a felső becslést ezt nem befolyásolja. A teljes hálózatot nézve a csomagtovábbítások száma legfeljebb $(N - 1)(P - 1)S = NPS - NS - PS + S$.

A nagyobb MAC címhez tartozó levél switch az első, vagy a korábbinál nagyobb erősségű *Learn* csomag fogadása esetén egyetlen *Confirmation* csomaggal válaszol, ami a megtanult útvonalon visszafelé továbbításra kerül a kisebb MAC címhez tartozó levél switchhez. Tehát legfeljebb S *Confirmation* csomag kerülhet létrehozásra, amelyeket maximum $N - 1$ switchnek kell továbbítania, mielőtt azok elérik a kisebb MAC címhez tartozó levél switchet. Ez legfeljebb $S(N - 1) = NS - S$ csomagtovábbítást jelent.

A korábbi értékeket összeszámolva a tanulási folyamat legfeljebb $\frac{1}{2}(N^2 - N) + NPS - PS$ többlet csomagtovábbítással járhat. Bár a kapott érték a hálózatban megtalálható switchek számával négyzetesen arányos, a switchek száma a switchek által normál működés során továbbított csomagok számához képest elhanyagolható. Fontos továbbá figyelembe venni azt is, hogy a türelmi időszakok következtében a kiszámolt csomagmennyiség csak bizonyos időközönként kerülhet kiküldésre, nem pedig folyamatosan.

4. fejezet

Megvalósítás

A bemutatott csomagtovábbítási, terheléelosztási algoritmust a P4 [3] adatsík programozási nyelven megvalósítottuk. A forráskódot a P4C [21] fordító segítségével a BMv2 szoftver switchre [22] fordítottuk le.

4.1. Vezérlősík

A megvalósításhoz nincsen központi vezérlősík alkalmazva, hanem a switchek egymással kizárólag az adatsíkon keresztül kommunikálnak. Viszont, ahogy a szerzők [13]-ben is ismertették, egyes funkciók megvalósítása nem lehetséges a P4 programozási nyelvben egy vezérlősík segítségével nélkül. Ennek érdekében a minden switchhez tartozik egy úgynevezett lokális vezérlősík, azaz egy olyan vezérlősík, amely kizárólag az (egyetlen) hozzá párosított switch konfigurálásáért felel.

A lokális vezérlők egyrészt felelősek a switch multicast csomagtovábbítás, illetve csomag klónozás konfigurálásáért. Létrehozásra kerültek olyan multicast és klónozási csoportok, amelyek segítségével broadcast (switch minden portján keresztül történő), split horizon (egy adott port kivételével minden porton keresztül történő) csomag küldés, illetve csak egy adott portra való csomag klónozás megvalósítható.

A lokális vezérlők másik feladata az útvonal és folyamfüggetlen bejegyzések tárolásának megvalósítása, illetve a már régóta nem használt útvonal bejegyzések törlése szükség esetén. Ezekről a 4.3. szakaszban lesz szó.

A vezérlősík és a P4 programozható adatsík közötti kapcsolatát a P4Runtime specifikáció [23] írja le. A terheléelosztási algoritmus megvalósítás során a lokális vezérlőnek csak olyan műveleteket kell végrehajtani, amelyet a P4Runtime specifikáció megenged,

viszont egyes ilyen műveletek bár már specifikálva vannak, még nincsenek implementálva. Ezért a P4Runtime szabvánnyal párhuzamosan egy régebbi, Thrift-alapú interfész is fel lett használva, amely lehetőséget ad a hiányzó műveletek végrehajtására. Továbbá csak a Thrift interfész segítségével van lehetőség a BMv2 szoftver switch teljesítményének korlátozására, azaz annak a megadására, hogy egységnyi idő alatt a switch hány csomag továbbítására legyen képes.

4.2. Folyamok azonosítása

Ahogy a 3.4. szakaszban bemutatásra került, a folyamatok egyes Ethernet, IP és UDP/TCP fejlécek segítségével vannak azonosítva. Azonban a forrás és a cél Ethernet MAC címek kivételével ezek a fejlécek egy hasító algoritmus által fel vannak dolgozva és csak egy 32 bites hasító érték (valamint a két 48 bites MAC cím) kerül eltárolásra. Ezzel a folyamat azonosítók tárolásához szükséges tárhely lényegesen lecsökken: hasítás nélkül 216 bitre lenne szükséges egyetlen folyamat azonosító eltárolására, a hasításnak köszönhetően viszont csak 128 bitre. Ez a megközelítés hasonlít az IPv6 csomagok felépítésére, ahol a forrás és a cél címen kívül egy 20 bites mező segít a folyamatok azonosításában [20].

A hasítás következtében viszont előfordulhat, hogy eltérő folyamatok azonos folyamatként vannak azonosítva (hasítóértékek ütköznek), ezzel a terheléelosztás maximális hatékonyságát csökken, hiszen a két folyamat számára nem határozható meg két eltérő útvonal. Ahhoz, hogy egy ilyen hasítóérték ütközés bekövetkezésének esélye 0,1% legyen, legalább $\sqrt{\frac{0,1\%}{100\%}} * 2 * 2^{32} \approx 2931$ folyamat szükséges két adott hoszt között (két adott MAC cím között) [24]. Az ütközés esélye csökkenthető, ha 32 bit helyett a hasító algoritmus egy nagyobb számosságú halmazra képez.

4.3. Útvonal és folyamfüggetlen bejegyzések tárolása

Útvonal, illetve folyamfüggetlen bejegyzések tárolásánál nagyon sok azonosítót (MAC címet vagy folyamat azonosítót) tudunk megkülönböztetni. Ebből fakadóan mindegyik folyamat számára párhuzamosan nem képes egy switch memóriát lefoglalni (hiszen nincsen ennyi memória egy switchben). A megoldás valamilyen asszociatív tároló használata, amely egy folyamazonosítóhoz (vagy MAC címhez) egy bejegyzést tud hozzárendelni. Az általános rendeltetésű programozási nyelvekben ilyenkor például egy hasító-táblát, vagy egy AVL-fát lehetne alkalmazni. A P4 adatsík programozási nyelv standard

könyvtára azonban ilyen adatszerkezetek nem tartalmaz, megvalósításuk pedig normál körülmények között nem is lehetséges: a hasítótáblákban a kulcsütközések feloldásához, illetve a fák bejárásához ciklusok szükségesek, a P4 programozási nyelv azonban nem tartalmaz ciklusokat; csak konstans időigényű műveletek megvalósítását teszi lehetővé.

Mivel a folyamfüggetlen, illetve az útvonal bejegyzések tárolása során hasonló nehézségeket kell megoldani, ezért a perzisztenciájuk közösen van megoldva: a folyamfüggetlen bejegyzések számára egy pszeudo folyam azonosító kerül létrehozásra, aminek segítségével az útvonalbejegyzésekkel megegyező módon lehet őket eltárolni (a tárolás során csupán egyes mezők üresek, kihasználatlanok maradnak, ezzel egy minimális mennyiségű memóriát pazarolva).

4.3.1. Lehetséges perzisztencia megközelítések

Több lehetséges megközelítés meg lett vizsgálva az útvonal bejegyzések tárolásának megvalósítására. A legegyszerűbb eset, amikor egy *hasítótábla* van alkalmazva, amely nem képes hasítóérték ütközések kezelésére. A P4 programozási nyelvben ez megvalósítható a folyam azonosítók hasítóértékei által indexelt, rögzített méretű regiszterek segítségével. Ez a megoldás egyszerű, nem szükséges hozzá még egy lokális vezérlősíkc sem, viszont hasítóérték ütközések esetén az adatmentés nem lehetséges, amely a dolgozatban bemutatott algoritmus számára elfogadhatatlan.

Hasítóértékek használata helyett lehetséges lenne a P4 nyelv által nyújtott *match-action táblákban adatokat tárolni*. Ezek a táblák valamilyen kulcshoz (jelen esetben egy folyam azonosítóhoz vagy MAC címhez) rendelnek egy metódust és annak meghívásához szükséges paraméterértékeket. Ezek a táblák azonban az adatsíkcban (a P4 kódban) csak olvashatóak: módosításukra csak a vezérlősíkc, azaz jelen esetben a lokális kontroller képes. Tehát egy módosítás végrehajtásához az adatsíkcnak egy üzenetet kell küldenie a vezérlősíkc számára, és ennek az üzenetnek tartalmaznia kell az összes beszúrandó adatot. Az üzenetküldés meg is valósítható a P4 nyelv *digest* rendszere segítségével, azonban ez egy késleltetést vezet be a perzisztenciába: egy kért adatmódosítás nem feltétlenül lenne látható az adatmódosítást közvetlenül követő kiolvasás által, például amikor két egymást gyorsan követő csomag ugyan azt az útvonal bejegyzést szeretné használni.

Az előbbi megközelítés javítható, ha az adatok regiszterekben vannak tárolva, és a *regiszterekhez match-action táblák rendelnek indexeket*. Ilyenkor a már meglévő adatok módosíthatóak késleltetés nélkül, hiszen egy regiszterben található érték felülírható az

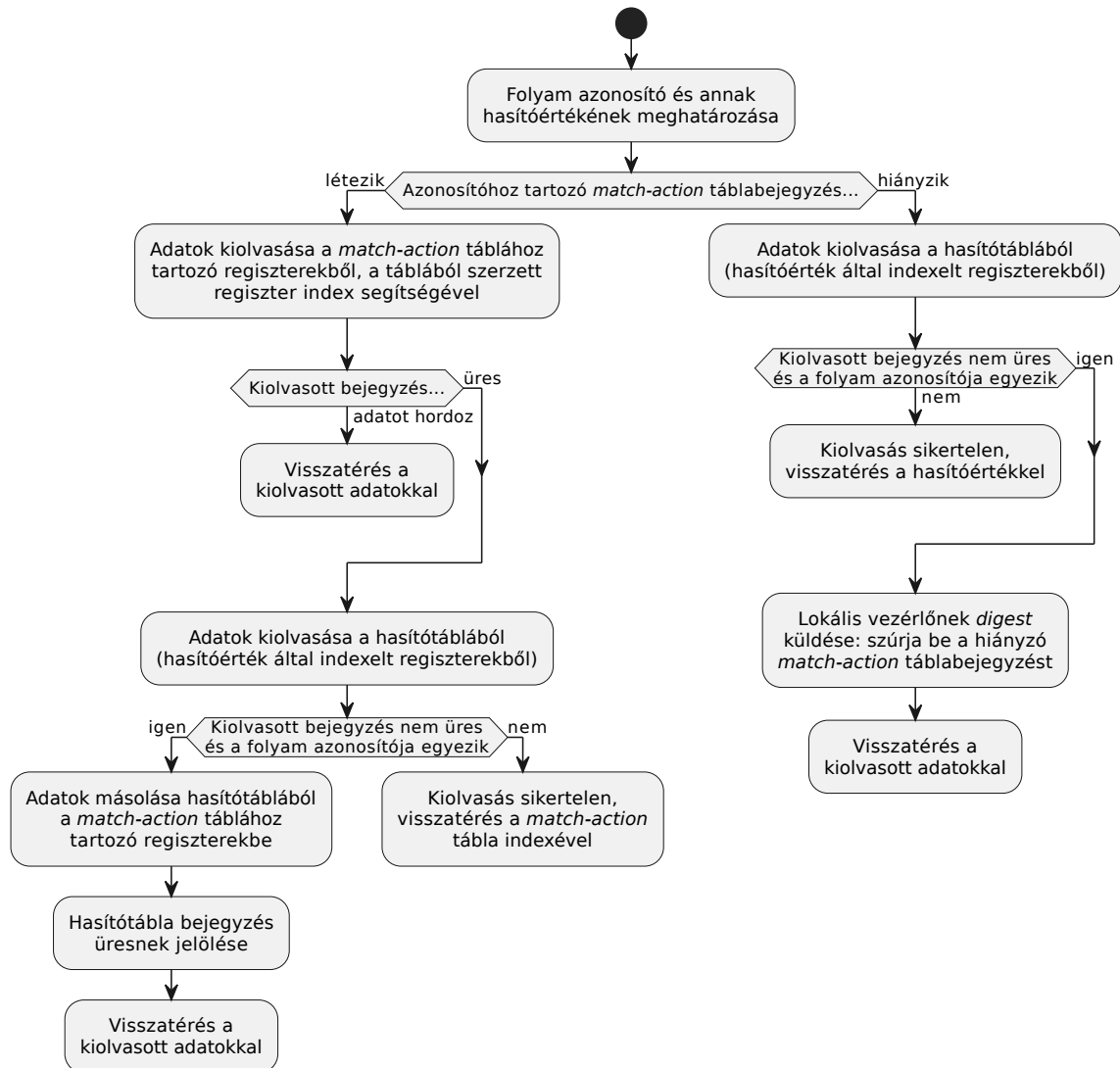
adatsíkból is. Viszont egy újabb bejegyzés beszúrásához továbbra is üzeni kell a lokális vezérőnek, hogy a *match-action* táblába az adott folyamhoz rendeljen hozzá egy szabad (nem használt) regiszter indexet, azaz a perzisztencia továbbra is szenved késleltetést. Ebben az esetben a *digest* (üzenet) csak a beszúrandó folyam azonosítót tartalmazza, a szabad indexet a lokális vezérő határozza meg (például egy számláló segítségével).

A végső megoldást a **korábbi megközelítések ötvöztetése** nyújtotta. Hogyha a *match-action* tábla már rendelt egy regiszter indexet az adott folyamhoz, akkor ezek a regiszterek vannak használva a perzisztencia megvalósítására. Amennyiben viszont hiányzik a *match-action* táblabejegyzés, akkor a folyam azonosító hasítóértékének segítségével más (eddigiektől különböző) regiszterek vannak indexelve, azaz egy hasítótábla van alkalmazva. A hasítótábla használatával párhuzamosan egy *digest* kerül kiküldésre a lokális vezérő számára, hogy a *match-action* táblába szűrje be a hiányzó bejegyzést. Azaz a hasítótábla csak ideiglenesen felel egy-egy bejegyzés tárolásáért: a bejegyzések, ahogy beszúrásra kerül a *match-action* táblabejegyzés, át vannak mozgatva a másik regiszterekbe, ezzel felszabadítva egy helyet a hasítótáblában. Ebből kifolyólag a hasítóérték ütközések csak ideiglenesen gátolják meg a perzisztenciát, amely a modell által meg van engedve.

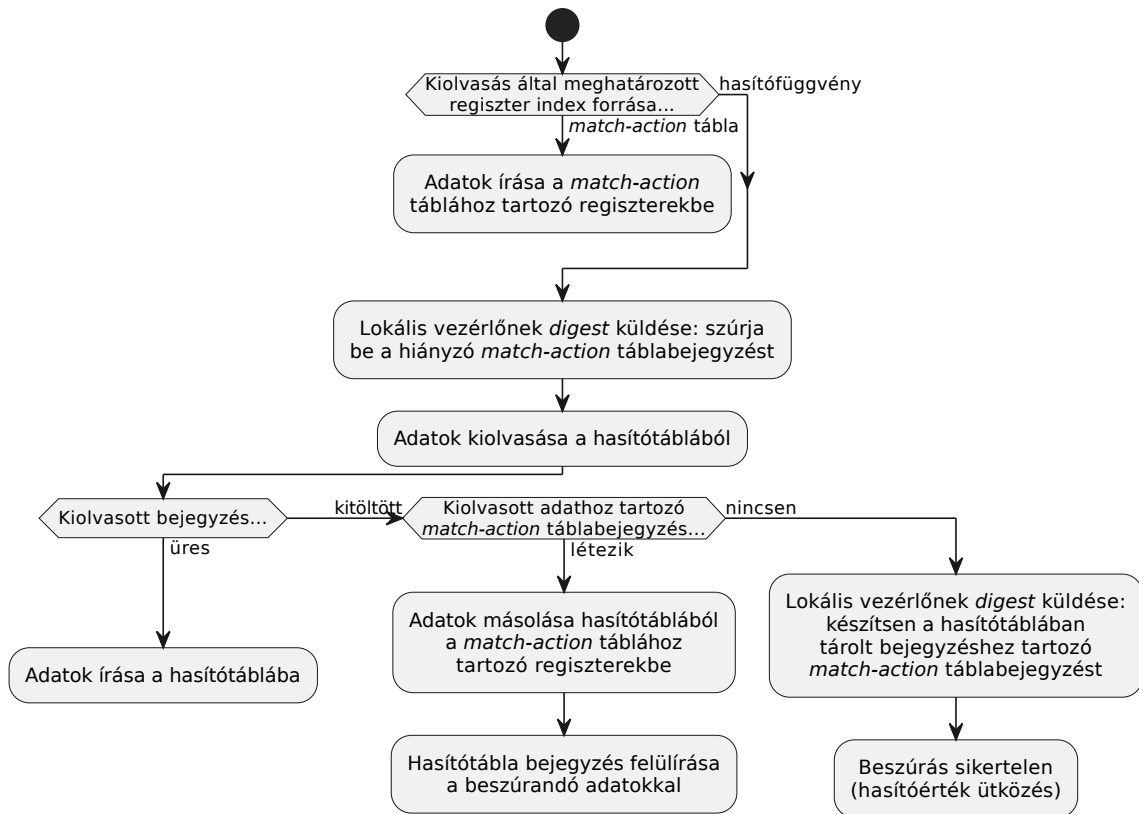
A hibrid adattárolásnak a teljes algoritmus a 4.1, 4.2. és a 4.3. ábrákon is megtekinthető. Látható, hogy nem csak egy újabb bejegyzés beszúrásának algoritmus küld egy *digest*-et, hanem hasítóérték ütközés, illetve hasítótáblából való kiolvasás esetén is küldünk. Erre azért van szükség, mert egy csomag feldolgozása során csak egy *digest* küldése van megengedve, így amennyiben több beszúrás is szükséges igényelni (azaz több *digest*-et is küldünk), akkor csak egy beszúrás igénylés fog megérkezni a lokális vezérőhöz. Annak köszönhetően, hogy az algoritmusban például hasítóérték ütközés esetén is küldünk egy beszúrás igénylést, már nem okoznak problémát az elveszett *digest*-ek.

Az (ideiglenes) hasítóérték ütközések esélye a hasítótábla méretétől, azaz a switchben elérhető memória mennyiségétől függ. Egy útvonal bejegyzés tartalmazza a legutóbbi útvonal tanulás időpontját (48 biten, a különböző türelmi időszakok megvalósítása érdekében), a tanulás és a megerősítés erősségét (4-4 bit), a kisebb és nagyobb MAC címhez vezető portok azonosítóját (9-9 bit), illetve hogy tárol-e bármilyen valós adatot (vagy üres-e) az adott bejegyzés (1 bit). Ezen felül a hasítótáblának a bejegyzéshez tartozó folyam azonosítót is el kell mentenie, hogy a hasítóérték ütközéseket a teljes azonosító összehasonlításával detektálni lehessen (128 bit). A felsorolt bithosszak közül a legtöbb (bizonyos korlátok között) szabadon állítható, azonban a megvalósítás során ezek az értékek lettek használva, így egy bejegyzés 203 biten, felfelé kerekítve 26 bájt, azaz

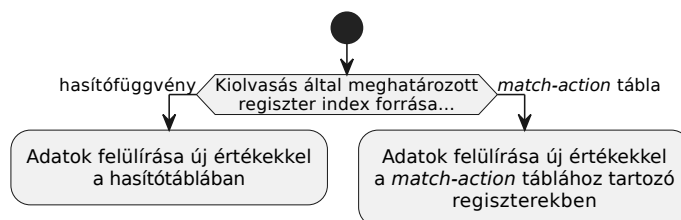
fér el a hasítótáblában. Amennyiben egy switch 10MB RAM-ot foglal le a hasítótáblának, akkor az $\frac{64 \cdot 10^6 \text{ byte}}{26 \text{ byte}} \approx 384615$ bejegyzés tárolására lesz alkalmas. Ilyenkor legalább $\sqrt{\frac{0,1\%}{100\%}} * 2 * \frac{64 \cdot 10^6 \text{ byte}}{26 \text{ byte}} \approx 27$ (folyamfüggetlen vagy útvonal) bejegyzésnek kell párhuzamosan beszúrás alatt állnia (azaz a hasítótáblában eltárolva lennie, arra várva, hogy a megfelelő match-action táblabejegyzés beszúrásra kerüljön) ahhoz, hogy egy hasítóérték ütközés 0,1%-kal következzen be.



4.1. ábra. Útvonal bejegyzés kiolvasásának algoritmus. A kiolvasás sikertelennek számít, ha eddig az adott folyamat azonosítóhoz nem volt egy bejegyzés hozzárendelve.



4.2. ábra. Új útvonal bejegyzés beszúrásának algoritmus.



4.3. ábra. Kioltvasott vagy újonnan beszúrt útvonal bejegyzés módosításának algoritmus.

4.3.2. Útvonal bejegyzések törlése

Ahogy a 3.7. szakaszban bemutatásra került, egy switchben nincsen elég memória az összes lehetséges útvonal bejegyzés párhuzamos eltárolására, ezért a már nem használt (inaktív) bejegyzéseket ki kell törölni, hogy a felszabadított memóriát egy aktív folyam tudja használni.

A nem használt útvonal bejegyzések meghatározásához minden bejegyzéshez feljegyezzük, hogy mikor volt utoljára használva (mikor érkezett utoljára olyan csomag, ami az adott bejegyzéshez tartozik). Hogy memóriát spóroljunk a megvalósítás során, ezt az időpillanatot a türelmi időszakok implementálására használt, legutóbbi tanulás időpont-

ját tartalmazó mezővel közösen tároljuk el. Ez úgy került megvalósításra, hogy a mező csak akkor tárolja a legutóbbi tanulás időpontját, ha még van (a pillanatnyi időpillanat és a mentett időpont alapján) aktív türelmi időszak. Ellenkező esetben a mező a bejegyzés utolsó használatának időpontjára jellemző értéket tárol: az utolsó használat időpillanatából kivonjuk a türelmi időszakok maximumát, és ezt adjuk értékül a mezőnek (emiatt a lejárt türelmi időszakok továbbra is inaktívak maradnak). Így mind a türelmi időszakokhoz, mind a nem használt útvonal bejegyzésekhez szükséges időpillanat rendelkezésünkre áll, egy memória hatékony módon.

Útvonal bejegyzéseket elég akkor törölni, amikor már nincsen szabad memória egy újabb bejegyzés számára: a törlés korábban való elvégzése kihasználatlan memóriát okoz, ami nem jár előnnyel. Ebből kifolyólag a bejegyzések törlésének szükségességét csak akkor kell ellenőriznie a lokális vezérőnek, amikor egy *digest* érkezik a switch adatsíkjától, hogy egy újabb bejegyzés beszúrása szükséges. (A törlést a lokális vezérőnek kell elvégeznie, hiszen a törléshez a match-action tábla módosítása szükséges, amire csak a vezérő képes.) A lokális vezérő a korábban bemutatott időpillanat segítségével azonban még nem tudja eldönteni, hogy egy adott bejegyzés használaton kívülnek számít-e: a lokális vezérő és a switch által futtatott P4 program két eltérő, egymással nem szinkronizált órát használ. Azaz bár a lokális vezérő két időpillanatot összehasonlítva meg tudja állapítani, hogy mennyi idő telt el közöttük, a pillanatnyi időpillanat meghatározására nincsen lehetősége. Ezt a problémát úgy oldottuk meg, hogy az újabb útvonal bejegyzés beszúrását kérő *digest* azt is tartalmazza, hogy a switch órája szerint mennyi (volt) a pillanatnyi idő (a *digest* kiküldésekor).

Bejegyzések törléséhez először meg kell állapítani a törlendő bejegyzés(ek)et, utána azok törlését végre kell hajtani (match-action táblából a megfelelő táblabejegyzést törölni kell, illetve az útvonal bejegyzéshez tartozó regisztereket is nullázni kell, hogy a régi adatok ne befolyásolják a következő bejegyzést, amely azon a memóriaterületen lesz tárolva). Ennek a több akciónak atomikus (egy megszakíthatatlan egységként történő) végrehajtására azonban a vezérő és az adatsík kapcsolatát leíró P4Runtime specifikáció nem ad lehetőséget. Ha a törlési folyamat nem atomikusan megy végbe, akkor előfordulhat, hogy egy bejegyzés törlendőnek megítélése után, viszont még a törlés végrehajtása előtt, újból használatba lép a bejegyzés: újra elkezd csomagokat forgalmazni a folyam, és akár egy újabb tanulási folyamat is el lesz indítva, ami után a törlést már nem lenne szabad megejteni.

Az ilyen inkonzisztens állapotok elkerülése érdekében a törölhető bejegyzések zá-

rolásra kerülnek azt megelőzően, hogy a vezérlő sík azonosítaná őket: mielőtt a lokális vezérlő kiolvassa az útvonal bejegyzéseket tartalmazó regisztereket (hogy megkeresse a törölendő bejegyzéseket), egy másik regiszterbe beleírja, hogy melyik időpillanatnál régebbi bejegyzéseket fogja majd törölni. Ez az érték ismert, hiszen mindig a (szintén ismert) pillanatnyi időpillanattól legalább egy adott konstans mértékkel (például 5-10 másodperccel) régebbi bejegyzések kerülnek törlésre. Az adatsík (P4 forráskód) ennek a regiszternek az értékét mindig ellenőrzi, és ha egy olyan útvonal bejegyzést kéne használnia, amely a regiszter értéke alapján törlés alatt áll, akkor az útvonal bejegyzést inkább nem használja, hanem hiányzóként (töröltként) kezeli azt. A vezérlő sík a törlési folyamat végeztével a zárolásra használt regiszter értékét nullázza, ezzel jelezve, hogy az összes útvonal bejegyzés szabadon használható.

4.4. Túlterheltség megállapítása, kezelése

A megvalósításhoz használt BMv2 szoftver switch a csomagfeldolgozási folyamatot két fő részre osztja: egy bemeneti feldolgozási fázisra (ahol a csomag továbbítására használt kimeneti port meghatározásra kerül), illetve egy kimeneti feldolgozási fázisra. A csomag fejléceinek módosítására mindkét fázisban van lehetőség, illetve a két fázist a kiküldésre váró csomagokat tartalmazó sor választja el. A kimeneti csomagfeldolgozás során le tudjuk kérdezni, hogy a feldolgozás alatt álló csomag mennyi időt töltött a sorban, illetve az is ismert, hogy egy teli sor kiürítéséhez (összes csomagjának kiküldéséhez) mennyi idő szükséges (azaz egy csomag maximum mennyi időt tölthet a sorban). Egy adott port túlterheltségét ezeknek a segítségével állapítjuk meg: amennyiben a csomag például a teli sor kiürítéséhez szükséges idő legalább 60%-át a sorban töltötte, akkor a port túlterheltnek van minősítve egy adott (például a teli sor kiürítéséhez szükséges idővel azonos) ideig.

A Modell áttekintés fejezetben bemutatott terheléelosztási, útvonalválasztási algoritmus nagyrészt a bemeneti feldolgozási fázisban került megvalósításra, hiszen csak itt van lehetőség a csomag továbbítására használt kimeneti port megadására, amely az algoritmus feladatai közé tartozik. Azonban a terheléelosztás megvalósításához az algoritmusnak ismernie kell a switch portjainak terheltségét is, ez az információ azonban csak a kimeneti csomagfeldolgozás során érhető el. Az algoritmus emiatt két részre van bontva, és az alábbi részei a kimeneti csomagfeldolgozási fázisban lettek megvalósítva:

- Tanulási csomagok továbbítása során a csomagok erősségének csökkentése (túlterheltség esetén).
- Túlterheltség miatti új útvonal igénylés.

Mindkét esetben a bemeneti csomagfeldolgozás először kiszámolja, hogy az adott művelet végrehajtása meg van-e engedve (például van-e aktív útvonal igénylési türelmi időszak). A kimeneti csomagfeldolgozás során pedig csak akkor van a művelet végrehajtva, ha az meg is van engedve, illetve túlterheltség is fennáll.

5. fejezet

Kiértékelés

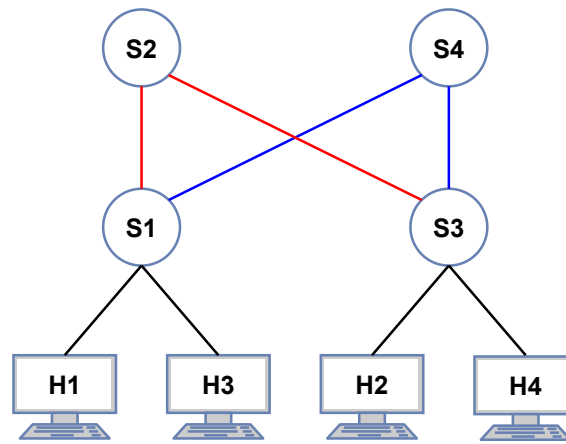
A bemutatott terheléelosztási algoritmust a korábban említett BMv2 szoftver switch [22], illetve a Mininet hálózati szimulátor [25] segítségével összehasonlítottuk pár, a Kapcsolódó munkák fejezetben bemutatott megoldással. Megvizsgáltuk, hogy egy adott sávszélességgel rendelkező hálózatban a különböző algoritmusok mennyi idő alatt, illetve milyen késleltetés mellett tudnak egy előre meghatározott adatmennyiséget átküldeni. Kitérünk továbbá a különböző esetekben tapasztalható jitter, illetve nem sorrendhelyesen fogadott csomagok mennyiségére is, valamint a bemutatott algoritmus használhatóságára egy teljesen túlterhelt hálózatban.

A kiértékelés során összehasonlítjuk az algoritmusunkat az ARP-Path [11] és a TCP-Path [14] nevű algoritmusok mintájára létrehozott terheléelosztási megoldásokkal. Az általunk használt implementációk bár a legfontosabb tulajdonságaikban megegyeznek az eredeti algoritmusokkal, pár részletben eltérnek:

- Az ARP-Path megvalósításunk nem csak ARP csomagokból, hanem bármely Ethernet csomagból képes tanulni. Az algoritmus fő tulajdonsága, hogy két hoszt (MAC cím) között csak egy (a tanulás pillanatban leggyorsabb) útvonal kerül megtanulásra, továbbra is fennáll.
- A TCP-Path megvalósításunk nem közvetlenül a *SYN*, illetve a *SYN+ACK* csomagokból tanul, hanem egy új folyam megjelenése esetén egy újabb csomag kerül létrehozásra (a *SYN* csomag mellé), amely segítségével történik a tanulás. Ebben az esetben is meg van tartva az algoritmus fő tulajdonsága: egy új folyam megjelenése esetén egy új útvonal kerül megtanulásra, és több lehetséges útvonal esetén a legkisebb késleltetéssel rendelkező útvonal kerül kiválasztásra.

5.1. Szimulációkhoz használt topológia

A kiértékeléshez az 5.1. ábrán látható hálózat lett használva. Két levél switch lett létrehozva, amelyekhez kettő-kettő hoszt kapcsolódik. A levél switchek között két lehetséges útvonal található (az egyik útvonal S2-n, a másik S4-en keresztül vezet). A kiértékelés során a H1-H2, illetve a H3-H4 hoszt páros kommunikál egymással (forgalmaz egymás között csomagokat). Egyes szimulációkban minden switch és minden link azonos tulajdonságokkal bír (egy csomag feldolgozásához szükséges idő, késleltetés), míg egyes szimulációk alatt asszimmetria kerül bevezetésre a hálózatba.



5.1. ábra. A kiértékeléshez használt topológia. Az S1-gyel és S2-mal jelölt levél switchek közötti két lehetséges útvonal kék, illetve piros színnel van feltüntetve.

5.2. Útvonalak sávszélességének összegét meg nem haladó küldési ráta

Ebben a szakaszban olyan szimulációs eredményeket ismertetünk, amelyekben a hálózat képes az összes küldött csomag célba juttatására, amennyiben az alternatív útvonalak is kihasználására kerülnek. Azaz hogyha csak egyetlen útvonal van használva a két levél switch között, akkor egyes csomagok eldobásra kényszerülnek (nem elegendő sávszélesség miatt), de mindkét útvonal (közel egyenletes) kihasználása esetén a csomagküldés megvalósítható alacsony késleltetéssel és csomagvesztés nélkül.

Az első szimulációk egy *szimmetrikus hálózatot* használtak, azaz a topológiában található két útvonal között nem volt különbség: mindkettő azonos sávszélességgel és késleltetéssel bír. Az útvonalak sávszélessége az S2-es és az S4-es switchek által van korlátozva (ezek a gyenge láncszemek a hálózatban). Mindkét switch 10 Mbps csomagto-

vábbítási rátára képes, és a kiküldésre váró csomagok sorának teljes tartalma 60 ms alatt küldhető ki az előbb meghatározott továbbítási ráta mellett.

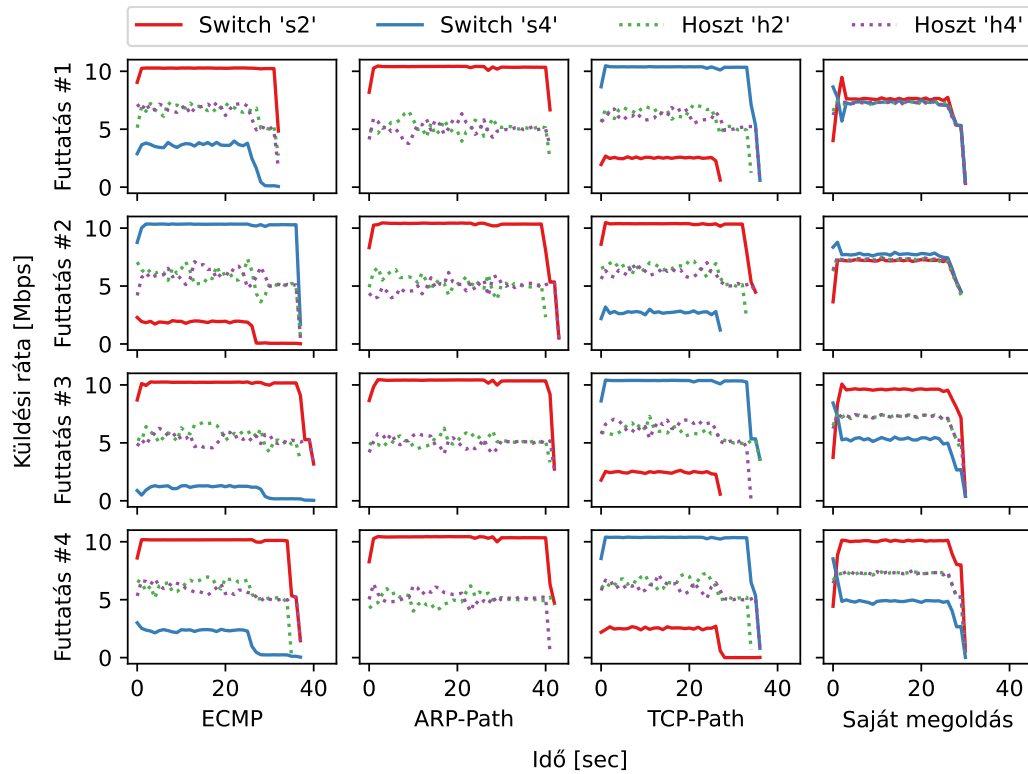
A H1-H2 és a H3-H4 hoszt párok egymás között háromféle folyamatot forgalmaznak párhuzamosan: alacsony, közepes, illetve magas küldési rátával rendelkezőt. Ezen folyamatpusok eloszlása az 5.1. táblázatból olvasható ki. Mindegyik folyamtnak annyi (előre meghatározott mennyiségű) adatot kell továbbítania, hogy a különböző folyamatpusok (ideális körülmények között) egyszerre végezzenek az adatküldéssel: az első csomag kiküldésétől számítva 30 másodperc múlva.

Folyamtípus	Folyamok száma		Küldési ráta		
	Darab	Arány	Egy folyam	Összes folyam	Arány
Kicsi	8	20%	62,5 Kbps	0,5 Mbps	3,57%
Közepes	28	70%	125 Kbps	3,5 Mbps	25,00%
Nagy	4	10%	2500 Kbps	10 Mbps	71,43%
Összesítés	40	100%	-	14 Mbps	100,00%

5.1. táblázat. Folyamtípusok tulajdonságai és eloszlása.

Az 5.2. grafikonon látható, hogy különböző szimuláció futtatások során az egyes hosztok és switchek mekkora küldési rátával forgalmaztak csomagokat. Leolvasható, hogy a korábbi megoldásokhoz képest a mi megoldásunk az előre meghatározott adatmennyiséget gyorsabban (ECMP-hez, ARP-Path-hoz, illetve TCP-Path-hoz képest 18, 29, illetve 17 százalékkal gyorsabban) képes célba juttatni. Az ábráról megállapítható továbbá, hogy a megoldásunk teljesítménye a futtatások során konzekvens: nincs lényeges különbség az adatmennyiség átküldéséhez szükséges időben a különböző futtatások során.

Leolvasható, hogy az ECMP, ARP-Path illetve a TCP-Path algoritmusok az egyik útvonal sávszélességét teljesen kihasználják a szimulált futtatások során: egy-egy switch eléri a 10 Mbps-os küldési rátát, ami a szimulált switchek maximális teljesítménye. Sőt, az útvonalon több, mint 10 Mbps-os rátával próbálnak csomagokat forgalmazni az algoritmusok, ezért csomagvesztésre kerül sor, hiszen a 10 Mbps-on túl forgalmazott csomagok eldobásra kerülnek. A csomagvesztés következtében életbe lép a TCP folyamvezérlési algoritmus, a folyamatok küldési rátája lecsökken (majd a csomagvesztés megszűnését követően újra megemelkedik), ahogy az ábrán a hosztok küldési rátájainak ingadozásai (szaggatott vonalak) is mutatják. Az ábráról leolvasható az is, hogy amikor az egyik hoszt küldési rátája lecsökken, akkor a másik hoszt küldési rátája megnövekszik, kihasználva a felszabadult sávszélességet.



5.2. ábra. Switchek és hosztok csomagküldési rátája az idő függvényében. Többféle terheléselosztási megoldással, mindegyik esetben négyszer meg lett ismételve a szimuláció.

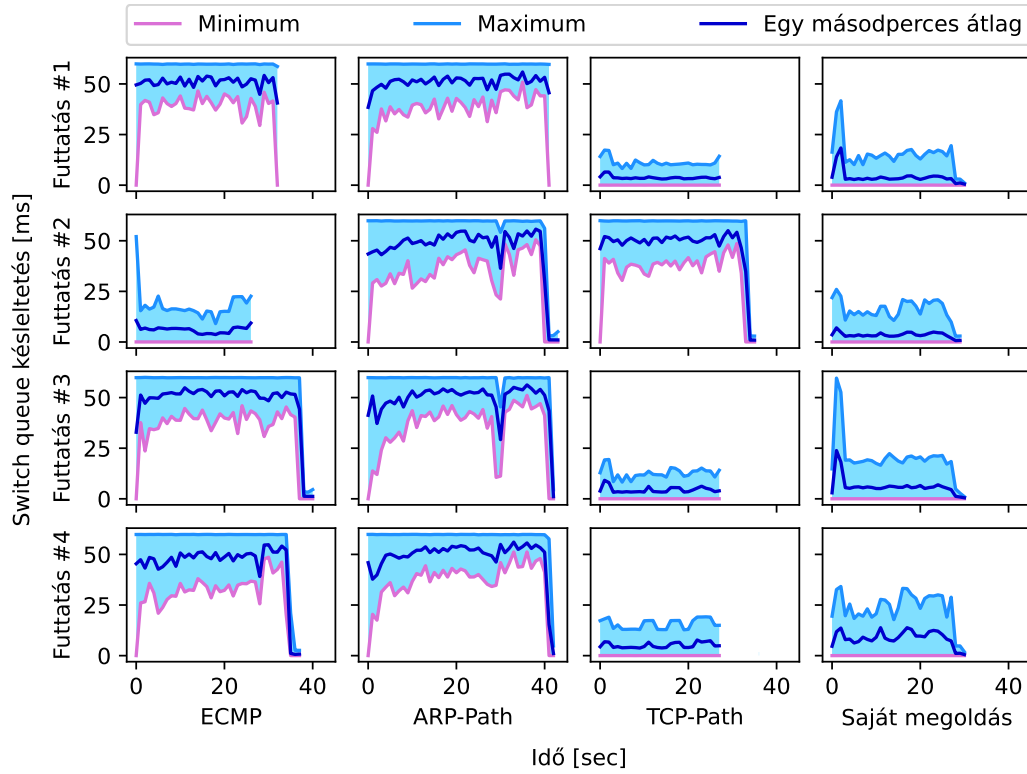
Ideális esetben mind a hosztok, mind a switchek küldési rátájainak összege megegyezik a folyamatok maximális küldési rátájainak összegével, azaz nem lép fel csomagvesztés és a TCP folyamatok konstans a nekik beállított maximális küldési rátával tudnak adatot forgalmazni. A saját megoldásunk esetében ezt láthatjuk: bár a különböző futtatások során változik, hogy az S1-es, illetve az S2-es switchen keresztül mekkora adatmennyiség van forgalmazva, a két switchen tartozó küldési ráták összege mindig 14 Mbps, azaz a szimulációt alkotó folyamatok küldési rátájainak összegével megegyezik. Ezzel ellentétben például az ECMP algoritmus esetében van, amikor 12 Mbps és 2 Mbps arányban vannak elosztva a folyamatok, azaz az egyik útvonalhoz tartozó folyamat torlódást és csomagvesztést tapasztalnak, a másik útvonalon forgalmazott folyamatok azonban, csomagvesztés hiányában, gyorsabban átküldik a számukra előre definiált adatmennyiséget (amiután az útvonalon a küldési ráta pedig lecsökken nullára). Az ARP-Path algoritmus esetében ennek az egyenlőtlen eloszlásnak az extrém esete figyelhető meg: csak az egyik útvonal került kihasználására, azaz az algoritmus a két hoszt pár között mindegyik futtatás során azonos (S2-es switchen át vezető) útvonalat tanult meg.

A saját megoldásunk esetében megfigyelhető, hogy a futtatások elején a switchek kül-

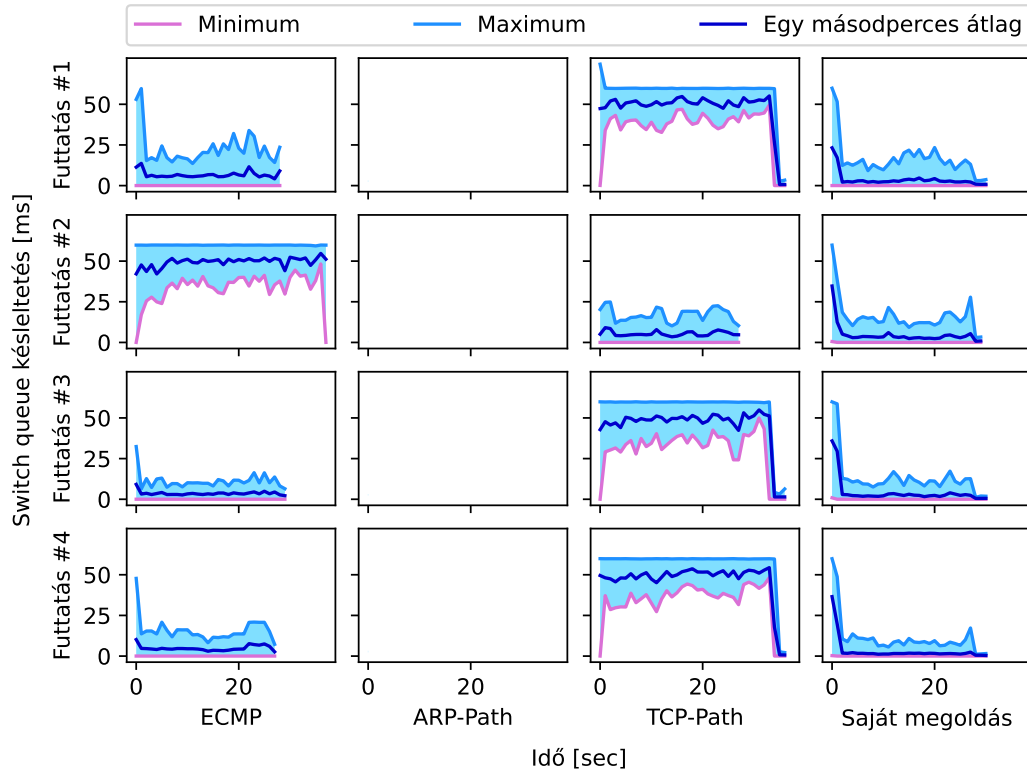
dési rátái jelentősen megváltoznak (megnövekednek vagy lecsökkennek). Ennek az az oka, hogy induláskor a többi megoldáshoz hasonlóan a mi megoldásunk is a legtöbb folyam számára megegyező útvonalat választ ki. Rövid időn belül azonban az algoritmusunk a torlódást érzékel, és egyes folyamokat a másik útvonalra irányít át, a torlódás megszüntetése érdekében. Például az első futtatás során azonban az átirányított folyamatok az induló folyamatokkal kiegészülve a másik útvonalon is torlódást okoztak, ezért egyes folyamokat az algoritmus megint alternatív útvonalra irányított át, a torlódást ezúttal ténylegesen megszüntetve és a folyamokat közel pontosan két egyenlő részre osztva a két útvonal között.

Az 5.3. és 5.4. grafikonokon az S2-es, illetve az S2-es switch által okozott késleltetés látható: hogy a switcheken áthaladó csomagok mennyi időt töltöttek a switch kiküldésére váró csomagok sorában. Leolvasható, hogy a korábbi megoldások nem osztják szét elég egyenletesen a folyamokat, így a késleltetés magasra nő: átlagosan ~50 milliszekundumot töltenek a csomagok a sorban, míg a saját megoldásunk esetében ez az érték 7 milliszekundum. Az is látható az ábrán, hogy a korábbi megoldások esetében a switch sorok betelnek (ezzel potenciálisan csomagvesztést okozva), hiszen egyes csomagok 60 milliszekundumot várákoztak a sorban, ami a sor maximális méretével egyeztethető meg. Az egyenletes szétosztás hiánya az ARP-Path algoritmus esetén a leglátványosabb: ahogy korábban említésre kerül, mindegyik futtatás során kizárólag az S2-es switchet tartalmazó útvonal került megtanulásra.

A saját megoldásunk esetében a szimulációs futtatások elején megfigyelhető, hogy a késleltetés egy másodperces átlaga magasra ugrik, azután viszont (a többi megoldással ellentétesen) a késleltetés lecsökken. Ennek az az oka, ahogy az korábban is ismertetésre került, hogy a megoldásunk induláskor a legtöbb folyamat azonos útvonalra irányítva, azután viszont, a torlódást (megnövekedett késleltetést) érzékelve, egyes folyamatok átirányításra kerülnek egy alternatív útvonalra, ezzel a switch soroknak lehetőségük van kiürülnie, a késleltetés csökken.

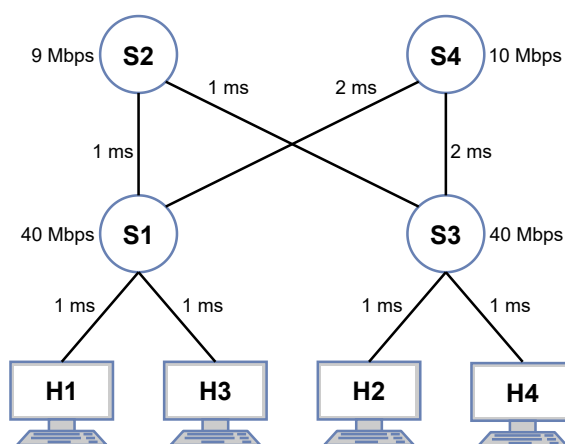


5.3. ábra. Csomagok által töltött idő az S2-es switch kiküldésre váró csomagok sorában, az idő függvényében.



5.4. ábra. Csomagok által töltött idő az S2-es switch kiküldésre váró csomagok sorában, az idő függvényében. Az ARP-Path algoritmus esetében nincs adat, mert az algoritmus egy csomagot sem az S2-es switchen keresztül továbbított.

Megvizsgálásra került továbbá egy *aszimmetrikus hálózat* esete is. Az alábbi eredmények olyan szimulációkból származnak, amelyek során csak egyféle folyamat típus volt használva (20 darab folyam, mindegyik 700 Kbps küldési rátával), valamint a hálózat minden linkje legalább 1 ms többlet késleltetést szimulált az előző kiértékelésekhez képest. A hálózat aszimmetriája abban merült ki, hogy az S1-S2-S3 útvonal 10 Mbps helyett csak 9 Mbps továbbítására volt képes, az S1-S4-S3 útvonal pedig a másikkal képest 2 ms többlet késleltetést tartalmazott. A bevezetett késleltetések és sávszélességek az 5.5. ábráról olvashatóak le. Az aszimmetriát tartalmazó topológiák olyan esetekben fordulhatnak elő, amikor például eltérő típusú switchekből került egy hálózat kiépítésre, vagy a hálózatban található switchek fokozatosan (nem pedig egyszerre) kerülnek lecserélésre.

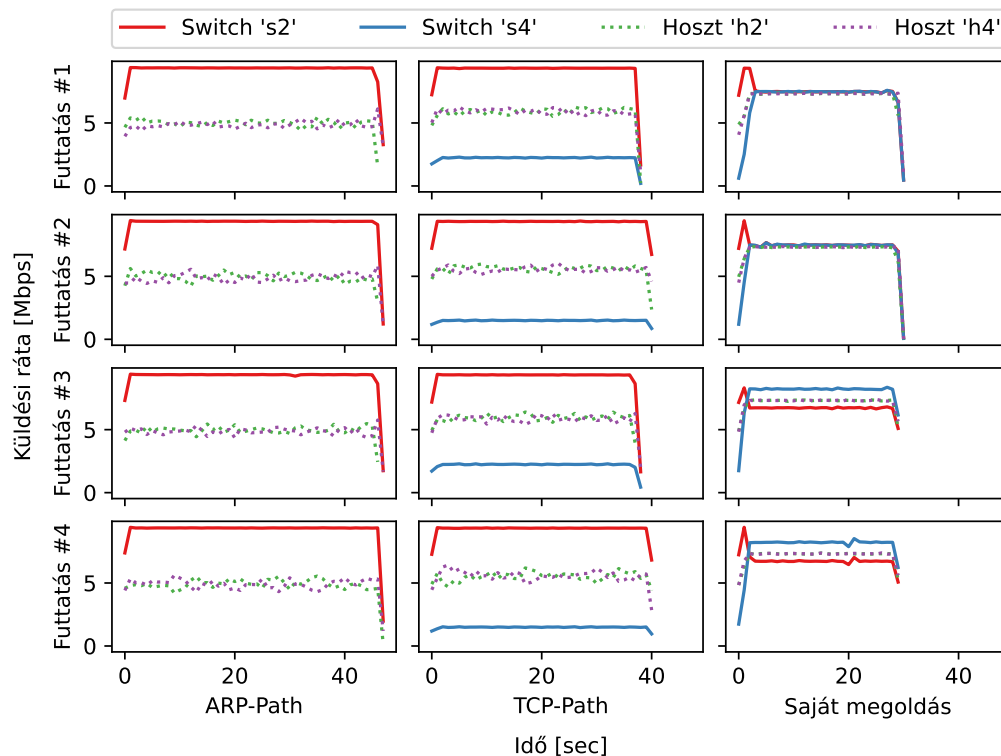


5.5. ábra. A kiértékeléshez használt aszimmetrikus topológia. Leolvasható az egyes linkek által okozott terjedési késleltetés, és a switchek által maximálisan elérhető csomagtovábbítási ráta.

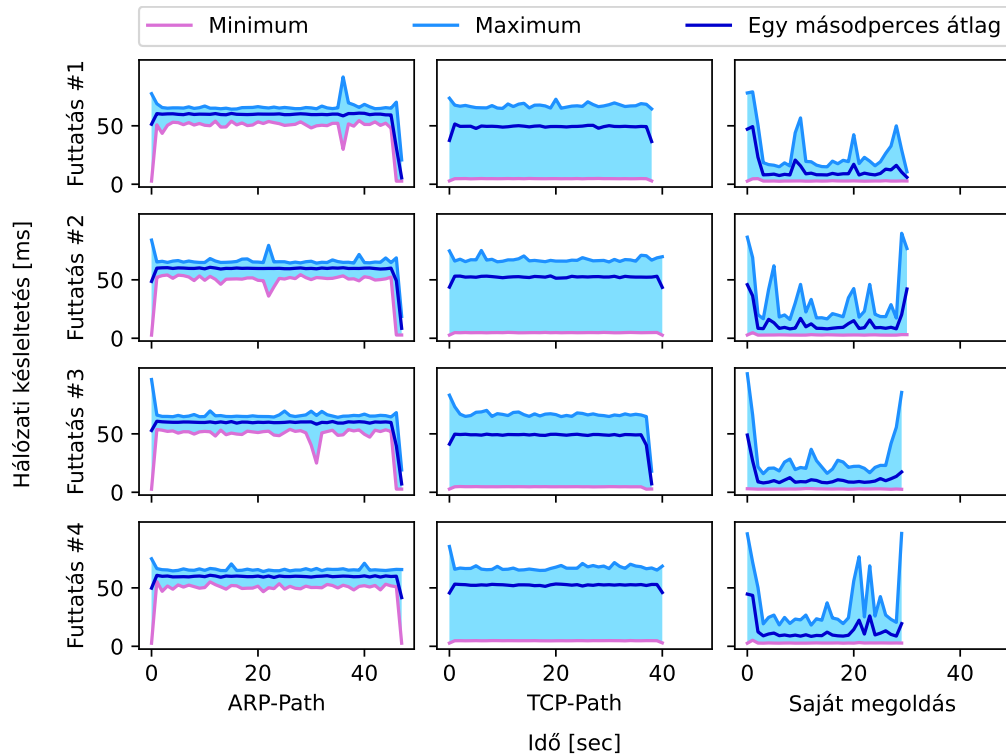
Az 5.6. grafikonon a küldési ráták láthatóak az idő függvényében. Az algoritmusok az S2-es switchet tartalmazó útvonalat tanulták meg elsősorban, hiszen ehhez az útvonalhoz kevesebb terjedési késleltetés lett hozzárendelve a használt aszimmetrikus topológiában. Az ARP-Path algoritmus ezt az egy útvonalat tanulta meg mindkét hoszt pár közötti kommunikáció számára, azaz a hálózat összesített 19 Mbps-os sávszélessége ellenére csak 9 Mbps-mal volt képes adatot továbbítani, magas késleltetés illetve csomagvesztés mellett. A TCP-Path algoritmus kisebb részben az alternatív útvonalat is használja, azonban a folyamatok eloszlása nem megfelelő, az S2-es switchet tartalmazó útvonalon torlódás, illetve csomagvesztés lép fel. A saját megoldásunk viszont, ahogy torlódást érzékelt, dinamikusan irányított át megfelelő mennyiségű folyamatot az alternatív útvonalra, ezzel a torlódást megszüntetve és alacsony hálózati késleltetést érve el a forgalmazott csomagok számára. Ezek következményeként a saját megoldásunk ebben a szimulációs esetben is gyorsabban

volt képes az előre definiált adatmennyiséget továbbítani: ARP-Path-hoz képest 37%-kal, TCP-Path-hoz képest 24%-kal.

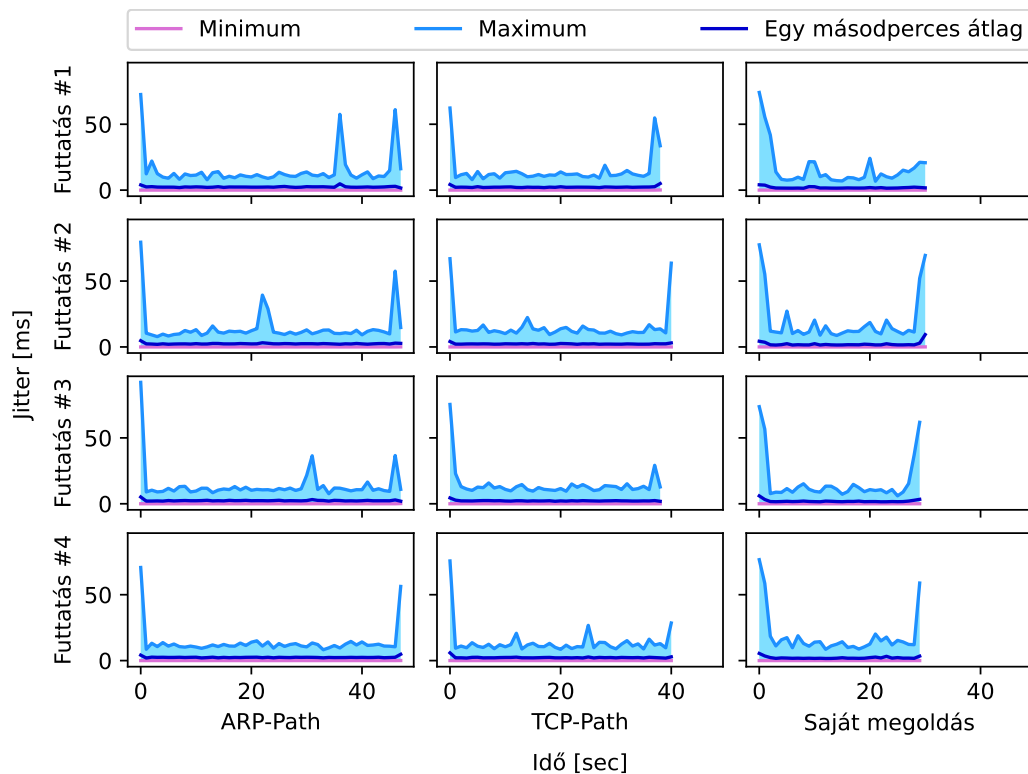
Az idő függvényében ábrázolt hálózati késleltetések az 5.7. ábráról olvashatóak le. Az ARP-Path algoritmus esetében a minimális és maximális késleltetés közötti különbség nem nagy, hiszen csak egyetlen útvonal van használva, amelyen a switch kiküldésre váró csomagokat tartalmazó sorának hossza közel azonos marad. A TCP-Path algoritmus viszont bár nagyrészt egy túlterhelt útvonalon keresztül forgalmazza a folyamatok, kisebb részben egy alacsony kihasználtságú útvonal is van alkalmazva, aminek következményeként a minimális késleltetés lényegesen kisebb az átlagos késleltetésnél. A futtatásokat összesítve, az ARP-Path algoritmus esetében az átlagos késleltetés 59 ms, a TCP-Path esetében 51 ms, a saját megoldásunk esetében pedig 12 ms. A késleltetés mértékének szimulációk közbeni ingadozásai a valós idejű hálózati szimuláció tökéletlenségével magyarázhatóak. A hálózati jitter (azonos folyamba tartozó, egymás után érkező csomagok hálózati késleltetésének különbsége) pedig az 5.8. grafikonnál olvashatók le. A futtatások elején és végén a jitter érték azért nő meg, mert ekkor telítődnek meg, illetve ürülnek ki a switch sorok, azaz ilyenkor a hálózati késleltetés hirtelen nagyot változik.



5.6. ábra. Switchek és hosztok csomagküldési rátája az idő függvényében. Az asszimmetria miatt az algoritmusok elsősorban az S2-es switchen át vezető utat tanulják meg, de a mi megoldásunk a torlódást érzékelve több folyamat áterrel az alternatív útvonalra.

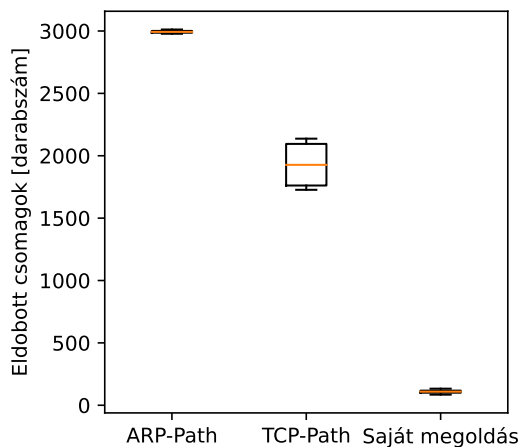


5.7. ábra. Csomagok által elszenvedett hálózati késleltetés az idő függvényében. TCP-Path esetében a legtöbb csomag magas késleltetéssel jut célba, viszont pár folyam egy alternatív, alacsony késleltetésű útvonalon halad, ezért kevés a minimális késleltetés.

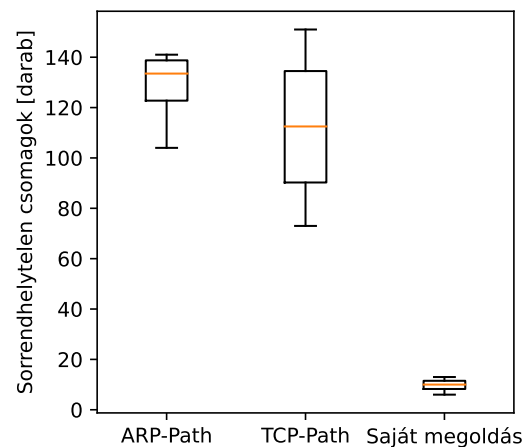


5.8. ábra. Csomagok által elszenvedett hálózati jitter az idő függvényében. A szimulációk elején és végén azért magas a jitter érték, mert a switchek kiküldésre váró csomagokat tartalmazó sorai ekkor telnek meg, illetve ürülnek ki.

Az 5.9. ábráról az egyes futtatások során kiküldésre került, de célba nem juttatott adatmennyiség olvasható le: a különböző futtatásokhoz tartozó értékek egy dobozdiagramon kerültek megjelenítésre. Csomagvesztés akkor fordult elő a szimulációk során, amikor egy switch kiküldésre váró csomagokat tartalmazó sora megtelt, tehát az újabb csomagot már nem lehetett belehelyezni. A mi megoldásunk csupán ~100 csomag eldobására kényszerült, a TCP-Path esetében ez a szám ~1900, az ARP-Path esetében pedig ~3000. Az 5.10. grafikonon pedig a nem sorrendhelyesen fogadott csomagok száma található le a különböző algoritmusok esetében: ARP-Path és TCP-Path esetében ~120 a darabszám, a mi megoldásunk esetében pedig ~10. Fontos kiemelni, hogy TCP esetében a csomagvesztés is nem sorrendhelyesen fogadott csomagokat eredményezhet például az eldobott csomagok újraküldése miatt.



5.9. ábra. Megtelt switch sorok miatt eldobott csomagok száma a különböző algoritmusok esetében, az egyes futtatások során.



5.10. ábra. Nem sorrendhelyesen fogadott csomagok száma a különböző algoritmusok esetében, az egyes futtatások során.

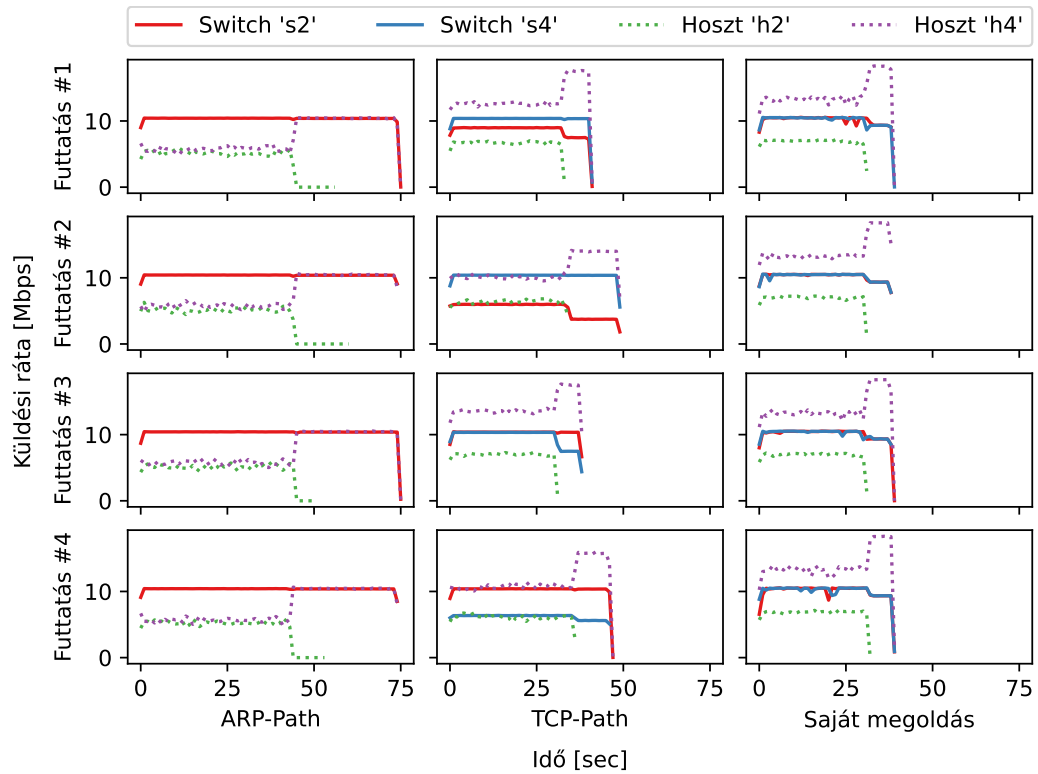
5.3. Hálózat sávszélességét meghaladó küldési ráta

Amennyiben a folyamatoknak nem létezik olyan csoportosítása, hogy az egy-egy csoporthoz tartozó útvonal sávszélessége meghaladja a csoportba tartozó folyamatok összesített küldési rátáját, akkor legalább a hálózat egyik útvonalán folyamatosan túlterheltség fog fennállni, ezáltal az általunk bemutatott algoritmus folyamatosan újabb és újabb tanulási folyamatokat fog elindítani, amik akár egy-egy útvonal változtatással is végződhetnek (vagy megtanulhatják újra az aktuális útvonalukat). A többi, kiértékelés során megvizsgált, terheléelosztási megoldás esetében ilyen probléma nem lép fel, hiszen ezek a megoldások csak egy útvonalat határoznak meg, amit később nem tudnak megváltoztatni.

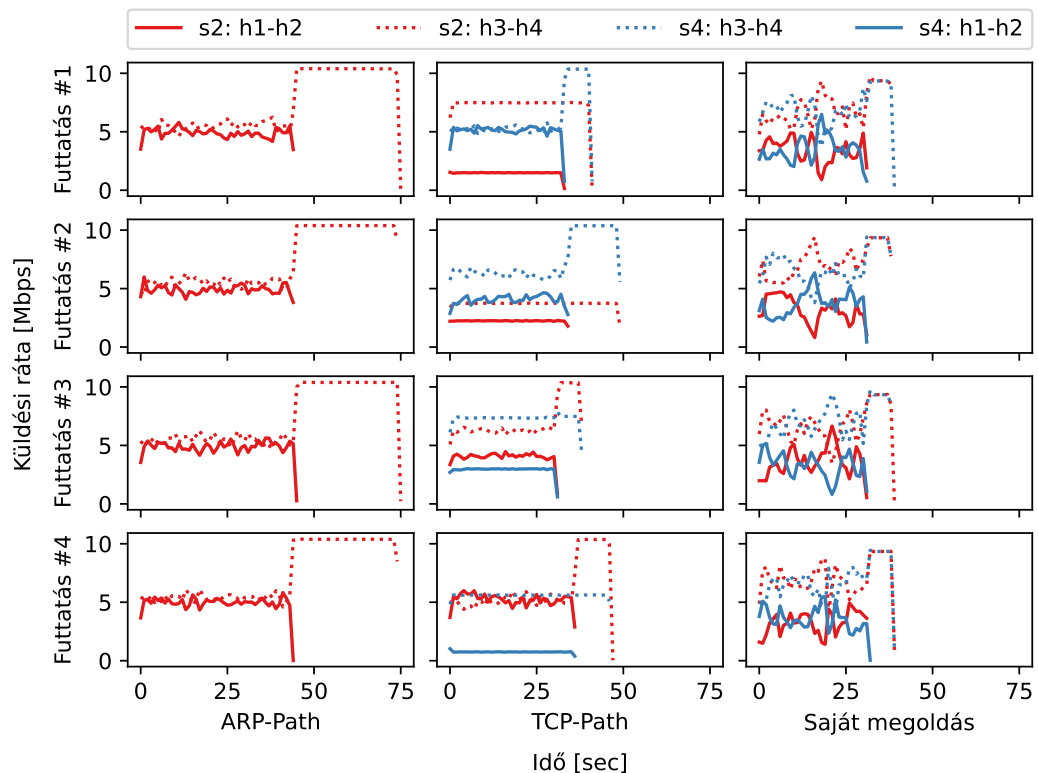
Ebben a szakaszban megvizsgáljuk, hogy az általunk bemutatott algoritmus egy ilyen szituációban (folyamatos túlterheltség mellett) milyen eredményeket képes elérni. A szimulációk során a már korábban ismertetett szimmetrikus hálózati topológiát alkalmaztuk. Mindkét hoszt páros között 10 darab folyamot forgalmaztunk, de amíg a H1-H2 hoszt páros folyamai a korábban is használt 700 Kbps küldési rátát alkalmazták, addig a H3-H4 hoszt páros 1750 Kbps küldési rátát használt. Mindegyik folyam továbbra is egy előre definiált adatmennyiség átküldéséért felel, amely egy torlódásmentes ideális hálózatban 30 másodperc alatt megvalósítható lenne minden egyes folyam által.

A korábbiakhoz hasonlóan, az 5.11. grafikonon a küldési ráták láthatóak az idő függvényében. Az ábráról leolvasható, hogy a mi megoldásunk megbízhatóan azonos eredményt ad a megismételt futtatások során, továbbá átlagosan 48%-kal, illetve 12%-kal gyorsabban továbbítja az előre definiált adatmennyiséget az ARP-Path, illetve a TCP-Path algoritmusnál. Látható, hogy ARP-Path esetében, ahol csak egyetlen útvonal van használva, a TCP folyamvezérlési algoritmusnak köszönhetően a két hoszt pár egyenlően osztozik az elérhető sávszélességen, az általuk forgalmazott csomagok eltérő küldési rátájának ellenére. Emiatt tapasztalható az ARP-Path esetében is, illetve a többi terhelésmegosztási algoritmus esetében is, hogy a kisebb küldési rátával rendelkező folyamok (H1-H2 hoszt páros) gyorsabban végeznek, hiszen a sávszélesség egyenletes eloszlása következtében az aktuális és maximális küldési ráták aránya a kisebb küldési rátával rendelkező folyamok esetében lesz nagyobb. Amikor pedig véget érnek egyes folyamok, akkor a fennmaradó folyamok a felszabaduló sávszélességet (amennyiben van) kihasználják, így az ábrán megugró küldési ráta olvasható le a kisebb küldési rátájú folyamok befejeztével megegyező időpillanatban.

Az előzőhöz hasonló felépítésű, 5.12. ábrán az S2-es és az S2-es switchen áthaladó forgalom két részre lett bontva: a H1-H2, illetve a H3-H4 hoszt párok között forgalmazott részekre. Így az ábrán jobban megfigyelhető, hogy a TCP folyamvezérlésnek köszönhetően a különböző folyamok hogyan versengenek a switchek által biztosított csomagtovábbküldési rátáért. A TCP-Path esetében minden futtatás során csak az egyik útvonalon történik versengés, mert a másik útvonalon forgalmazott folyamok küldési rátája nem éri el az útvonal sávszélességét, azaz minden futtatás során az egyik útvonal részben kihasználatlan marad. A saját megoldásunk esetében pedig a folyamatos útvonalváltások következtében meghatározóbb torlódások képesek kialakulni, amik magyarázzák a küldési ráták jelentős ingadozását.

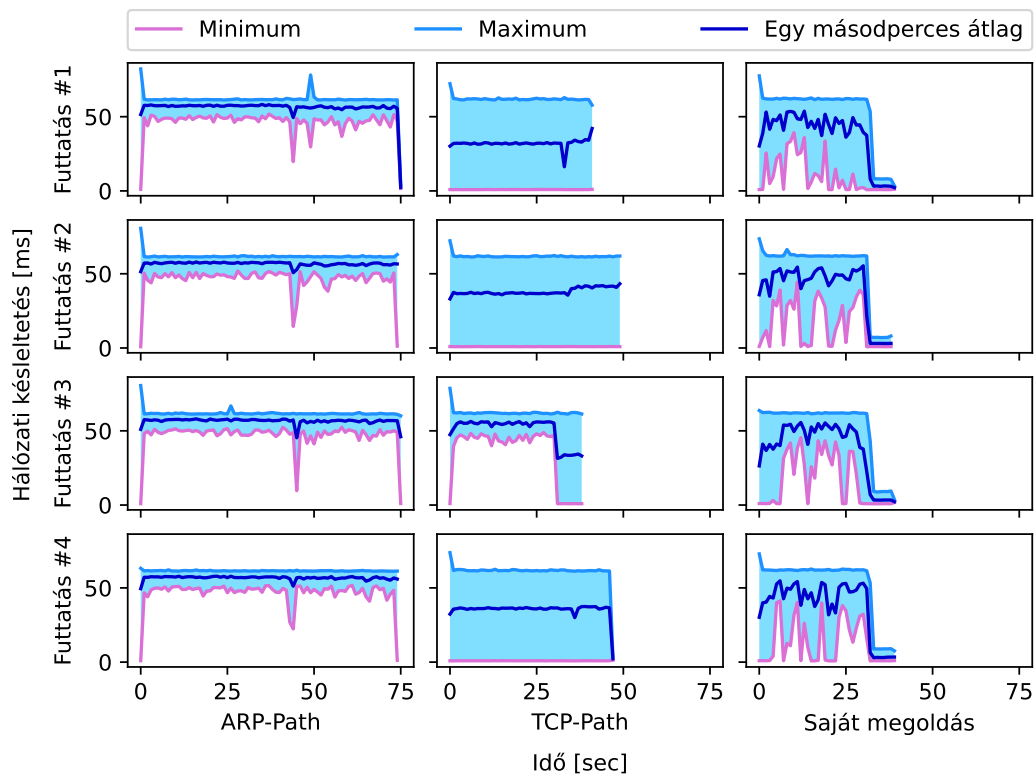


5.11. ábra. Switchek és hosztok csomagküldési rátája az idő függvényében.

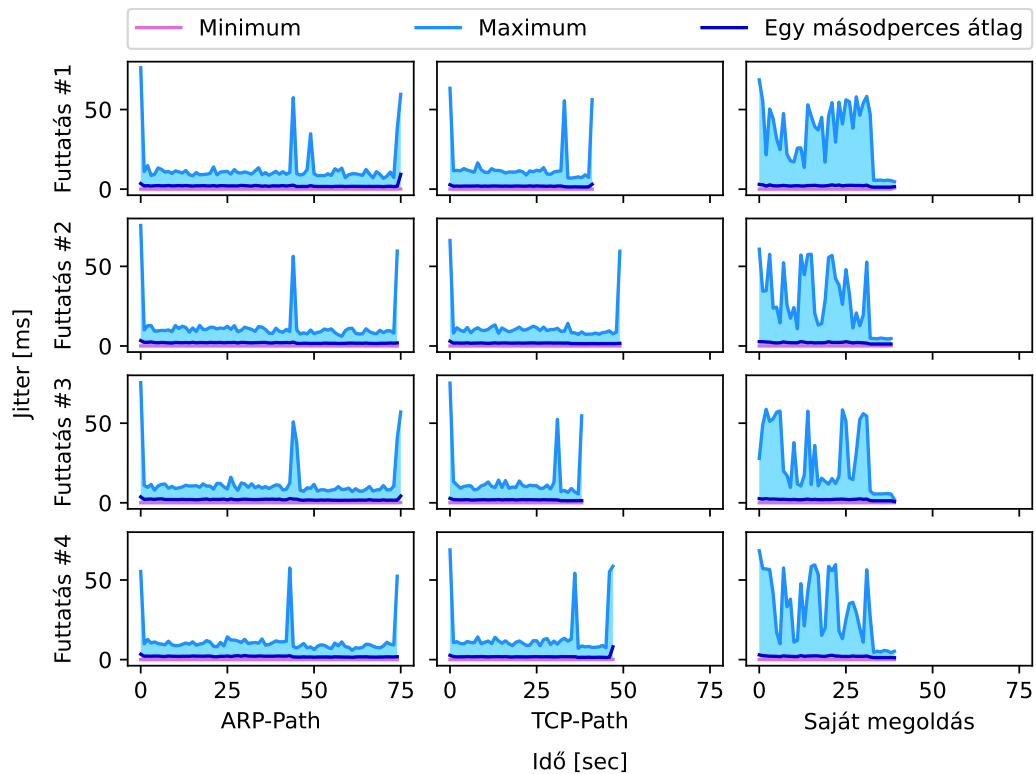


5.12. ábra. Idő függvényében a H1 -H2, illetve a H3-H4 hoszt párok között forgalmazott csomagok eloszlása a S2-es és az S4-es switcheken. A küldési ráták a TCP torlódásvezérlési algoritmusa miatt ingadoznak.

Az 5.13. grafikonról a csomagok által elszenvedett hálózati késleltetés, az 5.14. grafikonról pedig a hálózati jitter olvasható le az idő függvényében. Mindkét esetben a megszokottnál magasabb értékek olvashatóak le az ábrákról: ARP-Path esetén az átlagos késleltetés ~75ms, TCP-Path illetve a saját megoldásunk esetén pedig ~39 ms. Ennek az oka, hogy a hálózat teljesen túl van terhelve, azaz nem lehetséges torlódás nélkül a csomagtovábbítást megvalósítani. A perzisztens torlódás miatt az algoritmusunk folyamatosan útvonalváltásokat kezdeményez, ingadozó hálózati késleltetést eredményezve, amelyet a jitter értékekből is leolvashatunk. ARP-Path és TCP-Path esetén a futtatások során látható késleltetés/jitter értékbeli ugrás a kisebb küldési rátával rendelkező folyamatok befejeztével magyarázható: a folyamatok hirtelen véget érésekor a switch kiküldésre váró csomagokat tartalmazó sora kiürülhet egy pillanatra, amíg a többi folyamat a felszabadult sávszélességet fel nem használja. A TCP-Path harmadik futtatása során pedig a folyamatok egyenletes szétosztásának köszönhetően először mindkét útvonal túl volt terhelve, így a késleltetés szórása alacsony volt (hiszen mindkét útvonalon közel azonos volt a késleltetés), de egyes folyamatok befejeztével az egyik útvonal túlterheltsége megszűnt, ezzel a késleltetése is lecsökkent, ami miatt az ábrán mind az átlagos, mind a minimális késleltetés is lecsökkent.

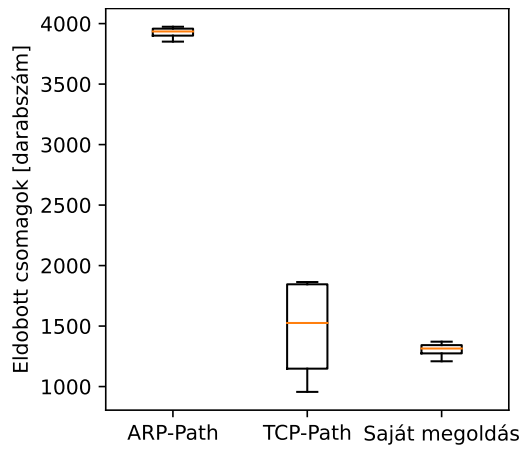


5.13. ábra. Csomagok által elszenvedett hálózati késleltetés az idő függvényében. A saját megoldásunk esetében a rendszeres útvonalváltások miatt változó a késleltetés.

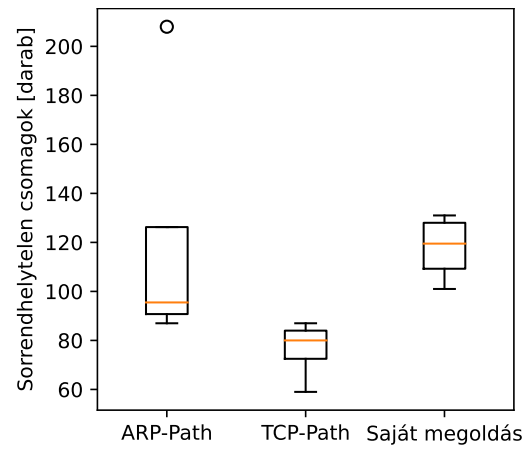


5.14. ábra. Csomagok által elszenvedett hálózati jitter az idő függvényében. A saját megoldásunk esetében a rendszeres útvonalváltások miatt változó az érték.

Ez előzőekhez hasonlóan az 5.15. grafikonon a kiküldésre került, de célba nem juttatott adatmennyiség található, az 5.16. grafikonon pedig a nem sorrendhelyesen fogadott csomagok száma olvasható le. A folyamatos útvonalváltások következményeként a mi megoldásunk a többi algoritmushoz képest enyhén több csomagot továbbított sorrend helytelenül: az ARP-Path, TCP-Path, illetve a saját algoritmusunk ~100, ~80, illetve ~120 sorrendiségi hibát vétett. Azonban az eldobott csomagok számát nézve a mi megoldásunk továbbra is jobban teljesít a többi megoldáshoz képest: amíg a mi megoldásunk ~1300 csomagot dobott el, addig az ARP-Path algoritmus ~3900, a TCP-Path algoritmus pedig ~1500 csomagot nem juttatott el a megfelelő hoszthoz.



5.15. ábra. Megtelt switch sorok miatt eldobott adatmennyiség a különböző algoritmusok esetében, az egyes futtatások során.



5.16. ábra. Nem sorrendhelyesen fogadott csomagok száma a különböző algoritmusok esetében, az egyes futtatások során.

6. fejezet

Összefoglalás

A dolgozatban egy olyan algoritmust mutattunk be, amely központi vezérlő nélküli, ismeretlen topológiájú hálózatban képes unicast, illetve multicast/broadcast csomagok továbbítására, miközben a hálózat pillanatnyi terhelését figyelembe véve terheléselosztást végez. Megoldásunk túlterheltség megjelenésekor egyes meglévő folyamatokat alternatív, kevesebb csomagot forgalmazó útvonalakra terel, ezzel a hálózat által biztosított sávszélességet nagyobb mértékben kihasználja. Ennek következményeként a hosztok közötti kommunikáció alacsonyabb késleltetéssel, illetve magasabb küldési ráta mellett történhet. Az algoritmusunkhoz egy megvalósítást is készítettünk a P4 adatsík programozási nyelven, ezzel alátámasztva az algoritmus gyakorlati megvalósíthatóságát.

A terheléselosztási megoldásunk, a Kapcsolódó munkák fejezetben bemutatott, önkonfiguráló hálózatokban is alkalmazható algoritmusokkal ellentétben képes váltakozó hálózati viszonyokra (például megváltozott folyam sávszélesség igényekre) önmagától reagálni, nem pedig csak újonnan megjelenő folyamatok számára választ egy, abban az időpillanatban ideális útvonalat. Megoldásunk bármely Ethernet-alapú hálózati protokollt (például TCP, UDP) támogatja, továbbá az egyirányú (UDP alapú) kommunikáció esetén is képes terheléselosztásra: az algoritmusunk számára nem elvárás, hogy a kommunikációban résztvevő mindkét hoszt küldjön is, és fogadjon is csomagokat. Illetve a megoldásunk a hálózati késleltetést a hálózat mindkét irányában figyelembe veszi: akkor is képes terheléselosztásra, ha például a hálózat TCP szervertől TCP kliensig vezető irányban túlterhelt (ebben az irányban teltek be a switch sorok), és akkor is, ha az ellenkező irányban van megnövekedett késleltetés. Beláttuk továbbá, hogy az általunk definiált, útvonal tanulásához használt csomagok elvesztése sem okoz problémát a bemutatott megoldás számára: csomagvesztés esetén is helyesen működik a terheléselosztási megoldásunk.

6.1. Fejlesztési lehetőségek

6.1.1. Link meghibásodás, megváltozott hálózati topológia kezelése

A bemutatott csomagtovábbítási, terheléelosztási megoldásunk bár a számítógépes hálózat működés közbeni bővülését (újabb switchek, újabb hosztok csatlakoztatását) támogatja, hálózati eszközök meghibásodása esetén nem tudja a sikeres csomagtovábbítást garantálni. Amennyiben egy switch, vagy egy switchet a hálózattal összekötő link meghibásodik (és a rajtuk keresztül történő csomagtovábbítás meghiúsul), a terheléelosztási algoritmusunk ezt nem érzékeli és nem próbálja a forgalmat egy alternatív útvonalra terelni. Ehhez hasonlóan, a megoldásunk nem érzékeli, hogyha egy hoszt (MAC cím) a hálózat egyik pontjából eltávolításra kerül, és egy másik pontján (egy másik switchen keresztül) újra csatlakozik a hálózathoz: a switchek nem feltétlenül törlik a MAC címhez tartozó bejegyzéseiket, ezáltal az elavult bejegyzések alapján, jelenleg már nem feltétlenül megfelelő portokon próbálják az áthelyezett hoszthoz címzett csomagokat továbbítani.

6.1.2. Gyakrabban útvonal váltások engedélyezése

A tanulási folyamat részeként, a 3.5.2. szakaszban definiáltunk egy türelmi időszakt, ami megadja, hogy legalább mennyi időnek kell eltelni két tanulási folyamat indítása között. Minél nagyobb ez a türelmi időszak, annál kisebb gyakorisággal lehet újabb tanulási folyamatokat indítani, azaz a bemutatott algoritmus annál lassabban tud hirtelen és gyakran változó hálózati körülményekhez alkalmazkodni.

A türelmi időszak csökkentése érdekében a tanulási folyamatokhoz hozzá lehetne rendelni egy (számláló alapú) azonosítót. Korábban a türelmi időszak kihasználásával, időpillanatok segítségével került eldöntésre, hogy egy adott csomag az előző, vagy pedig egy új tanulási folyamat részét képez-e. Amennyiben ez a feladat azonosítókkal kerül megvalósításra, akkor a türelmi időszak csökkenthető, a megoldásunk javítható.

A. függelék

Dolgozathoz felhasznált mesterséges intelligencia alapú eszközök

A dolgozat elkészítése során használatba lett véve a GitHub Copilot nevű szolgáltatás. Ez a mesterséges intelligencián alapuló eszköz automatikus kódkiegészítés nyújtásával segítette elő a dolgozat elkészültét.

Irodalomjegyzék

- [1] Open Networking Foundation. „Software-defined networking: The new norm for networks”. *ONF White Paper* 2.2-6 (2012), 11. old.
- [2] Weverton Luis da Costa Cordeiro, Jonatas Adilson Marques és Luciano Paschoal Gaspar. „Data plane programmability beyond openflow: Opportunities and challenges for network and service operations and management”. *Journal of Network and Systems Management* 25 (2017), 784–818. old.
- [3] Pat Bosshart és tsai. „P4: Programming protocol-independent packet processors”. *ACM SIGCOMM Computer Communication Review* 44.3 (2014), 87–95. old.
- [4] Péter Vörös és tsai. „T4p4s: A target-independent compiler for protocol-independent packet processors”. *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE. 2018, 1–8. old.
- [5] Richelle Adams. „Active queue management: A survey”. *IEEE communications surveys & tutorials* 15.3 (2012), 1425–1476. old.
- [6] Radia Perlman. „An algorithm for distributed computation of a spanningtree in an extended lan”. *ACM SIGCOMM computer communication review* 15.4 (1985), 44–53. old.
- [7] Christian Hopps. *Analysis of an equal-cost multi-path algorithm*. Techn. jel. 2000.
- [8] Costin Raiciu és tsai. „Improving datacenter performance and robustness with multipath TCP”. *ACM SIGCOMM Computer Communication Review* 41.4 (2011), 266–277. old.
- [9] Dimitrios Serpanos és Tilman Wolf. *Architecture of network systems*. Elsevier, 2011.
- [10] Belma Turkovic és tsai. „Fast network congestion detection and avoidance using P4”. *Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies*. 2018, 45–51. old.

- [11] Guillermo Ibáñez és tsai. „ARP-Path: ARP-based, shortest path bridges”. *IEEE communications letters* 15.7 (2011), 770–772. old.
- [12] Elisa Rojas és tsai. „All-Path bridging: Path exploration protocols for data center and campus networks”. *Computer Networks* 79 (2015), 120–132. old.
- [13] Isaias Martinez-Yelmo és tsai. „ARP-P4: deep analysis of a hybrid SDN ARP-Path/P4Runtime switch”. *Telecommunication Systems* 72.4 (2019), 555–565. old.
- [14] Joaquin Alvarez-Horcajo és tsai. „TCP-path: Improving load balance by network exploration”. *2017 IEEE 6th International Conference on Cloud Networking (CloudNet)*. IEEE. 2017, 1–6. old.
- [15] Mohammad Alizadeh és tsai. „CONGA: Distributed congestion-aware load balancing for datacenters”. *Proceedings of the 2014 ACM conference on SIGCOMM*. 2014, 503–514. old.
- [16] Naga Katta és tsai. „Hula: Scalable load balancing using programmable data planes”. *Proceedings of the Symposium on SDN Research*. 2016, 1–12. old.
- [17] Radia Perlman és tsai. *Routing bridges (RBridges): Base protocol specification*. Techn. jel. 2011.
- [18] David Allan és tsai. „Shortest path bridging: Efficient control of larger ethernet networks”. *IEEE Communications Magazine* 48.10 (2010), 128–135. old.
- [19] Hari Balakrishnan, Venkata N Padmanabhan és Randy H Katz. „The effects of asymmetry on TCP performance”. *Proceedings of the 3rd annual ACM/IEEE international conference on Mobile computing and networking*. 1997, 77–89. old.
- [20] Shane Amante és tsai. *IPv6 flow label specification*. Techn. jel. 2011.
- [21] *p4c: P4-16 reference compiler*. <https://github.com/p4lang/p4c>. (Elérés dátuma 2023. 06. 20.).
- [22] *Behavioral model: the reference P4 software switch*. <https://github.com/p4lang/behavioral-model>. (Elérés dátuma 2023. 06. 20.).
- [23] Nick McKeown. *P4 Runtime - Putting the Control Plane in Charge of the Forwarding Plane - Open Networking Foundation — opennetworking.org*. <https://opennetworking.org/news-and-events/blog/p4-runtime-putting-the-control-plane-in-charge-of-the-forwarding-plane/>. 2017. (Elérés dátuma 2023. 07. 04.).

- [24] Mihir Bellare és Phillip Rogaway. *Introduction to modern cryptography*. 2005.
- [25] Bob Lantz, Brandon Heller és Nick McKeown. „A network in a laptop: rapid prototyping for software-defined networks”. *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. 2010, 1–6. old.