

Slovenská technická univerzita

Fakulta informatiky a informačných technológií
Ilkovičova 2, 812 19 Bratislava

Dokumentácia k inžinierskemu dielu

Tímový projekt II

SealfisticatedD Networkers

Číslo a názov tímu: 21. – SealfisticatedD Networkers
Vedúci tímu: Ing. Peter Trúchly, PhD.
Členovia tímu: Bc. Maroš Hrobák, Bc. Matúš Kováč, Bc. Hana Kuntová, Bc. Marko Ondruš, Bc. Erika Štefanková, Bc. Matej Uhlík, Bc. Peter Válka
Akademický rok: 2017/2018

Obsah

| | |
|--|----|
| 1. Úvod | 1 |
| 1.1. Ciele pre zimný semester..... | 2 |
| 1.2. Ciele pre letný semester..... | 2 |
| 1.3. Celkový pohľad na systém | 2 |
| 2. Softvérovo definované siete | 3 |
| 2.1. Technológia SDN | 3 |
| 2.1.1. Vlastnosti SDN | 3 |
| 2.2. Protokol OpenFlow | 5 |
| 2.3. SDN forwardery | 8 |
| 2.3.1. Prepínač | 8 |
| 2.3.2. Open vSwitch (OVS)..... | 9 |
| 2.3.3. Pantou/OpenWRT | 10 |
| 2.3.4. OFSoftSwitch13 | 10 |
| 2.3.5. Indigo Virtual Switch (IVS) | 10 |
| 2.3.6. Linc..... | 10 |
| 2.4. SDN kontroléry | 11 |
| 2.4.1. NOX | 11 |
| 2.4.2. POX | 12 |
| 2.4.3. Beacon | 13 |
| 2.4.4. Floodlight | 14 |
| 2.4.5. MUL | 14 |
| 2.4.6. Maestro | 14 |
| 2.4.7. Iné | 14 |
| 2.5. Kontrolér Ryu..... | 15 |
| 2.6. Simulačné/emulačné prostredia..... | 17 |
| 2.6.1. EstiNet | 17 |
| 2.6.2. Ns-3 | 17 |
| 2.6.3. Trema..... | 17 |
| 2.6.4. Mininet | 18 |
| 3. Generátory premávky | 19 |

| | | |
|--------|--|----|
| 3.1. | Hping | 19 |
| 3.2. | Ostinato..... | 20 |
| 3.3. | Scapy | 21 |
| 3.4. | packETH..... | 22 |
| 4. | QoS z hľadiska poskytovania služieb na sieti | 24 |
| 4.1. | VOIP..... | 24 |
| 4.1.1. | Latencia | 24 |
| 4.1.2. | Odchýlka..... | 24 |
| 4.1.3. | Stratovosť paketov..... | 24 |
| 4.2. | Video interaktivita..... | 25 |
| 4.2.1. | Latencia | 25 |
| 4.2.2. | Stratovosť paketov..... | 25 |
| 4.2.3. | Odchýlka..... | 25 |
| 4.2.4. | Šírka pásma | 25 |
| 4.3. | Video stream..... | 25 |
| 4.3.1. | Latencia | 25 |
| 4.3.2. | Stratovosť paketov..... | 26 |
| 4.3.3. | Odchýlka..... | 26 |
| 4.3.4. | Šírka pásma | 26 |
| 4.4. | Dáta..... | 26 |
| 5. | Meranie QoS v SDN..... | 27 |
| 5.1. | Odchýlka, stratovosť, priepustnosť | 27 |
| 5.2. | Oneskorenie..... | 28 |
| 5.3. | TCP oneskorenie | 28 |
| 5.4. | UDP oneskorenie..... | 29 |
| 6. | Kvalita služieb v reálnom čase | 30 |
| 7. | QoS algoritmus plánovania tokov | 31 |
| 7.1. | Genetický algoritmus..... | 31 |
| 7.1.1. | Kríženie | 31 |
| 7.1.2. | Mutácia | 31 |
| 7.1.3. | Iterácia | 32 |
| 7.2. | Network Calculus model | 32 |
| 8. | Návrh dynamického pridelovania pre kontrolér z DP..... | 33 |

| | | |
|---------|--|----|
| 8.1. | Úprava controller.py | 33 |
| 8.2. | Úprava globals.py | 34 |
| 8.3. | Úprava QOS_linkDB.txt | 34 |
| 9. | Analýza metódy LARAC | 35 |
| 9.1. | Problém DCLC | 35 |
| 9.2. | Grafové vektory | 35 |
| 9.3. | Algoritmus | 36 |
| 10. | Analýza metódy SAQR | 38 |
| 11. | Analýza metódy ASA | 42 |
| 11.1. | Generický systémový model | 42 |
| 11.2. | SDN algoritmy | 44 |
| 11.2.1. | Fixed Priority Scheduling strategy | 44 |
| 11.2.2. | Balanced scheduling strategy | 46 |
| 11.2.3. | Adaptive cost scheduling strategy | 50 |
| 11.2.4. | Simulačné prostredie | 51 |
| 11.2.5. | Hodnotenie navrhovaného systému | 53 |
| 11.2.6. | Záver | 55 |
| 12. | Analýza MILP algoritmu | 56 |
| 13. | SDN v reálnom čase | 59 |
| 13.1. | Úvod | 59 |
| 13.2. | Definovanie problému | 60 |
| 13.3. | Smerovací modul | 61 |
| 13.3.1. | Constraint-Based Routing (CBR) | 61 |
| 13.3.2. | MILP | 61 |
| 13.4. | Cluster-based Adaptive Routing (CAR) | 62 |
| 13.4.1. | Skonštruovanie skupiny | 62 |
| 13.4.2. | Dynamické smerovanie v skupinách | 62 |
| 13.4.3. | Skonštruovanie cesty | 63 |
| 13.5. | Plánovací modul | 63 |
| 13.5.1. | Holistická analýza | 63 |
| 13.5.2. | Priradovanie priority | 65 |
| 13.5.3. | OPA | 65 |

| | |
|---|-----|
| 13.5.4. Feedback interface | 66 |
| 14. Testovanie..... | 67 |
| 14.1. Testovací scenár č.1 (Hlavný testovací scenár)..... | 67 |
| 14.1.1. Testovacie prostredie | 67 |
| 14.1.2. Testovacia topológia..... | 68 |
| 14.1.3. Zhodnotenie | 68 |
| 14.2. Testovací scenár č.2 (QoS)..... | 70 |
| 14.2.1. Testovacie prostredie | 70 |
| 14.2.2. Testovacia topológia..... | 70 |
| 14.2.3. Zhodnotenie | 70 |
| 14.3. Testovací scenár č.3 (Mininet - wifi)..... | 71 |
| 14.3.1. Testovacie prostredie | 71 |
| 14.3.2. Testovacia topológia..... | 72 |
| 14.3.3. Zhodnotenie | 72 |
| 14.4. Testovací scenár č.4..... | 72 |
| 14.4.1. Testovacie prostredie | 73 |
| 14.4.2. Testovacia topológia..... | 73 |
| 14.4.3. Zhodnotenie | 74 |
| 14.5. Testovanie topológie z článku guck2014 | 74 |
| 14.5.1. Pribeh testovania..... | 75 |
| 14.6. Testovanie topológie z diplomovej práce podľa článku guck2014 | 76 |
| 14.6.1. Pribeh testovania..... | 76 |
| 14.7. Testovanie metódy z diplomovej práce | 77 |
| 15. Implementácia metódy z DP Michala Palatinusa | 79 |
| 15.1. Problémy pri implementácii | 80 |
| 15.2. Testovanie metódy..... | 80 |
| 16. Implementácia metódy LARAC | 93 |
| 16.1. Testovanie metódy LARAC | 95 |
| 17. Implementácia metódy SAQR..... | 109 |
| 17.1. Testovanie metódy SAQR..... | 109 |
| 18. LLDP problém..... | 124 |
| 18.1 LLDP protokol..... | 124 |
| 18.2 Riešenie | 125 |

| | |
|--|-----|
| 19. Literatúra | 128 |
| Príloha A – Inštalčná príručka | 132 |
| Príloha B – Používateľská príručka | 133 |
| B.1 Mininet..... | 133 |
| Príloha C – Priebeh testovania..... | 137 |
| C.1 Testovací scenár č.1 | 137 |
| C.2 Testovací scenár č.2..... | 138 |
| C.3 Testovací scenár č.3 | 139 |
| C.4 Testovací scenár č.4..... | 139 |
| Príloha D – Technická dokumentácia..... | 140 |
| D.1 Analýza kódu z diplomovej práce Michala Palatinusa | 140 |
| D.1.1 Diagram prepojenia funkcií | 148 |
| D.2 Diagram algoritmu LARAC | 149 |
| D.3 Opis metódy LARAC | 150 |
| D.4 Opis metódy SAQR | 150 |
| Príloha E – testovanie treťou stranou | 152 |
| E.1 Úvod..... | 152 |
| E.2 Testovanie | 152 |
| E.2.1 Testovanie prostredia Mininetu..... | 152 |
| E.2.2 Testovanie implementovaných metód..... | 152 |
| E.2.3 Testovanie upravenej metódy p. Palatinusa | 152 |
| E.2.4 Testovanie metódy LARAC..... | 153 |
| E.2.5 Testovanie metódy SAQR..... | 153 |
| E.3 Zhrnutie | 153 |
| Príloha F – Testovacia topológia 1 | 156 |
| Príloha G – Testovacia topológia 2 | 157 |
| Príloha H – nesprávne testovanie | 158 |

Slovník

| Slovo | Vysvetlenie |
|--------------|---|
| Cookie | Malé množstvo dát, ktoré WWW server pošle prehliadaču, ktorý ho uloží na počítači |
| Debug | Postup pre nájdenie a znižovanie chýb |
| Forwarder | Prepínač v softvérovom definovaných sieťach |
| Framework | Aplikačný rámec |
| Open source | Voľne šíriteľný zdrojový kód |
| Plug in | Prídavný modul |
| Root bridge | Prepínač, ktorý je koreň stromu |
| Video stream | Video prenos |
| Wildcard | Rozmedzie hodnôt |

Zoznam skratiek

| Skratka | Vysvetlenie |
|----------------|--|
| ICMP | Internet Control Message Protocol |
| IP | Internet Protocol |
| IT | Internetové technológie |
| OVSDB | Databáza Open vSwitch |
| QoS | Kvalita služieb |
| SDN | Softvérovo definované siete |
| SLA | Service Level Agreement |
| SSH | Zabezpečený prístup k príkazovému interpretovaču |
| SSL | Vrstva bezpečných paketov |
| TCP | Protokol riadenia prenosu |
| UDP | User datagram protocol |
| VRRP | Virtual Router Redundancy Protocol |
| RTT | Round Trip Time |

1. Úvod

Súčasný rýchly rozvoj internetu, techniky a počítačových systémov priniesol okrem iného aj množstvo zariadení na prevádzku a správu počítačových a komunikačných sietí, ako napríklad prepínače a smerovače. Každým dňom sa zvyšuje počet ľudí pripojených do internetu a preto je veľké dátové centrá v IT spoločnostiach stále náročnejšie manažovať zväčšujúci sa nárast prenosovej a výpočtovej kapacity. S týmto problémom rastie aj potreba jednoduchej škálovateľnosti zariadení. Mnoho výrobcov sieťového hardvéru prichádza stále s novými inováciami ako tento problém doriešiť, avšak ani jedno z nich neponúka komplexné vyriešenia náročnej úlohy.

Jedným z riešení sú softvérovo definované siete, ktoré umožňujú riadiť, meniť a manažovať správanie siete dynamicky cez rozhranie. Hlavným bodom SDN architektúr je centralizovaný bod riadenia a manažmentu komunikačnej a počítačovej siete nazývaný SDN kontrolér. Tento kontrolér sa nachádza na vrchu celej siete a pomocou neho je možné jednoducho, inteligentne a efektívne riadiť smerovanie a využívanie sieťových zdrojov. Motiváciou pre tento projekt je zabezpečiť implementáciu algoritmov ktoré dokážu nielen nájsť vhodnú cestu sieťou pre každý tok, ale aj zohľadniť kvalitatívne parametre tokov alebo realizovať optimálne rozdeľovanie zátáže v sieti.

Tento dokument ponúka okrem „big picture“ projektu aj ciele vytýčené v zimnom semestri, podrobnú analýzu problematiky softvérovo definovaných sietí, návrh testovacích scenárov spolu s ich vyhodnotením, vzhľadom na následnú implementáciu vybraných algoritmov, ktoré budú efektívne riadiť zátáž v sieti.

1.1. Ciele pre zimný semester

- Oboznámiť sa s problematikou softvérovo definovaných sietí.
- Nainštalovať celé potrebné prostredie.
- Overiť náležitosti prostredia potrebné pre implementáciu.
- Získať zručnosti a základné princípy fungovania softvérovo definovaných sietí.
- Implementovať jednu metódu.

Z vybraných cieľov sa nám podarilo splniť všetky ciele, okrem implementácie jednej metódy. Implementáciu jednej metódy sme úplne dokončili až v letnom semestri a v tomto semestri sme ju len začali implementovať.

1.2. Ciele pre letný semester

- Analyzovať všetky predložené metódy.
- Implementovať aspoň dve zvolené metódy.
- Otestovať funkcionality na rôznych topológiách.
- Porovnať určené charakteristiky daných metód.
- Vytvoriť funkčné prenosné prostredie s implementovanými metódami.

Z vybraných cieľov sa nám podarilo splniť všetky ciele. Analyzovali sme všetky predložené články s metódami. Dokončili sme implementáciu metódy zo zimného semestra a implementovali sme ďalšie dve metódy. Následne sme všetky tri implementované metódy dôkladne otestovali na rôznych topológiách a jasne sme porovnali ich výsledky v tabuľkách a grafoch. Výstupom je prenosné prostredie vo forme virtuálnej mašiny, kde sú naše výsledky tímového projektu.

1.3. Celkový pohľad na systém

Systém je postavený na SDN, implementovali sme modul kontrolér, ktorý riadi všetku premávku v sieti. Funkcionalita SDN sietí je vysvetlená v nasledujúcej kapitole. Zmena v architektúre predstavuje vždy len zmenu modulu kontrolér.

Prehľad systému môžeme vidieť v prílohe D.

2. Softvérovo definované siete

Kapitola opisuje princíp fungovania softvérovo definovaných sietí. Vysvetľuje základné vlastnosti, ako aj protokol OpenFlow, či popisuje a porovnáva forwarderi, či prepínače alebo kontroléri.

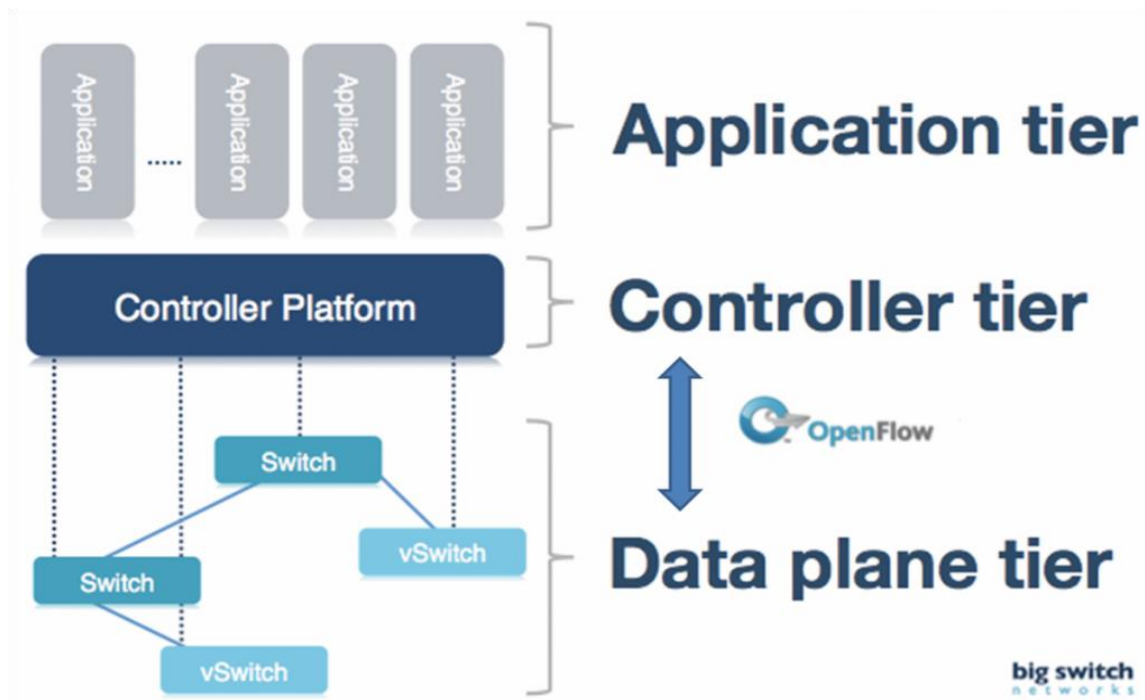
2.1. Technológia SDN

Inicializovať, kontrolovať, meniť a manažovať správanie siete dynamicky cez rozhranie.

2.1.1. Vlastnosti SDN

Vlastnosti SDN sietí sú nasledovné [30] [31] [32]:

- **Priamo programovateľné** – sieťová kontrola je priamo programovateľná pretože je oddelená od smerovacích funkcií
- **Agilná** – oddelenie kontroly od smerovania umožňuje administrátorom dynamicky prispôbovať sieťovú premávku aby spĺňala meniace sa potreby
- **Centrálne manažovateľná** – sieťová inteligencia je centralizovaná v SDN (Software Defined Networking) kontroléroch, ktoré udržujú celkový pohľad na sieť. Toto sa ukazuje vo viacerých aplikáciách ako jeden logický prepínač.
- **Programovateľná** – umožňuje meniť konfiguráciu, manažovať, zabezpečiť alebo optimalizovať sieť rýchlo cez automatické SDN programy, ktoré si môžu ľudia písať sami, pretože nie sú proprietárne.



Obrázok 1. SDN architektúra [31]

Dátová rovina

Je zodpovedná za narábanie s paketmi na základe príkazov z riadiacej roviny. Môže zahadzovať, meniť a preposielať pakety. [30] [31] [32]

Operačná rovina

Je zodpovedná za informácie ako, či je daný forwarder zapnutý alebo vypnutý, koľko voľných portov má, stav na každom porte a iné. Táto rovina je koncovým bodom manažovacej roviny. Táto rovina referuje ku zdrojom sieťových zariadení ako porty, pamäť, atď. Môže to byť kľudne aj súčasť smerovacej roviny, ale definícia roviny je o rozdelení operácií nad nejakou časťou. [30] [31] [32]

Riadiaca rovina

Je zodpovedná za rozhodnutia, kam by mal byť paket poslaný jedným alebo viacerými sieťovými zariadeniami a tlačí tieto rozhodnutia dole na forwardery na vykonanie. Táto rovina sa hlavne zameriava na smerovaciu rovinu, a informácie pre ňu. Môže ale používať informácie z operačnej roviny, pre rozhodnutia o tom ako je daný port vyťažený, atď. Jeho hlavnou úlohou je vyladiť smerovaciu tabuľku ktorá ovláda smerovaciu rovinu, na základe sieťovej topológie alebo externých servisných žiadostí. Medzi riadiacou rovinou, čo je niekedy nazývaná aj ako „east-west“ interface, sa používa napr. BGP. [30] [31] [32]

Úlohou riadiacej roviny je [30] [31] [32]:

- Zistenie topológie a jej údržba
- Výber trasy paketu a konkretizácia
- Mechanizmus prerušenia trasy

Manažovacia rovina

Je zodpovedná za monitorovanie, konfigurovanie a udržiavanie sieťových zariadení. Zameriava sa hlavne na riadiacu rovinu. Môže byť použitá na konfiguráciu smerovacej roviny ale robí to málo krát, je komplikovanejšia ako riadiaca rovina. Napríklad môže nastaviť časť alebo celú tabuľku smerovacích pravidiel, ale táto akcia bude braná ako odporúčenie pre riadiacu rovinu, a nie ako prikazovanie. Hlavnou úlohou manažovacej roviny je [30] [31] [32]:

- Správa chýb a ich monitorovanie
- Správa konfigurácie

Aplikačná rovina

Miesto kde sa aplikácie a servisy ktoré definujú sieťové správanie nachádzajú. Aplikácie, ktoré priamo podporujú operáciu smerovacej roviny (napríklad smerovanie v riadiacej rovine) nie sú chápané ako časť aplikačnej vrstvy. Aplikácie môžu byť implementované tak, aby ovplyvňovali viacero vrstiev. [30] [31] [32]

2.2. Protokol OpenFlow

Pre praktické uplatnenie SDN sietí je potrebné splniť 2 požiadavky[12] [13]:

- v sieti musí byť spoločná logická architektúra v rámci všetkých prepínačov, smerovačov a iných sieťových zariadeniach, riadených SDN kontrolérom
- je potrebný bezpečný protokol medzi SDN kontrolérom a sieťovými zariadeniami

Obe tieto podmienky rieši OpenFlow. Stručne sa teda dá povedať, že je to protokol medzi kontrolérom a sieťovými zariadeniami, ako aj nástroj pre špecifikáciu logickej štruktúry siete. Ako už vyplýva z vyššie spomenutého, kontrolér a prepínač sú dve rozdielne súčasti týchto sietí. Kontrolér slúži na spravovanie siete „na diaľku“, pričom ovláda práve prepínače, ktorým posiela príkazy a smeruje do nich pakety. [12] [13]

Pracuje v rámci TCP (Transmission Control Protocol), kde by mal počúvať na porte 6653, na ktorý sa hlásia prepínače, ktoré chcú nadviazať spojenie. Vzdialené riadenie prebieha na vrstve 3, kde sa môžu vykonať rôzne akcie [12] [13]:

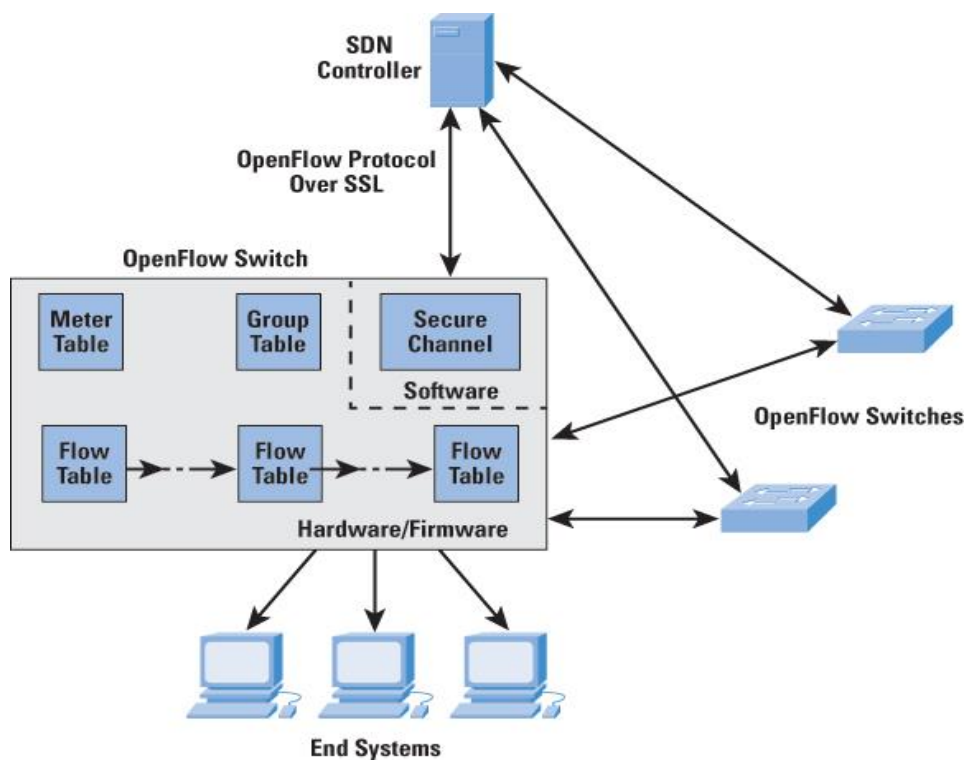
- pridávanie, zmena alebo vyradovanie paketov, podľa vopred definovaných pravidiel a akcií
- smerovanie akceptovaných paketov prepínačom
- neakceptované pakety sú smerované do kontroléra, ktorý môže:
 - zmeniť pravidlá smerovacej tabuľky na jednom alebo viacerých prepínačoch
 - nastaviť nové pravidlá, aby tak predišiel veľkej komunikácii medzi prepínačom a kontrolérom

V SDN sieťach sa vyskytuje viacero typov tabuliek, ako napríklad [12] [13]:

- prietoková tabuľka – spája pakety do určitých tokov a špecifikuje funkcie, ktoré sa majú vykonať na paketoch (môže ich byť aj viac)
- skupinová tabuľka – prietoková tabuľka sem môže smerovať tok, čo spustí viacero akcií, ktoré ovplyvňujú jeden alebo viacero tokov
- metrová tabuľka – môže spustiť viacero rôznych činností súvisiacich s výkonom na toku

Každá z tabuliek má určité súčasti, na základe ktorých pracujú. Ako príklad si môžeme uviesť súčasti prietokovej tabuľky [12] [13]:

- vyhovujúce polia – na ich základe vyberá vyhovujúce pakety
- priorita – relatívna priorita záznamov v tabuľke
- počítadlá – rôzne počítadlá a časovače definované OpenFlow
- inštrukcie – akcia, ktorá sa má vykonať, ak nastane zhoda
- pauzy – maximálny čas nečinnosti prepínača
- cookie – potrebné pre filtre, zmenu toku alebo jeho zmazanie (nepoužívajú sa, keď sa pakety spracovávajú)
- tabuľka chýbajúcich tokov – wildcard pre vyhovujúce polia



Obrázok 2. OpenFlow komunikácia [11]

Obrázok popisuje približnú komunikáciu v rámci OpenFlow protokolu [12] [13]:

- SDN kontrolér komunikuje s prepínačmi, ktoré sú kompatibilné s OpenFlow, pomocou OpenFlow protokolu bežiaceho cez SSL (Secure Sockets Layer)
- každý prepínač sa pripojí k zariadeniam cieľového používateľa, ktoré sú zdrojmi a cieľmi paketových tokov
- každý prepínač má niekoľko tabuliek (spomenutých vyššie), implementovaných hardvérom alebo firmvérom, ktoré sa používajú na riadenie toku cez prepínače

Bezpečnosť

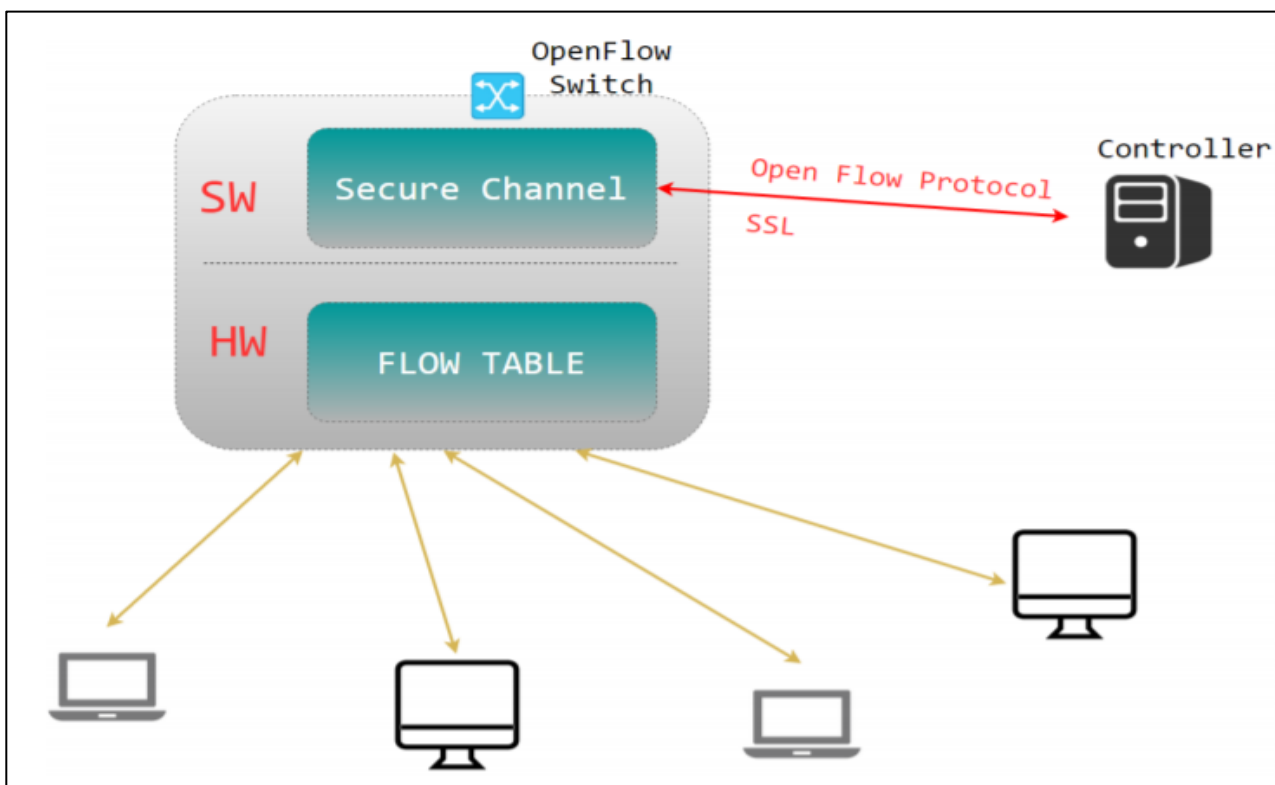
Ako všetky iné protokoly a komunikácie je aj OpenFlow náchylná na možné útoky. Medzi najtypickejšie útoky pre tento typ siete patria[12] [13]:

- man-in-the-middle útok
- single point útok a zlyhanie
- problémy spojené s programovacími a komunikačnými kanálmi

2.3. SDN forwardery

Preposielacie zariadenie pozostáva z dvoch častí [14]:

- prietokovej tabuľky (flow table) obsahujúcej záznam a akciu na prijímanie aktívnych tokov
- abstrakčnej vrstvy, ktorá bezpečne komunikuje s riadiacou jednotkou (kontrolérom) o nových vstupoch, ktoré sa v danom momente nenachádzajú v tokovej tabuľke.



Obrázok 3. Zobrazenie SDN prepínača a jeho vnútornej štruktúry pri komunikácii s kontrolérom [14]

2.3.1. Prepínač

Prepínače v SDN často predstavujú preposielací hardvér, ktorý je prístupný cez otvorené rozhranie. V sieti OpenFlow sa prepínače delia do dvoch základných skupín [14]:

- čisté (pure) – nemajú žiadnu funkcionality a spoliehajú sa len na kontrolér, ktorý rozhoduje o preposielaní paketov
- hybridné (hybrid) – okrem „tradičných“ protokolov a operácii podporujú aj OpenFlow.

OpenFlow prepínač pozostáva z jednej alebo viacerých tabuliek tokov, ktoré ukladajú záznamy pre vyhľadanie alebo presmerovanie paketov. [14]

Po príchode paketov na OpenFlow prepínač sa hlavička paketu extrahuje a porovná s políčkom zhody (match field) z tabuľky tokov. Ak sa zhoduje hlavička paketu s políčkom v tabuľke, tak prepínač použije príslušnú sadu inštrukcií spojenú s daným tokom. V prípade, že sa nenájde žiadna zhoda s tabuľkou tokov, tak nasledujúca akcia prepínača bude závisieť od inštrukcií definovaných v tabuľke chýbajúcich tokov (table-miss flow entry). [14]

Akcie obsiahnuté v tabuľke chýbajúcich tokov sú napr. zahodenie paketu, odoslanie paketu kontroléru cez OpenFlow kanál. [14]

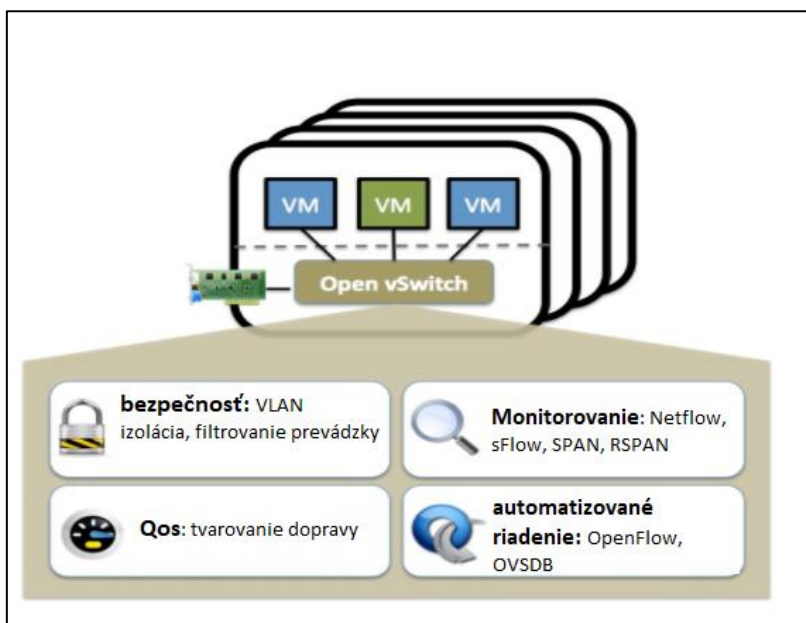
Tabuľka 1. Vlastnosti OpenFlow prepínačov [14]

| Soft. prepínač | Implementácia | Opis | Ryu | Verzia |
|----------------|---------------|--|-----|--------|
| Open vSwitch | C/Python | Podporuje štandardný manažment rozhraní a umožňuje programové rozšírenie a riadenie funkcií preposielania. Je možné ho aplikovať do štandardných prepínačov. [1] | ✓ | v1.0 |
| Pantou/OpenWRT | C | Mení komerčný bezdrôtový smerovač alebo prístupový bod na OpenFlow prepínač. [1] | X | v1.0 |
| OFSwitch13 | C/C++ | Kompatibilný s OpenFlow 1.3. [1] | X | v1.3 |
| Linc | Erlang | LINC je navrhnutý tak, aby pracoval v rôznych operačných systémoch | ✓ | v1.2 |
| Indigo | C | OpenFlow implementácia na fyzických prepínačoch využívajúca ich funkcie na spustenie OpenFlow. [1] | ✓ | v1.0 |

2.3.2. Open vSwitch (OVS)

Open vSwitch je kvalitný viacvrstvový virtuálny prepínač. Je navrhnutý tak, aby umožňoval masívnu automatizáciu siete pomocou programového rozšírenia a zároveň podporoval štandardný manažment rozhraní a protokolov. Open vSwitch podporuje OpenFlow verzie 1.0, 1.1, 1.2, 1.3. [15]

Open vSwitch má podporu pre štandardizované fyzické servery, linuxové distribúcie a Microsoft Windows. [16]



Obrázok 4. Vlasnosti Open vSwitch [17]

2.3.3. Pantou/OpenWRT

Pantou je založená na verzii BackFire 10.03 OpenWRT (Linux 2.6.32). Tento upravený balík OpenWRT obsahuje virtuálny prepínač, ktorý funguje ako aplikácia umožňujúca transformáciu bezdrôtového smerovača na prepínač. [15]

2.3.4. OFSoftSwitch13

Modul OFSoftSwitch13 poskytuje podporu pre protokol OpenFlow verzie 1.3, čím prináša prepínač a rozhranie kontroléra na simulátor ns-3. Komunikácia medzi kontrolérom a prepínačom je realizovaná cez protokol ns-3. Modul využíva externý OpenFlow 1.3 softvérový prepínač, ktorý je kompilovaný ako knižnica. [17]

2.3.5. Indigo Virtual Switch (IVS)

Projekt Indigo podporuje OpenFlow protokol na fyzických prepínačoch. Je navrhnutý pre vysoký výkon s nízkou administráciou. Indigo projekt podporuje len OpenFlow verziu 1.0. [15]

2.3.6. Linc

Linc je open source projekt, ktorý podporuje protokoly OpenFlow verzie 1.2 a 1.3. LINC je navrhnutý tak, aby využíval všeobecne dostupné komodity, x86 hardvér a pracoval v rôznych operačných systémoch (Linux, Mac, Windows a pod.) [15] Projekt je momentálne pozastavený.

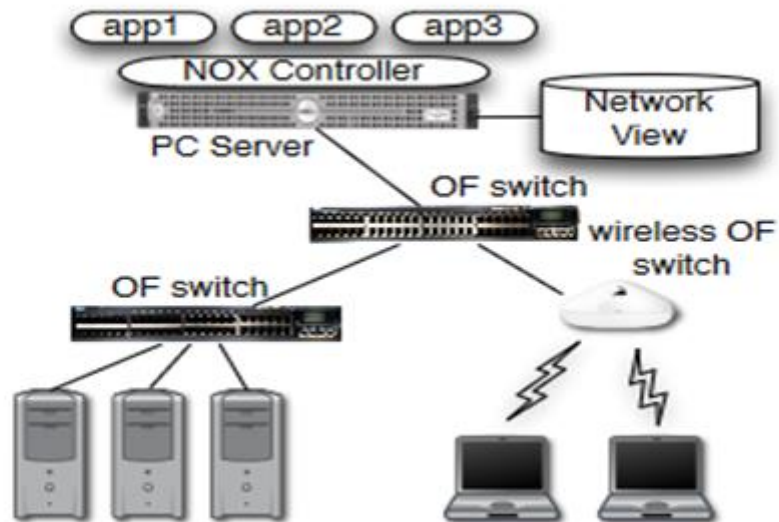
2.4. SDN kontroléry

OpenFlow kontrolér je typ SDN kontroléra, ktorý používa OpenFlow protokol. OpenFlow kontrolér používa OpenFlow protokol aby spojil a konfiguroval sieťové zariadenia, ako smerovače a prepínače, na nájdenie najlepšej cesty v premávke. SDN kontroléry teda uľahčujú riadenie siete a dokážu riadiť celú komunikáciu medzi aplikáciami a zariadeniami, tak aby sa čo najefektívnejšie upravil tok premávky, tak ako je v danom momente potrebné. Keď nie je riadiaca rovina implementovaná vo firmvéri, ale je implementovaná v softvéri, administrátori dokážu manažovať sieť oveľa jednoduchšie, viac dynamicky a s lepšou granularitou. OpenFlow kontrolér teda tvorí centrálnu kontrolnú jednotku v sieti, ktorá riadi všetku premávku, všetky zariadenia v sieti vykonávajú akcie tak ako od nich požaduje kontrolér a podporuje OpenFlow. OpenFlow je najpopulárnejší štandardný protokol používaný v SDN. [1] [2] [3]

2.4.1. NOX

NOX bol prvým open source kontrolérom pre OpenFlow, vydaným v roku 2008. NOX je open source OpenFlow kontrolér, ktorý poskytuje kontrolu nad sieťou s OpenFlow prepínačmi. Podporuje aplikácie v jazyku Python a C++, samotný kontrolér je napísaný v jazyku C++, reprezentácia pohľadu na sieť je formou indexovaných hashovacích tabuliek s vyrovnávacou pamäťou pre zrýchlenie toku v sieti. Jeden kontrolér vie spracovať 100 000 tokových požiadaviek za sekundu. NOX má len jednu globálnu dátovú štruktúru, pohľad na sieť, ktorý sa mení dosť pomaly na to aby bol udržiavaný centrálny a to aj vo veľmi veľkých sieťach. Informácie z tejto dátovej štruktúry slúžia na rozhodovanie sa a využívajú rozhodovanie v riadení systému. [4][5]

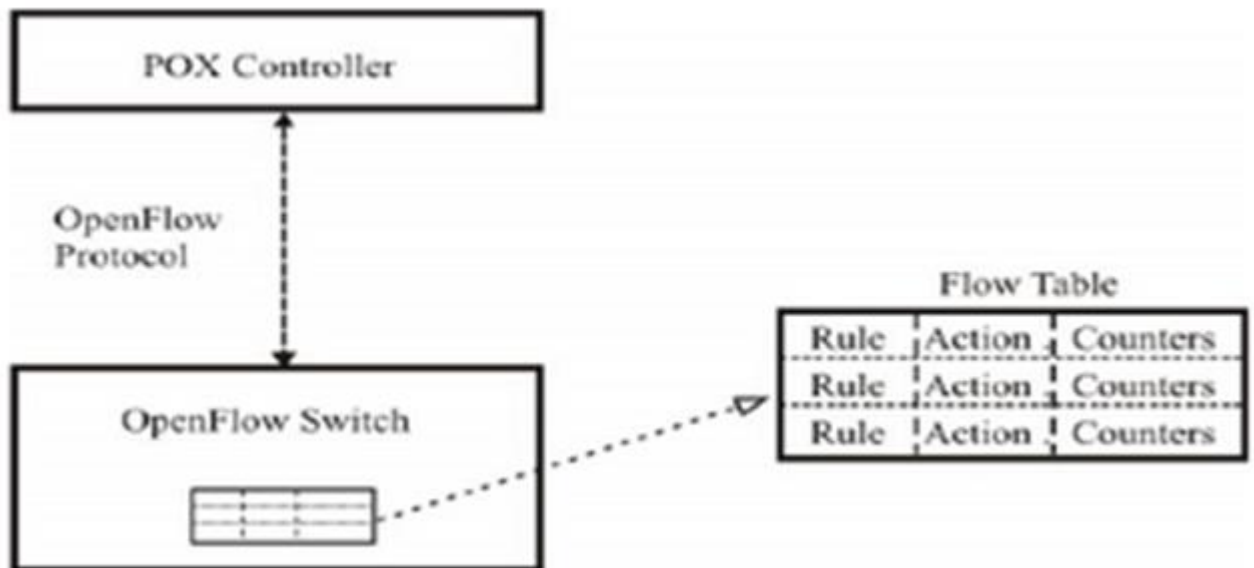
NOX používa rovnakú reprezentáciu. Pre každý paket, ktorý sa zhoduje v poli hlavička, je aktualizované pole počítadiel a sú vykonané príslušné akcie. Ak sa paket zhoduje s viacerými tokovými vstupmi, vyberie sa vstup s najväčšou prioritou. NOX podporuje OpenFlow verzie 1.3. [4] [5]



Obrázok 5. NOX kontrolér[4]

2.4.2. POX

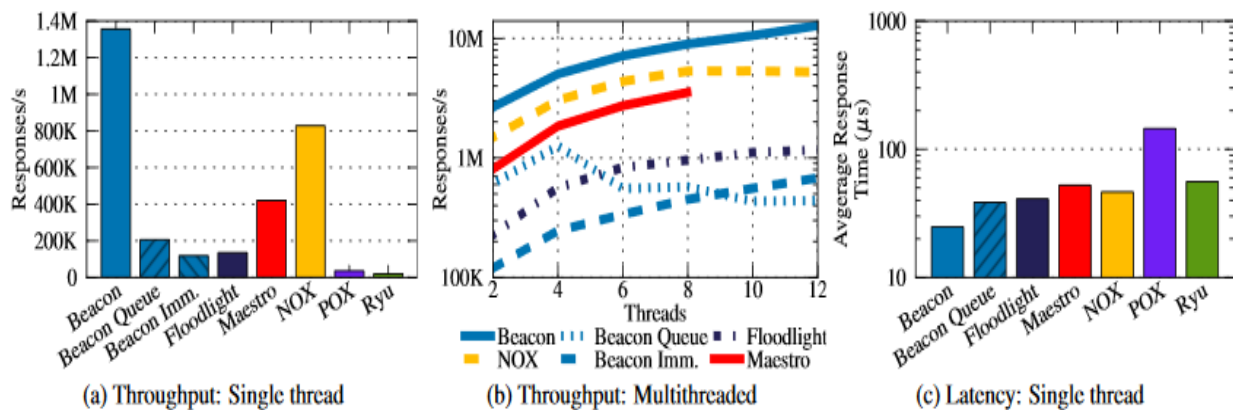
POX je open source kontrolér pre vývoj SDN aplikácií. POX kontrolér umožňuje efektívnu implementáciu OpenFlow protokolu do SDN. Tcp dump packet capture je nástroj, ktorý dokáže zachytiť pakety tečúce v sieti medzi kontrolérom a OpenFlow zariadeniami. POX kontrolér je založený na predošlom NOX kontroléri. Je napísaný v jazyku Python a je menej komplexný ako NOX. POX sa používa najmä na rýchle prototypovanie jednoduchších aplikácií pre SDN. Môže byť skombinovaný s reálnym hardvérom a je dobre kompatibilný s Mininetom, nemá však GUI rozhranie. Podporuje len však OpenFlow verzie 1.0. [6]



Obrázok 6. POX kontrolér[6]

2.4.3. Beacon

Beacon je OpenFlow kontrolér založený na Java a frameworku Spring. Bol vyvinutý najmä pre pohodlnejšie programovanie programátorov, rýchlejšiu kompiláciu a lepší výkon. Pri jazyku C++/Python kompilácia aj jednoduchších sietí mohla trvať aj 10minút, čo pri vývoji robilo problémy. Beacon dokáže pracovať tak aby v reálnom čase mohla začať alebo ukončiť aktuálnu alebo novú aplikáciu. Beacon na komunikáciu požíva Java knižnicu OpenFlowJ. Výhodou Beaconu je aj webové užívateľské rozhranie a stabilita. Podporuje však taktiež len OpenFlow verzie 1.0. [5]



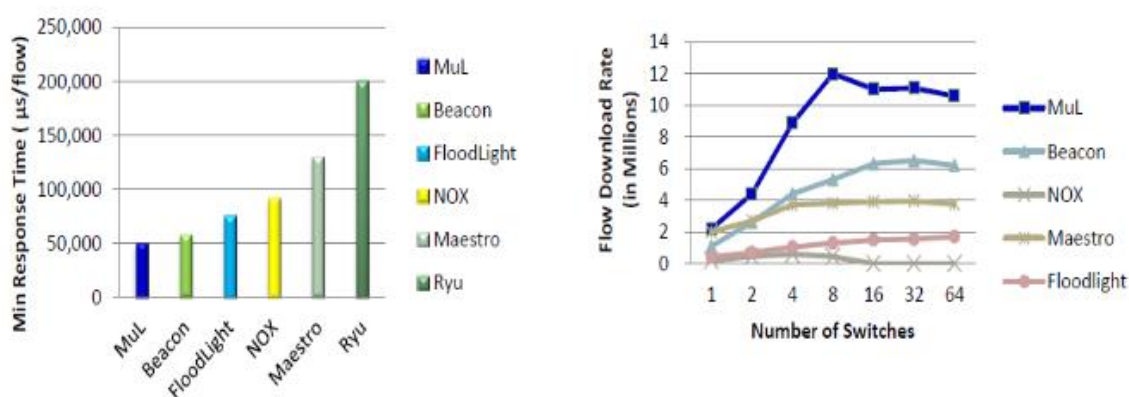
Obrázok 7. Porovnanie kontrolérov [5]

2.4.4. Floodlight

Floodlight je OpenFlow kontrolér založený na Java a frameworku Netty. Floodlight obsahuje modul OFSwitchManager pomocou, ktorého komunikuje s pripojenými switchami. Od ostatných kontrolérov sa líši najmä tým, že je synchronný. Návrh Floodlightu je však príliš náročný na dnešné produkčné prostredie, preto nevieme využiť jeho plný potenciál. Výhodou Floodlightu je naozaj jednoduchá inštalácia s minimálnymi závislosťami. Podporuje OpenFlow verzie 1.0, avšak Floodlight-plus podporuje aj verziu 1.3. [7]

2.4.5. MUL

MUL je OpenFlow kontrolér založený na jazyku C, vďaka ktorému dosahuje najlepšie výkonnostné hodnoty. Okrem jazyka C MUL kontrolér ponúka dve funkcie FirmFlow a FlexPlug, ktoré zvyšujú najmä bezpečnosť siete a flexibilitu siete. MUL podporuje najnovšiu verziu OpenFlow. [8]



Obrázok 8. Porovnanie kontrolérov[8]

2.4.6. Maestro

Maestro je OpenFlow kontrolér založený na Java a je prvým kontrolérom, ktorý umožňuje paralelizmus, aby zabezpečil čo najlepšiu výkonnosť. Okrem dobrého výkonu a paralelizmu neposkytuje žiadne veľké výhody. Podporuje OpenFlow verzie 1.0. [9]

2.4.7. Iné

Ďalšími alternatívami kontrolérov môžu byť aj Trema, Flowe a Nette, no tie žiaľ v dnešnej dobe ešte nespĺňajú podmienky plnohodnotných kontrolérov, pretože sú ešte vo vývoji a je na nich obmedzená funkcionálnosť. [2]

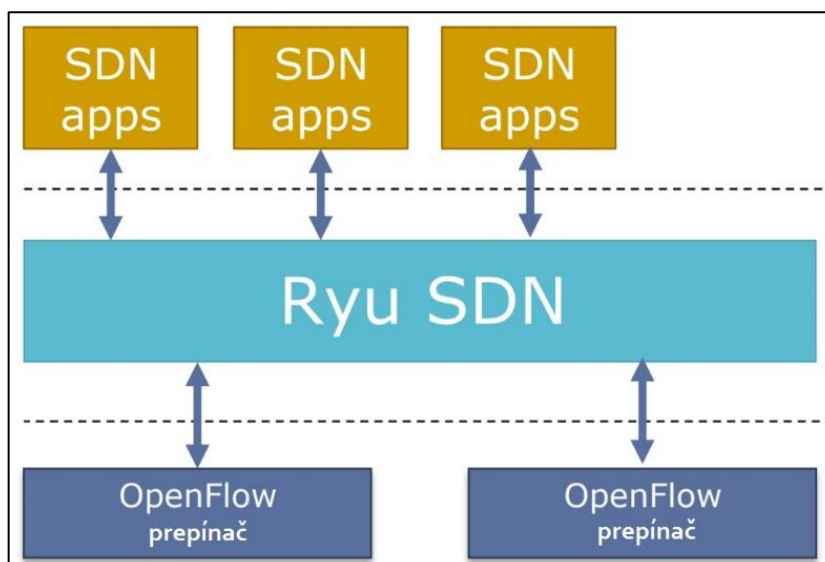
| Use-Cases \ Controllers | Trema | Nox/Pox | RYU | Floodlight | ODL | ONOS*** |
|---|---------|---------|---------|------------|---------|---------|
| Network Virtualization by Virtual Overlays | YES | YES | YES | PARTIAL | YES | NO |
| Hop-by-hop Network Virtualization | NO | NO | NO | YES | YES | YES |
| OpenStack Neutron Support | NO | NO | YES | YES | YES | NO |
| Legacy Network Interoperability | NO | NO | NO | NO | YES | PARTIAL |
| Service Insertion and Chaining | NO | NO | PARTIAL | NO | YES | PARTIAL |
| Network Monitoring | PARTIAL | PARTIAL | YES | YES | YES | YES |
| Policy Enforcement | NO | NO | NO | PARTIAL | YES | PARTIAL |
| Load Balancing | NO | NO | NO | NO | YES | NO |
| Traffic Engineering | PARTIAL | PARTIAL | PARTIAL | PARTIAL | YES | PARTIAL |
| Dynamic Network Taps | NO | NO | YES | YES | YES | NO |
| Multi-Layer Network Optimization | NO | NO | NO | NO | PARTIAL | PARTIAL |
| Transport Networks - NV, Traffic-Rerouting, Interconnecting DCs, etc. | NO | NO | PARTIAL | NO | PARTIAL | PARTIAL |
| Campus Networks | PARTIAL | PARTIAL | PARTIAL | PARTIAL | PARTIAL | NO |
| Routing | YES | NO | YES | YES | YES | YES |

Obrázok 9. Porovnanie kontrolérov[10]

2.5. Kontrolér Ryu

Ryu kontrolér je jeden z najznámejších a zároveň najpoužívanejších kontrolérov v SDN sieťach. Jeho názov vychádza z japončiny, kde v dvoch významoch znamená tok, resp. Japonský drak. Ryu je voľne šíriteľný sieťový operačný systém – programovacie sieťové rozhranie (logicky centralizované), voľne dostupné pod licenciou Apache 2. Najnovšia verzia k písaniu tohto dokumentu je 4.18. [24] [25] [26]

Ryu poskytuje veľmi silný pomer medzi jeho výhodami a nevýhodami, v prospech výhod. Ryu vďaka svojmu rozšíreniu a štandardu kontroléra má širokú základňu aktívnych používateľov, ktorí vytvárajú masívny zdroj informácií a spätných väzieb pre vývojárov, preto je stále v aktívnom vývoji a udržiavaní kódu. Taktiež je považovaný za formálny štandard pre OpenStack, čo je voľne šíriteľný softvér pre stavanie privátnych a verejných cloudových riešení. Medziiným poskytuje konzistentnú topologizáciu na druhej vrstve OSI modelu nezávisle od tej fyzickej. [24] [25] [26]



Obrázok 10. Štruktúra SDN z hľadiska vrstiev[24] [25] [26]

Ryu si stavia medzi jej hlavne piliere výhod tri charakteristiky: kvalita kódu, funkcionálna a použiteľnosť. Podporuje niekoľko protokolov pre správu sieťových zariadení, tými sú napr. Netconf, OF-config, no primárne OpenFlow vo verziách až po aktuálnu 1.5 (k písaniu dokumentu) spolu s rozšíreniami Nicira. Napriek množstvu typových produktov na trhu je principiálne nezávislý na výrobcovi a tak schopný spolupráce. Ryu je najmä kvôli prvému bodu vhodná pre rozsiahle produkčné prostredia. Na druhej strane, práve implementáciou kódu, ktorá je v plnej miere v programovacom jazyku Python, je oproti ostatným riešeniam značne pomalšia. Z hľadiska kompatibility, pracuje s verziami jazyka Python 2.7, 3 a 3.4. [24] [25] [26]

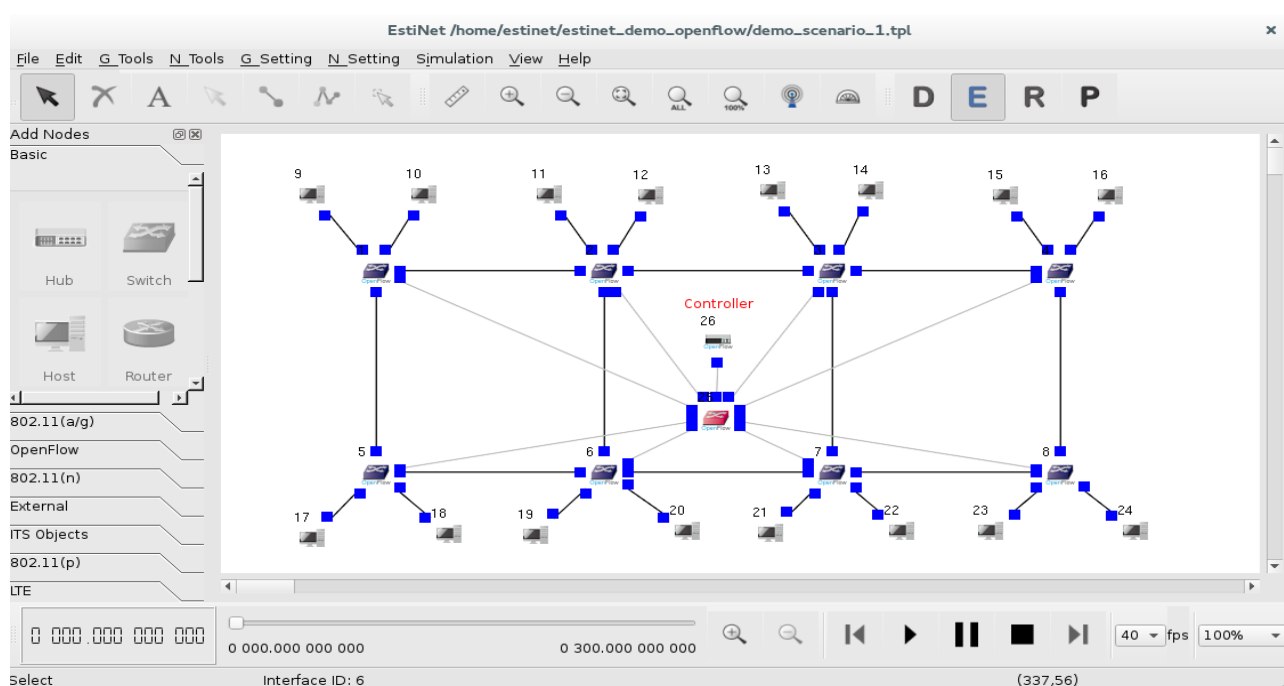
Ryu ako softvér pozostáva z určitých častí – nazývané komponenty. Komponent je samostatný blok, ktorý slúži ako rozhranie pre kontrolu a vytváranie udalostí. Takýchto komponentov poskytuje Ryu v základe niekoľko. Tie existujúce sa dajú modifikovať, taktiež je možné pridávať a kombinovať nové a komunikácia prebieha na základne nezávislých správ. Vstavané komponenty sú v jazyku Python a pozostávajú z vlákien a procesov operačného systému. Dodatočné časti môžu byť naviazané aj na už existujúce, musí byť však dodržaná metodika správ. Medzi dostupné komponenty patria napríklad OF-wire, Topology, VRRP, OF REST, Endpoint alebo Stats. Zoskupením niektorých častých častí pre potreby efektivity sú vytvorené knižnice. Tie obsahujú metódy, ktoré sú volané priamo komponentmi. Takéto knižnice sú Netconf, OF-conf, sFlow, NetFlow, či OVSDB JSON. [24] [25] [26]

2.6. Simulačné/emulačné prostredia

Existuje niekoľko vývojových platforiem, pomocou ktorých je možné simulovať SDN projekty. Umožňuje vytváranie a vykonávanie experimentov. [15]

2.6.1. EstiNet

EstiNet umožňuje simuláciu niekoľkých sieťových protokolov. Jeden z nich je OpenFlow protokol. EstiNet má kvalitné grafické vlastnosti pri vizualizácii experimentu. Vizualizuje animáciu paketov plus grafická reprezentácia jednotlivých uzlov v sieti. [15]



Obrázok 11. Prostredie EstiNet[15]

2.6.2. Ns-3

Ns nemá kompatibilitu s novou verziou OpenFlow protokolu. Rovnako nepodporuje openflow kontroléry: NOX, POX alebo Floodlight bez modifikácií. [15]

2.6.3. Trema

Trema poskytuje všetky prostriedky pre vývoj a experimentovanie SND siete. Má integrované testovacie a debugovacie prostredie, ktoré monitoruje, diagnostikuje celú sieť (Trema shark, Wireshark plug-in). Avšak neposkytuje grafické rozhranie a samotný simulátor je napísaný v jazyku C a Ruby, čo limituje popularitu danej platformy. [15]

2.6.4. Mininet

Mininet je sieťový emulátor. Umožňuje vytvárať koncové zariadenia, prepínače, smerovače, a linky medzi nimi na jednom Linuxovom kerneli. Mininet host sa správa rovnako ako reálna mašina a je možné sa naň pripojiť pomocou SSH. Čo sa týka OpenFlow kontrolérov, Mininet je veľmi flexibilný a umožňuje pridať do simulácie množstvo typov kontrolérov. [15]

Nevýhody Mininetu[15]:

- Je kompatibilný jedine s operačným systémom Linux
- Neposkytuje siete s rýchlosťou 100Gbps.

3. Generátory premávky

V mininete beží takmer každý linuxový program. Takže je možné využiť takmer akýkoľvek klientský alebo serverový program pre tvorbu premávky (ping, iperf, wget, curl, netperf, netcat,...). Premávku je možné zachytiť pomocou programov wireshark a tcpdump. Vygenerovať pakety je tiež možné pomocou jazyka python za využitia Scapy. [18]

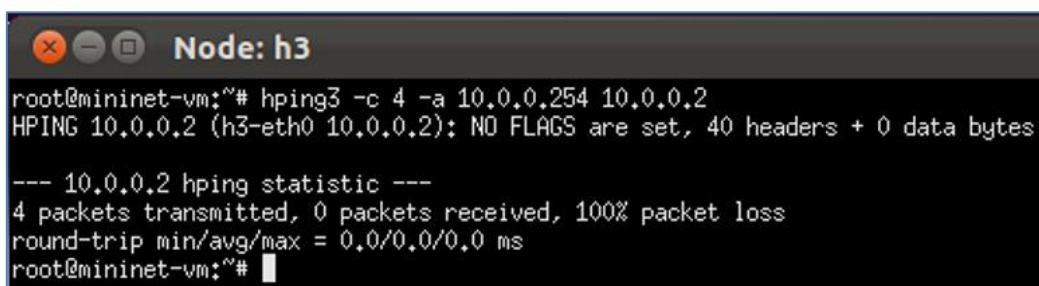
Kapitola opisuje a porovnáva voľne dostupné generátory premávky, akými sú napr. Hping, Ostinato, Scapy, packETH.

3.1. Hping

Hping je jedným z najznámejších a bezplatných nástrojov pre tvorbu premávky. Umožňuje zostaviť vlastné ICMP, UDP, TCP a Raw IP pakety. Nástroj bol v minulosti používaný hlavne na bezpečnosť, ale je možné ho využívať aj v iných smeroch ako: [19]

- Testovanie firewall
- Rozšírené skenovanie portov
- Testovanie siete pomocou rôznych protokolov

Nástroj nemá používateľské rozhranie, teda je možné ho spúšťať len cez príkazový riadok. Hping má traceroute režim a tiež schopnosť odosielania súborov medzi krytým kanálom. [19]



```
root@mininet-vm:~# hping3 -c 4 -a 10.0.0.254 10.0.0.2
HPING 10.0.0.2 (h3-eth0 10.0.0.2): NO FLAGS are set, 40 headers + 0 data bytes

--- 10.0.0.2 hping statistic ---
4 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
root@mininet-vm:~#
```

Obrázok 12. Príklad výsledku použitia hping3 príkazu

Hping nie je v štandardnej inštalácii a treba ho doinštalovať pomocou príkazu: `sudo apt-get install hping3`

Základné testovanie

Hping3 umožňuje testovať všetky vyššie spomenuté prípady, umožňuje posielanie súborov, aj keď sa nepodarilo nájsť posielanie videí. Hping3 umožňuje testovanie viacerých funkcionalít naraz ale z rôznych koncových zariadení či prepínačov. Tiež je možné otestovanie rôznych prípadov napadnutí.

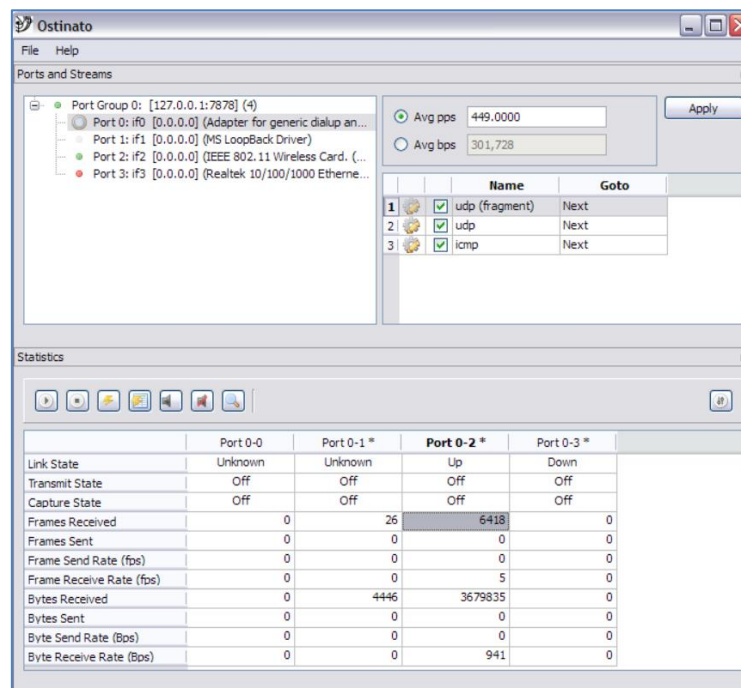
3.2. Ostinato

Ostinato je voľne dostupný paketový generátor a analytický nástroj. Na rozdiel do Hping využíva používateľské rozhranie, ktoré umožňuje jednoduché používanie. [18]

Nástroj podporuje najbežnejšie protokoly:

- Ethernet/802.3/LLC SNAP
- VLAN (aj QinQ)
- ARP, IPv4, IPv6, IP Tunneling
- TCP, UDP, ICMP
- každý textový protokol (http, SIP, RSTP,...)

Pomocou programu je možné ľahko upraviť ľubovoľné pole každého protokolu. [18]

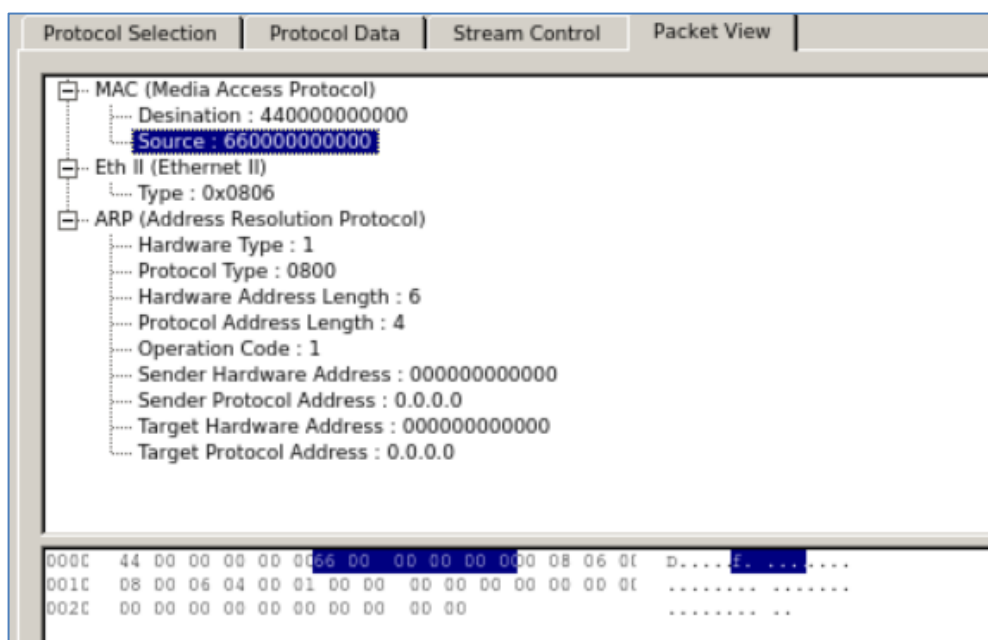


Obrázok 13. Používateľské rozhranie

Ostinato podobne ako Hping treba stiahnuť pomocou príkazu: `sudo apt-get install ostinato`. Princíp spúšťania nástroja na virtuálnom zdroji je popísaný v [20].

Základné testovanie

Pri Ostinato je pekné, že je možné vidieť štatistiky jednotlivých portov a tiež je možné v ňom si rozbaľiť pakety. Tiež má veľmi dobrú používateľskú príručku¹. Od verzie 6.0 Ostinato podporuje Python skriptovanie, vďaka ktorému je možné rozšíriť Ostinato o ďalšie protokoly. Ostinato umožňuje otváranie a zmenu PCAP súborov a tiež je možné ho spúšťať a testovať na viacerých koncových uzloch naraz. Nikde sa ale nenašla možnosť preposielania videa.



Obrázok 14. Rozbalenie paketu pomocou Ostinato

3.3. Scapy

Scapy je ďalší nástroj na vytváranie paketov, ktorý bol napísaný v Pythone. Môže dekódovať alebo vytvárať pakety pre širokú škálu protokolov. Môže vykonávať rôzne úlohy vrátane skenovania, vyhľadávania, skúšania alebo testovania siete. [18]

Na rozdiel od iných nástrojov umožňuje aj odosielanie neplatných rámcov, vytváranie si vlastných 802.11 rámcov, kombinovanie rôznych technológií (VLAN hopping + ARP cache poisoning, VOIP na šifrovanom kanále WEP,...). [21]

¹ <https://userguide.ostinato.org/Quickstart.html>

Princíp fungovania Scapy je popísaný v [22].

Základné testovanie

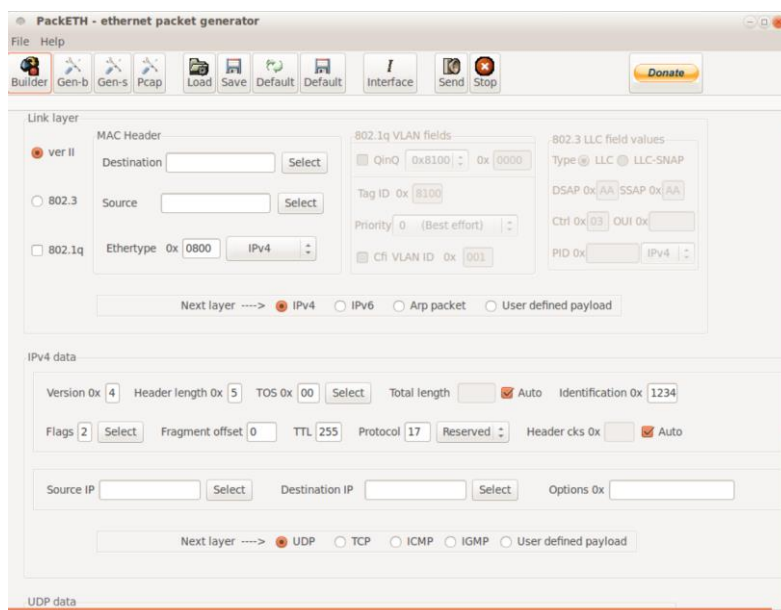
Príkazom `sudo scapy` spustíme scapy rozhranie. Nevýhodou je, že je potreba manuálne si tvoriť pakety na posielanie. Výhodou zas je, že je možné si testovacie scenáre ukladať do Python skriptov a tie následne testovať. Do skriptu si je možné zvoliť aj typ prenášania (video, obrázok).

```
"Node: h3"
root@mininet-vm:~/mininet/custom# sudo scapy
INFO: Can't import python gnuplot wrapper . Won't be able to plot.
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().
INFO: No IPv6 support in kernel
WARNING: No route found for IPv6 destination :: (no default route?)
Welcome to Scapy (2.2.0)
>>> █
```

Obrázok 15. Scapy rozhranie

3.4. packETH

PackETH je linuxový nástroj pre vytváranie paketov pomocou používateľského rozhrania. Umožňuje rýchlo vytvoriť a odoslať sekvenciu paketov. Rovnako ako ostatné nástroje podporuje rôzne protokoly. Je možné tiež nastaviť počet paketov a oneskorenie medzi nimi. [18]



Obrázok 16. Používateľské rozhranie

Nástroj podporuje vytvorenie a odoslanie akéhokoľvek ethernetového rámcu. Podporované protokoly sú [23]:

- ethernet II, ethernet 802.3, 802.1q, QinQ, užívateľom definovaný ethernetový rámec
- ARP, IPv4, IPv6
- UDP, TCP, ICMP
- RTP
- JUMBO rámce

Základné testovanie

Generátor je možné stiahnuť pomocou príkazu `sudo apt-get install packeth`. A spustiť ho je možné pomocou príkazu `packeth`, kedy sa zobrazí používateľské rozhranie. Cez rozhranie je možné posielat' pakety buď po jednom alebo aj viacej naraz. Tiež je možné si v súbore uložiť IP a z daného súboru ich načítavať (žiaľ súbor musí mať pevne definovaný formát). S tým, že je možné posielat' aj minimálne pakety obsahujúce zvuk².

Pri tomto generátore ale nastal problém, že aj keď je možnosť posielanie len jedného paketu, pošle sa ich omnoho viac naraz, ako je znázornené v obrázku nižšie.

| | | | | | | | |
|------|--------------|-------------------|-------------------|-----|----|-------------------------------------|------------------|
| 8698 | 60.710995000 | 66:00:00:00:00:00 | 44:00:00:00:00:00 | ARP | 60 | Who has 192.168.0.4? | Tell 192.168.0.6 |
| 8699 | 60.711069000 | 66:00:00:00:00:00 | 44:00:00:00:00:00 | ARP | 60 | Who has 192.168.0.4? | Tell 192.168.0.6 |
| 8700 | 60.711071000 | 66:00:00:00:00:00 | 44:00:00:00:00:00 | ARP | 60 | Who has 192.168.0.4? | Tell 192.168.0.6 |
| 8701 | 60.711104000 | 66:00:00:00:00:00 | 44:00:00:00:00:00 | ARP | 60 | Who has 192.168.0.4? | Tell 192.168.0.6 |
| 8702 | 60.711105000 | 66:00:00:00:00:00 | 44:00:00:00:00:00 | ARP | 60 | Who has 192.168.0.4? | Tell 192.168.0.6 |
| 8703 | 60.711127000 | 66:00:00:00:00:00 | 44:00:00:00:00:00 | ARP | 60 | Who has 192.168.0.4? | Tell 192.168.0.6 |
| 8704 | 60.711128000 | 66:00:00:00:00:00 | 44:00:00:00:00:00 | ARP | 60 | Who has 192.168.0.4? | Tell 192.168.0.6 |
| 8705 | 60.711203000 | 66:00:00:00:00:00 | 44:00:00:00:00:00 | ARP | 60 | Who has 192.168.0.4? | Tell 192.168.0.6 |
| 8706 | 60.711269000 | 66:00:00:00:00:00 | 44:00:00:00:00:00 | ARP | 60 | Who has 192.168.0.4? | Tell 192.168.0.6 |
| 8707 | 60.711314000 | 44:00:00:00:00:00 | 66:00:00:00:00:00 | ARP | 42 | 192.168.0.4 is at 44:00:00:00:00:00 | |

Obrázok 17. Zobrazenie ARP pre PackETH generátor

² <https://packeth.wordpress.com/2014/09/04/sending-rtp-stream-with-packeth/>

4. QoS z hľadiska poskytovania služieb na sieti

4.1. VOIP

Voice over IP, jedná sa o hovorovú komunikáciu cez internet. [27]

4.1.1. Latencia

Ľudia, ktorí bežne telefonujú zachytia oneskorenie 250ms alebo viac. ITU-T G.114 odporúčajú maximálnu latenciu 150ms jedným smerom. Takže sieť by mala mať porovnateľne menej ako 150ms latenciu. Rôzni poskytovatelia a ich maximálny limit pre latenciu [28]:

- Axiowave SLA 64ms max
- Internap SLA 45ms max
- Qwest SLA 50ms max
- Verio SLA 55ms
- Max 150 ms

4.1.2. Odchýlka

Rôzni poskytovatelia a ich maximálny limit pre odchýlku [29]:

- Axiowave SLA 0.5ms max
- Internap SLA 0.5ms max
- Qwest SLA 2ms max
- Verio SLA 0.5ms priemerne, nesmie prekročiť 10ms viac ako 0.1 percento času
- Viterla SLA 1ms max
- max 30ms

4.1.3. Stratovosť paketov

VOIP nie je tolerantné ku strate paketov. Už jedno percentné straty výrazne znižujú VOIP volanie používajúce G.711, a iné kompresné algoritmy tolerujú ešte menšie straty. Ideálne sú žiadne straty paketov v VOIP.

Rôzny poskytovatelia a ich maximálny limit pre stratovosť paketov [29]:

- Axiowave SLA 0 max
- Internap SLA 0.3 max
- Qwest SLA 0.5 max
- Verio SLA 0.1 max

4.2. Video interaktivita

Interaktívna komunikácia pomocou videa. Napríklad cez službu Skype, keď máme video konferenciu.

4.2.1. Latencia

Nie viac ako 150ms. V živej video konferencii alebo pri hraní hry, sa považuje pod 100ms ako nízka latencia, pretože ľudské oko ju nevie ešte výrazne zaznamenať. Pri komunikácii počítača s obsahom videa, je normálne uvažovať ešte menšie ako 30, 10 alebo dokonca pod 1 ms. [27]

4.2.2. Stratovosť paketov

Strata nie viac ako 1 percento. [27]

4.2.3. Odchýlka

Nie viac ako 30ms. [27]

4.2.4. Šírka pásma

Je 20 a viac percent z potrebnej prenosovej rýchlosti. Napríklad 384-kbps video konferenčné stretnutie potrebuje 460-kbps garantovanej šírky pásma. [28]

4.3. Video stream

Jednoduchý stream videa z Internetu, napríklad z portálu Youtube.

4.3.1. Latencia

Napríklad 100ms/33.3ms na 1 rámeček je rovný strate 3 rámcov. Nemala by byť väčšia ako 4 až 5 sekúnd. [28]

4.3.2. Stratovosť paketov

Strata by nemala byť väčšia ako 5 percent. [28]

4.3.3. Odchýlka

Nemá žiadne signifikantné obmedzenia. [28]

4.3.4. Šírka pásma

Záleží od rozlíšenia a kódovania. [28]

| QCI | Bearer Type | Application Example | Packet Delay | Packet Loss | Priority |
|-----|-------------|---|--------------|-------------|----------|
| 1 | GBR | Conversational VoIP | 100ms | 10^{-2} | 2 |
| 2 | | Conversational Video (Live Streaming) | 150ms | 10^{-3} | 4 |
| 3 | | Non-Conversational Video (Buffered Streaming) | 300ms | 10^{-6} | 5 |
| 4 | | Real Time Gaming | 50ms | 10^{-3} | 3 |
| 5 | NON-GBR | IMS Signaling | 100ms | 10^{-6} | 1 |
| 6 | | Voice, Video, Interactive Games | 100ms | 10^{-3} | 7 |
| 7 | | Video (Buffered Streaming) | 300ms | 10^{-6} | 6 |
| 8 | | TCP apps (web, email, ftp) | | | 8 |
| 9 | | Platinum vs. gold user | | | 9 |

Obrázok 18. Stratovosť a oneskorenia paketov VoIP, Video, Hry, TCP aplikácie [27] [28] [29]

4.4. Dáta

Nad dátami pre QoS sú štandardy dosť všeobecné. Majú mať dostatočne veľkú šírku pásma, ale nemajú veľmi obmedzovať celkovú priepustnosť siete. V zozname sa nachádzajú na konci, pretože nemajú veľmi špecifické nároky na QoS.

5. Meranie QoS v SDN

5.1. Odchýlka, stratovosť, priepustnosť

Po krátkom hľadaní sme zistili, že túto funkcionality umožňuje samotný Mininet. Po zadaní pár príkazov, si dokážeme nastaviť obmedzenia aké chceme.

Obmedzovať linku môžeme dvoma spôsobmi. Jedným z nich je posielat' priamo http správy na prepínač. Tieto správy majú rovnaký obsah v podstate ako tie, ktoré budeme používať z príkazového riadku. Líšia sa len tým, že musia mať formát JSON.

Príklad vytvorenia radu, s maximálnou priepustnosťou 4MB a minimálnou 2MB, pomocou príkazu v príkazovom riadku. [35] Príkaz nastaví parametre pre linku eth1 na požadovanom prepínači.

```
ovs-vsctl -- set port eth1 qos=@newqos -- \
--id=@newqos create QoS type=linux-htb other-config:max-rate=10000000 queues=0=@q0 -- --id=@q0
create Queue other-config:min-rate=4000000 other-config:max-rate=8000000
```

```
ovs-vsctl -- set port eth1 qos=@newqos -- \
--id=@newqos create QoS type=linux-htb other-config:max-rate=4000000
```

Príkaz sa na začiatku používa tak, že sa zadá len maximálna priepustnosť. Tento príkaz ovplyvňuje všeobecnú premávku cez tento port. Pri pridaní ďalšieho radu, sa už ale dá pridať aj minimálna priepustnosť.

Nastavenie na začiatku pre „port“ je konkrétne rozhranie na zariadení. Teda je potrebné si dať najprv výpis pripojených rozhraní a podľa neho priradovať konkrétne rozhranie, do nášho príkazu.

Pre vymazanie vytvoreného radu sa použije tento príkaz, na zvolenom prepínači.

```
ovs-vsctl -- --all destroy QoS -- --all destroy Queue
```

Pre zobrazenie konfigurácie na konkrétnom porte použijeme tento príkaz, na zvolenom prepínači. Je veľmi potrebné si strážiť názov rozhrania.

```
ovs-appctl -t ovs-vswitchd qos/show eth1
```

Vymazanie konfigurácie z konkrétneho rozhrania.

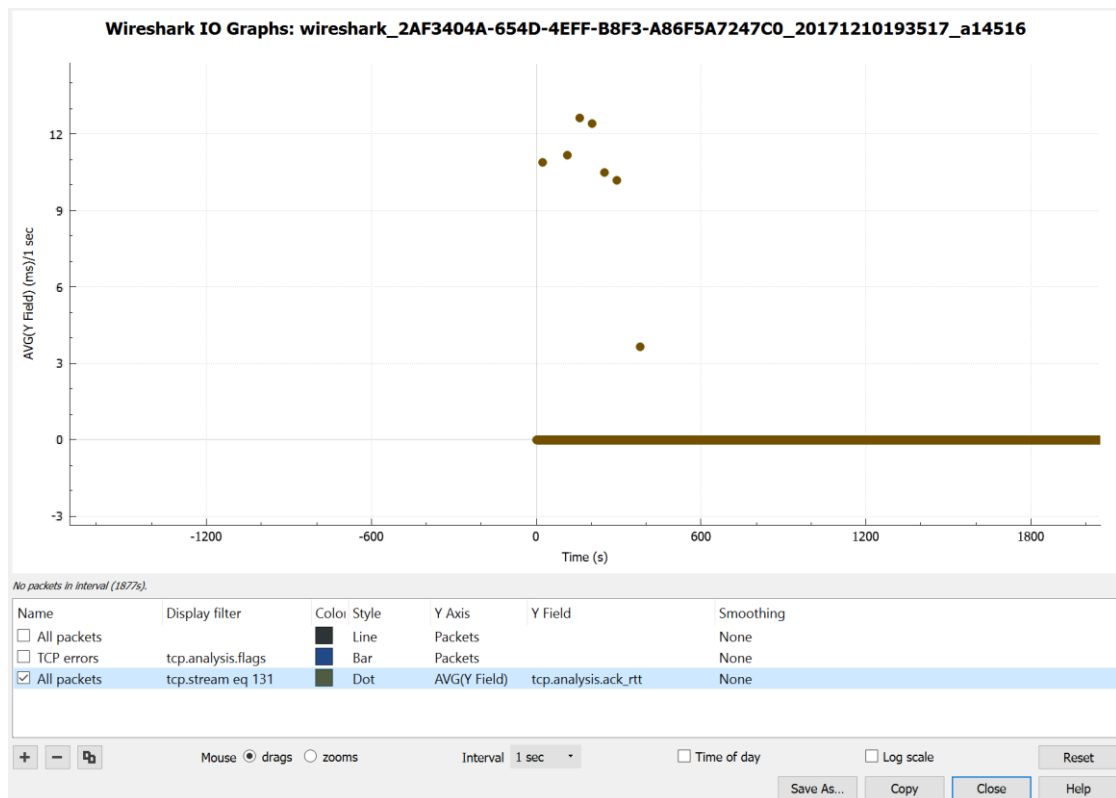
```
Ovs-vsctl -- clear Port eth1 qos
```

5.2. Oneskorenie

Tento postup ale platí pre analyzovanie len QoS kritérií ako „jitter“, stratovosť paketov a priepustnosť, pre oneskorenie je to výrazne náročnejšie. Pre UDP toky správy je to veľmi náročné, pretože musia byť oba stroje zosynchronizované časovo, ale pre TCP sa to dá merať priamo v programe Wireshark.

5.3. TCP oneskorenie

Pre meranie TCP oneskorenia stačí v programe Wireshark zvoliť konkrétnu TCP komunikáciu, a pri pravom kliknutí na jej inicializačný paket, zvoliť „Follow TCP stream“. Označí sa nám konkrétna TCP komunikácia, a odfiltruje sa ostatná. Následne už stačí zvoliť v hornom menu „statistics“ a „TCP stream graphs“->„Round Trip Time“ a na grafe vidíme RTT (Round trip time) tejto komunikácie. Oneskorenie sa dá odhadnúť aj ako polovica z RTT času. Pre lepšie zobrazenie si môžeme ešte zobrazit' iný graf, ktorý nájdeme tiež v možnosti „statistics“ v hornom menu a zvolíme „I/O Graph“. Na tomto grafe si môžeme pridávať rôzne komunikácie a aj ich porovnávať, nás ale zaujíma jedna konkrétna a tú máme stále vyfiltrovanú v zozname zachytených paketov. Skopírujeme si nastavenie filtra tesne nad týmto zoznamom, napríklad „tcp.stream eq 131“. V okne „IO Graph“ vidíme nižšie zoznam dát, ktoré sú nastavené a vizualizujú sa na grafe. Pri ich zaškrtnutí naľavo, sa aktivuje dané nastavenie a zobrazia sa jeho údaje. My si jeden z týchto nastavení ale zmeníme na to čo potrebujeme sledovať. Pri možnosti „Display filter“ nakopírujeme, už spomínané nastavenie filtra. V „Y Axis“ si zvolíme „AVG(Y Field)“ a do „Y Field“ si pridáme „tcp.analysis.ack_rtt“. Ak sme nastavili správne, môžeme vidieť na grafe priemerné časy RTT nášho zvoleného TCP toku.



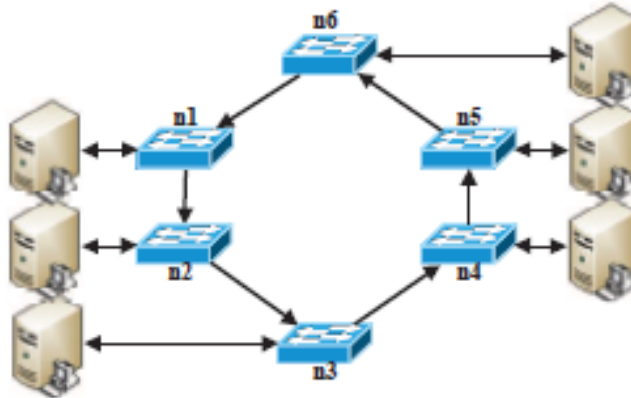
Obrázok 19. Nastavenie IO Grafu vo Wiresharku

5.4. UDP oneskorenie

Pre meranie tejto veličiny sme našli program „ULTRA_PING“. [36] Tento program umožňuje vytvárať rôzne frekvencie tokov aj veľkostí pre UDP transportný protokol a následne aj zobrazit' ich na grafe. Umožňuje aj meranie oneskorenia paketov z viacerých zariadení naraz. Spúšťa sa tak, že sa na jednom počítači spustí tento program ako server a na druhom ako klient. Je schopný merať aj RTT aj „jednosmerné RTT“, teda oneskorenie. Programy sú dva echo.py a quack.py. Echo.py vykonáva RTT a quack.py len jednosmerné oneskorenie.

6. Kvalita služieb v reálnom čase

Pre overenie modelu článku guck2014 sa porovnávali v článku spomenuté algoritmy – Greedy a MIP. Analyzovala sa 6-uzlová, jednosmerná, kruhová topológia (Obrázok 20. Topológia).



Obrázok 20. Topológia

Každý uzol pridáva premávku (traffic mix) do ringu a každý port má 4 výstupné rady (output queues) skombinované plánovaním priorít (priority scheduling). Maximálna dĺžka hopov v systéme je 3 a sleduje nasledovné požiadavky:

- 70% premávky bude prenášaných na jeden hop
- 20% premávky bude prenášaných na 2 hopy
- 10% premávky bude prenášaných na 3 hopy

Premávka obsahuje štyri triedy prevádzky (service classes), uvedených v tabuľke nižšie. Rozdelenie týchto tried je rôzne, aby každá trieda zahŕňala aspoň 5% premávky. Postupne sa premávka zvyšuje alebo znižuje, v kroku = 5%.

Tabuľka 2. Služby pre hodnotenie

TABLE II. SERVICE CLASSES FOR THE EVALUATION

| Name | r_c | b_c | t_c |
|---------|-----------|---------|-------|
| $c = 1$ | 10kByte/s | 100Byte | 5ms |
| $c = 2$ | 10kByte/s | 100Byte | 10ms |
| $c = 3$ | 10kByte/s | 100Byte | 20ms |
| $c = 4$ | 10kByte/s | 100Byte | 50ms |

7. QoS algoritmus plánovania tokov

Na plánovanie tokov v sieti využijeme QoS algoritmus plánovania tokov. Algoritmus pozostáva z dvoch častí, genetický algoritmus a mierne modifikovaný Network Calculus model. Genetický algoritmus má na starosti nájdenie vhodnej cesty v sieti pre daný tok. Nájdené cesty sú vyhodnotené pomocou trojice metrik: oneskorenie, priepustnosť a odchýlka. Metriky sa vyrátajú použitím Network Calculus modelu. Každý tok v sieti má okrem zdrojového a cieľového uzla, začiatočného času a príchodovej krivky $A(t)$ ešte trojicu QoS parametrov (w_d, w_j, w_t) , predstavujúcu oneskorenie, odchýlka a priepustnosť, ktoré sú požadované pre daný typ toku. Algoritmus teda hľadá cestu, ktorá splní požiadavky toku:

$$x_d \leq w_d \text{ a } x_j \leq w_j \text{ a } x_t \geq w_t \quad (1)$$

kde (x_d, x_j, x_t) reprezentuje celkové oneskorenie, odchýlka a priepustnosť na trase.

7.1. Genetický algoritmus

Algoritmus sa snaží nájsť najlepšiu cestu pre konkrétny tok. Na ohodnotenie každej cesty využíva jej QoS metriky, ktoré získa použitím network calculus modelu. Algoritmus prebieha 10 iterácií, pričom sa snaží získať pri každej iterácii lepšiu cestu, než akú má doteraz. Po 10 iterácii končí a vráti najlepšiu cestu. Algoritmus môže skončiť predčasne v prípade ak nájde cestu, ktorú ohodnotí hodnotou 0. Počiatočné 2 cesty získa použitím Dijkstrovho algoritmu.

7.1.1. Kríženie

Pokiaľ majú 2 cesty P1, P2 spoločný uzol (okrem začiatočného a koncového), tak môžeme aplikovať kríženie. Krížením vzniknú 2 nové cesty. Nová cesta N1 obsahuje začiatočné uzly z P1 po stredný uzol, stredný uzol, uzly až po koncový uzol z cesty P2. Nová cesta N2 má obrátené použitie ciest P1 a P2.

7.1.2. Mutácia

Zvolí sa vnútorný uzol v ceste, ktorý má najväčšie oneskorenie. Vyhodnotíme či, existuje cesta z predošlého uzla na nasledujúci uzol bez použitia cesty na zvolený uzol, ktorý má najväčšie oneskorenie. Pokiaľ existuje, získame novú cestu.

7.1.3. Iterácia

V nasledujúcej časti je opísaný priebeh konkrétnej iterácie. Vyberú sa 2 najlepšie cesty z predošlej iterácie (v prvom kole sú to cesty získané aplikovaním Dijkstrovho algoritmu). Na oboch cestách sa aplikuje kríženie a mutácia. Použitím kríženia a mutácie získame 8 ciest, z ktorých iba 2 najlepšie putujú do nasledujúcej iterácie.

7.2. Network Calculus model

Výpočet oneskorenia toku získame sumou oneskorení jednotlivých uzlov v sieti. Na výpočet oneskorenia konkrétneho uzla sa aplikuje nasledujúci výpočet:

$$T_{lq} = \frac{\sum_{k=1}^q B_{lk} + b_{max}}{CAP_{maxl} - \sum_{k=1}^{q-1} R_{lk}} \quad (2)$$

Kde T_{lq} je oneskorenie, ktorému podlieha tok v rade q na linke l , B_{lk} je veľkosť vyrovnávacej pamäte radu k na linke l , b_{max} je maximálna veľkosť ethernet rámcu a berie v úvahu, že je vždy možné, že sa niektorý paket práve odosiela. R_{lk} je rýchlosť radu k na linke l a CAP_{maxl} je maximálna kapacita linky l .

Výpočet celkovej priepustnosti cesty sa rovná najmenej priepustnosti linky, ktorú obsahuje cesta. Priepustnosť linky vypočítame rozdielom maximálnej rýchlosti linky a obsadenej kapacity linky. Obsadenú kapacitu linky vypočítame súčtom všetkých požadovaných priepustností, ktoré vedú danou linkou.

Implementovaný algoritmus neberie do úvahy hodnoty odchýlky.

8. Návrh dynamického pridelovania pre kontrolér z DP

8.1. Úprava controller.py

V pôvodnej verzii sú v kontrolérovi staticky priradené linky a IP pre koncové zariadenia. Tiež je problém, že pre konkrétne IP adresy sú priradené konkrétne QoS služby alebo že sa porovnáva počet liniek iba pre prípad keď ich počet je rovný 32, v tomto prípade ostatné topológie s menším počtom liniek nie sú funkčné pre daný kontrolér.

Pri mazaní sa vymaže iba daný prepínač, pričom statické linky ostávajú v topológii, keďže sú staticky zadané v súboroch, ktoré sa následne neupravujú a tým pádom sa neprepočítavajú toky pre upravenú topológiu.

V súbore controller.py bude potreba upraviť nasledujúce funkcie:

- `setStaticRoutesOnEdgeSwitch` – staticky sa nastavujú IP pre koncové zariadenia, funkcia sa volá keď sa zmení stav prepínača (*forwarder_state_changed*). Funkcia kontroluje, či sedí datapath ID a ak áno, iba v tom prípade priradí staticky IP adresu a nastaví akciu a pridá tok.

Úprava: vymaže sa statické nastavovanie IP pre konkrétny datapath ID, ako ďalší argument do funkcie sa pridá zdrojová adresa z novo prijatého paketu.

- `categorizeFlowAndDefineMatch` – pre konkrétne IP adresy sú priradené konkrétne QoS. Pre paket kontroluje, či sedia jednotlivé protokoly a potom pre konkrétnu QoS službu kontroluje, či sedí statická IP adresa.

Úprava: využije sa pole HOST, ktoré v sebe uchováva všetky už pre prepínače známe IP adresy koncových zariadení. Pole HOST bude zväčšené o QoS službu pre jednotlivé koncové zariadenia. Postup:

- zdrojová alebo cieľová adresa sa nachádza v poli HOST,
 - a) nastaví sa potrebné parametre podľa existujúcej metódy,
 - b) v prípade, že jedna z adries sa nenachádza v poli HOST, pridá sa IP a daná QoS služba
- zdrojová a cieľová adresa sa nenachádza v poli HOST,
 - a) rozbalí sa prijatý paket, aby sa zistilo o aký typ komunikácie sa jedná,
 - b) uložia sa IP adresy koncových zariadení do poľa HOST aj s prislúchajúcou QoS

službou,

c) nastaví sa potrebné parametre podľa existujúcej metódy.

- updateTopologyFromIcmp – kontrolovanie pre počet liniek 32.

Úprava: dynamicky sa bude prepočítavať počet liniek pre konkrétnu topológiu.

- set_queue – pri vytváraní radu je staticky zadaný začiatok mien pri linkách.

Úprava: jeden z argumentov funkcie je aj konkrétny prepínač, na ktorom sa nastavujú rady. Pre prepínač sa budú zisťovať názvy jednotlivých portov, podľa príkazu alebo API volaní, ktoré sa následne uloží do pomocného poľa, z ktorého sa potom budú získavať mená jednotlivých portov pre nastavenie radu.

8.2. Úprava globals.py

Premenná LINKS_CAPACITY:

Je to slovníkový typ, ktorý obsahuje informácie o stave jednosmerných liniek nachádzajúcich sa v sieti. Je potrebné dynamicky vkladať linky v topológií. Kľúč bude reťazec v tvare ‘začiatočný uzol – koncový uzol’ a hodnota je ďalší slovník. Vnorený slovník obsahuje informácie o radoch nachádzajúcich sa na linke. Pri pridávaní novej cesty stačí zachovať pôvodný tvar, pretože to nebudeme meniť.

```
'1-2': {0: {'max_cap': 5600000, 'remaining_cap': 5600000}, 1: {'max_cap': 4400000, 'remaining_cap': 4400000}}
```

8.3. Úprava QOS_linkDB.txt

Rovnako ako pri úprave globals.py je potrebné pridávať do zadaného súboru informácie o nových linkách v topológií. Jeho štruktúra je podobná ako pri LINKS_CAPACITY. Rozdiel je v tom, že obsahuje informácie obojsmerne.

9. Analýza metódy LARAC

Táto kapitola objasňuje fungovanie algoritmu LARAC použitého v článku Lagrange Relaxation Based Method for the QoS Routing Problem[37] určeného na výber trasy v sieťovej topológii systému fungujúceho na softvérovo riadených sieťach rozdelením nepolynomiálneho problému na vhodné menšie časti. Ponúka jasné vysvetlenia pre nosné body štruktúry výberu optimálnej cesty a uvádza jednotlivé pseudo-operácie pre následnú implementáciu do zvoleného individuálneho riešenia.

9.1. Problém DCLC

Je všeobecne známe, že problém smerovania v komunikačných sieťach je NP-úplný problém, v prípade ak počet parametrov, ktoré by mali byť minimalizované je viac alebo rovné dvom. Riešenie takéhoto problému rastie asymptoticky rýchlejšie ako polynomiálne. Dôsledkom čoho, je že čas potrebný na riešenie tohto problému prekračuje stovky rokov pri akejkoľvek veľkej výpočtovej sile dnešných počítačov a prostriedkov. Väčšina algoritmov preto delí tento problém na menšie podproblémy. Jedným z týchto problémov je aj problém obmedzením oneskorenia pre cestu s najmenšou cenou (angl. Delay Constrained Least Cost Path Problem, DCLC). [37]

9.2. Grafové vektory

V počítačovej sieti majme graf $G = (V,E)$, kde V reprezentuje uzly a E reprezentuje linky. Každá linka $e \in E$ má oneskorenie $d(e)$, ktoré je konštantná hodnota propagačného a radového oneskorenia. Zároveň má aj cenu $c(e)$, ktorá môže byť stanovená podľa viacerých parametrov, štandardne podľa predurčených vlastností použitých v konkrétnom systéme. Jednosmerné smerovacie protokoly môžeme deliť na dve základné kategórie; vektorové a stavové (angl. distance vector, link state). Sú založené na algoritmoch Bellman-Ford a Dijkstra. [37]

Ak hľadáme najkratšiu cestu pre cenu liniek, tak je riešenie definované nasledovne, kde s reprezentuje počiatkový bod (angl. source) a t určuje koncový uzol (angl. target) [37]:

$$\min_{p \in P'(s,t)} \sum_{e \in p} c(e),$$

(3)

Ak hľadáme cestu pre minimálne oneskorenie linky, tak je riešenie definované nasledovne[37]:

$$\sum_{e \in p} d(e) \leq \Delta_{delay}.$$

(4)

Cesta pre minimálnu cenu liniek je relatívne drahšia ako cesta pre minimálne oneskorenie a cesta pre minimálne oneskorenie je relatívne drahšia ako cesta pre minimálnu cenu liniek. Preto je nutné kooperovať medzi cenou a oneskorením, definovaným ako DCLC. [37]

9.3. Algoritmus

Nasledujúce body postupne opisujú vykonávanie algoritmu pre rôzne podmienky. [37]

- a. Na začiatku je stanovená globálna požiadavka pre oneskorenie, označená ako Δ_{delay} .
- b. Pomocou Dijkstrovho algoritmu sa vypočíta najkratšia cesta medzi dvoma uzlami označenými ako (s,t) berúc do úvahy cenu liniek ako meradlo vzdialenosti. Výsledok je uložený ako p_c . Ak je získaný výsledok menší alebo rovný ako globálna požiadavka pre oneskorenie, algoritmus končí a výsledkom je nájdená cesta.
- c. Pomocou Dijkstrovho algoritmu sa vypočíta najkratšia cesta medzi dvoma uzlami označenými ako (s,t) berúc do úvahy oneskorenie liniek ako meradlo vzdialenosti. Výsledok je uložený ako p_d . Ak je získaný výsledok väčší ako globálna požiadavka pre oneskorenie, algoritmus končí s výsledkom nemožnosti nájsť cestu pre dané hodnoty.
- d. Vypočíta sa hodnota lambda λ pomocou vzorca:

$$\lambda := \frac{c(p_c) - c(p_d)}{d(p_d) - d(p_c)}.$$

(5)

- e. kde $c()$ a $d()$ označujú rátanú cenu, resp. oneskorenie pre zvolený argument cesty.
- f. Pre každú linku sa vypočíta jej hodnota $c\lambda$ pomocou vzorca:

$$c_\lambda := c + \lambda \cdot d$$

(6)

- g. Pomocou Dijkstrovho algoritmu sa vypočíta najkratšia cesta medzi dvoma uzlami označenými ako (s,t) berúc do úvahy vypočítanú hodnotu c_λ liniek ako meradlo vzdialenosti. Výsledok je uložený ako r.
- h. Ak je suma hodnôt c_λ pre cesty r a pc rovnaká, algoritmus končí a výsledkom je cesta pd. Všetky cesty sú c_λ minimálne.
- i. Ak suma oneskorení na nájdenej ceste r je menšia alebo rovná ako globálna podmienka pre oneskorenie, cesta r sa uloží ako cesta pd.
- j. Ak suma oneskorení na nájdenej ceste r je väčšia ako globálna podmienka pre oneskorenie, cesta r sa uloží ako pc.
- k. Algoritmus sa vracia na krok číslo 4.

Výstupom algoritmu je nájdenie najlepšej cesty medzi dvoma bodmi použitím optimálnej hodnoty lambda. [37]

V niektorých špecifických prípadoch môže nastať, že algoritmus nedokáže identifikovať optimálnu cestu pre akúkoľvek hodnotu lambda – tak ako nedokáže ani žiadny iný algoritmus založený na princípe súhrnov cien liniek. [37]

10. Analýza metódy SAQR

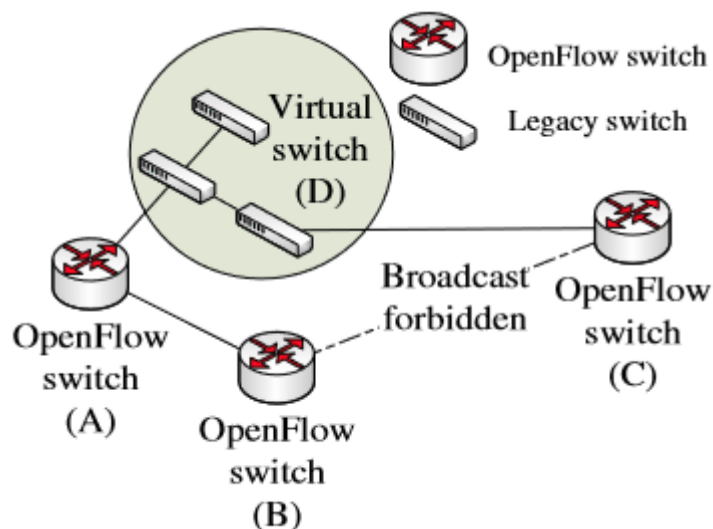
Zmena celej klasickej siete na SDN sieť je prakticky nemožná, kvôli vysokej cene náhrady. Existujú hybridné SDN siete, kde fyzické prepínače kooperujú s SDN prepínačmi. Sú dve možnosti zostavenia SDN hybridnej topológie, určiť topológiu staticky, čo je jednoduchší variant alebo ďalej preberaný variant a to je naučiť SDN kontrolér topológiu dynamicky. Vytvorenie hybridnej SDN siete nie je triviálny problém a preto autori v článku zameranom na spomínané hybridné SDN siete rozdelili svoj algoritmus na tri časti[38]:

- Zostavenie hybridnej topológie,
- Nájdenie koncového zariadenia a prevencia proti slučkám,
- Aplikácia SA (Simulated Annealing) na QoS požiadavky

Každý SDN prepínač odosiela LLDP (Link Layer Discovery Protocol) paket do SDN kontroléra. Klasický prepínač však takéto pakety neodosle a SDN kontrolér tak o ňom nič nevie. Na týchto prepínačoch beží STP (Spanning Tree Protocol) a LBP (Learning Bridge Protocol) a na základe takýchto paketov vieme zostaviť celú topológiu. Z STP paketu vieme získať MAC adresu a číslo portu a vytvoriť v SDN kontroléri virtuálny prepínač. Jeden virtuálny prepínač sa môže skladať z jedného alebo viac klasických prepínačov. Virtuálny prepínač ma nasledujúce atribúty[38]:

- Switch id - MAC adresa získaná od root id v hlavičke STP paketu,
- Port id - reťazec z id prepínača a id portu v hlavičke STP paketu.

Napríklad SDN kontrolér nájde v nasledujúcej sieti (Obrázok č.21.) jeden virtuálny prepínač s dvoma portami. [38]



Obrázok 21. Príklad topológie[38]

Po zostavení hybridnej topológie je ešte potrebné vyriešiť nájdenie klientského PC. Na lokalizáciu klientského PC vieme využiť ARP pakety. Ak SDN prepínač dostane ARP paket prvý krát z portu pripojeného k neznámemu objektu, potom je klientský PC pripojený na tento SDN prepínač na danom porte. Ak je ten port pripojený k virtuálnemu prepínaču, tak koncové zariadenie je pripojené na virtuálnom prepínači. Ďalej je potrebné vyriešiť problém slučiek. Na niektorých linkách v topológií treba vypnúť broadcast (broadcast forbidden linka z obrázka 21), toto sa dá jednoducho vyriešiť pomocou STP. [38]

Ak chceme v takejto topológii (obrázok 21) odoslať paket z prepínača A do C, cez prepínač D, tak na prepínačoch A a C vieme dať vstupný tok, ktorý povie z akého rozhrania majú odoslať paket. V prepínači D, sa preposielanie urobí samé pomocou LBP. Na základe LBP vieme poslať Packet-out informáciu, ktorá bude obsahovať "falošný paket" z prepínača C do prepínača D. Paket bude obsahovať adresu cieľa a upozorní prepínač D, že pakety obsahujúce cieľovú adresu prepínača C majú byť odoslané do prepínača C. [38]

Aplikácia SA na QoS sa rozdeľuje na tri moduly[38]:

- Nájdenie topológie - tento modul prijíma pakety z SDN prepínačov, normálnych prepínačov a koncových zariadení a vytvára virtuálnu topológiu v SDN kontroléry
- Zbierač informácií o sieti - tento modul periodicky zisťuje stav siete od SDN prepínačov a predpokladaný stav siete od normálnych prepínačov aby pomohol rozdeleniu tokov nájsť sieťové zdroje (prostriedky)
- Rozdelenie tokov - tento modul nájde najlepšiu cestu podľa SAQR na základe SLA (Service Level Agreement - zadané používateľom) a dvoch predošlých modulov

Keď modul prijme Packet-in požiadavku, QoS požiadavky (SLA) sa skontrolujú prvé. Potom modul vygeneruje prvú cestu na základe Dijkstrovho algoritmu a vypočíta cenu tejto cesty (C_p). Rovnica pre W_x (váhu) slúži na konkrétnejšie upravenie QoS parametrov, kvôli rôznym stavom siete, x môže byť nahradené oneskorením, stratovosťou alebo šírkou pásma. Váha bude väčšia ak daný QoS parameter, nenájde uplatnenie príliš veľa krát, čo môže byť upravené MR_x (miss rate) parametrom. Pokiaľ sa premenná T (počet prepínačov v topológii) dostane na nulovú hodnotu, algoritmus iteratívne vykonáva dva kroky 1) Generuje susednú cestu a počíta váhu cesty. Nahradzuje časť cesty cez port, ktorý nebol využitý na pôvodnej ceste. 2) `move_to_neighbor_state` - ak susedná cesta má nižšiu váhu, nahradí ju v pôvodnej ceste. V opačnom prípade nahradí cestu s pravdepodobnosťou podľa vzorca. Pričom hodnota c je nastavená na -0.75 , tým algoritmus predíde nájdenie iba lokálneho optima. [38]

$$p(C_P, C_N, t) = \begin{cases} 1 & , \quad C_N < C_P \\ e^{-\frac{c|C_N - C_P|}{t}} & , \quad C_N \geq C_P \end{cases}$$

(7)

V nasledujúcej tabuľke sú vysvetlené premenné použité v pseudo kóde. [38]

Tabuľka 3 Premenné v pseudo kóde a rovniach

| Notation | Definition |
|------------------------------------|--|
| P | Selected path |
| N | Neighbour path generated by selected path |
| C_P | Cost of selected path |
| C_N | Cost of neighbour path |
| t | Iterations count |
| T | Maximal iterations count |
| $\langle P_d, P_l, P_b \rangle$ | Delay, loss rate and bandwidth of selected path |
| $\langle R_d, R_l, R_b \rangle$ | Delay, loss rate and bandwidth of QoS requirements vector |
| $\langle W_d, W_l, W_b \rangle$ | Weights for delay, loss rate and bandwidth |
| $\langle MR_d, MR_l, MR_b \rangle$ | Miss rates for delay, loss rate and bandwidth |
| W_x, MR_x | Weight and miss rate for x (delay, loss rate or bandwidth) |
| c | Coefficient for probability function (equation (4)) |

O premenných vieme nasledujúce vzťahy. [38]

$$C_P = W_d \frac{(P_d - R_d)}{R_d} + W_l \frac{(P_l - R_l)}{R_l} + W_b \frac{(R_b - P_b)}{R_b}$$

$$W_x = \frac{MR_x}{MR_d + MR_l + MR_b}$$

$$MR_x = \frac{\text{Number of flows that cannot meet requirement } x}{\text{Number of flows in history}}$$

$$p(C_P, C_N, t) = \begin{cases} 1 & , \quad C_N < C_P \\ e^{-\frac{c|C_N - C_P|}{t}} & , \quad C_N \geq C_P \end{cases}$$

$$(P_d \leq R_d) (P_l \leq R_l), (P_b \geq R_b)$$

(8)

Pseudo kód ku dynamickému zostaveniu topológie.

```
packet = rozbal_packet()
If packet == "STP"
    pridad_port()
    prirad_MAC()
    zostav_VS()
If packet == "ARP"
    pridad_port()
    prirad_MAC()
    zostav_HOST()
```

Pseudo kód algoritmu SA (Simulated Annealing).

```
Input:
    src (source address),
    dst (destination address),
    r_vec (QoS requirements vector)
Output:
     $P$  (routing path)
 $P = \text{Dijkstra}(src, dst)$ 
 $C_P = \text{calculate\_path\_cost}(P, r\_vec)$ 
for  $t = T$  to 0 do
     $N = \text{generate\_neighbor\_path}(P)$ 
     $C_N = \text{calculate\_path\_cost}(N, r\_vec)$ 
    if  $\text{move\_to\_neighbor\_state}(C_P, C_N, t)$  then
         $P = N$ 
         $C_P = C_N$ 
    end if
end for
return  $P$ 
```

11. Analýza metódy ASA

V sieťach na báze prepínania paketov sú súbory rozdeľované na menšie pakety, ktoré sú odosielané cez čo najefektívnejšie susedné smerovače. Výber suseda prebieha pomocou výberového mechanizmu, implementovaného v prepínacom alebo smerovacom zariadení. Tento mechanizmus je taktiež zodpovedný za zahadzovanie niektorých paketov a dát, čo tvorí veľký problém v sieti, nakoľko je veľmi dôležité vyhnúť sa príliš veľkému počtu zahodených paketov v internete.[39]

Ak je paket zahodený predtým, ako dorazí do cieľovej stanice, všetky zdroje, ktoré sa využili na jeho prenos sú zbytočne zaťažované. V extrémnych prípadoch môže táto situácia viesť ku kolapsu siete. Práve z tohto dôvodu bolo vytvorených viacero dynamických metodológií na správu zdrojov v internete, ako napríklad Weighted Fair Queue (WFQ) [1], Stochastic Fairness Queuing (SFQ) [2], alebo aj Random Early Detection (RED) [3] a podobne. Taktiež je medzi nimi možné radiť aj autormi predstavený algoritmus – Adaptive Scheduling Algorithm (ASA). [39]

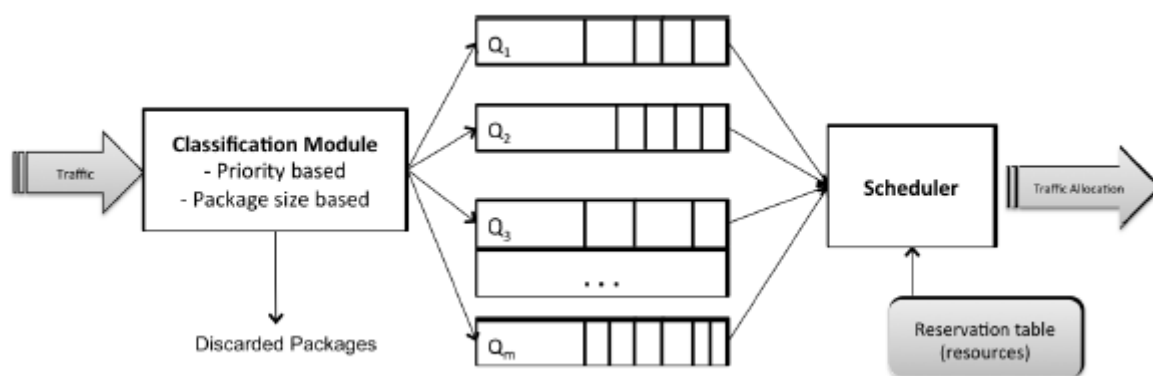
Predstavený algoritmus má rozšíriť všeobecný SDN model, pomocou integrácie softvérového ovládaného plánovača pre radenie toku v dátovej rovine - autori sa snažia zredukovať zložitosť stratégie plánovača. Tvrdia, že koncové prepínače v dátových centrách s podporou OpenFlow protokolu a iných prvkov SDN sietí, sú schopné podporiť pokročilé úrovne konfigurácií. [39]

Príspevok autorov je nasledovný[39]:

- vypracovanie plánovacej stratégie s viacerými pravidlami, v snahe o čo najväčšiu optimalizáciu vytťažnosti v SDN sieť,
 - stratégia založená na adaptívnom mechanizme, kde sú toky prioritizované ako podľa heterogenosti zdrojov siete, tak aj podľa aplikačných tokov,
- poskytnutie teoretickej a empirickej analýzy efektivity manažmentu zdrojov v SDN sieti, pri rôznych pravidlách a podmienkach.

11.1. Generický systémový model

Jednotlivé toky paketov sú definované: $\{F_i\}_{1 \leq i \leq n}$. Všetky pakety v toku F_i majú rovnakú veľkosť D_i . Všetky toky z globálnej rady Q , môžu byť rozdelené medzi interné toky $\{Q_j\}_{1 \leq j \leq m^2}$. Rozdelenie globálneho toku do interných je definované na základe priority. Priorita môže byť napríklad veľkosť paketov v toku. [39]



Obrázok 22. Systémová architektúra založená na manažmente radu[39]

Na obrázku 22 je znázornený koncept. Reservation table (rezervačná tabuľka RT) je využívaná algoritmom pre statickú alokáciu na zachovanie pracovného zaťaženia zdrojov (W_p) – v našom prípade portov na prepínačoch (P). Pre jednoduchosť budeme predpokladať, že všetky porty majú rovnakú kapacitu, toky prichádzajú s poissonovým rozdelením³ s rovnakým parametrom λ . [39]

Tabuľka 4. Skratky a ich význam[39]

| Notation | Description |
|---------------|---|
| $n; n_{max}$ | The number of Flows; The maximum number of Flows. |
| m | The number of resources (ports). |
| F_i | The Flow. |
| D_i | The size of a package in Flow F_i . |
| Q_i | The sub-Queue of a global queue (Q). |
| W_i | The resource workload (used bandwidth). |
| λ | Variance of a Poisson distribution. |
| μ | The parameter of the exponential distribution, often called the <i>rate parameter</i> . |
| μ_n | The exponential rate parameter for a specific number of Flows, n . |
| $t; \Delta t$ | Time; A short interval of time denoting a transition step. |
| $N(t)$ | Number of Flows at moment t . |
| $p_n(t)$ | Probability to have n Flows at moment t . |
| p_n | Probability to have n Flows in a steady state (at equilibrium). |
| $E(F)$ | Efficiency metric for a set of Flows (F). |
| ρ | The amount of processing-time per unit time (λ/μ) divided by the processing capabilities per unit time (number of resources, m). |
| T_L | Lower throughput. |
| T_U | Upper throughput. |
| T_M | Utilization rate. |

³ https://sk.wikipedia.org/wiki/Poissonovo_rozdelenie

11.2. SDN algoritmy

Pre zabezpečenie optimálneho procesu pridelovania musí plánovač nájsť optimálne riešenie pre problém s využívaním zdrojov za predpokladu, že tok F_i namapovaný P_j nebude preplánovaný na iný zdroj. Z tohto dôvodu plánovač dodržiava prísnu politiku pridelovania tokov na základe ich priority a najvyššie toky sú pridelené ako prvé. Pridelovanie zdrojov je poskytnuté v troch rôznych scenároch založených na nasledujúcich definíciách minimálnej šírky pásma[39]:

- Minimálna šírka pásma sa rovná vysielanému času (transmission window time) pre tok o veľkosti D_i (základný scenár).
- Minimálna šírka je väčšia ako súčet všetkých alokovaných tokov s minimálnou šírkou (najhorší scenár).
- Minimálna šírka je menšia ako súčet všetkých tokov s minimálnou šírkou („typický“ scenár).

Boli vyvinuté tri verzie algoritmu na základe tokovej úrovne[39]:

- *Fixed priority scheduling strategy* – uplatňuje sa špecifická stratégia výberu (FIFO, náhodný, ...).
- *Balanced scheduling strategy* – podobná k Max-Min-max vyberaniu.
- *Adaptive cost scheduling strategy* – prispôbený súčasnému využívaniu zdrojov.

11.2.1. Fixed Priority Scheduling strategy

Algoritmus zvažuje rôzne stratégie pre každý rad a spracováva všetky rady začínajúce od najvyššej priority. Slovný opis algoritmus[39]:

1. Priradiť každému portu nulové zaťaženie $W_p = 0$; W_p (bandwidth).
2. Ulož všetky toky do poľa Q.
3. Kým nie je „tokové pole“ Q prázdne opakuj.
 - a. Začni od najvyššej priority po najnižšiu – prioritou sa určuje podľa veľkosti paketov v toku.
 - b. Zisti si potrebné informácie o danom tokoch s rovnakou prioritou z poľa Q do premennej Q_{pr} .

- c. Zníž pole Q o dané toky Q_{pr} .
- d. Kým pole tokov s rovnakou prioritou Q_{pr} nie je prázdne.
 - i. Vyber si určitý tok F_q – určitý tok si možno vybrať ľubovoľnou stratégiou (FIFO, náhodne).
 - ii. Nájdi port s najmenšou šírkou pásma W_p medzi všetkými portami m .
 - iii. Zisti, či šírka pásma na porte W_p je menší ako maximálna šírka pásma W_{MAX} .
 - 1. Áno, priradiť tok k portu, zväčši šírku pásma W_p na porte o veľkosť toku D_q a zmenši pole tokov s rovnakou prioritou Q_{pr} o daný tok F_q .
 - 2. Nie, zahodiť všetky toky z Q_{pr} .
- 4. Vráť pole portov s ich vyťaženosťou (využitou šírkou pásma) W .

Algorithm 1 Fixed Priority Scheduling using Specific Selection Strategy

```
1: procedure FIXEDPRIORITYSCHEDULING( $Q, strategy$ )
2:   for all ports,  $p$  do
3:      $W_p \leftarrow 0$ ;
4:   end for
5:    $Q \leftarrow$  Set of Flows;
6:   while  $Q \neq \Phi$  do
7:     for  $pr = HighestPriority \dots LowestPriority$  do
8:        $Q_{pr} \leftarrow extractFlows(Q, pr)$ ;
9:        $Q \leftarrow Q \setminus Q_{pr}$ ;
10:      while  $Q_{pr} \neq \Phi$  do
11:        Select flow  $F_q \in Q_{pr}$  using specified  $strategy$ ;
12:                                                 $\triangleright strategy$  can be: FCFS, random, etc.
13:         $W_p \leftarrow \min_{j=1,m} \{W_j\}$ ;
14:        if  $W_p < W_{MAX}$  then
15:          Allocate Flow  $F_q$  on Port  $p$ ;  $W_p \leftarrow W_p + D_q$ ;  $Q_{pr} \leftarrow Q_{pr} \setminus F_q$ ;
16:        else
17:          Consider other Port.
18:          if all Ports are used at maximum capacity then
19:            Drop all Flows from  $Q_{pr}$ .
20:          end if
21:        end if
22:      end while
23:    end for
24:  end while
25:  Return  $W$ ;
26: end procedure
```

\triangleright Array with Ports' workload.

Obrázok 23. Algoritmus pre Fixed Priority Scheduling strategy[39]

11.2.2. Balanced scheduling strategy

Algoritmus dva predstavuje stratégiu pre vyrovňovanie plánovania tokov tým, že sa spracuje najprv najdlhší tok a potom najkratší. Táto zmes zaisťuje dobrú rovnováhu v spracovaní radu. Po každom pridelení toku do portu sa aktualizuje pracovné zaťaženie každého portu (šírka pásma). Slovný opis algoritmu 2[39]:

1. Priradiť každému portu nulové zaťaženie $W_p = 0$; W_p (*bandwidth*).
2. Uložiť všetky toky do polia Q , nastaviť premennú n na počet tokov v poli Q
3. Nastaviť premennú m na počet portov v topológii
4. Zistiť, či počet tokov n je menší ako počet portov m
 - a. Áno, priradiť každému portu i , jeden tok F_i . Zvýšiť šírku pásma pre port W_i o tok D_i (veľkosť paketov).

- b. Nie, prechádzaj postupne jednotlivými portami
- i. Vypočítaj minimálnu cenu jednotlivých tokov – využíva sa algoritmus 3.
 - ii. Vyber najväčší tok spomedzi všetkých na konkrétnom porte C_{pq}
 - iii. Zisti, či šírka pásma na porte W_p je menší ako maximálna šírka pásma W_{MAX} .
 1. Áno, prirad' tok F_q . k portu p, zväčši šírku pásma W_p na porte o veľkosť toku D_q a zmenši pole tokov Q o daný tok F_q .
 2. Nie, nájdi iný port.
 - a. Ak všetky porty sú vyt'azené, zahod' všetky toky v poli Q.
 - iv. Kým nie je pole tokov Q prázdne pokračuj.
 1. Vypočítaj minimálnu cenu jednotlivých tokov – využíva sa algoritmus 3.
 2. Vyber najmenšiu cenu pre tok spomedzi všetkých na konkrétnom porte C_{pq} .
 3. Zisti, či šírka pásma na porte W_p je menšia ako maximálna šírka pásma W_{MAX} .
 - a. Áno, prirad' tok F_q . k portu p, zväčši šírku pásma W_p na porte o veľkosť toku D_q a zmenši pole tokov s Q o daný tok F_q .
 - b. Nie, nájdi iný port.
 - i. Ak všetky porty sú vyt'azené, zahod' všetky toky v poli Q.
 - v. Ak pole tokov Q je prázdne, skonči.

- vi. Vypočítaj minimálnu cenu jednotlivých tokov – využíva sa algoritmus 3.
- vii. Vyber najväčší tok spomedzi všetkých na konkrétnom porte C_{pq} .
- viii. Zisti, či šírka pásma na porte W_p je menší ako maximálna šírka pásma W_{MAX} .
 - 1. Áno, priradiť tok F_q k portu p , zväčši šírku pásma W_p na porte o veľkosť toku D_q a zmenši pole tokov Q o daný tok F_q .
 - 2. Nie, nájsi iný port.
 - a. Ak všetky porty sú vytážené, zahod' všetky toky v poli Q .
- ix. Vráť pole portov s ich vytážením – využitou šírkou pásma W .

Slovný opis k algoritmu 3:

Do algoritmu vstupujú atribúty – pole tokov Q a pole portov s ich využitou šírkou pásma W .

- 1. Nastav pole cien tokov na nulu $Costs = 0$.
- 2. Pre každý tok F_i v poli Q vykonaj.
 - a. Pre každý port P_j v poli $Ports$ vykonaj.
 - i. Priradiť do premennej C_{ij} veľkosť toku D_i sčítaj s využitou šírkou pásma W_j – laicky povedané, snažíme sa skombinovať všetky možnosti priradenia portu k toku a zistiť tak jednotlivé ceny.
 - b. Do premennej C_{pi} priradiť najmenšiu cenu toku pre každý port.
 - c. Premennú C_{pi} zlúč s poľom $Costs$.

Algorithm 2 Balanced Scheduling

```
1: procedure BALANCEDSCHEDULING( $Q$ )
2:   for all ports,  $p$  do
3:      $W_p \leftarrow 0$ ;
4:   end for
5:    $Q \leftarrow$  Set of Flows;  $n \leftarrow |Q|$ ;
6:    $m \leftarrow$  Number of Ports;
7:   if  $n \leq m$  then ▷ Bootstrap Strategy (Allocate each Flow on a Port).
8:     for  $i = 1, n$  do
9:       Allocate Flow  $F_i$  on Port  $i$ ;  $W_i \leftarrow W_i + D_i$ ;
10:    end for
11:  else
12:    for  $k = 1, m$  do ▷ Allocate a Port for a larger Flow.
13:       $Costs =$  ComputeMinAllocationCosts( $Q, W$ );
14:      Select  $C_{pq} \leftarrow \max \{C_{ji}\}, C_{ji} \in Costs$ ;
15:      if  $W_p < W_{MAX}$  then
16:        Allocate Flow  $F_q$  on Port  $p$ ;  $W_p \leftarrow W_p + D_q$ ;  $Q \leftarrow Q \setminus F_q$ ;
17:      else
18:        Consider other Port.
19:        if all Ports are used at maximum capacity then
20:          Drop all Flows from  $Q$ .
21:        end if
22:      end if
23:    end for
24:    while  $Q \neq \Phi$  do
25:       $Costs =$  ComputeMinAllocationCosts( $Q, W$ );
26:      Select  $C_{pq} \leftarrow \min \{C_{ji}\}, C_{ji} \in Costs$ ;
27:      if  $W_p < W_{MAX}$  then
28:        Allocate Flow  $F_q$  on Port  $p$ ;  $W_p \leftarrow W_p + D_q$ ;  $Q \leftarrow Q \setminus F_q$ ;
29:      else
30:        Consider other Port.
31:        if all Ports are used at maximum capacity then
32:          Drop all Flows from  $Q$ .
33:        end if
34:      end if
35:      if  $Q == \Phi$  then
36:        exit;
37:      end if
38:       $Costs =$  ComputeMinAllocationCosts( $Q, W$ );
39:      Select  $C_{pq} \leftarrow \max \{C_{ji}\}, C_{ji} \in Costs$ ;
40:      if  $W_p < W_{MAX}$  then
41:        Allocate Flow  $F_q$  on Port  $p$ ;  $W_p \leftarrow W_p + D_q$ ;  $Q \leftarrow Q \setminus F_q$ ;
42:      else
43:        Consider other Port.
44:        if all Ports are used at maximum capacity then
45:          Drop all Flows from  $Q$ .
46:        end if
47:      end if
48:    end while
49:  end if
50:  Return  $W$ ; ▷ Array with Ports' workload.
51: end procedure
```

Obrázok 24. Algoritmus pre Balanced Scheduling[39]

Algorithm 3 Compute Minimum Allocation Costs for a Flow

```
1: procedure COMPUTEMINALLOCATIONCOSTS( $Q, W$ )
2:    $Costs \leftarrow \Phi$ ;
3:   for each flow  $F_i \in Q, i = 1, n$  do
4:     for each port  $P_j \in Ports, j = 1, n$  do
5:        $C_{ji} \leftarrow W_j + D_i$ ;
6:     end for
7:      $C_{pi} \leftarrow \min \{C_{ji}\}, j = 1, m$ ;
8:      $Costs \leftarrow Costs \cup C_{pi}$ ;
9:   end for
10:  Return  $Costs$ ;
11: end procedure
```

Obrázok 25. Algoritmus pre výpočet minimálnej ceny toku[39]

11.2.3. Adaptive cost scheduling strategy

Slovný opis algoritmu 4[39]:

1. kým nie je pole tokov Q prázdne pokračuj.
2. Do premennej $NormCosts$ vlož nulu.
3. Pre každý tok F_i v poli Q vykonaj.
 - b. Pre každý port P_j v poli $Ports$ vykonaj.
 - i. Prirad' do premennej C_{ij} veľkosť toku D_i sčítaj s využitou šírkou pásma W_j – laicky povedané, snažíme sa skombinovať všetky možnosti priradenia portu k toku a zistiť tak jednotlivé ceny.
 - c. Do premennej C_{pi} prirad' najmenšiu cenu toku pre každý port.
 - d. Do premennej D_p vyber minimálnu veľkosť paketu.
 - e. Do premennej E_{pi} vydeľ premennú C_{pi} s premennou D_p .
 - f. Premennú E_{pi} zlúč s poľom $NormCosts$.
2. Vyber najväčšiu položku E_{pq} z poľa $NormCosts$.
3. Ak šírka pásma na porte W_p je menšia ako maximálna možná W_{MAX}
 - a. Áno, alokuj tok F_q na port p , , zväčši šírku pásma W_p na porte o veľkosť toku D_q a zmenši pole tokov Q o daný tok F_q .
 - b. Nie, zahod' všetky toky z poľa Q .
4. Vráť pole portov s ich využitou šírkou pásma.

Algorithm 4 Adaptive Cost Scheduling

```
1: procedure ADAPTIVECOSTSCHEDULING( $Q$ )
2:   while  $Q \neq \Phi$  do
3:     NormCosts  $\leftarrow \Phi$ ;
4:     for each flow  $F_i \in Q, i = 1, n$  do
5:       for each port  $P_j \in Ports, j = 1, n$  do
6:          $C_{ji} \leftarrow W_j + D_i$ ;
7:       end for
8:        $C_{pi} \leftarrow \min \{C_{ji}\}, j = 1, m$ ;
9:        $D_p \leftarrow \min \{D_i\}, i = 1, n$ ;
10:       $E_{pi} = C_{pi}/D_p$ ;
11:      NormCosts  $\leftarrow$  NormCosts  $\cup E_{pi}$ ;
12:    end for
13:    Select  $E_{pq} \leftarrow \max \{E_{ji}\}, E_{ji} \in NormCosts$ ;
14:    if  $W_p < W_{MAX}$  then
15:      Allocate Flow  $F_q$  on Port  $p$ ;
16:       $W_p \leftarrow W_p + D_q$ ;  $Q \leftarrow Q \setminus F_q$ ;
17:    else
18:      Drop all Flows from  $Q$ .
19:    end if
20:  end while
21:  Return  $W$ ;
22: end procedure
```

▷ Array with Ports' workload.

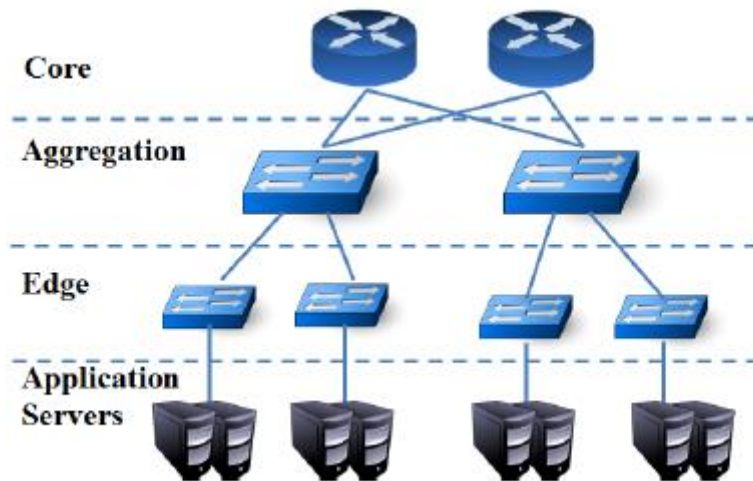
Obrázok 26. Algoritmus pre Adaptive Cost Scheduling[39]

11.2.4. Simulačné prostredie

Dátové centrá sú väčšinou konštruované ako veľké skupiny počítačov, ktoré sú vlastnené a používané jednou organizáciou, ako napríklad veľké univerzity, alebo súkromné podniky. Počet serverov v rámci týchto centier sa líši od niekoľko tisíc, po niekoľko desaťtisíc kusov. [39]

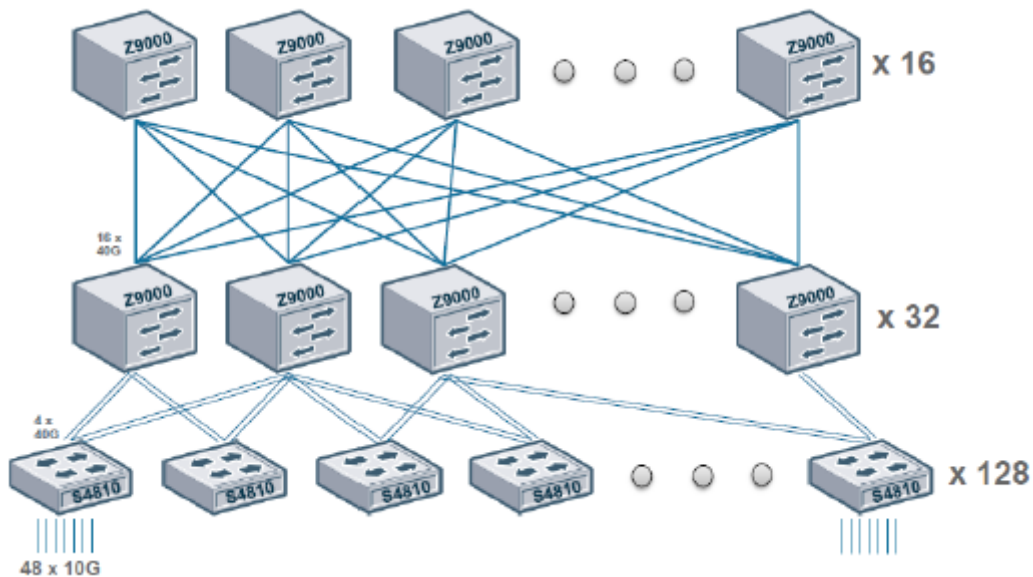
Typické dátové centrum je organizované do 3-vrstvej architektúry (v prípade menších centier sa jadro a agregáčna vrstva spájajú do jednej a tvoria tak 2-vrstvú architektúru) [39]:

- Okrajová vrstva
 - pozostáva z Top-of-Rack prepínačov, ktoré pripájajú serveri k sieťovej štruktúre
- Agregáčna vrstva
 - pozostáva zo zariadení, ktoré spájajú ToR prepínače s okrajovou vrstvou
- Jadro
 - pozostáva zo zariadení, ktoré prepájajú dátové centrum a WAN



Obrázok 27. Architektúra dátového centra[39]

Samotná sieťová architektúra, ktorú autori použili pri testovaní sa nazýva Clos sieť – viacstupňová prepínacia sieť, kde je možné použiť prepínače malých rozmerov. Je takisto možné túto sieť navrhnuť ako neblokujúcu, podobne ako aj cross-bar prepínač – pre každý vstupno-výstupný pár vieme nájsť usporiadanie ciest pre spojenie vstupov a výstupov cez prepínače v strednom stupni. [39]

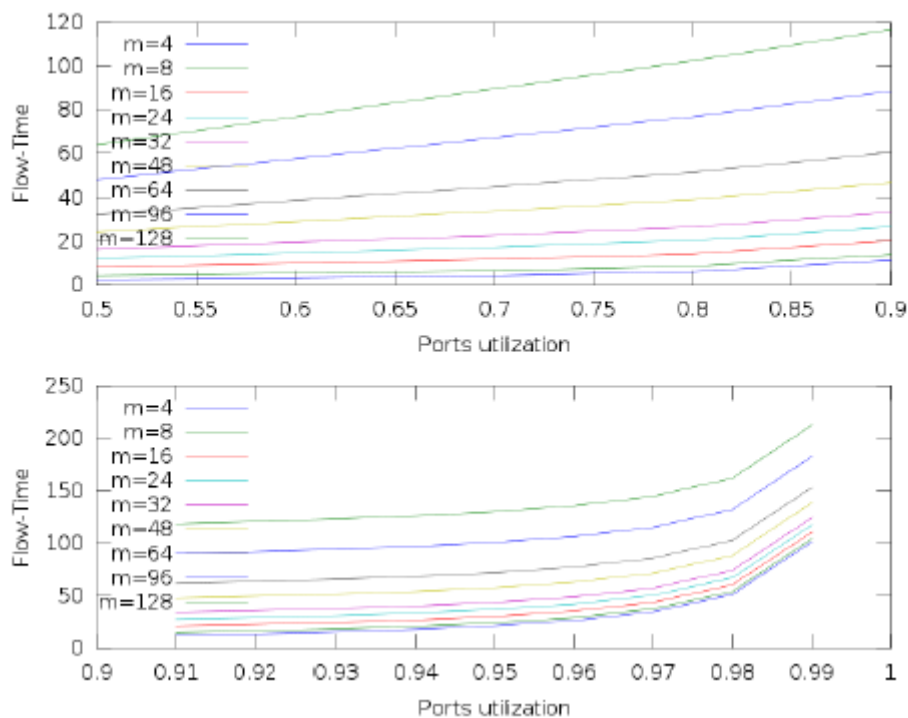


Obrázok 28. Jednoduchá Clos topológia[39]

11.2.5. Hodnotenie navrhovaného systému

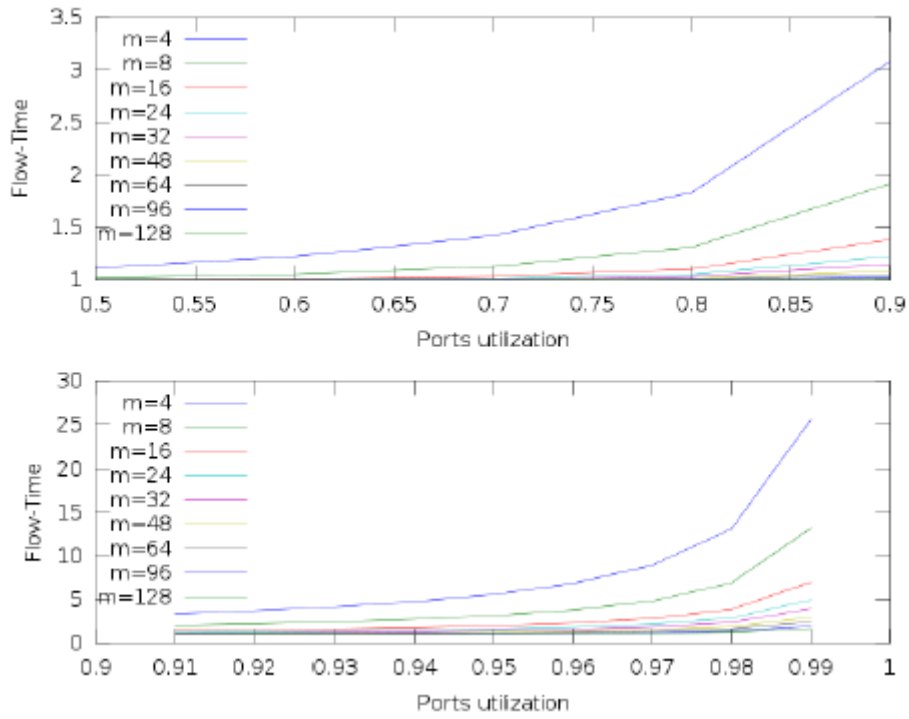
Hodnotilo sa podľa rôznych kritérií, pričom sa bral ohľad hlavne na flow-time (suma finálnych časov pre všetky existujúce toky v systéme). Optimálny plán je ten, ktorý minimalizuje tento čas. [39]

- Efektivita pri rôznom počte portov a pri fixnom čase príchodu (arrival time) [39]
 - flow-time sa zvyšuje, keď sa zvyšuje aj priemerné využitie
 - čím vyšší počet portov, tým lepšie výsledky

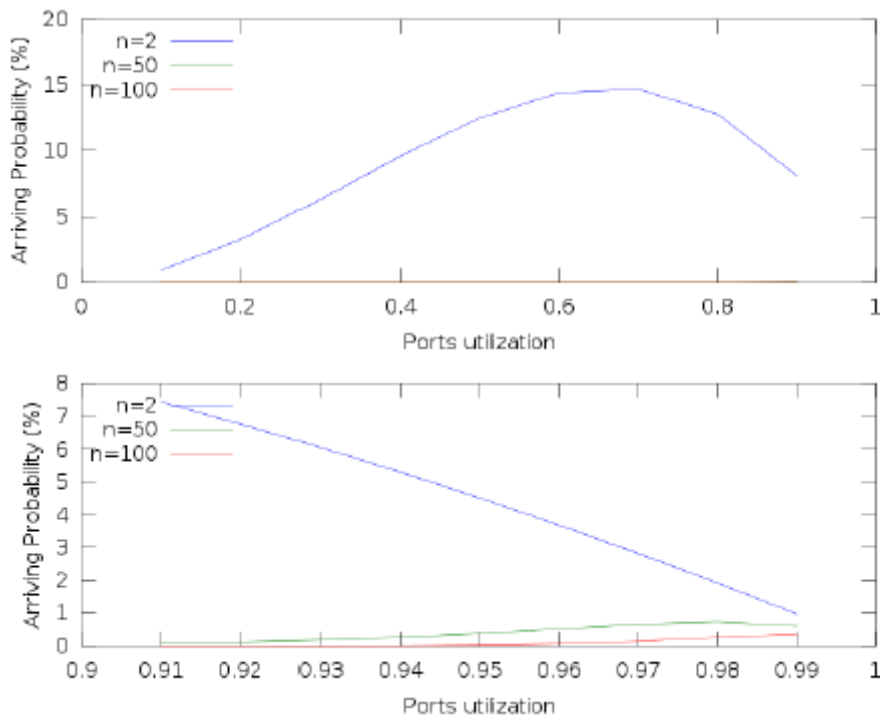


Obrázok 29. Očakávaný flow-time pre rozdielny počet portov (fixný čas príchodu) [39]

- Efektivita pri rôznom počte portov a zachovanej (fixnej) rýchlosti spracovania[39]
 - Flow-time sa zvyšuje, keď sa zvyšuje priemerné využitie
 - čím nižší počet portov, tým lepšie výsledky



Obrázok 30. Očakávaný flow-time pre rozdielny počet portov (fixná rýchlosť spracovania) [39]



Obrázok 31. Pravdepodobnosť, že v systéme sa bude nachádzať práve n tokov[39]

11.2.6. Záver

Autori predstavili nový prístup k riadeniu systému v SDN, ktorý zahŕňa softvérovo riadený plánovač pre radenie tokov v dátovej vrstve. Hlavnou metrikou pre alokáciu zdrojov v navrhnutom algoritme je minimálny čas spracovania toku.[39]

12. Analýza MILP algoritmu

Dnešné LAN siete pozostávajú z mnohých používateľských zariadení, ktoré sa pripájajú do internetu rôznymi sieťovými technológiami (Ethernet, 2,4 GHZ WiFi, 5 GHZ WiFi). Všeobecne sa tieto zariadenia pripájajú do internetu jednou technológiou kategorizovanou podľa priority. Toto statické nastavenie nedovoľuje sieťam odomknúť ich plný potenciál v zmysle latencie a priepustnosti. Na vyriešenie tohto problému bol predstavený algoritmus rovnomerného rozdelenia záťaže, ktorý dynamicky pridelí potrebnú technológiu a cestu cez sieť pre každý tok, založený na požiadavkách, dostupnosti šírky pásma a vlastnosti linky. Cieľom tohto algoritmu je nájsť optimálnu konfiguráciu ciest pre všetky toky v sieti a maximalizovať tak celkovú priepustnosť. Algoritmus sa dokáže dynamicky prispôbiť zmenám podmienok v sieti, príchodu nových tokov, zrušeniam tokov a chýbam liniek. Tento problém bol sformulovaný ako MILP problém. Riešenie MILP problému využíva heuristický algoritmus, ktorý je najrýchlejší. Bol otestovaný na ns-3 module, reálnych zariadeniach a rôznych tokoch.[40]

V posledných rokoch sa výrazne rozvinuli lokálne siete (LAN). Sú používané stále rastúcim počtom rôznych zariadení (napr. počítače, prenosné počítače, inteligentné televízory, tablety, smartphony), ktoré sú pripojené k internetu pomocou rôznych káblových a bezdrôtových sieťových technológií (napr. Ethernet, rozvody elektrického vedenia, rôzne štandardy Wi-Fi) a spotrebiteľské služby s čoraz prísnejšími požiadavkami (napr. Voice over IP, Video on Demand, IP-TV). Na jednej strane, moderné multimediálne služby vyžadujú striktnú kvalitu a sú veľmi citlivé na prerušenia a degradáciu (napr. vysoká latencia, preťaženie alebo poruchy spojenia). Na druhej strane LAN sa zväčša spravujú statickým spôsobom, a tým pádom nemôžu automaticky reagovať včas na dočasné narušenia, ktoré spôsobujú zníženie kvality QoE. V niektorých prípadoch môže používateľ manuálne prenastaviť priority používaných technológií, ale nie sú zmienky o dynamickom menení medzi týmito technológiami alebo používaním viacerých technológií naraz. [40]

Architektúra LAN siete je reprezentovaná grafom definovaným štvoricou (N, T, L, G) , kde[40]:

N - množina uzlov $\{n_1, n_2, n, \dots\}$ reprezentujúcich rôzne zariadenia v LAN sieti

T - množina technológií (Ethernet/WiFi) $\{t_1, t_2, \dots\}$, cT - reprezentuje šírku pásma danej technológie, $cT \geq 0$

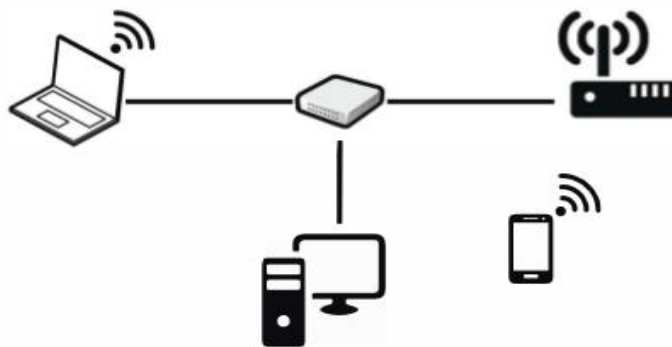
L - množina liniek, každá linka je definovaná trojicou $\langle sl, dl, tl \rangle$, pričom sl - zdrojový uzol, dl - cieľový uzol, tl - technológia

G - množina kolíznych skupín - $\{1,2,\dots \mid t_1 = t_2 \ \& \ ct_1 = ct_2\}$ - kolízna skupina spája všetky linky, ktoré zdieľajú šírku pásma technológie v rovnakej sieti a môže dôjsť k interferencii medzi nimi

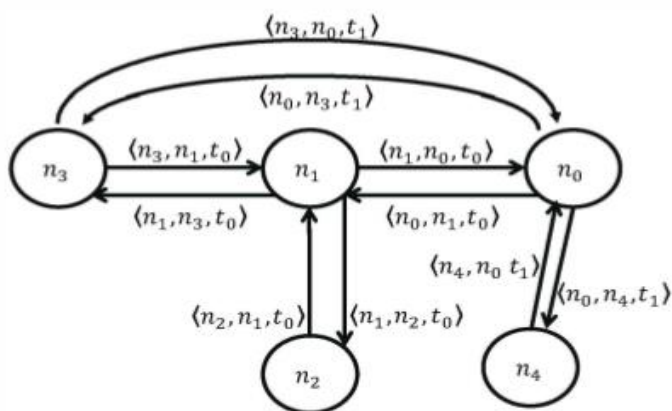
Je potrebné zdefinovať ešte množinu všetkých tokov F. [40]

F - množina všetkých tokov $\{f_1, f_2, \dots\} \langle sf, df, rf \rangle$, sf - zdrojový uzol, df - cieľový uzol, rf - šírka pásma pre daný tok

Na obrázku vidíme LAN sieť, ktorá pozostáva z prístupového bodu, prepínača, z dvoch zariadení pripojených ethernetom na prepínač a dvoch zariadení pripojených cez technológiu WiFi. Zariadenia sú reprezentované množinou uzlov $\{n_0, n_1, n_2, n_3, n_4\}$, technológie sú reprezentované množinou $\{t_0, t_1\}$. Medzi zariadeniami je v každom smere jedna linka, napríklad medzi AP a prepínačom sú dve linky $\langle n_0, n_1, t_0 \rangle$ a $\langle n_1, n_0, t_0 \rangle$. Na obrázku sa nachádza sedem kolíznych domén. Každá ethernet linka má vlastnú kolíznu doménu a jedna je v rámci WiFi. [40]



(a) Network topology



Obrázok 32. Príklad topológie[40]

Problém rozdelenia tokov je modelovaný MILP formuláciou, ktorá pozostáva z[40]:

- Vstupy
- Rozhodujúce premenné
- Cieľová funkcia
- Množina obmedzení

Tento model je vyriešený cez Gurobi solver, ktorý nájde optimálne riešenie pre dané parametre. Vstupom do Gurobi solvera sú nasledujúce parametre. [40]

Vstupy[40]:

Sln - všetky linky, ktoré majú n ako zdroj

Dln - všetky linky, ktoré majú n ako cieľ

Sfn - všetky linky, ktoré majú rovnaký zdroj daného toku

Dfn - všetky linky, ktoré majú rovnaký cieľ daného toku

Xfn - všetky linky, ktoré majú cieľ zdroja toku

Yfn - všetky linky, ktoré majú zdroj cieľu toku

Rozhodujúce premenné[40]:

Lambdal,f - cesta pre každý tok (ak prešla lamdal,f=1, ak nie lamdal,f=0)

Pi - definuje priradenie šírky pásma toku

Cieľová funkcia[40]:

$$\max \sum_{f \in F} \tau_f$$

(9)

Množina obmedzení[40]:

Maximálna kapacita v kolíznej skupine nie je prekročená,

Obmedzenia aby v toku nevznikla slučka:

Každá cesta má presne 1 linku odchádzajúcu zo zdroja toku.

Každá cesta má presne 1 linku prichádzajúcu do cieľa toku.

Každá cesta nemôže mať linku prichádzajúcu do zdroja toku.

Každá cesta nemôže mať linku odchádzajúcu z cieľa toku.

Pre každý uzol v ceste toku, okrem zdroja a cieľa toku, platí, že ide z neho maximálne jedna linka.

Pre každý uzol v ceste toku, okrem zdroja a cieľa toku, platí, že vchádza doň maximálne jedna linka.

Celkový počet prichádzajúcich a odchádzajúcich liniek z uzla sa rovná, okrem zdroja a cieľa toku.

Medzi dvoma uzlami môže byť aktívna len jedna linka.

Všetky toky rešpektujú TCP férovosť, ktorá je definovaná nasledovne. [40]

Rozhodujúca premenná[40]:

$$\forall l \in L, \forall f, g \in F : \zeta_{l,f,g} = \lambda_{l,f} \cdot \lambda_{l,g}$$

(10)

Množina obmedzení[40]:

$$\begin{aligned} \forall l \in L, \forall f, g \in F : \zeta_{l,f,g} &\leq \lambda_{l,f} \\ \forall l \in L, \forall f, g \in F : \zeta_{l,f,g} &\leq \lambda_{l,g} \\ \forall l \in L, \forall f, g \in F : \zeta_{l,f,g} &\geq \lambda_{l,f} + \lambda_{l,g} - 1 \\ \forall l \in L, \forall f \in F : \tau_f \cdot \sum_{g \in F} \zeta_{l,f,g} &\leq O_l \cdot c_l \end{aligned}$$

(11)

Autori uviedli, že ich výsledok práce nie je schopný pracovať v reálnom čase pri väčších sieťach. Pri použití heuristiky algoritmus pracuje rýchlejšie.

V článku je príliš málo detailov na to, aby sa dal spraviť návrh. Nie je spomenutý žiadny algoritmus ani heuristická funkcia. Rovnako chýbajú informácie o tokoch. Ďalej je vysvetlené len riešenie z Gurobi solvera, ktorého výstup je nejasný a nie je možné s ním ďalej pracovať. Rovnako táto technika nie je adaptovateľná v prostredí mininetu.

13. SDN v reálnom čase

13.1. Úvod

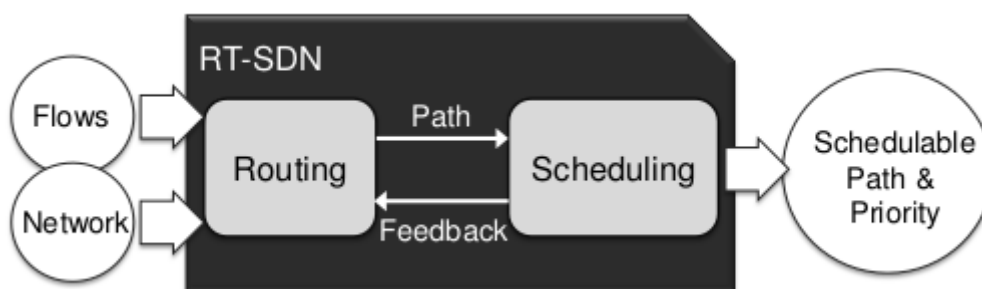
Požiadavka komunikácie v reálnom čase sa rýchlo šíri vo všetkých typoch sietí. Podpora komunikácie v reálnom čase so splnením všetkých QoS požiadaviek musí riešiť veľa problémov, ako je výber cesty alebo rozdelenie premávky. Pre každý tok v reálnom čase musí byť vybraná cesta od zdroja do cieľa spĺňať všetky požiadavky. Autori v článku navrhli adaptívny systém pre komunikáciu

v reálnom čase s názvom RT-SDN. Tento systém zabezpečuje smerovanie a rozdelenie premávky tak, aby zabezpečil garanciu všetkých požiadaviek v SDN sieťach. Aby systém spĺňal danú špecifikáciu bola najskôr vyvinutá škálovateľná smerovacia schéma, ktorá adaptívne rekonfiguruje existujúce cesty a hľadá nové cesty, ktoré spĺňajú požiadavku šírky pásma. Keďže viacero tokov môže mať cestu cez rovnaký smerovač bolo navrhnuté usporiadanie na základe priority, ktoré umožňuje dobre známy OPA (Optimal Priority Assignment) algoritmus, ktorý pracuje s hodnotou odchýlky. V článku pozorujú bonusy plánovania ciest a usporiadania na základe priority pomocou spätnej väzby. [41]

13.2. Definovanie problému

Sieťová topológia je charakterizovaná grafom s orientovanými hranami $G \langle V, E \rangle$, kde V je množina vrcholov a E je množina orientovaných liniek (hrán). Každá linka $l(a,b) \in E$, ukazuje z bodu $a \in V$ do bodu $b \in V$ a je asociovaná s obmedzením kapacity $c(a,b)$. Predpokladajme, že F bude množina tokov v reálnom čase. Každý tok $f_k \in F$ je špecifikovaný ako $f_k(s_k, t_k, T_k, M_k, D_k)$, s_k - začiatkový vrchol, t_k - cieľový vrchol, T_k - minimálny čas medzi dvoma nasledujúcimi správami, M_k - maximálna veľkosť správy. D_k - relatívny end-to-end deadline ($D_k < T_k$). Ďalšia premenná R_k - sekvencia liniek, ktoré spájajú s_k do t_k . Maximálna požadovaná šírka pásma je definovaná ako $b_k = M_k / T_k$. Ďalej P_k - priorita toku f_k . [41]

S danou sieťovou topológiou $G \langle V, E \rangle$ a množinou tokov F považujeme za problém, určiť cesty R_k a priority P_k každého toku f_k , tak aby všetky správy toku dorazili zo zdroja s_k do cieľa t_k zo splnením D_k . Tieto problémy sú považované za NP úplné. [41]



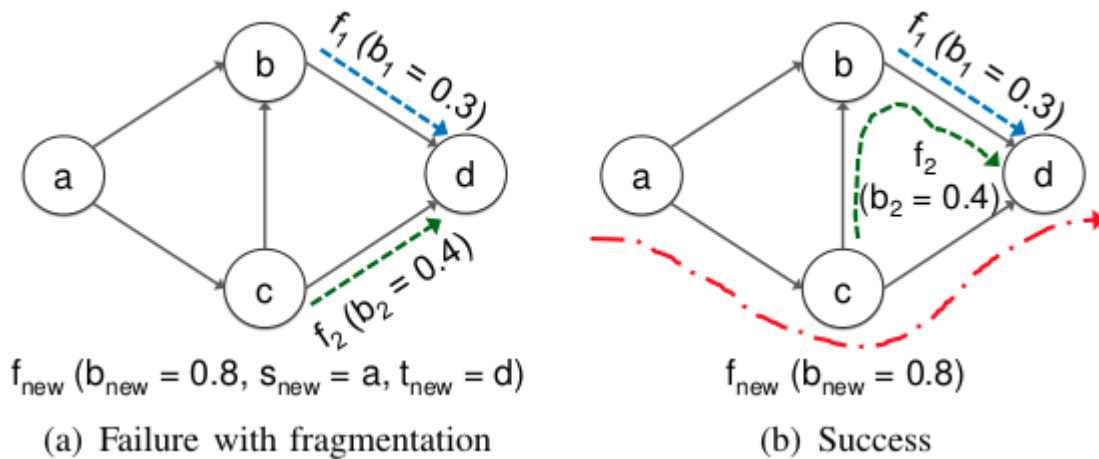
Obrázok 33. Štruktúra SDN v reálnom čase [41]

Systém funguje tak, že po nájdení cesty modulom Routing s vyhovujúcou šírkou pásma, túto cestu pošle do modulu Scheduler. Ktorý priradí toku prioritu a zistí či sa naplánovaním tohto toku neporušia splnenia D_k ostatných tokov. V prípade potreby odošle spätnú väzbu späť do modulu Routing, kvôli preplánovaniu cesty. [41]

13.3. Smerovací modul

13.3.1. Constraint-Based Routing (CBR)

$$\begin{aligned} & \text{minimize} && |R_{new}| \\ & \text{subject to} && \forall l(a,b) \in R_{new}, \sum_{\forall k:l(a,b) \in R_k} b_k \leq c(a,b) \end{aligned}$$



Obrázok 34. Ukážka algoritmu [41]

13.3.2. MILP

Algorithm 1 Multi-Commodity Flow (MCF)

input : $G(V,E)$, $F=\{f_k \mid f_k = (s_k, t_k, b_k)\}$, c_l : capacity of link l ,
 $E=\{E_p \mid p \in E_p, E_p = I_p \cup O_p\}$, I_p : ingress link, O_p : egress link

output: $\forall f_k \in F, \forall l \in E$,

$$X_k^l = \begin{cases} 1, & \text{if flow } f_k \text{ flows into link } l \\ 0, & \text{otherwise} \end{cases}$$

$$\text{minimize}_X \sum_{l \in E} \sum_{f_k \in F} X_k^l b_k$$

$$\text{subject to} \quad \forall l \in E, \sum_{f_k \in F} X_k^l b_k \leq c_l$$

$$\forall f_k \in F, \sum_{l \in O_{s_k}} X_k^l b_k = \sum_{l \in I_{t_k}} X_k^l b_k = b_k$$

$$\forall f_k \in F, \forall p \in V, p \neq s_k, t_k, \sum_{l \in I_p} X_k^l = \sum_{l \in O_p} X_k^l$$

$$\forall f_k \in F, \forall p \in V, \sum_{l \in E_p} X_k^l \leq 2$$

$$\forall f_k \in F, \sum_{l \in E_{s_k}} X_k^l = 1 \text{ and } \sum_{l \in E_{t_k}} X_k^l = 1$$

Obrázok 35. MILP algoritmus [41]

13.4. Cluster-based Adaptive Routing (CAR)

Systém zahŕňa Cluster-based Adaptive Routing (CAR), ktorý poskytuje dobrý pomer medzi výmenou efektívnosti a zložitosti riešenia s možnosťou škálovania. Pri reprezentácii topológie ako $G\langle V, E \rangle$ a množine tokov F , ktoré pozostávajú z $f \in F$ a sú priradené vyhovujúcej ceste R_k .

Keď sa akceptuje nový tok f_{new} s garanciou šírky pásma, potenciálne môže ísť o problém s kritickou linkou v toku f_{new} . Kritická linka je linka, ktorej dostupná šírka pásma je menšia ako b_k . Toto je riešené vytvorením virtuálnej topológie $G^*\langle V^*, E^* \rangle$, kde ak existuje cesta R^*_{new} pre f_{new} v G^* , tak musia existovať aj všetky cesty R_k pre všetky $f_k \in F$, so všetkými požiadavkami na šírku pásma. Pre pochopenie je potrebné si zaviesť nasledujúce pojmy. Pre transformáciu $G\langle V, E \rangle$ na $G^*\langle V^*, E^* \rangle$, najprv rozdelíme časť $G\langle V, E \rangle$ na množinu nesúvisiacich skupín $G_i\langle V_i, E_i \rangle$. Pre každý vrchol $v \in V$, platí, že patrí presne do jednej skupiny $G_i (v \in V_i)$. Každá linku $l(a,b) \in E$, ktorej a aj b patrí do jedného G_i je interná, alebo je externá, ak linka spája dve skupiny. Pre každý vrchol $v \in V_i$, platí, že vrchol je buď hraničný, ak ide doň aspoň jedna externá linka, inak je vrchol interný. Pre každý vrchol $v \in V$, platí, že je virtuálny, ak je zdrojom s_{new} pre cieľ t_{new} toku f_{new} alebo je hraničný vrchol. Pre každý pár virtuálnych vrcholov (a, b) , ktoré patria do tej istej skupiny, skonštruujeme obojsmernú virtuálnu linku $l^*(a, b)$ a $l^*(b, a)$ medzi vrcholmi a a b . Potom vytvoríme V^* , ako množinu všetkých vrcholov a E^* ako množinu všetkých virtuálnych a externých liniek. [41]

13.4.1. Skonštruovanie skupiny

Skonštruovanie skupiny sa robí nasledovným spôsobom [41]:

1. Najprv transformujeme $G\langle V, E \rangle$ na $G'\langle V', E' \rangle$ kde $V = V'$ a E' obsahuje len kritické linky f_{new} . Táto transformácia je urobená odstránením všetkým nekritických liniek. Potom vytvoríme jednotlivé skupiny G_i iteratívne v krokoch 2 a 3.
2. Prejdeme na bod 3 ak V' je prázdne, inak skončíme. Ak hodnota i nie je inicializovaná inicializujeme ju.
3. Pre ľubovoľný vrchol $v' \in V'$, nájdeme množinu V_i vrcholov spojených v G' ako $v' \in V_i$. Toto je docieľené prehládavaním do šírky. Potom pre každý $v' \in V_i$, odstránime $v' \in V_i$ z V' . Zvýšime hodnotu i o 1 a ideme na krok 2.

13.4.2. Dynamické smerovanie v skupinách

Po vytvorení jednotlivých skupín, pre každú skupinu G_i , zadefinujeme množinu tokov F_i , ktoré patria do skupín nasledovne. Pre každý tok $f_k \in F$, ktorý používa linku $l(a, b) \in E_i$ a teda patrí do G_i , vytvoríme nový tok f'_k , ktorý má začiatkový bod v s'_k a cieľový bod v t'_k v rámci skupiny G_i . s'_k, t'_k

$\in V_i$ nasledovne, $s'k$ je sk , ak $sk \in V_i$ alebo prichádzajúci hraničný vrchol $v' \in V_i$, a podobne, $t'k$ je tk , ak $tk \in V_i$ alebo odchádzajúci hraničný vrchol $v'' \in V_i$. Považujeme všetky scenáre, v ktorých f_{new} patrí do G_i , preskúmavajúc všetky kombinácie, kde prichádzajúce a odchádzajúce hraničné vrcholy budú používané f_{new} . V_i^* predstavuje množinu virtuálnych vrcholov, ktoré patria do G_i , $V_i^* \equiv V^* \cap V_i$. Potom, použijeme MILP na optimalizáciu pre každý možný scenár párovania $s'_{new} \in V_i^*$ a $t'_{new} \in V_i^*$ pre f_{new} . Keď nájdeme riešenie MILP optimalizácie, znamená to, že je možné postavenie všetkých ciest s garantovaním šírky pásma, keď f_{new} prechádza z s'_{new} do t'_{new} v rámci skupiny. Potom pridáme virtuálnu linku $l^*(s'_{new}, t'_{new})$ do virtuálnej siete $G^* \langle V^*, E^* \rangle$ s kapacitou nastavenou na nekonečnú hodnotu. [41]

13.4.3. Skonštruovanie cesty

Každá linka $l^*(a, b)$ vo virtuálnej sieti $G^* \langle V^*, E^* \rangle$, je buď externá linka s väčšou šírkou pásma ako b_{new} alebo virtuálna linka, ktorá umožňuje vyriešiť problém ľubovoľnej kritickej linky na podporu f_{new} . Potom spustíme CBR na nájdenie cesty R_{new} pre f_{new} v G^* a potom spustíme spracovanie všetkých virtuálnych liniek $l^*(a, b) \in R_{new}$ nasledovne. Pre každú virtuálnu linku $l^*(a, b) \in R_{new}$, musí existovať skupina G_i , na základe definície $l(a, b) \in V_i$, riešenia MILP optimalizácie pre G_i v scenári $a=s'_{new}$ a $b=t'_{new}$. Apelujúc na riešenie rozpoznávame cesty existujúcich tokov a nahrádzame virtuálnu linku $l^*(a, b)$, inou internou linkou. [41]

13.5. Plánovací modul

Predpokladáme, že topológia $G \langle V, E \rangle$ a množina F tokov v reálnom čase, kde každému toku $f_k \in F$ je priradená cesta R_k . Potom môžeme riešiť problém zisťovania priority P_k pre každý tok f_k , tak aby sme dodržali D_k .

13.5.1. Holistická analýza

Holistická analýza reprezentuje najhorší scenár doby odozvy pre distribuovaný model úloh, kde úloha $p_i (T_i, D_i, \{C_{i,j}\})$ je sekvencia podúloh $p_{i,j} \dots p_{i,N_i}$ s prednostnými obmedzeniami. Podúloha $p_{i,j+1}$ môže byť vykonávaná v najhoršom možnom vykonávanom čase $C_{i,j}$ aj na inom vrchole iba ak prednostná podúloha $p_{i,j}$ skončila vykonávanie. Keďže je ťažké považovať prednostné obmedzenia v plánovacej analýze presne, je tento problém považovaný ako NP-úplný. Odchýlka v tomto prípade slúži na zlepšenie prednostných obmedzení. Odchýlka $J_{i,j}$ podúlohy $T_{i,j}$ reprezentuje rozdiel času od ukončenia vykonávania prednostnej podúlohy $p_{i,j-1}$ po začatie vykonávania podúlohy $p_{i,j}$. [41]:

$$J_{i,j} = \sum_{k=1}^{j-1} w_{i,k}.$$

(12)

Holistická analýza ukazuje, že spojenie W_i na dobu odozvy úlohy p_i môže byť odvodené vypočítaním maximálneho oneskorenia $w_{i,j}$, ktoré má každá podúloha $p_{i,j}$. $w_{i,a}$ je vypočítané pomocou nasledovného vzťahu [41]:

$$w_{i,j}^{n+1} = C_i + \sum_{\forall k \in hp(i)} \left\lceil \frac{J_{k,j} + w_{i,j}^n}{T_k} \right\rceil C_k,$$

(13)

Kde $hp(i)$ je množina úloh p_i s vyššou prioritou. Iteratívny výpočet $w_{i,j}$ začína s $w_{0i,j} = 0$. Pokračuje, až kým $w_{i,j}^{n+1} = w_{i,j}^n$, alebo nedosiahne hodnotu väčšiu ako D_i . Najhorší možný čas doby odozvy W_i úlohy p_i je $W_i = \sum_j w_{i,j}$, a je naplánovaný, ak $W_i \leq D_i$. [41]

Maximálny čas prenosu správy M_k je daný vzťahom [41]:

$$C_k = \max_{l_a \in R_k} C_{k,a} + T_P \cdot |R_k|, \text{ where } C_{k,a} = (M_k + \text{pkt}_k^{\text{hd}}) / c_a,$$

(14)

A pkt_k^{hd} je veľkosť hlavičky paketu M_k a T_P je najväčšie možné oneskorenie spracovania vo vrchole. Jka je definované ako $w_{k,a-1}$, kde l_{a-1} predstavuje predošlú linku l_a v R_k . Potom maximálne oneskorenie $w_{k,a}$, ktoré má tok f_k na linke l_a je vypočítaný nasledovným vzťahom [41]:

$$w_{k,a}^{n+1} = B_{k,a} + \sum_{\forall i \in hp(k,a)} \left\lceil \frac{J_{i,a} + w_{k,a}^n}{T_i} \right\rceil C_{i,a},$$

(15)

, kde $B_{k,a}$ je maximálny čas blokovania od tokov z nižšou prioritou. $B_{k,a}$ sa vypočíta nasledovne [41]:

$B_{k,a} = \text{pkt}^{\text{size}}/c_a$, kde pkt size je veľkosť jednotlivého paketu. Iterácia začína s hodnotou $w_{0k,a} = C_{k,a}$. Odchýlka $J_{i,a}$ je definovaná nasledovne [41]:

$$J_{i,a} = \sum_{\forall l_b \in R_i^a} (w_{i,b} + \frac{\text{pkt}^{\text{size}}}{c_b}), \quad (16)$$

, kde $R_{i,a}$ reprezentuje podmnožinu R_i , ktorá zahŕňa iba predošlé linky l_a v ceste R_i . Potom najhorší možný scenár doby odozvy W_k toku f_k je vypočítaný nasledovne [41]:

$$W_k = \sum_{\forall l_a \in R_k} (w_{k,a} + \frac{\text{pkt}^{\text{size}}}{c_a}) + C_k. \quad (17)$$

13.5.2. Priradovanie priority

Na vyriešenie problému priradovania priority v reálnom čase je predstavený algoritmus OPA (Optimal Priority Assignment). Algoritmus OPA sa skladá z troch podmienok [41]:

1. Nastaviteľnosť toku je podmienená množinou tokov s vyššou alebo rovnakou prioritou, ale nie ich poradím.
2. Nastaviteľnosť toku je závislá aj od množiny tokov z nižšou prioritou, ale nie jej poradím.
3. Ak tok s nízkou prioritou bol naplánovaný, po priradení vyššej priority sa nemôže stať, že bude neplánovateľný.

Holistická analýza s odchýlkou $J_{k,k}$, ktorá je definovaná v rovniciach vyššie nie je kompatibilná s OPA algoritmom. [41]

Vieme zdefinovať odchýlku $J_{k,a}$, ktorá nie je senzitívna na prioritu a je definovaná ako $J_{k,a} = D_k - C_k$. Ak existuje tok f_k , ktorý $w_{k,a-1} > D_k - C_k$. Táto odchýlka už je kompatibilná s OPA algoritmom. [41]

13.5.3. OPA

```

zoradPriority(pole_priorit);
for each pole_priorit:
    for each nepriradene_toky:
        if planovatelnyTok(priorita) == True:
            priradPrioritu(tok);
        end if;

```

```

        break for;
    end for;
    planovatelny = False;
    return planovatelny;
end for;
planovatelny = True;
return planovatelny;

```

V prípade SDN prepínačov vieme priradiť 8 druhov priorít na port a príkaz break for; môže byť z algoritmu OPA vylúčený. V SDN prostredí je taktiež vypnutý MDNS a je používané statické ARP. Čas vo vrcholoch je synchronizovaný pomocou NTP, čo nám presne zaručí meranie času pre každý tok. Navyše bola implementovaná prioritná rada pre každú linku za použitia linux Traffic Control nástroja s Hierarchical Token Bucket. [41]

13.5.4. Feedback interface

Keď OPA algoritmus zlyhá pri priradení priority toku, tak vďaka feedback rozhraniu medzi schedule module a routing modulom je možné poslať signál na rekonfiguráciu ciest. Linka $BL_k \in R_k$ je linka s úzkym hrdlom toku f_k , ak

$$BL_k = \arg \max_{l \in R_k} w_{k,l}.$$

(18)

Ak OPA algoritmus nepriradí priority toku f_k , linku BL_k odstránime. Feedback rozhranie je prechádzané kým nie sú všetky toky naplánované, alebo nie je nájdená žiadna nová cesta. [41]

14. Testovanie

Vytvorenie testovacej topológie prebiehalo cez inicializačný skript v jazyku python, priebeh testovania, spolu s konkrétnymi krokmi sa nachádza v časti technická dokumentácia C – Priebeh testovania.

14.1. Testovací scenár č.1 (Hlavný testovací scenár)

Cieľom testovacieho scenára, je zistenie cesty, ktorú si kontrolér zvolil ako hlavnú pri preposielaní paketov a následne overenie, či cesta sa zmení v prípade vypnutia linky alebo zmenenia šírky pásma.

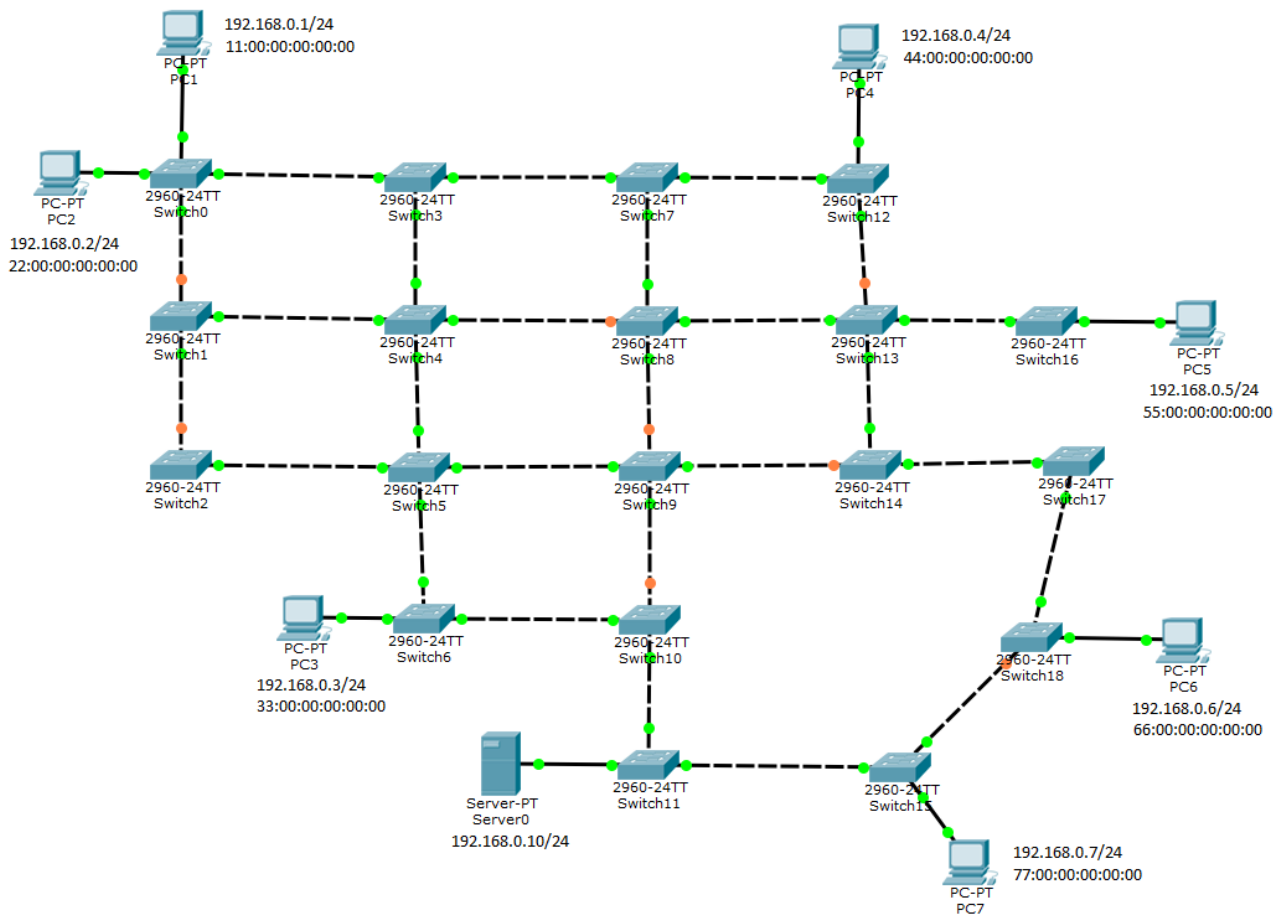
14.1.1. Testovacie prostredie

Operačný systém: Linux

OpenFlow prepínač: Open vSwitch

OpenFlow kontrolér: Ryu kontrolér

14.1.2. Testovacia topológia



Obrázok 36. Testovacia topológia č.1

14.1.3. Zhodnotenie

1. Sieť je zhotovená podľa topológie
2. Ping sa úspešne podaril na všetky zariadenia, aj keď kontroléru dlhšie trvá kým sa prepojí. V niektorých prípadoch dokonca neprešiel ping na zopár koncových zariadení a bolo treba reštartovať kontroléra a aj topológiu.
3. Pomocou príkazov: `sudo ovs-ofctl dump-flows sxx` a `sudo ovs-ofctl show sxx` (xx predstavujú poradové číslo prepínača) a ryu používateľského rozhrania ⁴ som zostavila cestu z h1 do h5. Cesta vedie cez nasledovné prepínače:

s00 – s03 – s07 – s08 – s13 – s16

⁴ `cd ryu`
`ryu-manager --observe-links ryu/app/gui_topology/gui_topology.py`

4. Pomocou príkazu `sudo ovs-ofctl mod-port sxx sxx-eth2 down` (prepínač a rozhranie k prepínaču) som vypla dané rozhranie a podľa 3. bodu som kontrolovala zmenu cesty. Zmenená cesta z h1 do h5 je nasledovná:

s00 – s03 – s04 – s08 – s13 – s16

5. Rovnako ako v predchádzajúcich krokoch som našla cestu z h3 na h7:

s06 – s05 – s09 – s10 – s11 – s15

6. Zmenila som šírku pásma medzi s09 a s10 na 1000Mbit

7. Cesta po zmene šírky pásma z h3 na h7 je nasledovná:

s06 – s10 – s9 – s14 – s17 – s18 – s15

Linka, na ktorej bola zmenená šírka pásma je aj naďalej využitá, len sa zmenila celková trasa.

8. Rovnako ako v predchádzajúcich úlohách som našla cestu medzi h2 a h4

s00 – s03 – s07 – s12

9. Na rozdiel od predchádzajúceho riešenia som zvolila menšiu šírku pásma na linkách, ktoré sú zaradené do cesty. Zvolila som si linky medzi prepínačmi s00 – s03 a s03 – s07 a dala som im šírku pásma 1Mbit.

10. Cesta po zmene šírky pásma z h2 na h4 je nasledovná:

s00 – s03 – s07 – s12

Ako je možné vidieť, tak cesta ostáva nezmenená aj po zmene šírky pásma.

11. Pomocou príkazov som spustila http server na h1 a poslala naň požiadavku cez h2. Výstup je možné vidieť na obrázku nižšie.

```
root@mininet-vm:~/mininet/custom# python -m SimpleHTTPServer 80 &
[1] 3659
root@mininet-vm:~/mininet/custom# Serving HTTP on 0.0.0.0 port 80 ...
192.168.0.2 - - [15/Nov/2017 15:51:07] "GET / HTTP/1.1" 200 -
192.168.0.2 - - [15/Nov/2017 15:51:29] "GET / HTTP/1.1" 200 -
192.168.0.2 - - [15/Nov/2017 15:52:46] "GET / HTTP/1.1" 200 -
192.168.0.2 - - [15/Nov/2017 15:52:48] "GET / HTTP/1.1" 200 -
```

Obrázok 37. Výstup HTTP servera

14.2. Testovací scénár č.2 (QoS)

Cieľom testovacieho scénára je využiť QoS rad na zistenie podpory QoS v mininete.

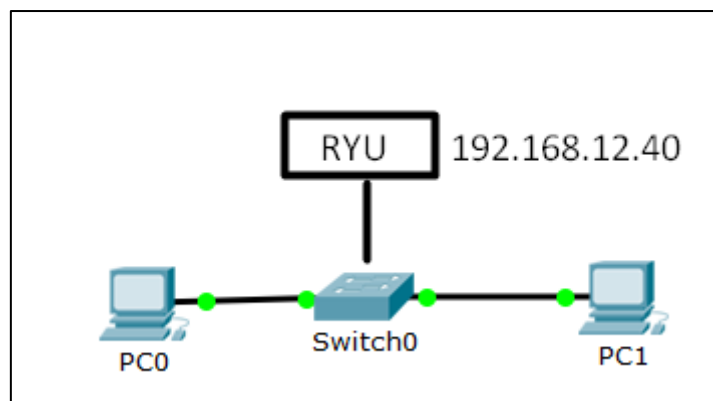
14.2.1. Testovacie prostredie

Operačný systém: Linux

OpenFlow prepínač: Open vSwitch

OpenFlow kontrolér: Ryu kontrolér

14.2.2. Testovacia topológia



Obrázok 38. Testovacia topológia č.2

Switch0 – Open vSwitch

PC0, PC1 – dve koncové zariadenia

14.2.3. Zhodnotenie

1. sieť je zhotovená podľa topológie
2. ping sa úspešne podaril na všetky zariadenia aj na VLAN

3. UDP komunikácia medzi stanicou a koncovým zariadením prebehla úspešne, ale šírka pásma nebola 5Mbit/s alebo 10Mbit/s
4. UDP komunikácia prebehla úspešne aj na zariadenia aj VLAN ale bez aplikovania radov
5. Nedosiahli sme rovnaké výsledky pretože sa nepodarilo aplikovať rozširovací modul L2QoS.py pre ryu-manager. Všade sme mali základnú 10Mbit/s priepustnosť z nastavenia.

Problémy

1. Zlý modul pre ryu-manager, musí sa použiť L2QoS.py pre podporu kontroly OF Switch QoS

14.3. Testovací scenár č.3 (Mininet - wifi)

Cieľom testovacieho scenára je otestovať funkcionality Wi-Fi a ad Hoc sietí v mininete-wifi⁵.

Testovací scenár testuje rozdelenie premávky medzi dva body prístupu.

14.3.1. Testovacie prostredie

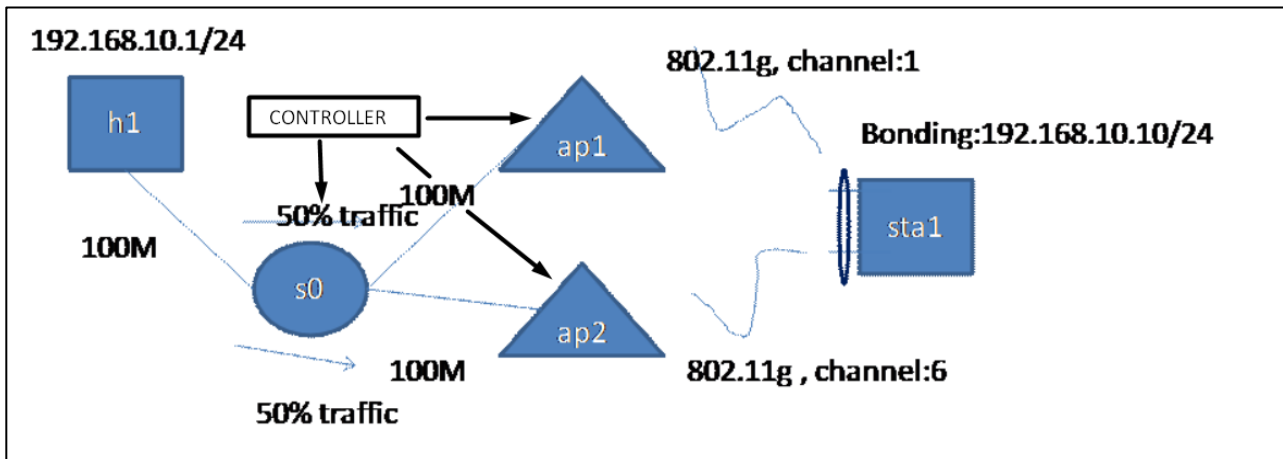
Operačný systém: Linux

OpenFlow prepínač: Open vSwitch

OpenFlow kontrolér: Ryu kontrolér

⁵ <http://sdncentral.ir/wp-content/uploads/2017/07/mininet-wifi-draft-manual.pdf>

14.3.2. Testovacia topológia



Obrázok 39. Testovacia topológia č.3

- s0 – Open vSwitch
- h1–koncové zariadenie
- controller – ryu kontrolér
- ap1, ap2 – body prístupu (access points)
- sta1 – stanica

14.3.3. Zhodnotenie

1. sieť je zhotovená podľa topológie
2. ping sa úspešne podaril na všetky zariadenia
3. UDP komunikácia medzi stanicou a koncovým zariadením prebehla neúspešne pri neuvedení manuálnych pravidiel
4. UDP komunikácia prebehla úspešne pri použití manuálnych pravidiel
5. Dosiahli sa podobné výsledky ako v práci na ktorú konkrétny testovací scenár bol odvodený, prepínač úspešne presmerovával komunikáciu medzi dvomi prístupovými bodmi.

14.4. Testovací scenár č.4

Cieľom testovacieho scenára je overiť fungovanie STP pri vyhodení jednej z liniek.

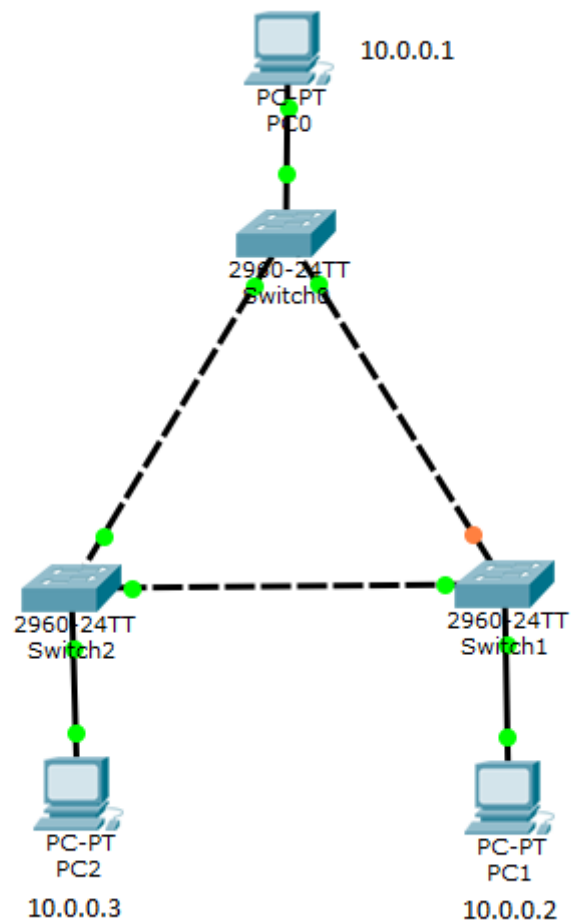
14.4.1. Testovacie prostredie

Operačný systém: Linux

OpenFlow prepínač: Open vSwitch

OpenFlow kontrolér: Ryu kontrolér

14.4.2. Testovacia topológia



Obrázok 40. Testovacia topológia č.4

14.4.3. Zhodnotenie

Názov testovacieho súboru:

- test3.py

Spúšťacie príkazy:

- *sudo mn --custom test3.py --topo test3 --controller=remote --switch ovs,failMode=standalone,stp=1*
- *ryu-manager /usr/local/lib/python2.7/dist-packages/ryu/app/simple_switch_stp.py*

Testovacie príkazy:

- *sudo ovs-ofctl mod-port s2 s2-eth2 down*
- *sudo ovs-ofctl mod-port s2 s2-eth2 up*
- *sudo ovs-ofctl mod-port s2 s2-eth1 down*
- *sudo ovs-ofctl mod-port s2 s2-eth1 up*
- *links*
- *pingall*

Testovanie zotavenia siete prebehlo úspešne najprv som vypol jednu linku medzi switchom 1 a 2 a potom druhú linku medzi switchom 0 a 2 v oboch prípadoch po naučení všetkých portov bola sieť schopná komunikácie.

14.5. Testovanie topológie z článku guck2014⁶

Cieľom testovania je otestovať funkčnosť a možnosti realizácie topológie a pomocou softvérových nástrojov uplatnených v článku guck2014[33]. Topológia je vytvorená ako jednosmerná kruhová šesť uzlová, pozostávajúca z jednotlivých zariadení, na ktorých je pripojené jedno koncové zariadenie.

⁶ <http://ieeexplore.ieee.org/document/6968971/>

14.5.1. Priebeh testovania

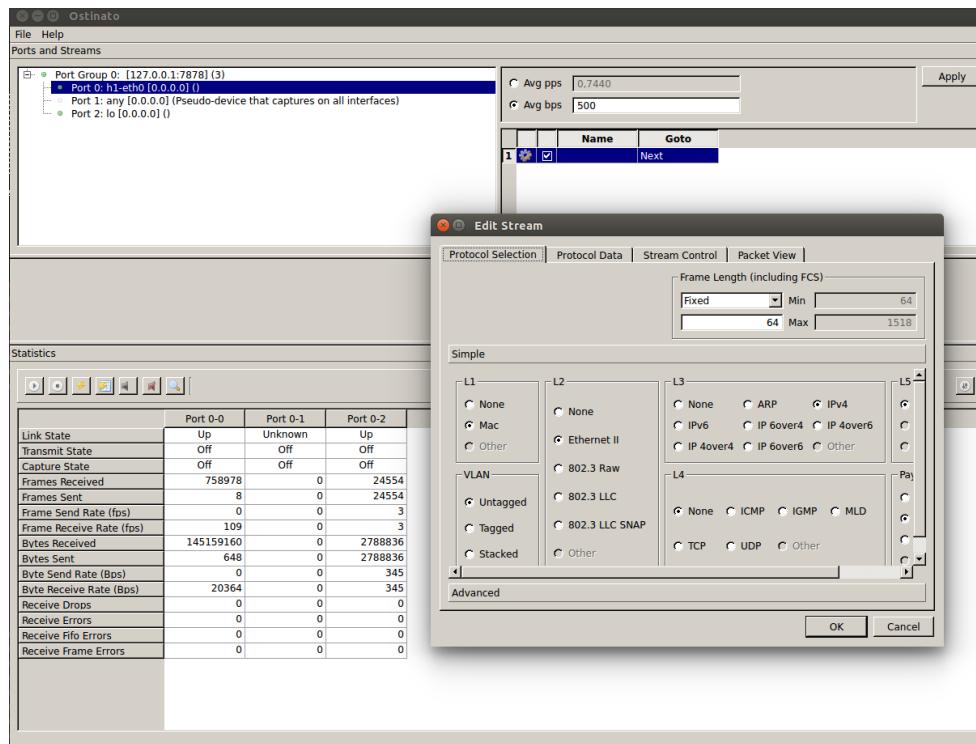
Testovanie pozostávalo z vytvorenia externej vlastnej topológie podľa článku guck2014[33] a zostavenej do skriptu guck.py. Táto je spustená príkazom:

- *sudo mn --custom guck.py --topo MyTopo*

Pre prepínače v kruhovej topológii je definovaná jednosmerná linka. Tá je štandardne emulátorom Mininet nepodporovaná, avšak je možné túto vlastnosť simulovať, napríklad definovaním štruktúr v tabuľkách tokov takým spôsobom, že celá premávka je smerovaná iba jedným rozhraním smerom von. Ďalšou časťou implementácie bolo aj zaradzovanie premávky na prepínačoch do prioritných radov, to je uskutočnené vykonaním príkazov na prepínačoch pre vytvorenie samotného radu, definovaním jeho úrovni a následne porovnávanie a zaradzovanie špecifických typov premávok do daného radu podľa niekoľkých porovnávacích pravidiel.

- *tc qdisc add dev h1-eth0 root handle 1: prio*
- *tc qdisc add dev h1-eth0 parent 1:1 handle 10: sfq*
- *tc qdisc add dev h1-eth0 parent 1:2 handle 20: sfq*
- *tc filter add dev h1-eth0 protocol ip parent 1: prio 1 u32 match ip protocol 0x11 0xff match ip tos 0x28 0xff flowid 1:1*

Po konfigurácii jednotlivých požadovaných radov a priradení na každom prepínači je možné pokračovať vytvorením požadovanej premávky na každom koncovom zariadení. Táto akcia je realizovaná aplikáciou Ostinato. Po otvorení terminálu konkrétneho zariadenia príkazom xterm sa program spustí zadaním príkazu Ostinato. Pridaním nového toku je možné modifikovať jeho špecifiká a tak upresniť aj jednotlivé zadané parametre podľa článku. Ukážka konfigurácie je na obrázku:



Obrázok 41. Generátor premávky Ostinato

Po modifikácii a uplatnení daných tokov je spustený generátor pre takto definované toky a premávka je dodávaná do kruhovej jednosmernej topológie s cieľom na jednotlivý počet skokov od zdrojového bodu s tým, že spracovanie tokov a paketov prioritných radov sa realizuje na prepínačoch.

14.6. Testovanie topológie z diplomovej práce podľa článku guck2014⁷

Cieľom je otestovať metódu z diplomového projektu[34] podľa topológie a testovania z článku guck2014[33]. Testovalo sa na jednosmernej, 6-uzlovej, kruhovej topológie.

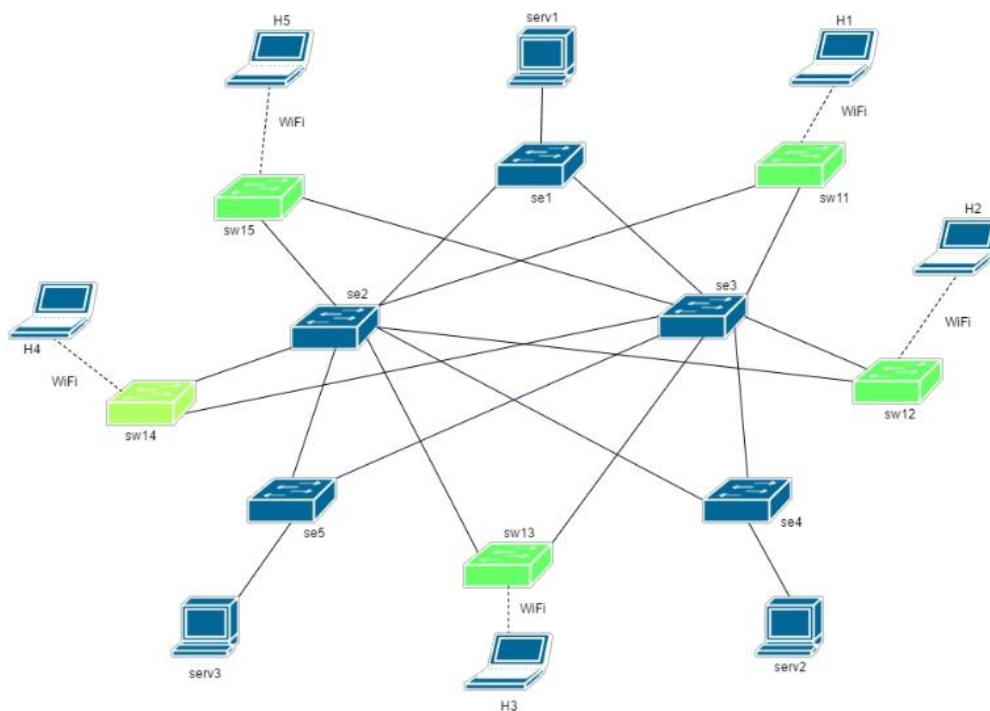
14.6.1. Pribeh testovania

Kontrolér bol spúšťaný podľa pokynov zistených pri jeho implementácii. Žiaľ po spustení topológie a spustením príkazu pingall, kedy by sa zistila dostupnosť všetkých hostov, sa nepodaril žiaľ ani tento základný príkaz. Po bližšom prezretí kódu controller.py bolo zistené, že k jednotlivým koncovým zariadeniam boli staticky priradené IP a teda, že sa robia jednotlivé porovnávaná podľa pevne daných IP adries. Z tohto dôvodu sme sa rozhodli zatiaľ ukončiť testovanie podľa tejto topológie, kým nebude tento problém vyriešený.

⁷ <http://ieeexplore.ieee.org/document/6968971/>

14.7. Testovanie metódy z diplomovej práce

Cieľom testovania je overiť funkčnosť implementovanej metódy z Diplomovej práce Michala Palatinusa [34]. Testovalo sa na topológii, ktorá obsahuje desať prepínačov, tri servery a päť klientských počítačov (ether_topo.py).



Obrázok 42. Topológia použitá pri testovaní

Testovanie generického algoritmu prebiehalo cez ICMP správy a bude sledovalo sa, či algoritmus nájde cesty medzi uzlami. Jednotlivé kroky testovania:

1. Spustenie topológie ether_topo.py
2. Príkaz pingall
3. Odsledovať príkazom net/links/tracert, aké linky sa vytvorili

Testovanie tejto topológie na našej virtuálnej mašine nebolo možné z dôvodu nefunkčnosti ns3 modulu. V diplomovej práci, v inštaláčnej ani používateľskej príručke sa nenachádza žiadna zmienka o tomto module. Testovanie prebehlo na jednej z topológií priloženej k diplomovej práci bez ns3 modulu (mininet_topo), ktorá obsahuje 5 klientských PC, 3 servery a 10 smerovačov. Výsledok testu ukázal nefunkčnosť liniek medzi klientskymi PC po zadaní príkazu pingall. Nefunkčnosť základného smerovania napovedá chybnou implementáciou, alebo inému problému. Testovanie pokračovalo na obraze z diplomovej práce, s cieľom zistiť, či je chyba na našej strane. Po spustení topológie s ns3 modulom (ether_topo.py) bol takisto výsledok testu neúspešný. Algoritmus nenašiel linky medzi

klientskymi PC, z čoho usudzujeme, že je nutná dodatočná konfigurácia, ktorá ale nie je spomínaná nikde v priloženej diplomovej práci[34].

15. Implementácia metódy z DP Michala Palatinusa

Kapitola je venovaná implementácii danej metódy a jednotlivým problémom. Ako bolo spomínané v návrhu, implementáciu sme rozdelili do troch súborov. Potrebná implementácia sa od návrhu ale líšila v jednotlivých bodoch:

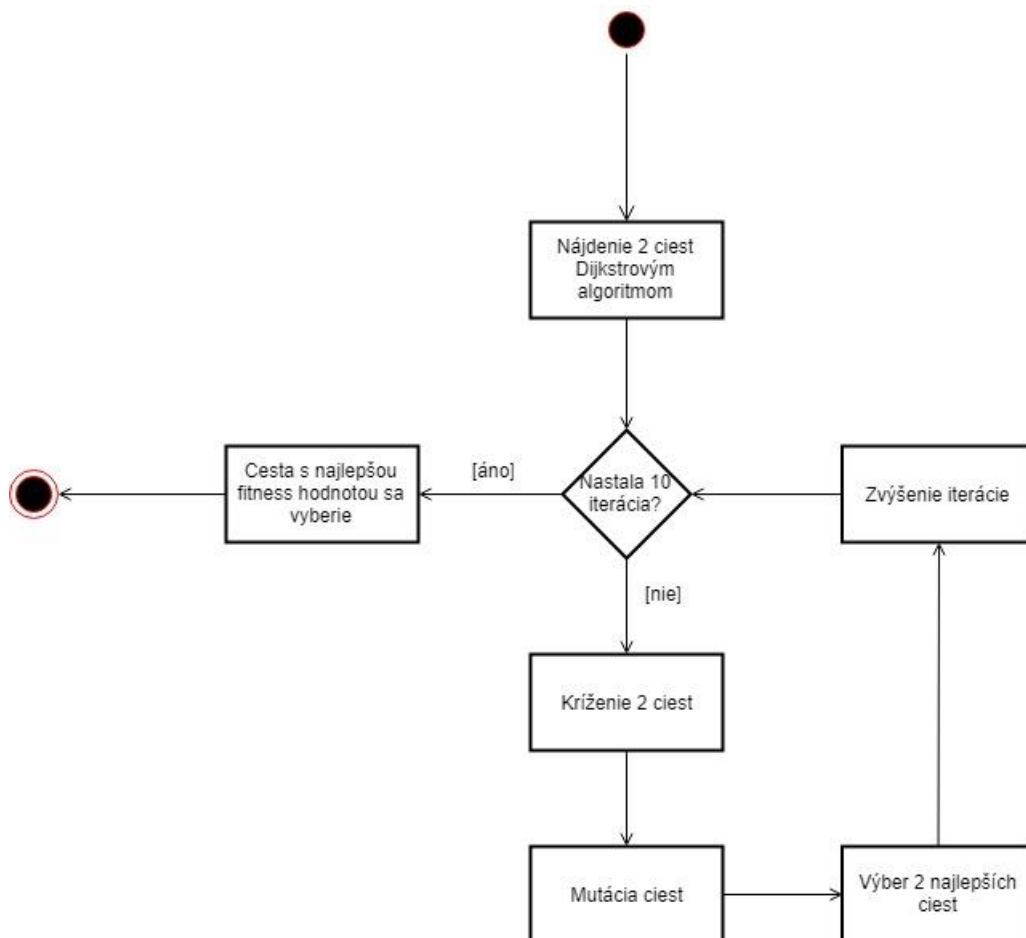
a) Súbor controller.py

Pole HOSTS nebolo treba upravovať. Síce na začiatku sú staticky priradené IP jednotlivých koncových zariadení, ďalej v súbore sa rieši aj prípad, kedy sa poli HOSTS daná IP nenachádza. V tomto prípade stačilo len vymazať staticky priradené informácie do poľa HOSTS.

a) Súbor globals.py

b) Súbor QoS_linkDB.txt

Implementácia metódy z DP Michala Palatinusa je znázornená nasledujúcim obrázkom.



Obrázok 43. Diagram toku pre metódu z DP Michala Palatinusa

15.1. Problémy pri implementácii

Pri implementácii došlo k viacerým problémom. Napríklad na tímovom serveri nebolo možné spúšťať danú metódu z viacerých dôvodov. Jeden z hlavných dôvodov je chybová hláška o využívaní už daného portu. Tento prípad bol riešený prepísaním portu, na ktorom by mala figurovať daná metóda. Žiaľ neúspešne. Z tohto dôvodu, sme začali implementáciu na lokálnych strojoch s mininet verzou 2.2.1. Na lokálnom stroji s ubuntu 16.04 LTS nasledovali ďalšie problémy. Pri spustení síce metóda už nevypisovala problém s využívaním rovnakého portu, avšak sa objavila chyba s indexovaním pri funkcii `topoToSet(self, topo)`. Chyba bola následne upravená o zníženie indexovania, kedy už pri spustení topológie neprebehla žiadna chyba. Následne po spustení príkazu `ping`, koncové zariadenia neboli schopné medzi sebou komunikovať, kvôli rovnakému problému, zlé indexovanie. Chyba bola následne opravená. Kontrolér bol spúšťaný príkazom `ryu-manager controller.py --observe-links`, avšak daný argument `--observe-links`, ktorý by mal zabezpečiť zobrazenie liniek, nefungoval správne. Namiesto poľa s linkami, cez ktoré by mal tok ísť, vypisovalo prázdne pole. Kontrolér sám o sebe je „nestabilný“. Pri spúšťaní vypisuje občasné chybové hlášky o zlom indexovaní, avšak nie vždy. Prípadné znovu spustenie kontroléra či samotnej topológie zvykne chybu vyriešiť.

15.2. Testovanie metódy

Metóda bola testovaná pomocou piatich rôznych tokov. Využitá bola vlastná topológia o 14 linkách. Nástroj na jednotlivé simulácie bol použitý *hping3* pre klienta a *iperf* pre server.

Jednotlivé toky, ktoré boli testované sú:

1. Prenos súborov

Protokol: TCP

Priepustnosť: 5,5Mbit/s

{'alpha': 0, 'beta': 0, 'gamma': 1, 'delay': 4.000, 'jitter': 0.01, 'thrput': 5500000}

Port: 5001

Veľkosť paketov: dynamicky dané TCP protokolom

Rýchlosť odosielania: dynamicky dané TCP protokolom

Odosielané dáta brané z existujúceho súboru (prepínač `-F /path/to/file.mp4`)

2. Skype hovor

Protokol: UDP (prepínač `-u`)

Priepustnosť: 100kbit/s

{'alpha': 1, 'beta': 0, 'gamma': 1, 'delay': 1.000, 'jitter': 0.01, 'thrput': 100000}

Port: 5002

Veľkosť paketov: 450B (prepínač *-l 450B*)

Rýchlosť odosielanie: 500 kbit/s (prepínač *-b 500k*)

3. Prenos videa

Protokol:UDP(preparač-*u*)

Priepustnosť:4,5Mbit/s

{'alpha': 1, 'beta': 0, 'gamma': 1, 'delay': 4.000, 'jitter': 0.01, 'thrput': 4500000}

Port: 5003

Veľkosť paketov: 1460B (prepínač *-l 1460B*)

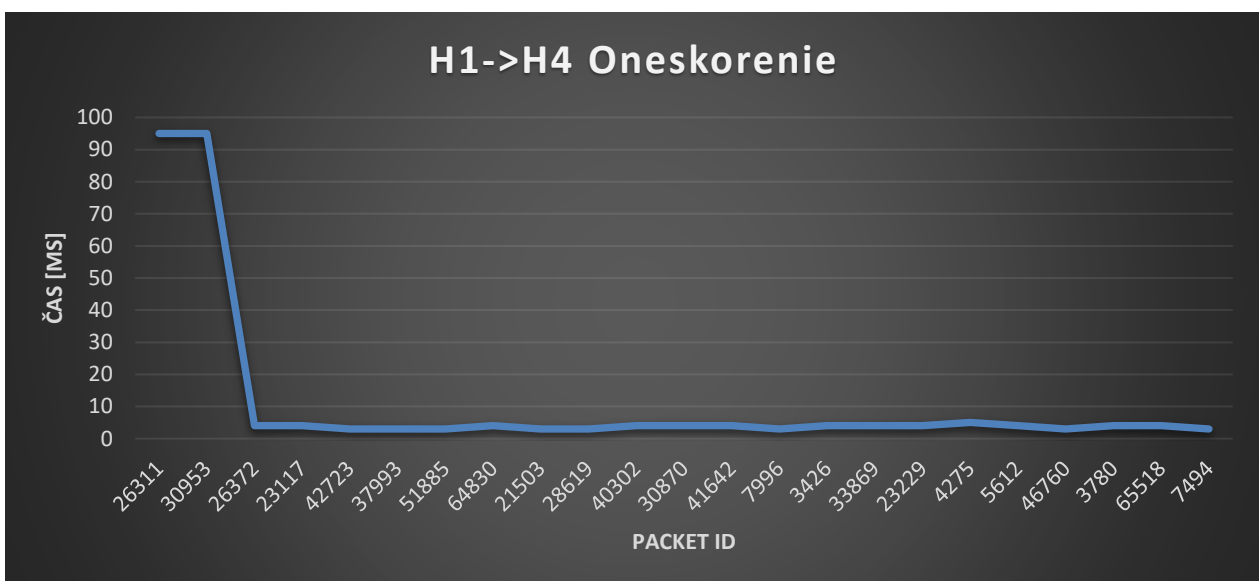
Rýchlosť odosielania: 4500 kbits/s (prepínač *-b 4500k*)

4. Manažment

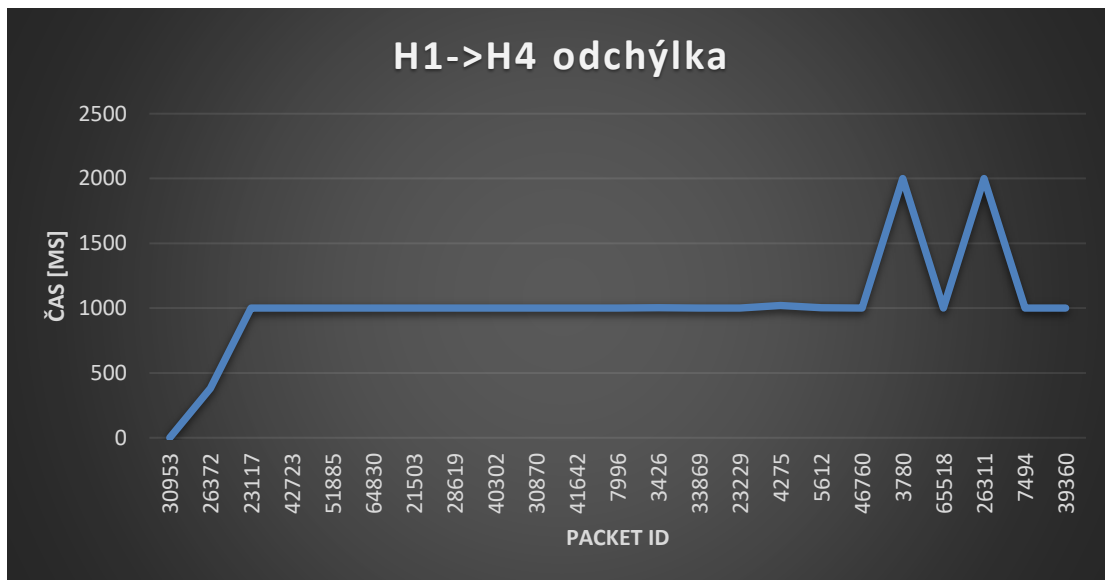
Manažment predstavuje premávku ako ICMP, ARP a ostatnú IP komunikáciu, ktorá nespadá do ostatných troch kategórií tokov.

Toky:

1. H4 iperf *-su -p 5002 -i 1*
H1 hping3 10.0.0.4 *-2 -V -p 5002 -d 450*



Graf 1. oneskorenie



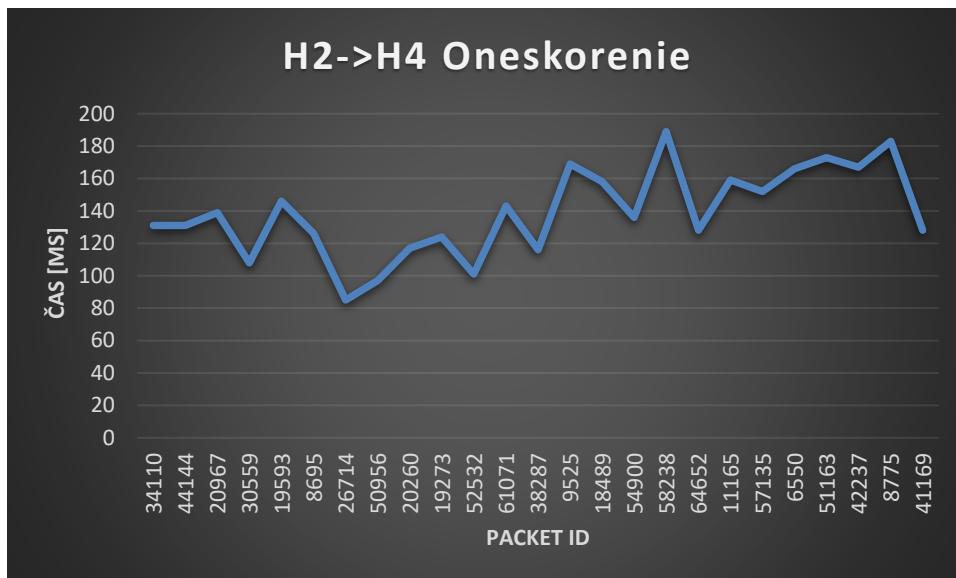
Graf 2. odchýlka

V prvom prípade sa testoval skype hovor (UDP komunikácia). Na grafe 1 a 2 možno vidieť oneskorenie a odchýlku v rámci toku medzi koncovými zariadeniami H1 a H4. Na grafe 1 možno vidieť oneskorenie jednotlivých paketov, ktorých maximum je 100 ms, pričom po konvergencii siete, sa oneskorenie konštantne drží pod 10 ms. Požadované maximálne oneskorenie bolo 1000 ms, čo daná metóda dodržala.

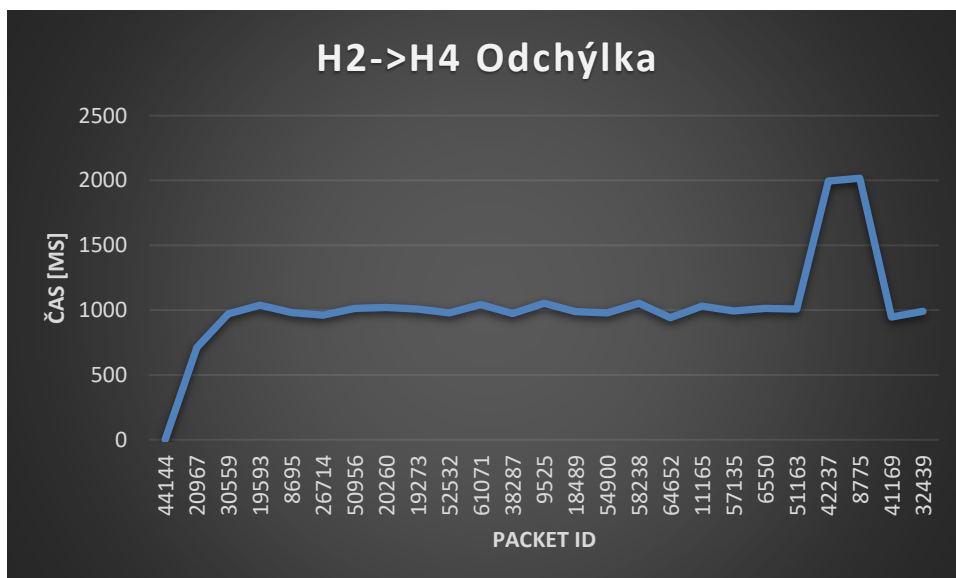
Na grafe 2 možno vidieť odchýlku, kedy *hping3* posielala pakety každú sekundu, z tohto dôvodu sa odchýlka drží okolo 1000 ms, pričom na konci je možné vidieť vyššie hodnoty.

2. H4 iperf -s -p 5001 -i 1

H2 hping3 10.0.0.4 -V -p 5001 -d 450 -E /home/peter/pythons/



Graf 3. oneskorenie



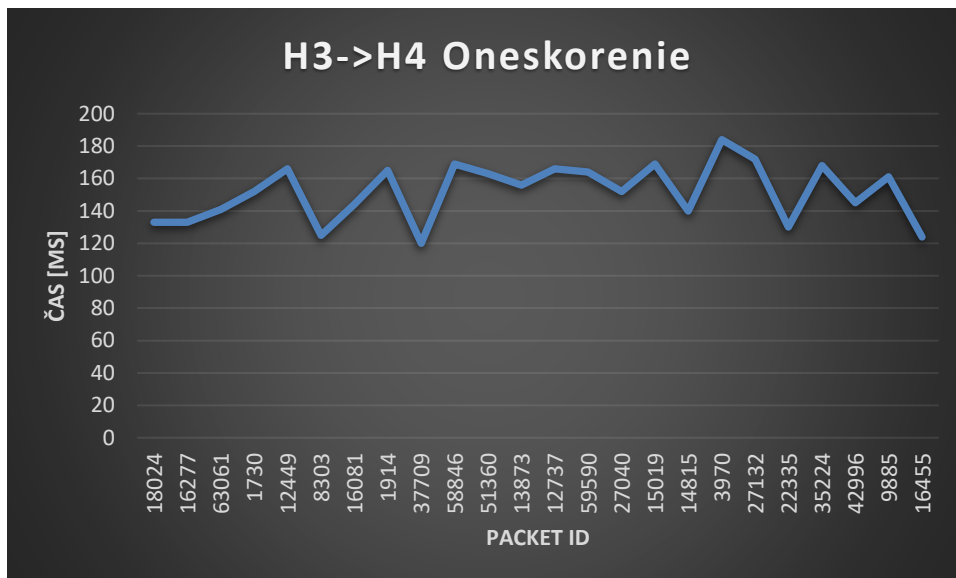
Graf 4. odchýlka

V druhom testovacom scenári sa skúmala TCP komunikácia, pri posielaní súborov z priečinka /home/peter/pythons/. V grafe 3 možno vidieť oneskorenie, kedy maximálne požadované oneskorenie je 4000 ms, kedy aj v tomto prípade metóda dodržala tento čas a jej maximálne oneskorenie bolo do 200 ms.

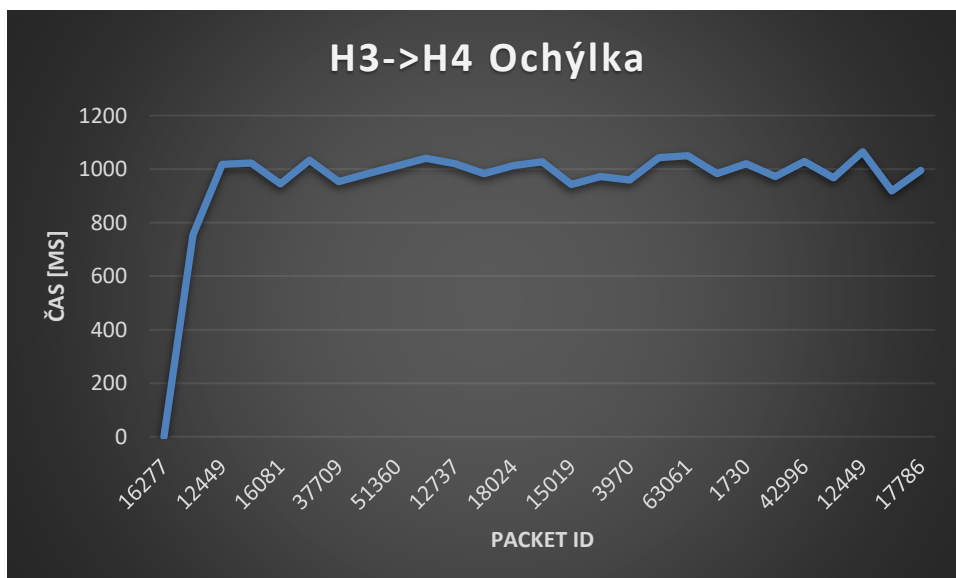
V prípade odchýlky na grafe 4, je možné vidieť „pohyblivosť“ okolo 1000 ms, kedy môžeme predpokladať, že maximálne požadovaná odchýlka 10 ms je prekročená.

3. H4 iperf -s -p 5001 -i 1

H3 hping3 10.0.0.4 -V -p 5001 -d 450 -E /home/peter/pythons/



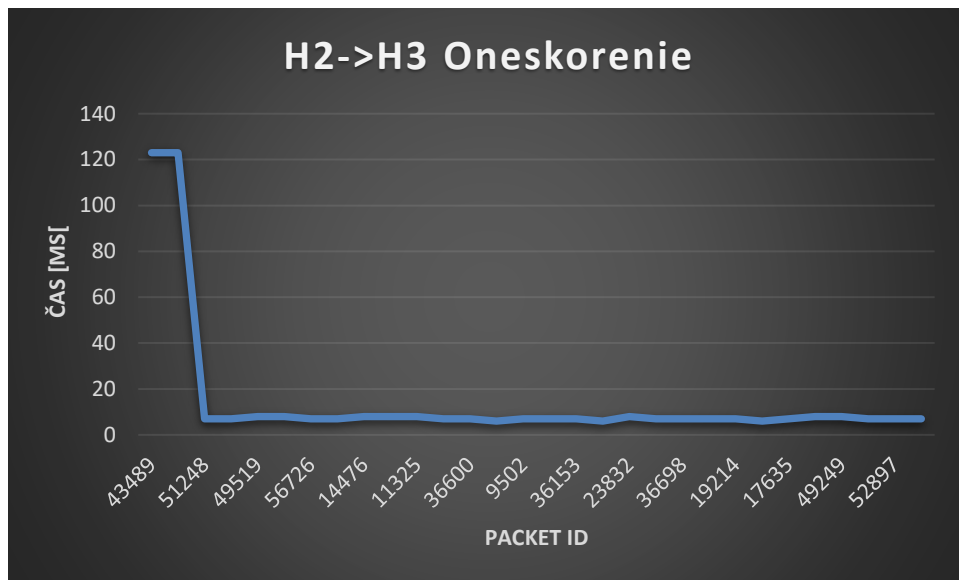
Graf 5. oneskorenie



Graf 6. odchýlka

Aj v tomto prípade, bola riešená TCP komunikácia s prenosom súborov z priečinku. Aj v tomto prípade platí to isté, čo je spomenuté pri testovacom scenári 2.

4. H3 iperf -su -p 5002 -i 1
H2 hping3 10.0.0.3 -2 -V -p 5002 -d 450



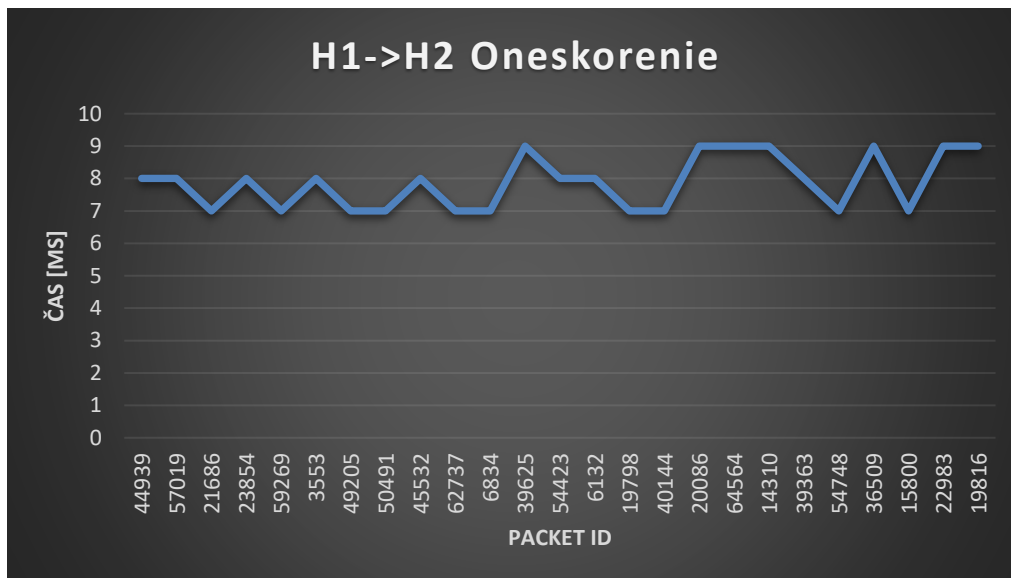
Graf 7. oneskorenie



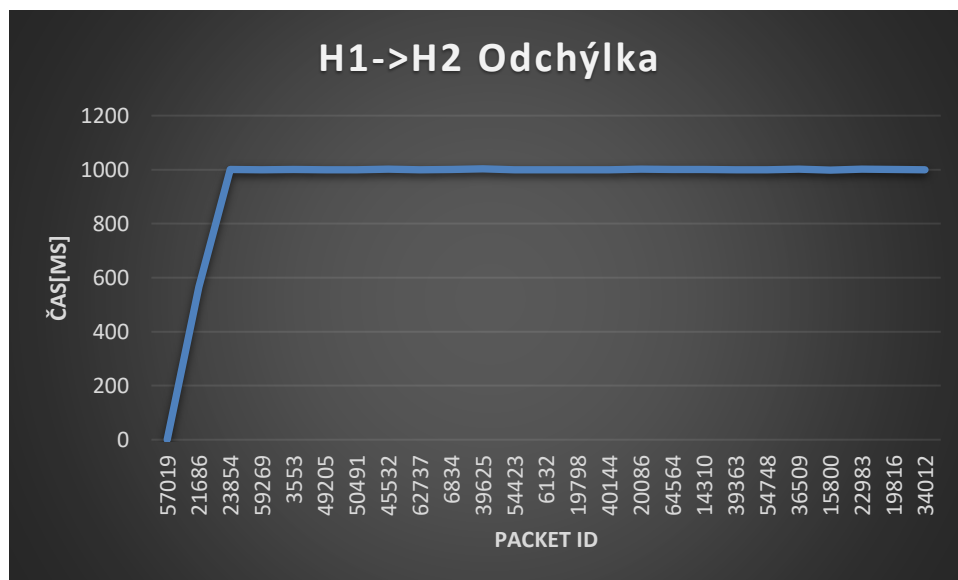
Graf 8. odchýlka

Hodnoty merania vo štvrtom testovacom scenári, sa približne zhodujú s meraniami v prvom testovacom, kedy v oboch prípadoch sa merala UDP komunikácia.

5. H2 iperf -su -p 5003 -i 1
H1 hping3 10.0.0.2 -2 -V -p 5003 -d 1460



Graf 9. oneskorenie



Graf 10. odchýlka

V poslednom testovacom scenári sa merala UDP komunikácia pre prenos videa. Aj v tomto prípade možno vidieť dodržanie maximálnych QoS parametrov.

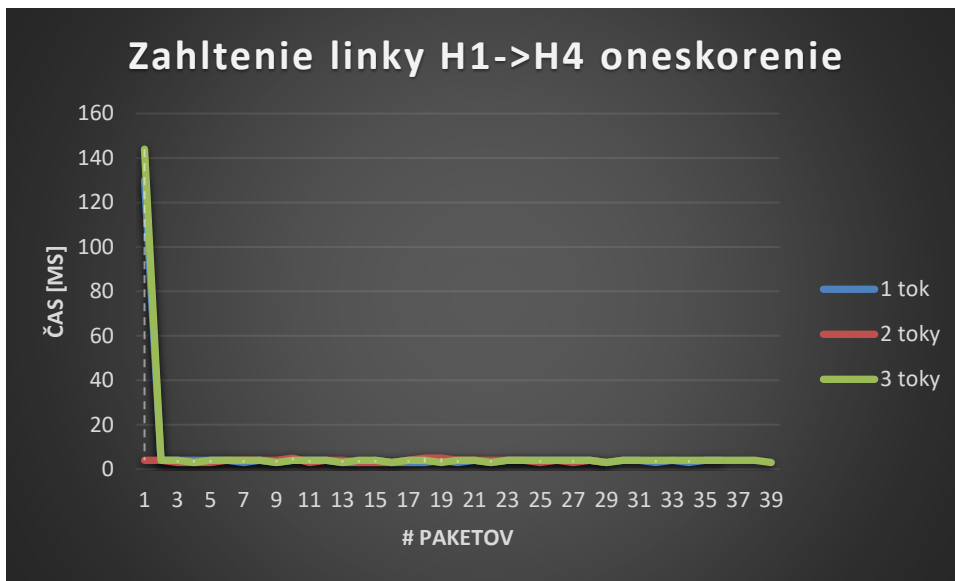
V rámci testovania sa testovali aj viaceré toky súbežne. V tomto prípade sme testovanie rozdelili na 3 hlavné testovacie scenáre.

1. Zahltienie linky – kedy sme testovali, ako sa kontrolér správa pri zahltení linky.

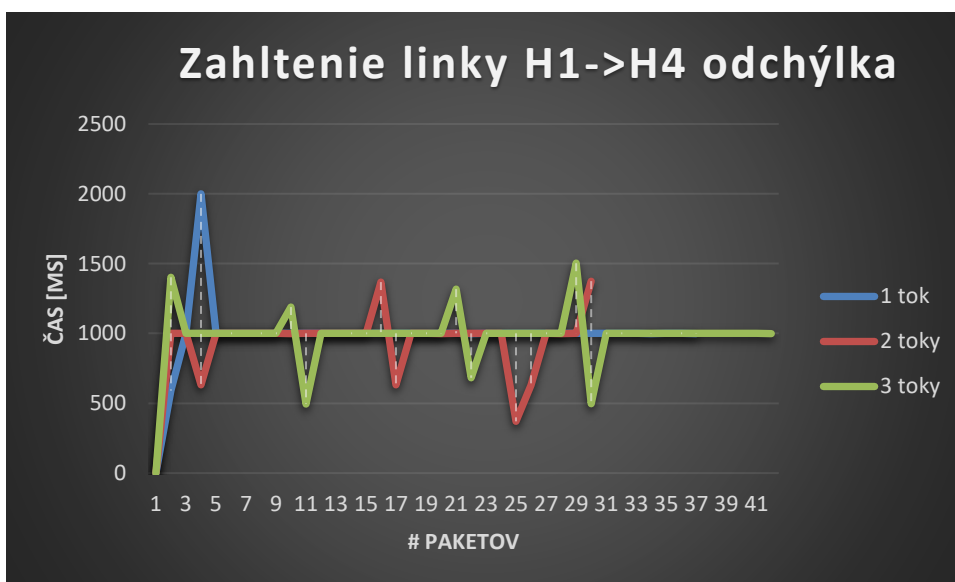
V rámci testovacieho scenára bolo potreba upraviť kapacitu linky, aby v prípade pridania tretieho toku nastal prípad, že linka už z hľadiska kapacity by nemala povoliť komunikáciu tretieho toku.

Avšak z technického dôvodu musela byť použitá topológia bez slučiek, kedy kontrolér, hoci vypíše nulovú kapacitu linky, tak nedokáže nájsť žiadnu inú zodpovedajúcu linku a tak ďalej púšťa komunikáciu rovnakou linkou, hoci nespĺňa kapacitu. Ako je možné vidieť na grafoch nižšie.

Oneskorenie aj po pridaní tretieho toku ostáva konštantne nad nulou. Avšak pri odchýlke je možno spozorovať väčšie množstvo anomálii, kedy by sa mala odchýlka udržiavať okolo 1000ms, ale je možné spozorovať aj vyššie či nižšie hodnoty.



Graf 11. zahltenie linky oneskorenie



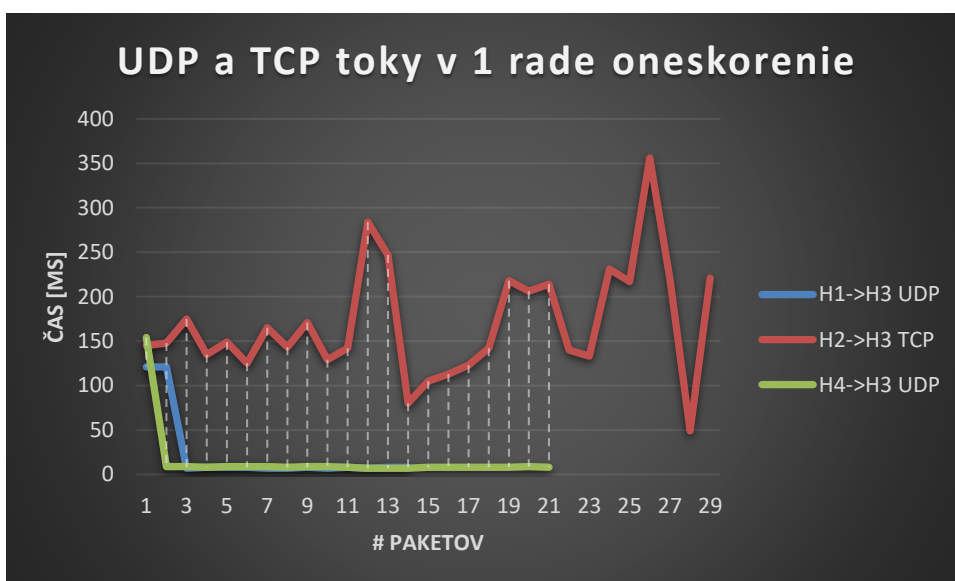
Graf 12. zahltenie linky odchýlka

2. Pridanie TCP toku do UDP tokov – kedy sme testovali, či kontrolér na základe QoS parametrov uprednostní TCP tok.

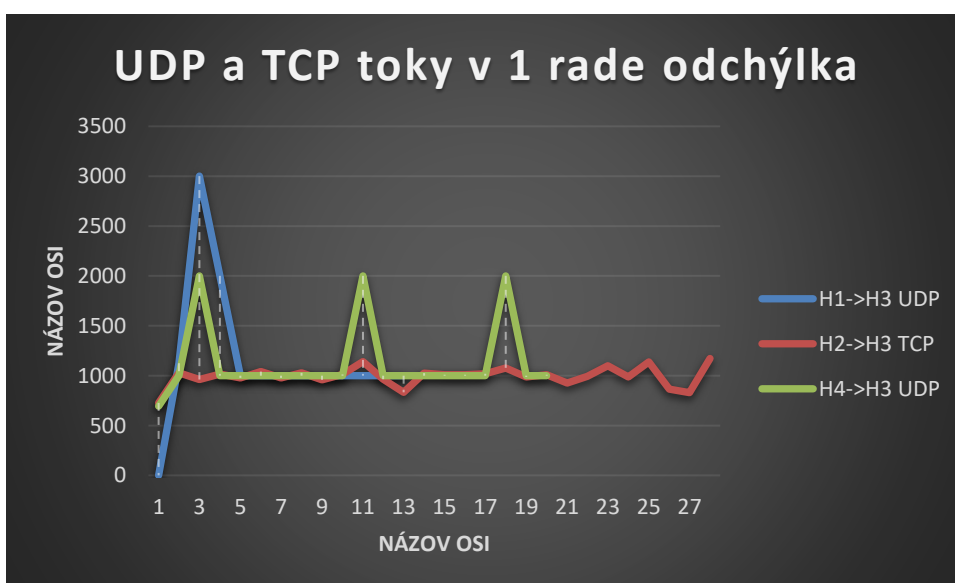
Kontrolér určuje priority jednotlivým radom a nie tokom. V tomto prípade sa testovalo v rámci jedného radu, kedy boli naraz spustené dva UDP toky a jeden TCP tok. V prípade viacerých radov, ak by boli rozdelené, že TCP tok je v jednom rade a UDP toky v druhom rade, kontrolér by mal uprednostniť TCP tok.

V prípade jedného radu, je možné spozorovať, že TCP tok je o niečo vyšší ako v prípade, keď bol TCP tok pustený sám na linke, kedy dosahoval pri oneskorení čas od 120ms do 180ms. Na nižšie zobrazenom grafe je možné vidieť, že oneskorenie sa pohybuje až do 350ms. Tiež je možné spozorovať, že UDP toky si stále udržiavajú konštantné oneskorenie okolo 0ms.

Na druhej strane, pri odchýlke je možné vidieť, že TCP tok na rozdiel UDP tokov nejaví žiadne anomálie, kedy pri TCP toku je konštantná odchýlka.



Graf 13. pridanie TCP toku oneskorenie



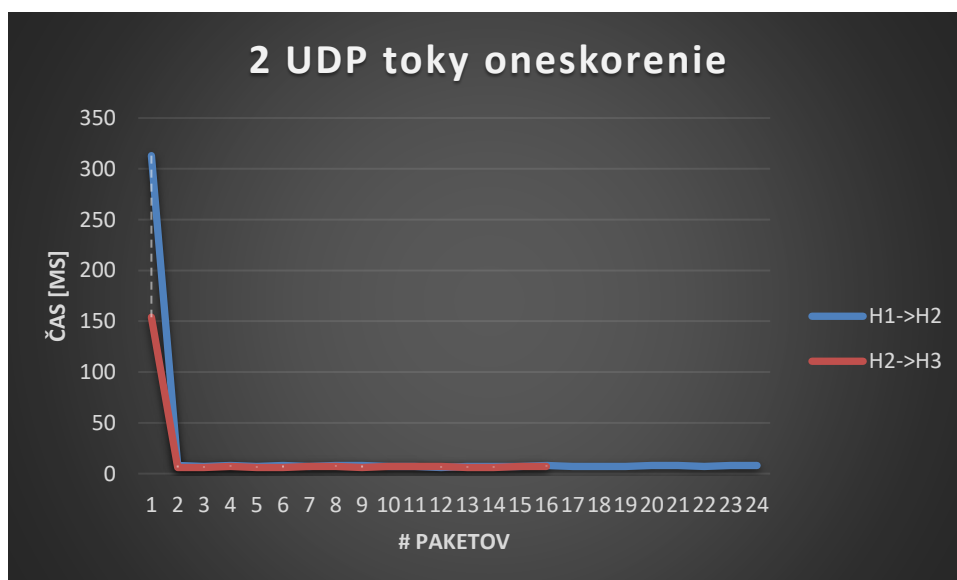
Graf 14. pridanie TCP toku odchýlka

3. Viaceré UDP toky – kedy sme testovali, či pri náraste UDP tokov, budú jednotlivé toky stále spĺňať QoS parametre.

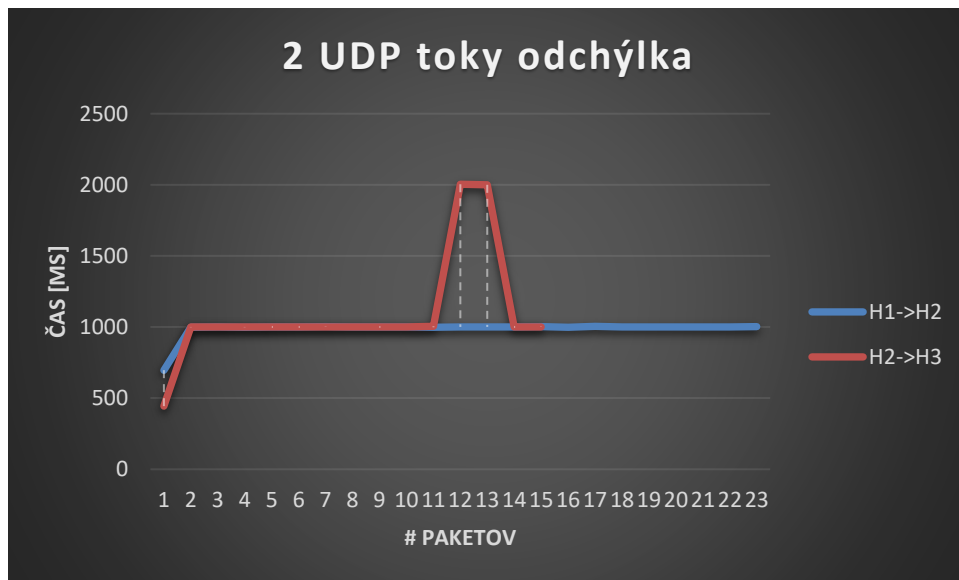
V rámci scenára sa testovali 2 rôzne typy UDP tokov – prenos videa a skype hovor.

Testovalo sa nasledovne: zo začiatku boli spustené len 2 UDP toky, pri ktorých sme si uložili výsledky (možno vidieť na grafoch). Následne sme pridali tretí UDP tok a postup opakovali. Nakoniec sme pridali posledný (štvrtý) UDP tok a namerané výsledky uložili.

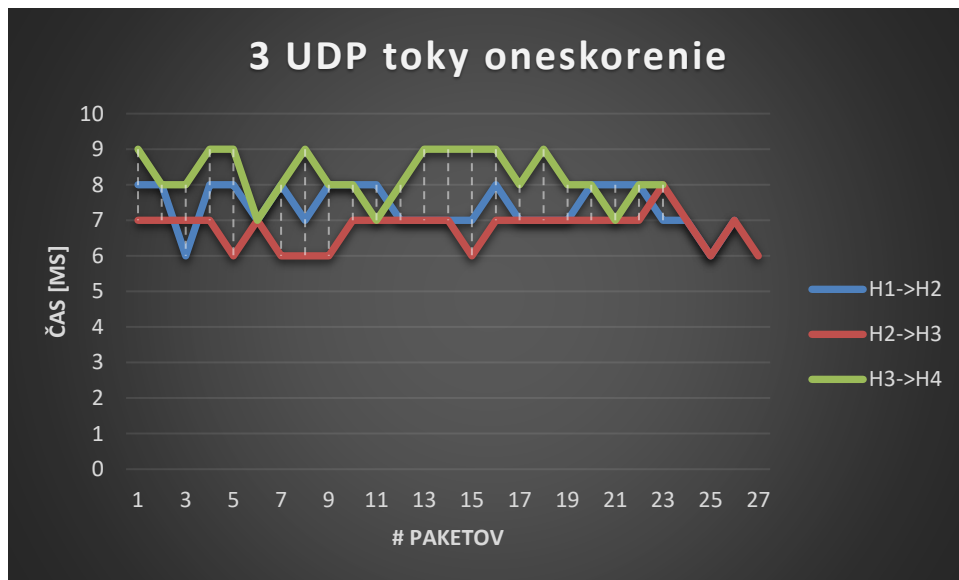
Na nižšie uvedených grafoch je možné vidieť oneskorenie a odchýlku od 2 UDP tokov po 4 UDP toky. Ako je možné spozorovať oneskorenie v rámci jednotlivých tokov nemá vplyv na ich počet. Avšak nárast počtu tokov ovplyvňuje odchýlku, kedy sa vyskytuje s vyšším počtom tokov aj viac anomálií v rámci grafu (2000ms).



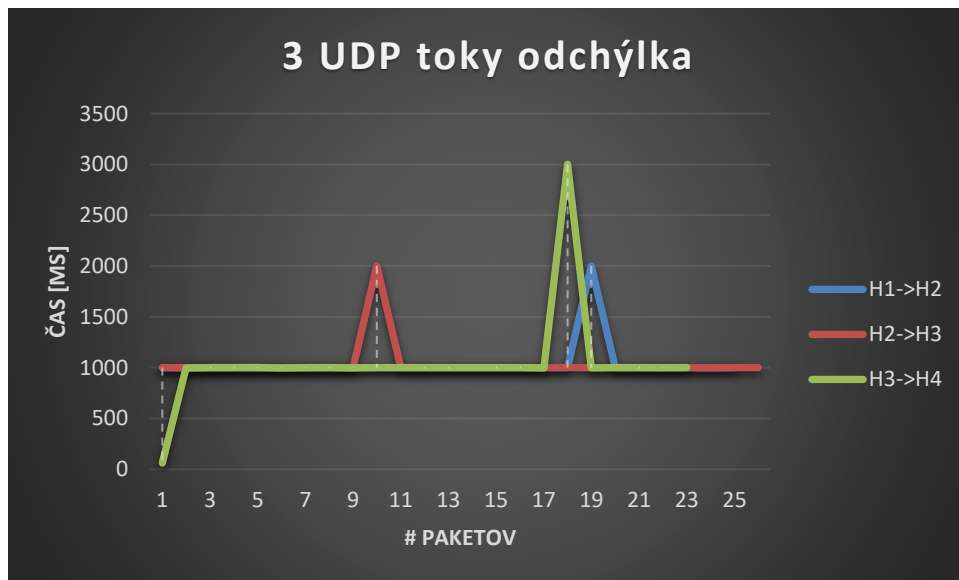
Graf 15. 2 UDP toky oneskorenie



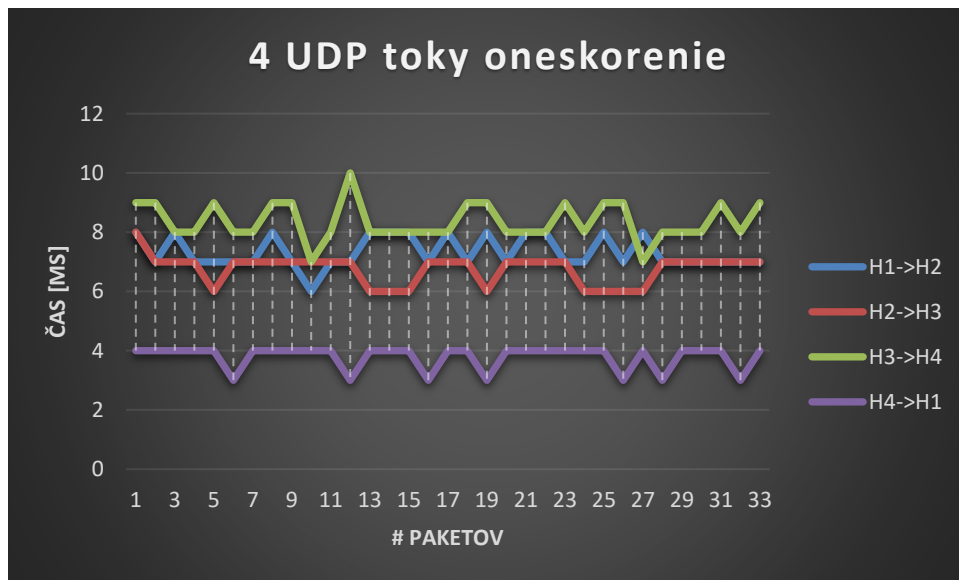
Graf 16. 2 UDP toky odchýlka



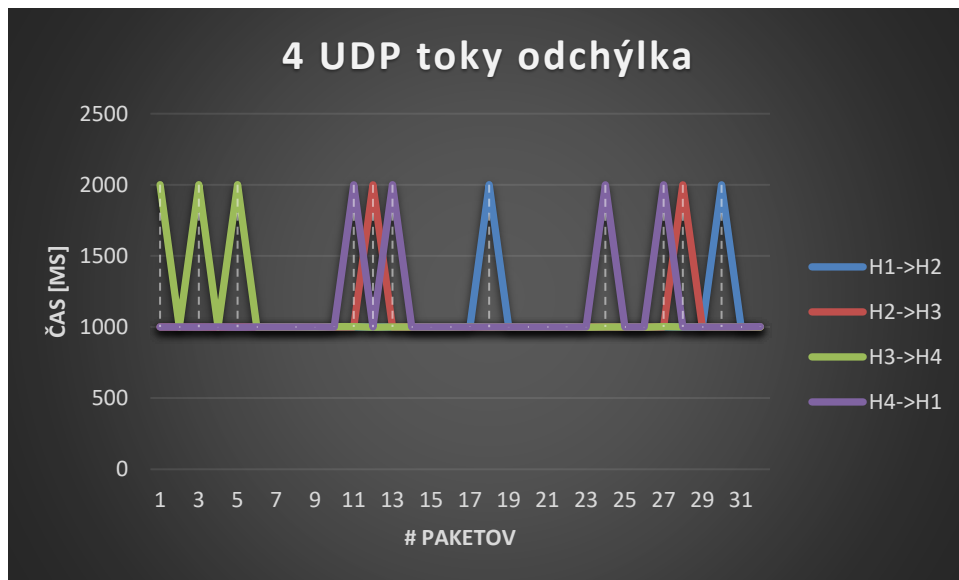
Graf 17. 3 UDP toky oneskorenie



Graf 18. 3 UDP toky odchýlka



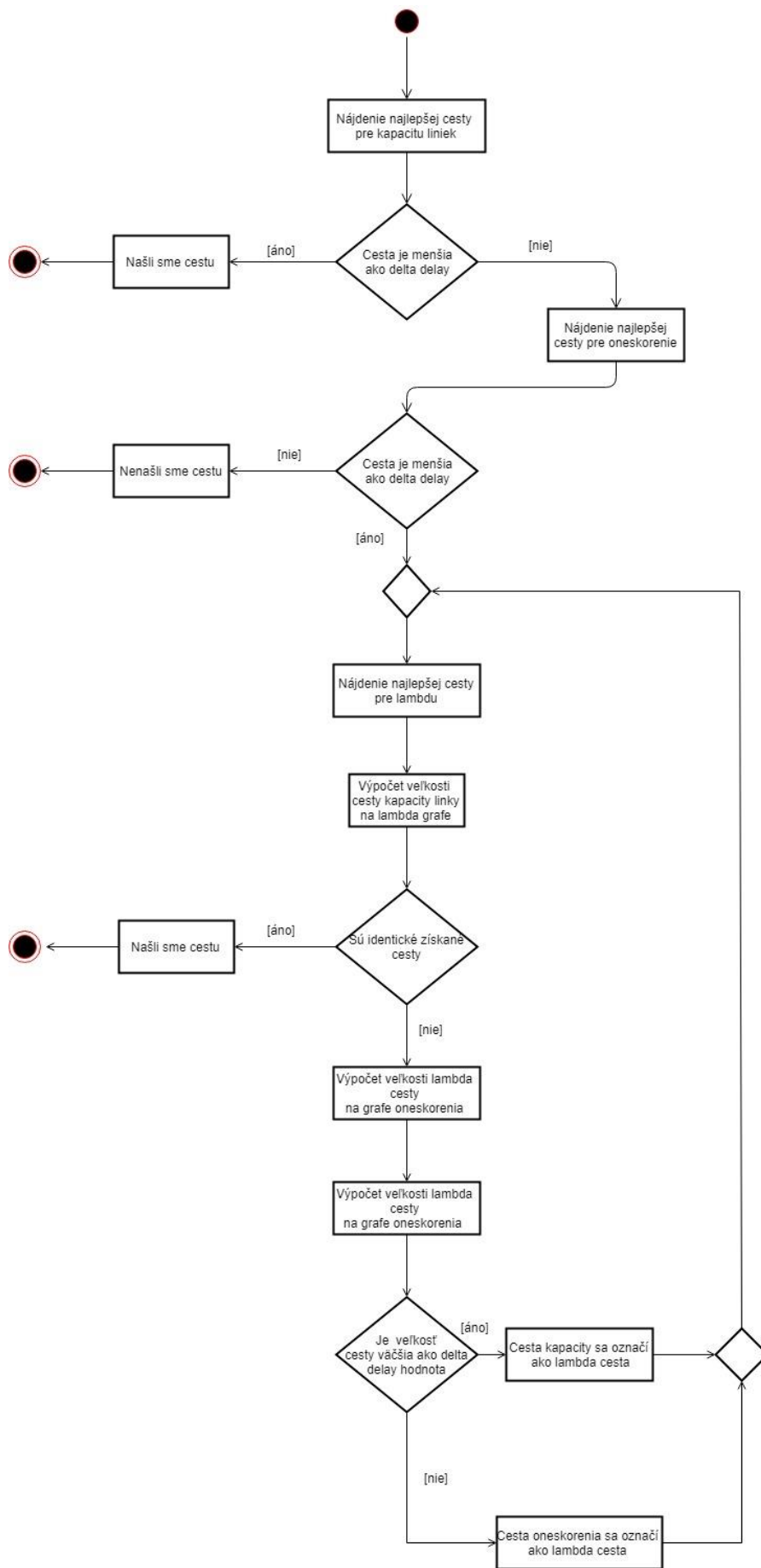
Graf 19. 4 UDP toky oneskorenie



Graf 20. 4 UDP toky odchýlka

16. Implementácia metódy LARAC

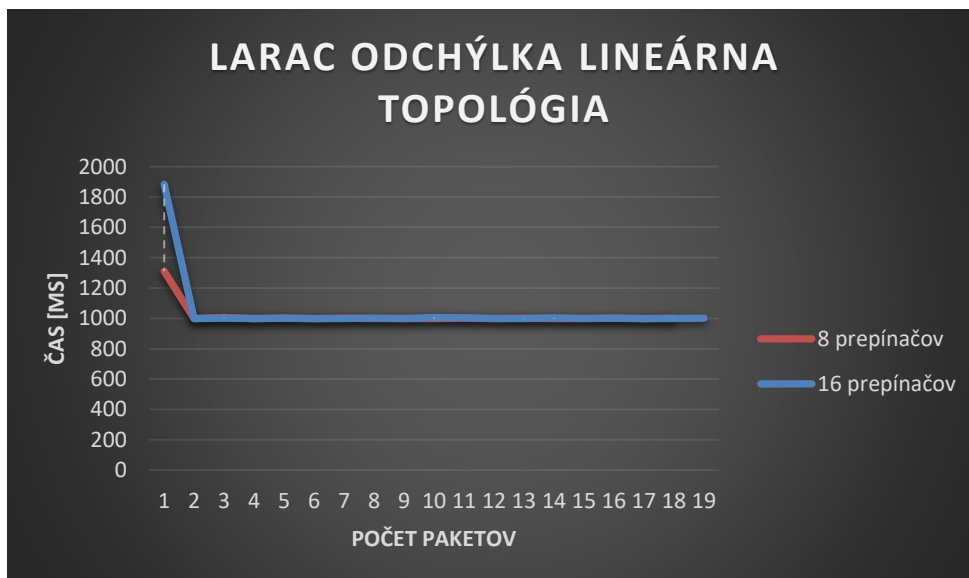
Kapitola je venovaná implementácii metódy LARAC. Celý algoritmus je implementovaný v súbore `path_finder.py`. Implementácia odpovedá z väčšej časti analýze LARAC metódy. Na nasledujúcom obrázku môžeme vidieť logiku algoritmu LARAC.



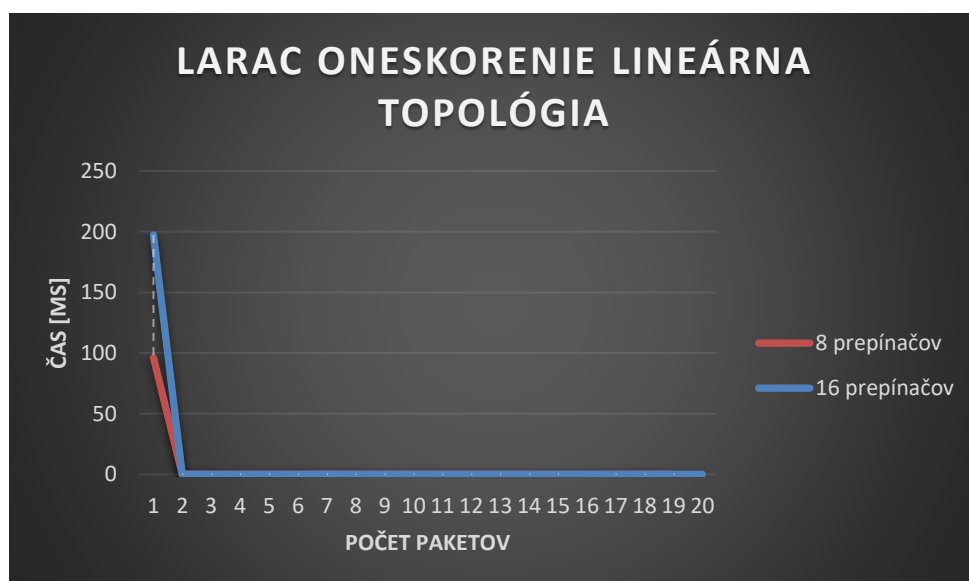
Obrázok 44. Diagram toku pre metódu LARAC

16.1. Testovanie metódy LARAC

Testovanie LARAC metódy na lineárnej topológii. Test pozostával len z pustenja pingu zo zariadení na jednom a druhom konci topológie.



Graf 21. Hodnota odchýlka v lineárnej topológii

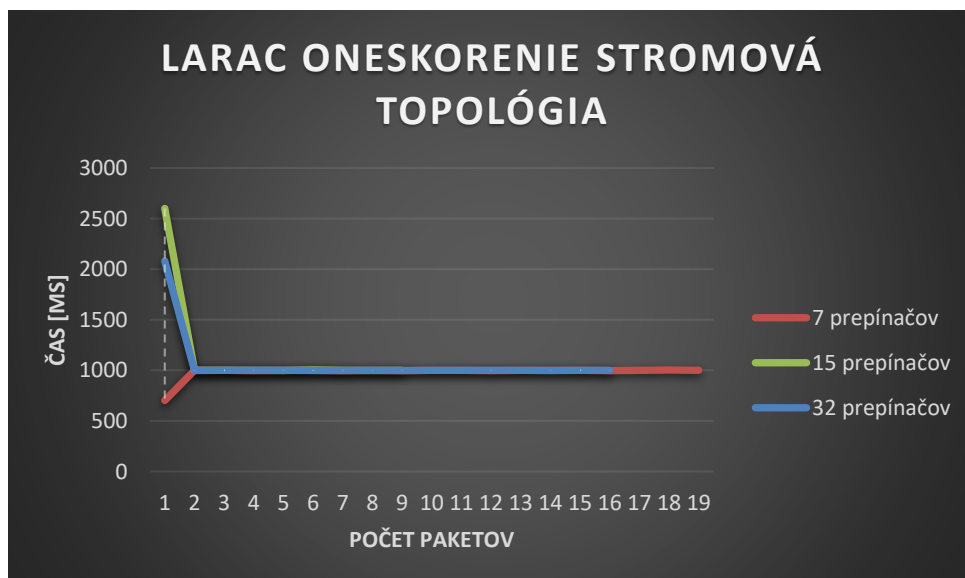


Graf 22. Hodnota oneskorenia v lineárnej topológii

Testovanie LARAC metódy na stromovej topológii. Test pozostával z pustenja pingu zo zariadení v uzloch čo najďalej od seba.



Graf 23. Hodnota odchýlka v stromovej topológii



Graf 24. Hodnota oneskorenia v stromovej topológii

Metóda bola testovaná aj pomocou piatich rôznych tokov. Využitá bola vlastná topológia o 14 linkách. Nástroj na jednotlivé simulácie bol použitý hping3 pre klienta a iperf pre server.

Jednotlivé toky, ktoré boli testované sú:

1. Prenos súborov

Protokol: TCP

Priepustnosť: 5,5Mbit/s

{'alpha': 0, 'beta': 0, 'gamma': 1, 'delay': 4.000, 'jitter': 0.01, 'thrput': 5500000}

Port: 5001

Veľkosť paketov: dynamicky dané TCP protokolom

Rýchlosť odosielania: dynamicky dané TCP protokolom

Odosielané dáta brané z existujúceho súboru (prepínač *-F /path/to/file.mp4*)

2. Skype hovor

Protokol: UDP (prepínač *-u*)

Priepustnosť: 100kbit/s

{'alpha': 1, 'beta': 0, 'gamma': 1, 'delay': 1.000, 'jitter': 0.01, 'thrput': 100000}

Port: 5002

Veľkosť paketov: 450B (prepínač *-l 450B*)

Rýchlosť odosielanie: 500 kbit/s (prepínač *-b 500k*)

3. Prenos videa

Protokol: UDP (prepínač *-u*)

Priepustnosť: 4,5Mbit/s

{'alpha': 1, 'beta': 0, 'gamma': 1, 'delay': 4.000, 'jitter': 0.01, 'thrput': 4500000}

Port: 5003

Veľkosť paketov: 1460B (prepínač *-l 1460B*)

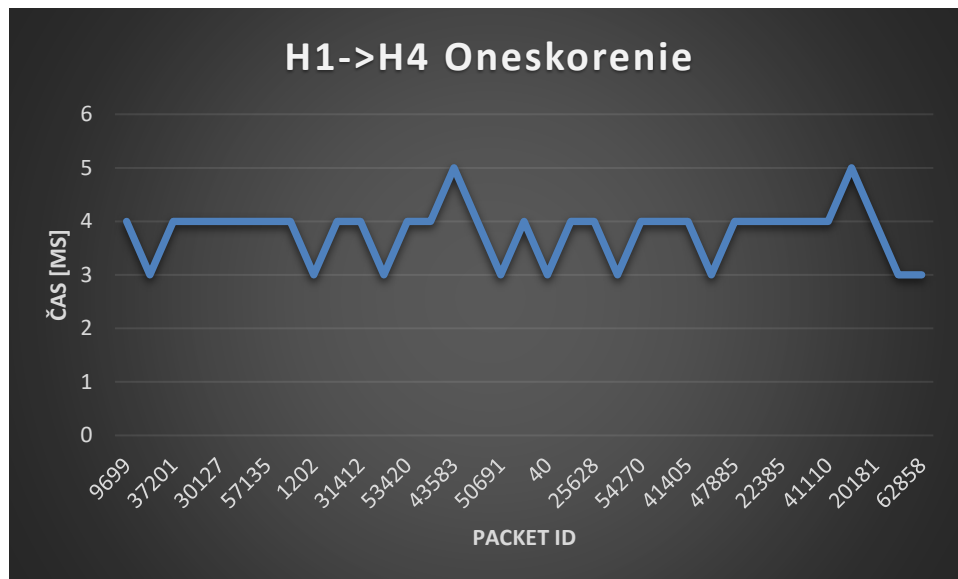
Rýchlosť odosielania: 4500 kbits/s (prepínač *-b 4500k*)

4. Manažment

Manažment predstavuje premávku ako ICMP, ARP a ostatnú IP komunikáciu, ktorá nespadá do ostatných troch kategórií tokov.

Toky:

1. H4 iperf -su -p 5002 -i 1
H1 hping3 10.0.0.4 -2 -V -p 5002 -d 450



Graf 25. oneskorenie



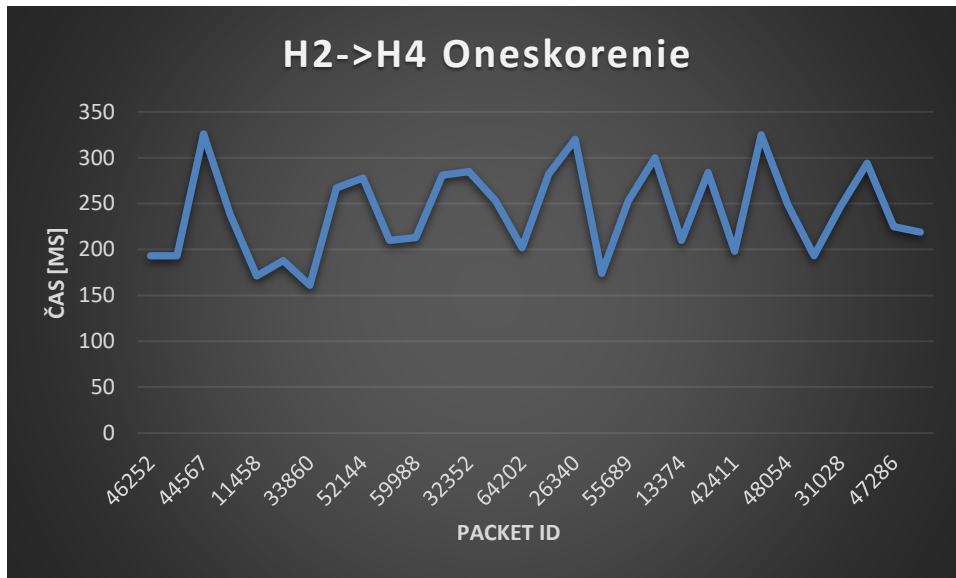
Graf 26. odchýlka

V prvom prípade sa testoval skype hovor (UDP komunikácia). Na grafe 25 a 26 možno vidieť oneskorenie a odchýlku v rámci toku medzi koncovými zariadeniami H1 a H4. Na grafe 25 možno vidieť oneskorenie jednotlivých paketov, ktorých maximum je 100 ms, pričom po konvergencii siete, sa oneskorenie konštantne drží na a pod 5 ms. Požadované maximálne oneskorenie bolo 1000 ms, čo daná metóda dodržala.

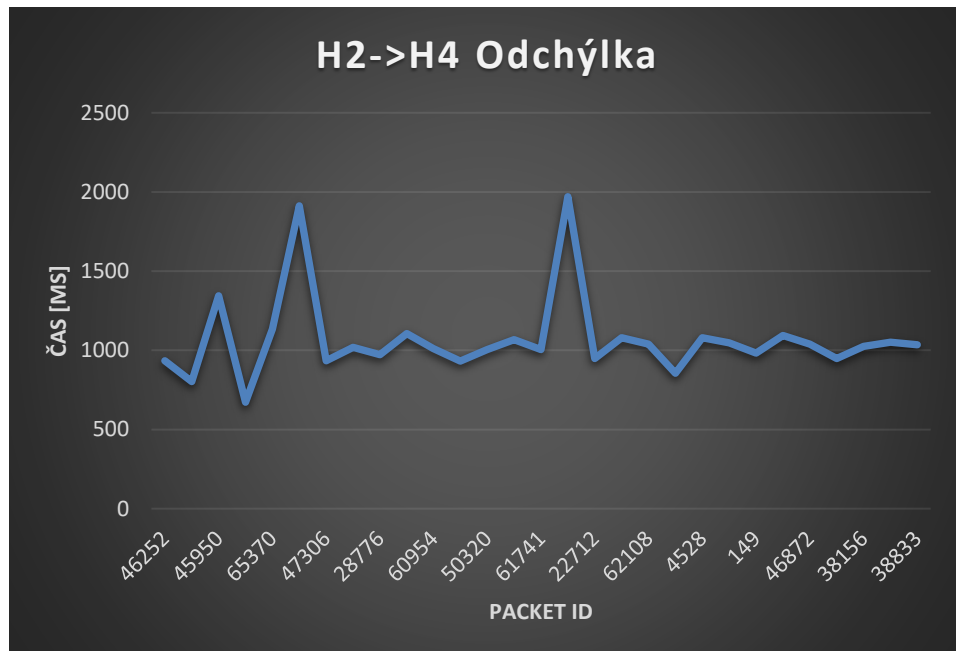
Na grafe dva možno vidieť odchýlku, kedy hping3 posiela pakety každú sekundu, z tohto dôvodu sa odchýlka drží okolo 1000 ms, pričom na konci je možné vidieť vyššie hodnoty.

2. H4 iperf -s -p 5001 -i 1

H2 hping3 10.0.0.4 -V -p 5001 -d 450 -E /home/peter/pythons/



Graf 27. oneskorenie



Graf 28. odchýlka

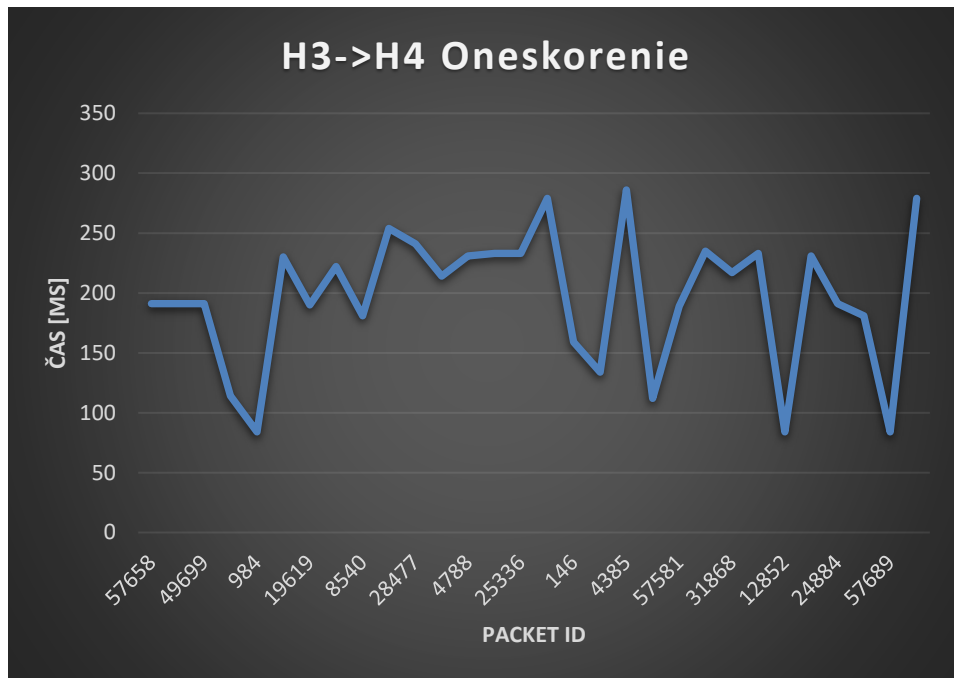
V druhom testovacom scenári sa skúmala TCP komunikácia, pri posielaní súborov z priečinka /home/peter/pythons/. V grafe 27 možno vidieť oneskorenie, kedy maximálne požadované

oneskorenie je 4000 ms. V tomto prípade metóda dodržala tento čas a jej maximálne oneskorenie bolo do 340 ms.

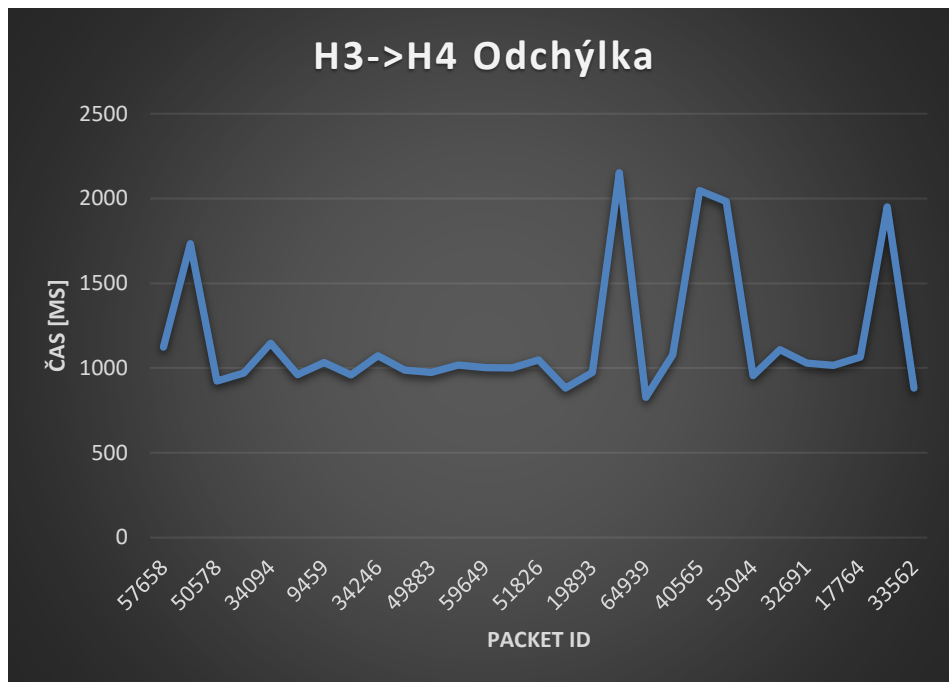
V prípade odchýlky na grafe 28, je možné vidieť „pohyblivosť“ okolo 1500 ms, kedy môžeme predpokladať, že maximálne požadovaná odchýlka 10 ms je prekročená.

3. H4 iperf -s -p 5001 -i 1

H3 hping3 10.0.0.4 -V -p 5001 -d 450 -E /home/peter/pythons/



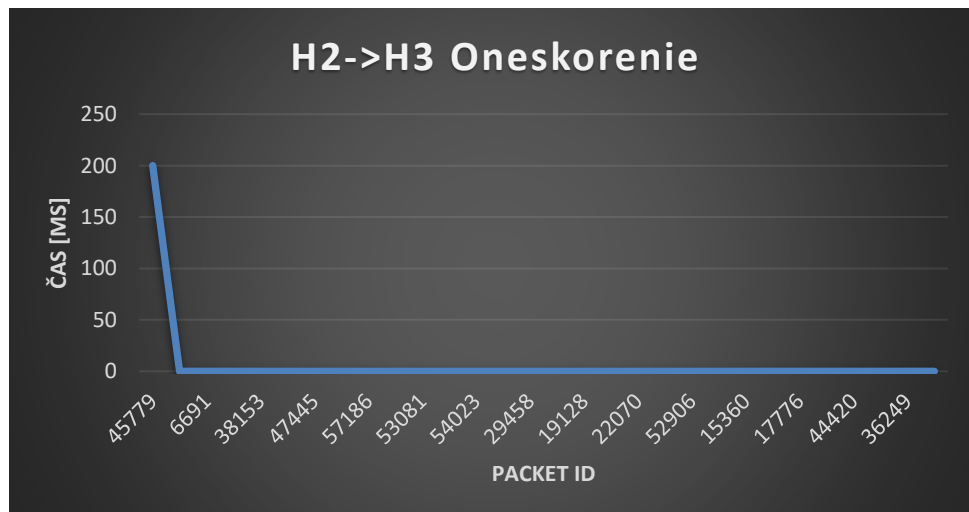
Graf 29. oneskorenie



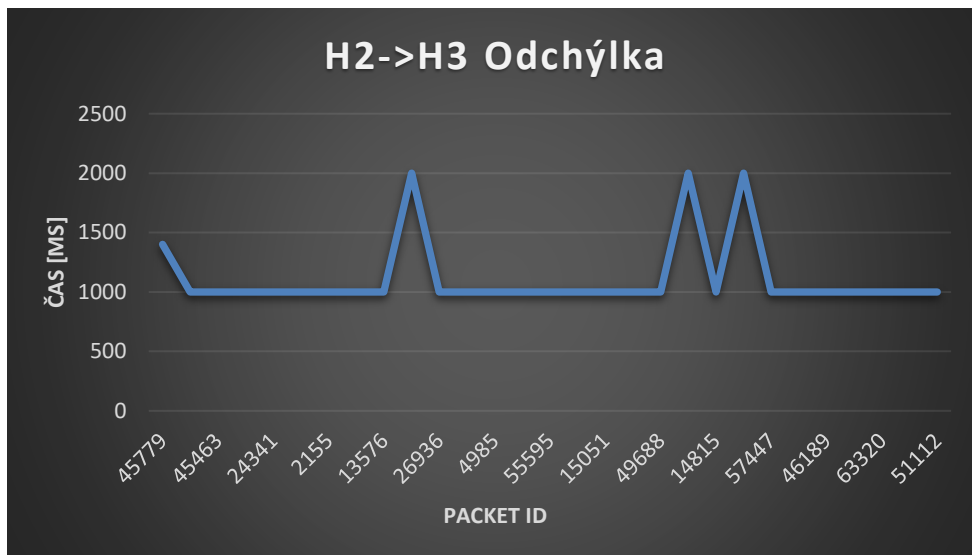
Graf 30. odchýlka

Aj v tomto prípade, bola riešená TCP komunikácia s prenosom súborov z priečinku. Aj v tomto prípade platí to isté, čo je spomenuté pri testovacom scenári 2.

4. H3 iperf -su -p 5002 -i 1
H2 hping3 10.0.0.3 -2 -V -p 5002 -d 450



Graf 31. oneskorenie

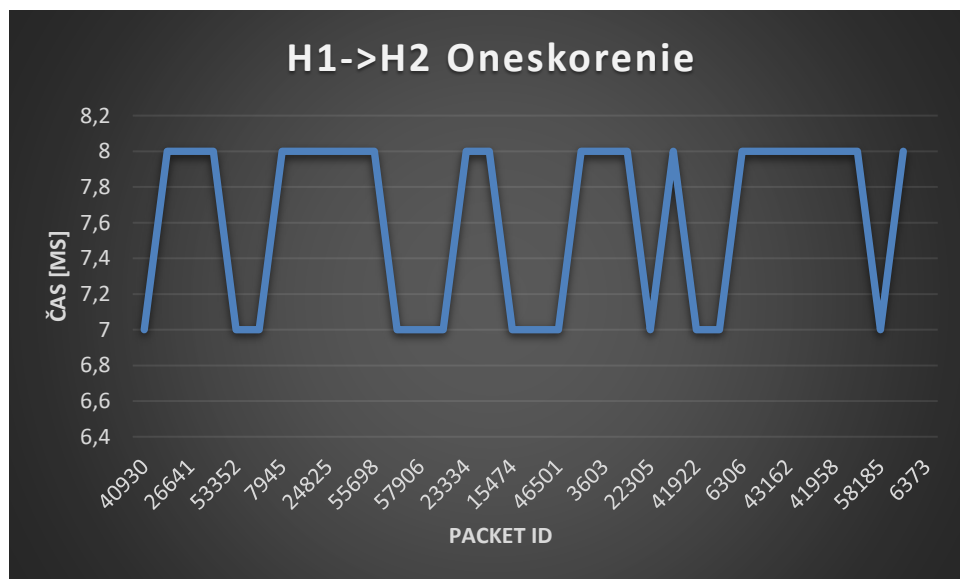


Graf 32. odchýlka

Meranie si môžeme všimnúť dopadlo veľmi dobre pre oneskorenie. Na začiatku nám vyskočila doba príjmu prvého paketu, ale ostatné majú oneskorenie okolo nuly, čo je veľmi dobrý čas.

5. H2 iperf -su -p 5003 -i 1

H1 hping3 10.0.0.2 -2 -V -p 5003 -d 1460



Graf 33. oneskorenie



Graf 34. odchýlka

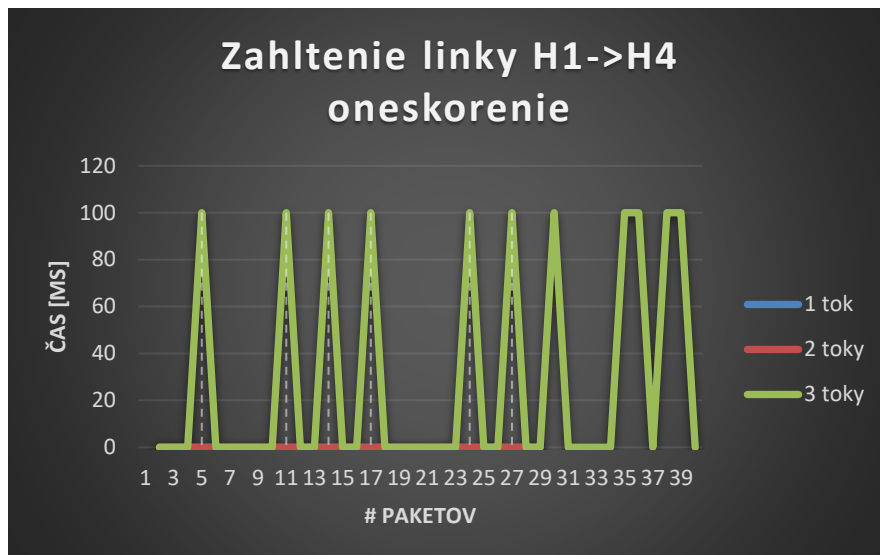
V rámci testovania sa testovali aj viaceré toky súbežne. V tomto prípade sme testovanie rozdelili na 3 hlavné testovacie scenáre.

4. Zahľtenie linky – kedy sme testovali, ako sa kontrolér správa pri zahľtení linky.

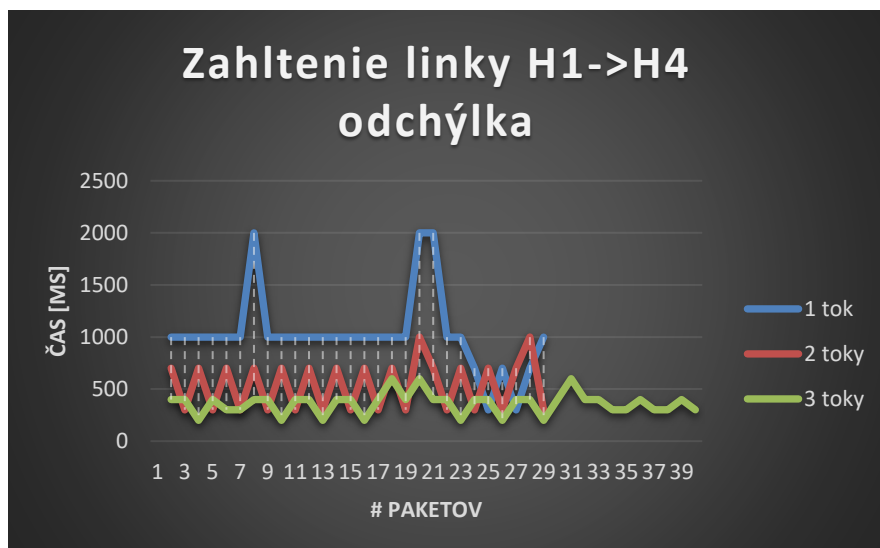
V rámci testovacieho scenára bolo potreba upraviť kapacitu linky, aby v prípade pridania tretieho toku nastal prípad, že linka už z hľadiska kapacity by nemala povoliť komunikáciu tretieho toku.

Avšak z technického dôvodu musela byť použitá topológia bez slučiek, kedy kontrolér, hoci vypíše nulovú kapacitu linky, tak nedokáže nájsť žiadnu inú zodpovedajúcu linku a tak ďalej púšťa komunikáciu rovnakou linkou, hoci nespĺňa kapacitu. Ako je možné vidieť na grafoch nižšie.

Oneskorenie aj po pridaní tretieho toku ostáva konštantne nad nulou. Avšak pri odchýlke je možno spozorovať väčšie množstvo anomálii, kedy by sa mala odchýlka udržiavať okolo 1000ms, ale je možné spozorovať aj vyššie či nižšie hodnoty.



Graf 35. zahltenie linky oneskorenie



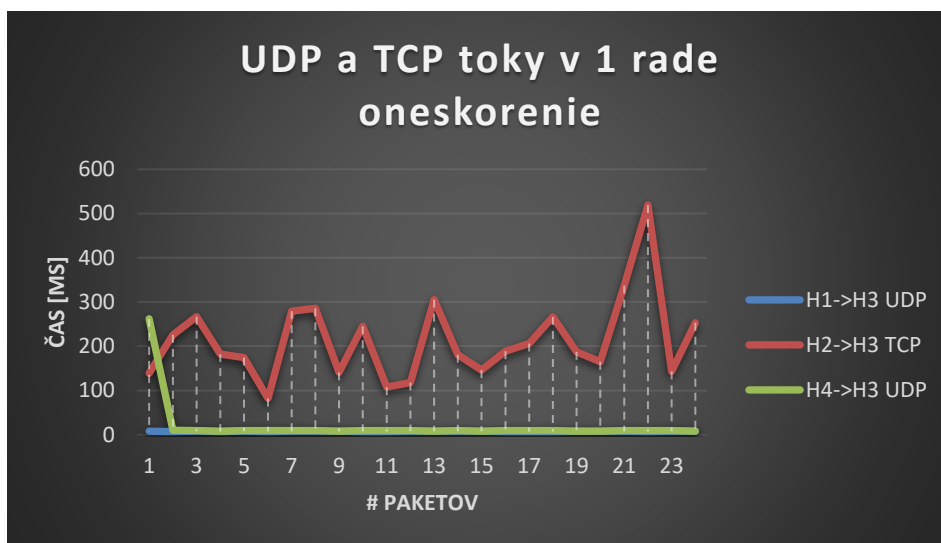
Graf 36. zahltenie linky odchylka

5. Pridanie TCP toku do UDP tokov – kedy sme testovali, či kontrolér na základe QoS parametrov uprednostní TCP tok.

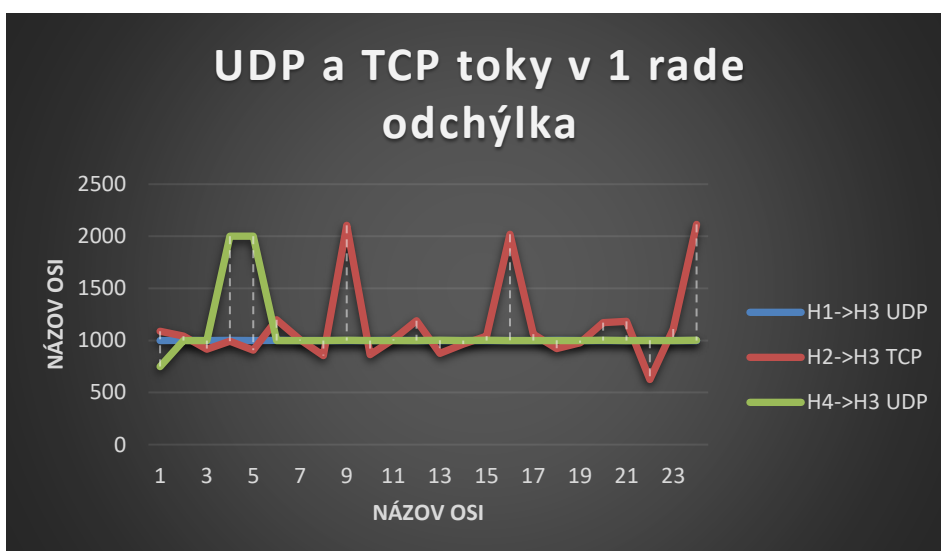
Kontrolér určuje priority jednotlivým radom a nie tokom. V tomto prípade sa testovalo v rámci jedného radu, kedy boli naraz spustené dva UDP toky a jeden TCP tok. V prípade viacerých radov, ak by boli rozdelené, že TCP tok je v jednom rade a UDP toky v druhom rade, kontrolér by mal uprednostniť TCP tok.

V prípade jedného radu, je možné spozorovať, že TCP tok je o niečo vyšší ako v prípade, keď bol TCP tok pustený sám na linke, kedy dosahoval pri oneskorení čas od 120ms do 180ms. Na nižšie zobrazenom grafe je možné vidieť, že oneskorenie sa pohybuje až o 350ms. Tiež je možné spozorovať, že UDP toky si stále udržiavajú konštantné oneskorenie okolo 0ms.

Na druhej strane, pri odchýlke je možné vidieť, že TCP tok na rozdiel od UDP tokov nejaví žiadne anomálie, kedy pri TCP toku je konštantná odchýlka.



Graf 37. pridanie TCP toku oneskorenie



Graf 38. pridanie TCP toku odchýlka

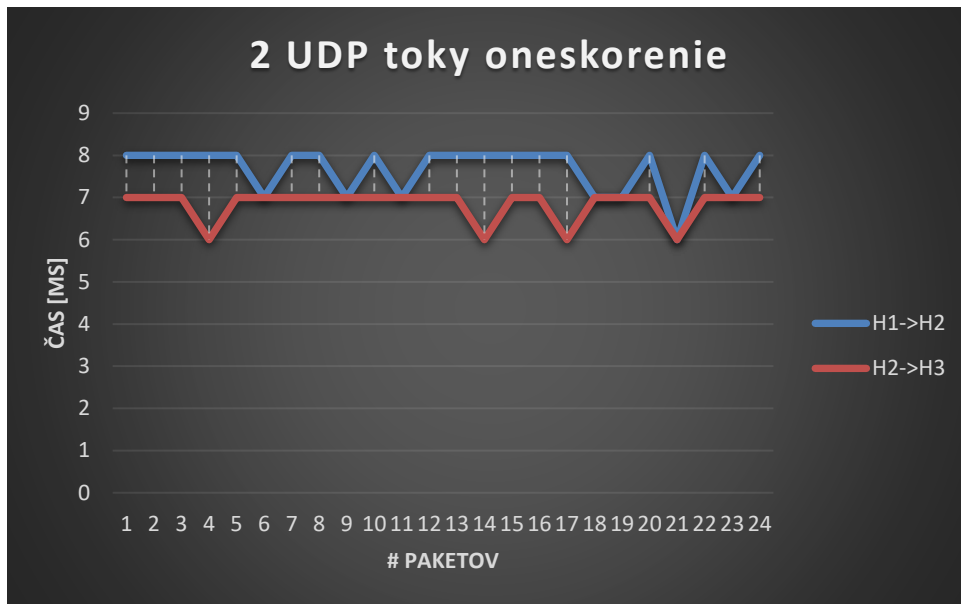
6. Viaceré UDP toky – kedy sme testovali, či pri náraste UDP tokov, budú jednotlivé toky stále spĺňať QoS parametre.

V rámci scenára sa testovali 2 rôzne typy UDP tokov – prenos videa a skype hovor.

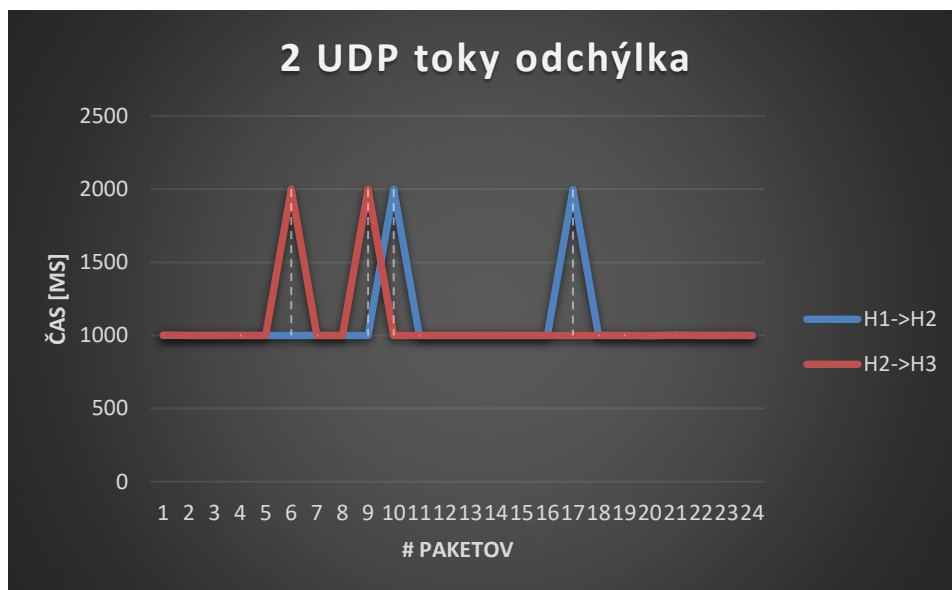
Testovalo sa nasledovne: zo začiatku boli spustené len 2 UDP toky, pri ktorých sme si uložili výsledky (možno vidieť na grafoch). Následne sme pridali tretí UDP tok a postup opakovali. Nakoniec sme pridali posledný (štvrtý) UDP tok a namerané výsledky uložili.

Na nižšie uvedených grafoch je možné vidieť oneskorenie a odchýlku od 2 UDP tokov po 4 UDP toky. Ako je možné spozorovať oneskorenie v rámci jednotlivých tokov nemá vplyv na ich

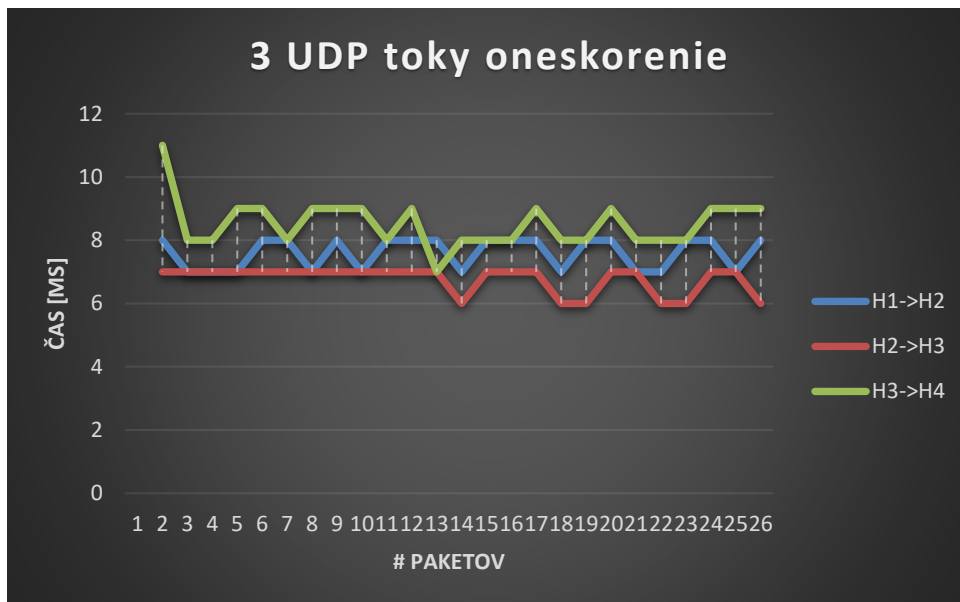
počet. Avšak nárast počtu tokov ovplyvňuje odchýlku, kedy sa vyskytuje s vyšším počtom tokov aj viac anomálií v rámci grafu (2000ms).



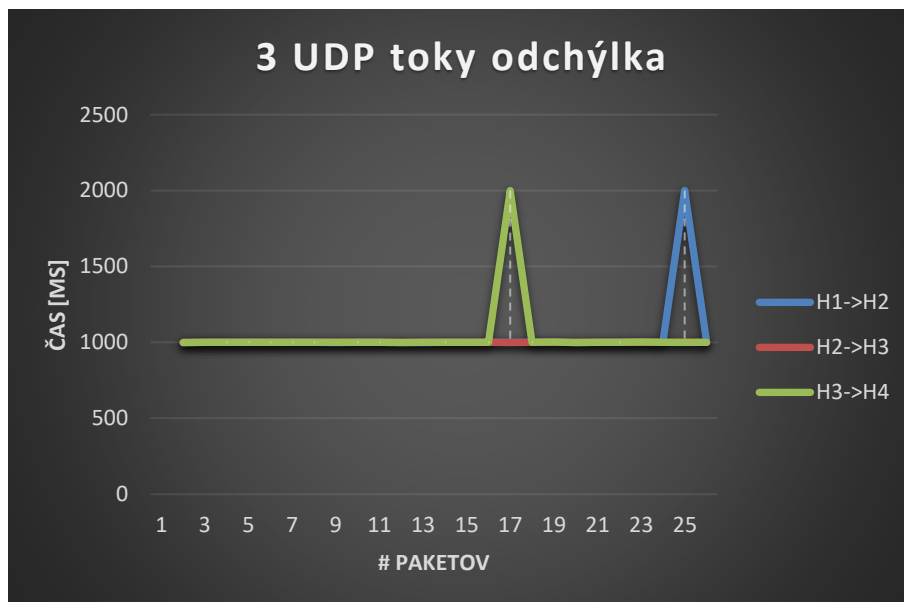
Graf 39. 2 UDP toky oneskorenie



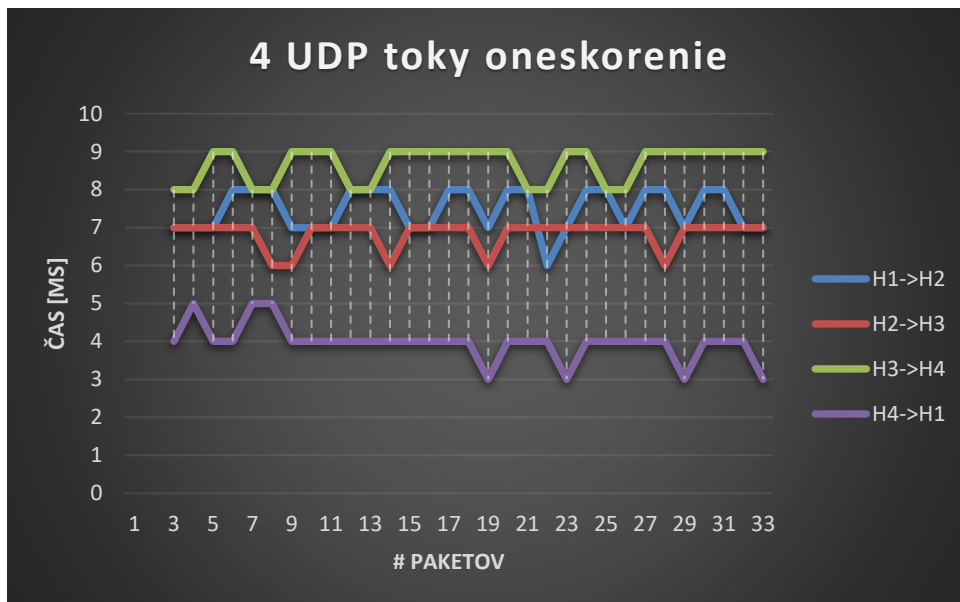
Graf 40. 2 UDP toky odchýlka



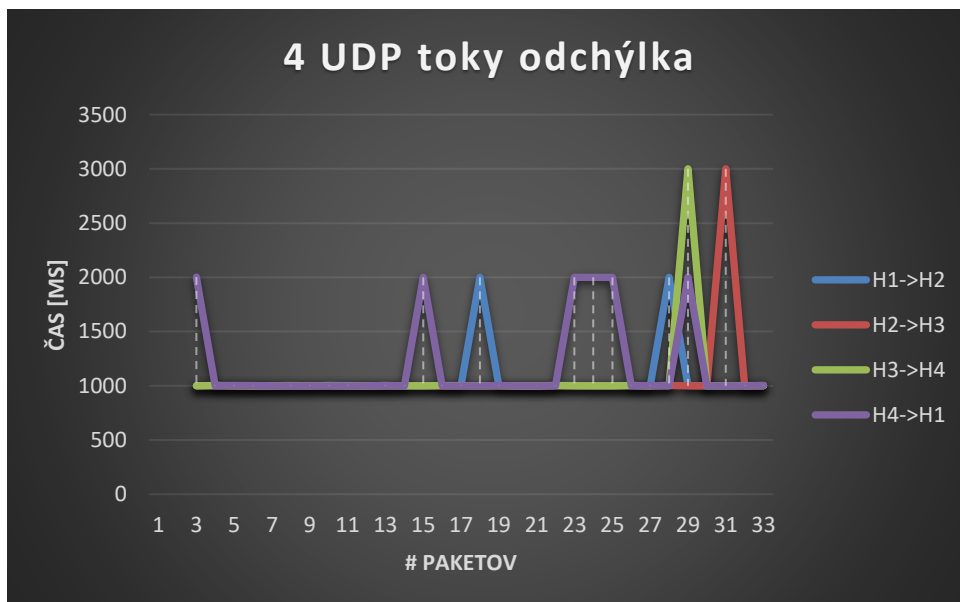
Graf 41. 3 UDP toky oneskorenie



Graf 42. 3 UDP toky odchýlka



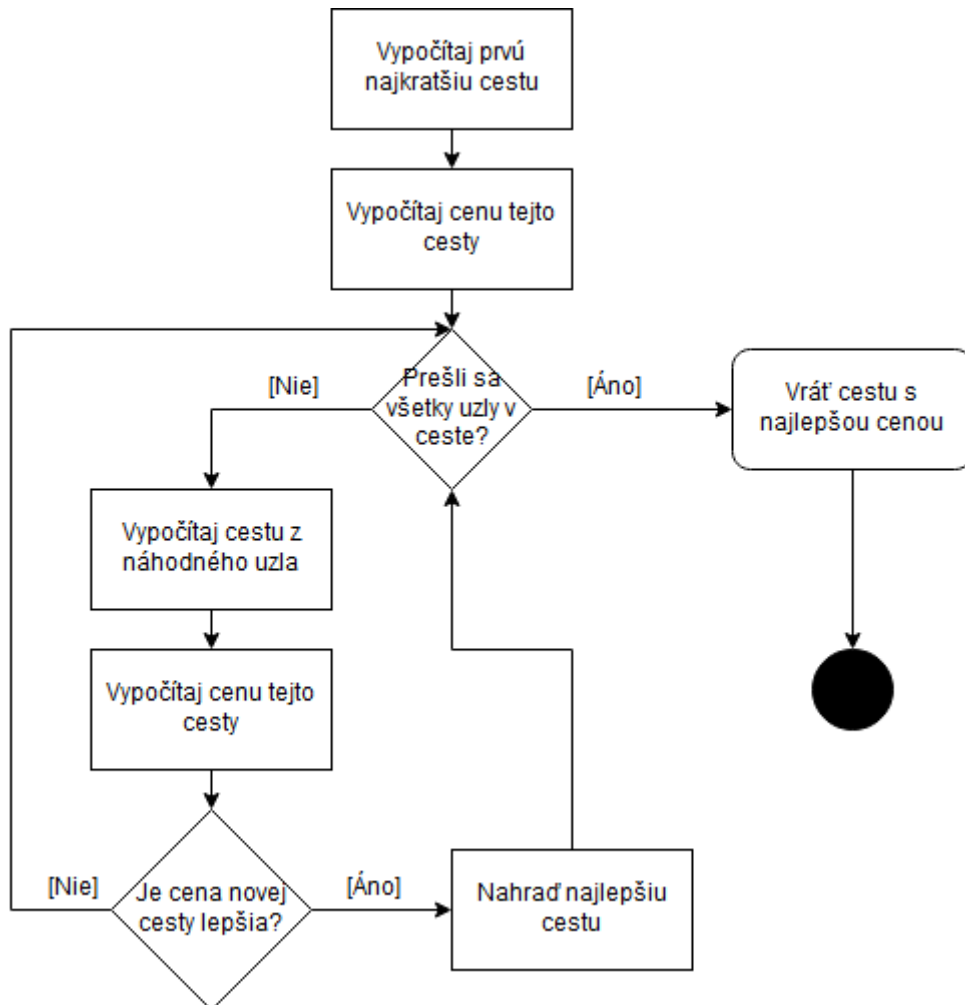
Graf 43. 4 UDP toky oneskorenie



Graf 44. 4 UDP toky odchýlka

17. Implementácia metódy SAQR

Kapitola je venovaná implementácii danej metódy a jednotlivým problémom. Metóda je implementovaná v súbore `path_finder.py`. Implementácia odpovedá z väčšej časti analýze SAQR metódy. Na nasledujúcom obrázku môžeme vidieť logiku algoritmu SAQR.



Obrázok 45. Flow diagram algoritmu SAQR

Pri implementácii algoritmu nevznikli žiadne problémy.

17.1. Testovanie metódy SAQR

Testovanie SAQR metódy na lineárnej topológii. Test pozostával len z pustenja pingu zo zariadení na jednom a druhom konci topológie.



Graf 45. Hodnota odchýlka v lineárnej topológii



Graf 46. Hodnota oneskorenia v lineárnej topológii

Testovanie SAQR metódy na stromovej topológii. Test pozostával z pustenja pingu zo zariadení v uzloch čo najďalej od seba.



Graf 47. Hodnota odchýlka v stromovej topológii



Graf 48. Hodnota oneskorenia v stromovej topológii

Metóda bola testovaná aj pomocou piatich rôznych tokov. Využitá bola vlastná topológia o 14 linkách. Nástroj na jednotlivé simulácie bol použitý *hping3* pre klienta a *iperf* pre server.

Jednotlivé toky, ktoré boli testované sú:

1. Prenos súborov

Protokol: TCP

Priepustnosť: 5,5Mbit/s

{'alpha': 0, 'beta': 0, 'gamma': 1, 'delay': 4.000, 'jitter': 0.01, 'thrput': 5500000}

Port: 5001

Veľkosť paketov: dynamicky dané TCP protokolom

Rýchlosť odosielania: dynamicky dané TCP protokolom

Odosielané dáta brané z existujúceho súboru (prepínač *-F /path/to/file.mp4*)

2. Skype hovor

Protokol: UDP (prepínač *-u*)

Priepustnosť: 100kbit/s

{'alpha': 1, 'beta': 0, 'gamma': 1, 'delay': 1.000, 'jitter': 0.01, 'thrput': 100000}

Port: 5002

Veľkosť paketov: 450B (prepínač *-l 450B*)

Rýchlosť odosielanie: 500 kbit/s (prepínač *-b 500k*)

3. Prenos videa

Protokol: UDP (prepínač *-u*)

Priepustnosť: 4,5Mbit/s

{'alpha': 1, 'beta': 0, 'gamma': 1, 'delay': 4.000, 'jitter': 0.01, 'thrput': 4500000}

Port: 5003

Veľkosť paketov: 1460B (prepínač *-l 1460B*)

Rýchlosť odosielania: 4500 kbits/s (prepínač *-b 4500k*)

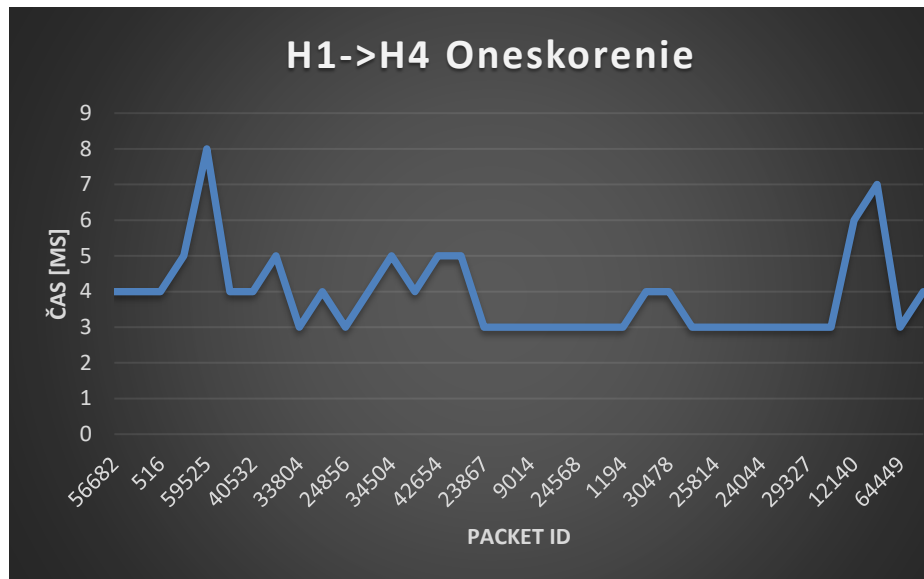
4. Manažment

Manažment predstavuje premávku ako ICMP, ARP a ostatnú IP komunikáciu, ktorá nespadá do ostatných troch kategórií tokov.

Toky:

1. H4 iperf *-su -p 5002 -i 1*

H1 hping3 10.0.0.4 *-2 -V -p 5002 -d 450*



Graf 49. oneskorenie



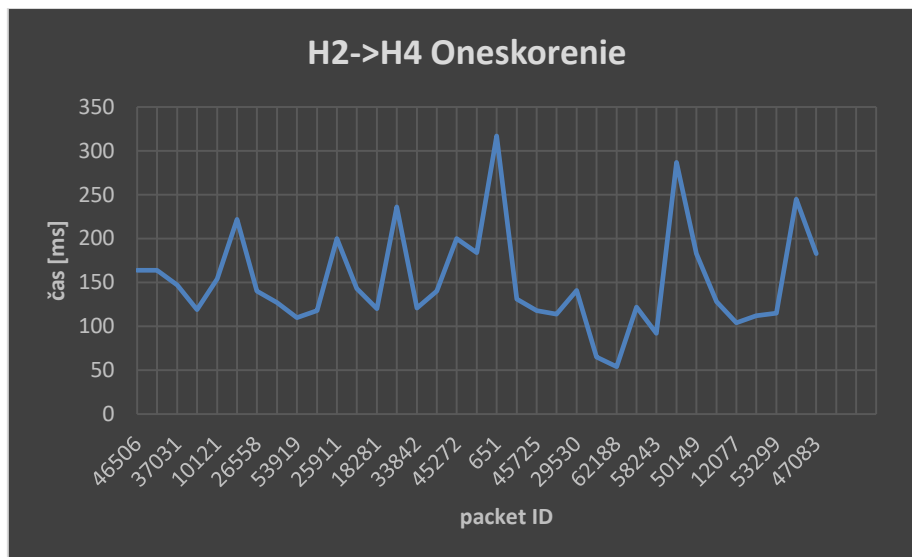
Graf 50. odchýlka

V prvom prípade sa testoval skype hovor (UDP komunikácia). Na grafe 49 a 50 možno vidieť oneskorenie a odchýlku v rámci toku medzi koncovými zariadeniami H1 a H4. Na grafe 49 možno vidieť oneskorenie jednotlivých paketov, ktorých maximum je 8 ms, pričom sa oneskorenie stabilne drží pod touto hodnotou. Požadované maximálne oneskorenie bolo 1000 ms, čo daná metóda dodržala.

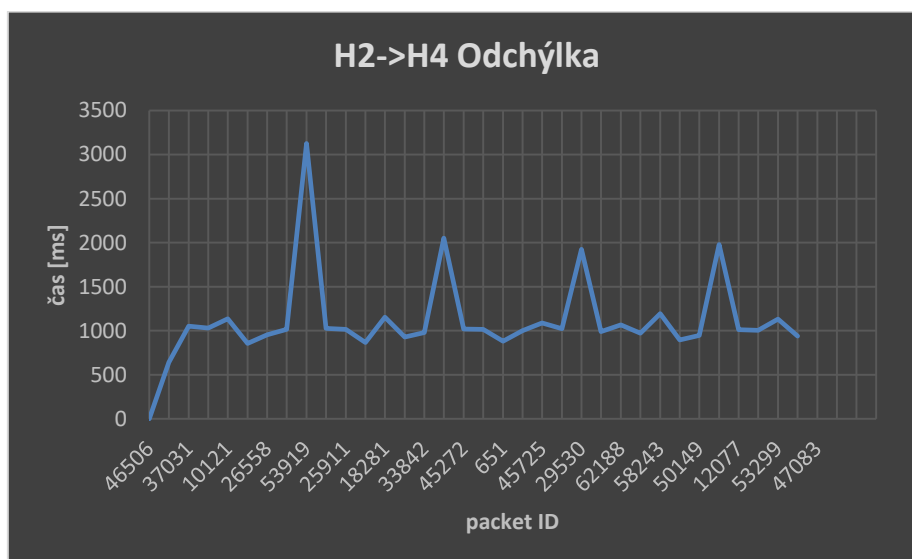
Na grafe 50 možno vidieť odchýlku, kedy *hping3* posielala pakety každú sekundu, z tohto dôvodu sa odchýlka drží okolo 1000 ms, pričom v strede testu je možné vidieť vyššie hodnoty.

2. H4 iperf -s -p 5001 -i 1

H2 hping3 10.0.0.4 -V -p 5001 -d 450 -E /home/peter/pythons/



Graf 51. oneskorenie



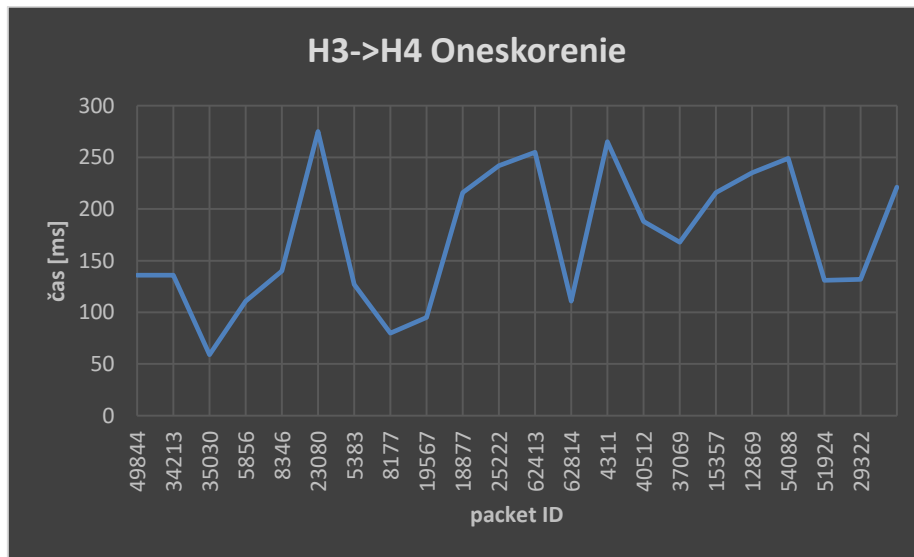
Graf 52. odchýlka

V druhom testovacom scenári sa skúmala TCP komunikácia, pri posielaní súborov z priečinka /home/peter/pythons/. V grafe 51 možno vidieť oneskorenie, kedy maximálne oneskorenie je až 330 ms, v tomto prípade metóda nedodržala požadované maximálne oneskorenie 200 ms. Z grafu a aj ostatných testovacích scenárov môžeme usúdiť, že táto metóda, nie je vhodná na TCP komunikáciu, čo môže byť spôsobené náhodným výberom hľadania lepšej cesty.

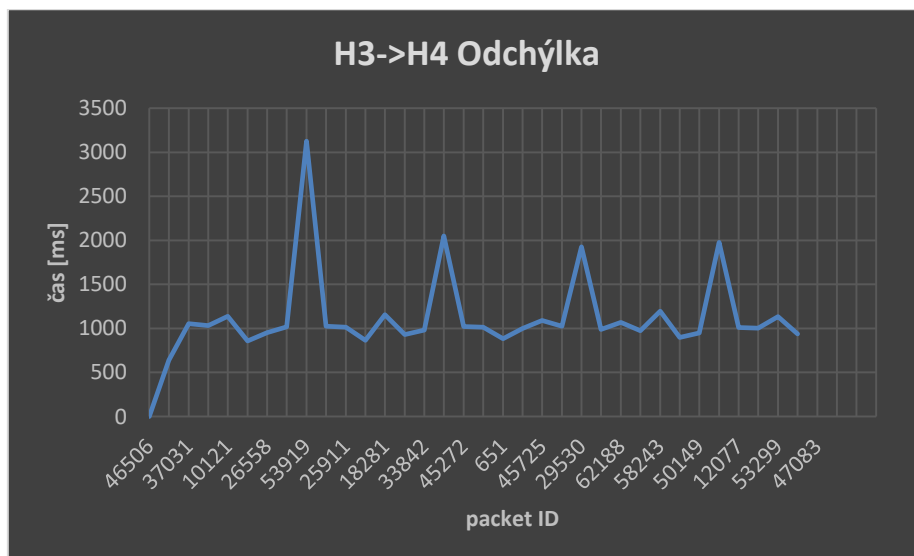
V prípade odchýlky na grafe 52, je možné vidieť „pohyblivosť“ okolo 1000 ms s väčšími, či menšími odchýlkami, kedy môžeme predpokladať, že maximálne požadovaná odchýlka 10 ms je prekročená.

3. H4 iperf -s -p 5001 -i 1

H3 hping3 10.0.0.4 -V -p 5001 -d 450 -E /home/peter/pythons/



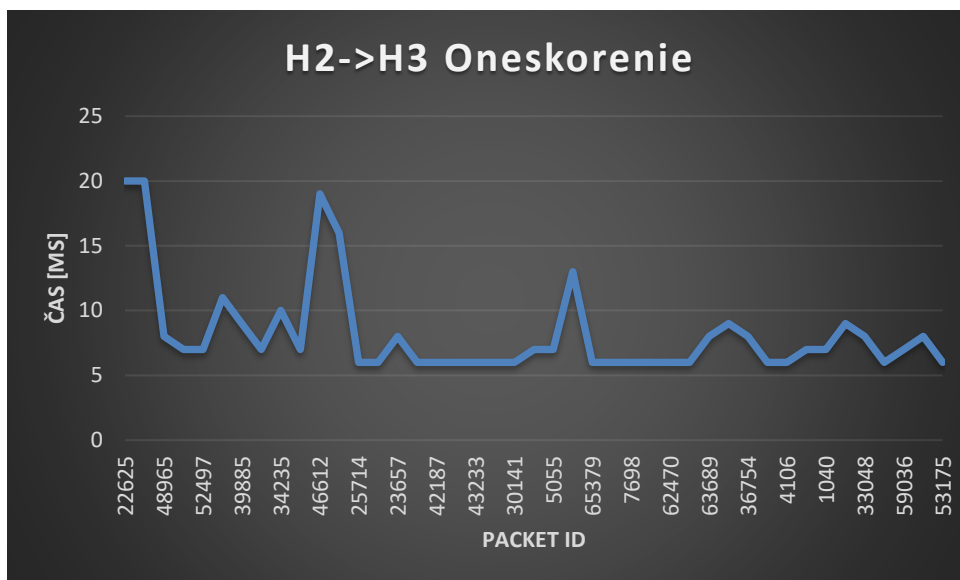
Graf 53. oneskorenie



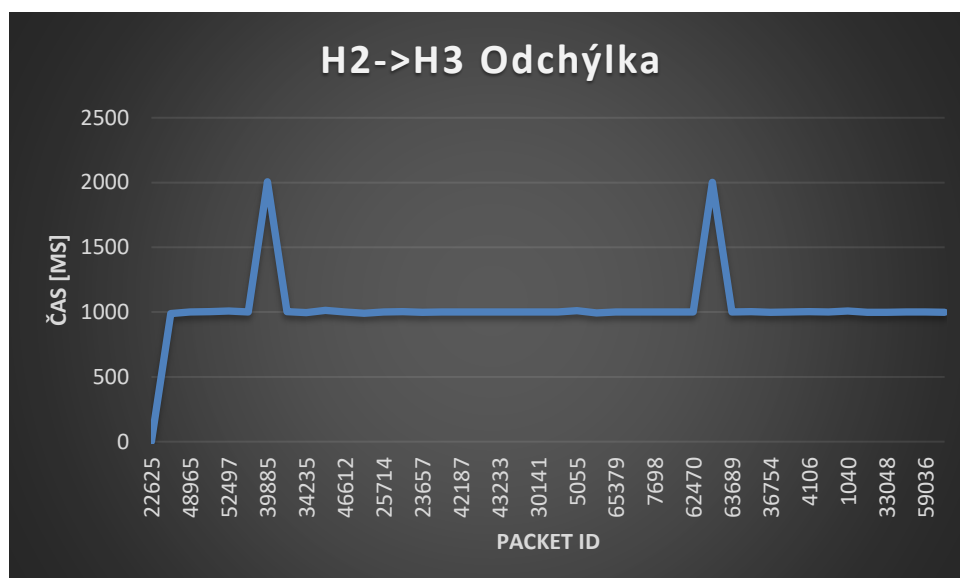
Graf 54. odchýlka

Aj v tomto prípade, bola riešená TCP komunikácia s prenosom súborov z priečinku. Aj v tomto prípade platí to isté, čo je spomenuté pri testovacom scenári 2.

- 4. H3 iperf -su -p 5002 -i 1
- H2 hping3 10.0.0.3 -2 -V -p 5002 -d 450



Graf 55. oneskorenie

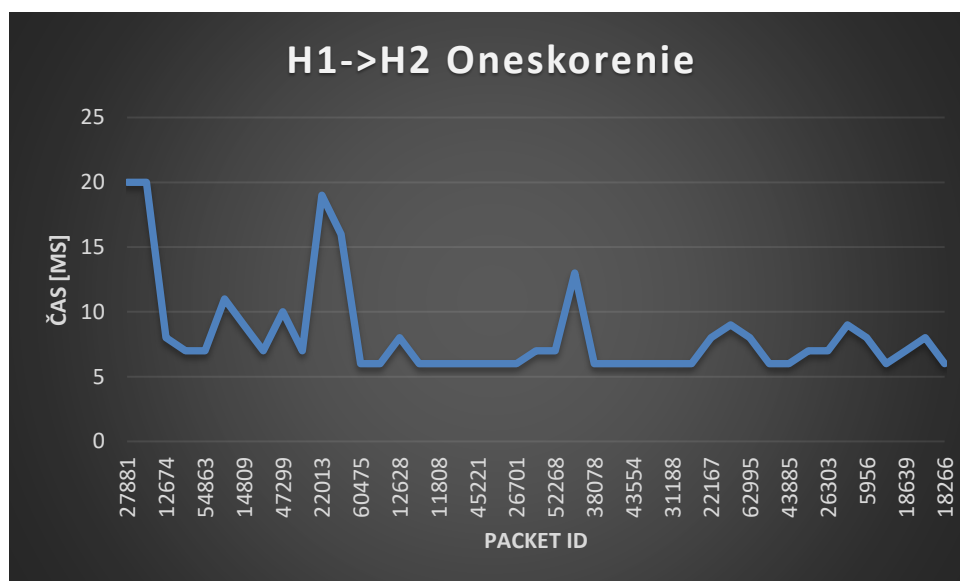


Graf 56. odchýlka

Hodnoty merania vo štvrtom testovacom scenári, sa približne zhodujú s meraniami v prvom testovacom, kedy v oboch prípadoch sa merala UDP komunikácia.

5. H2 iperf -su -p 5003 -i 1

H1 hping3 10.0.0.2 -2 -V -p 5003 -d 1460



Graf 57. oneskorenie



Graf 58. odchýlka

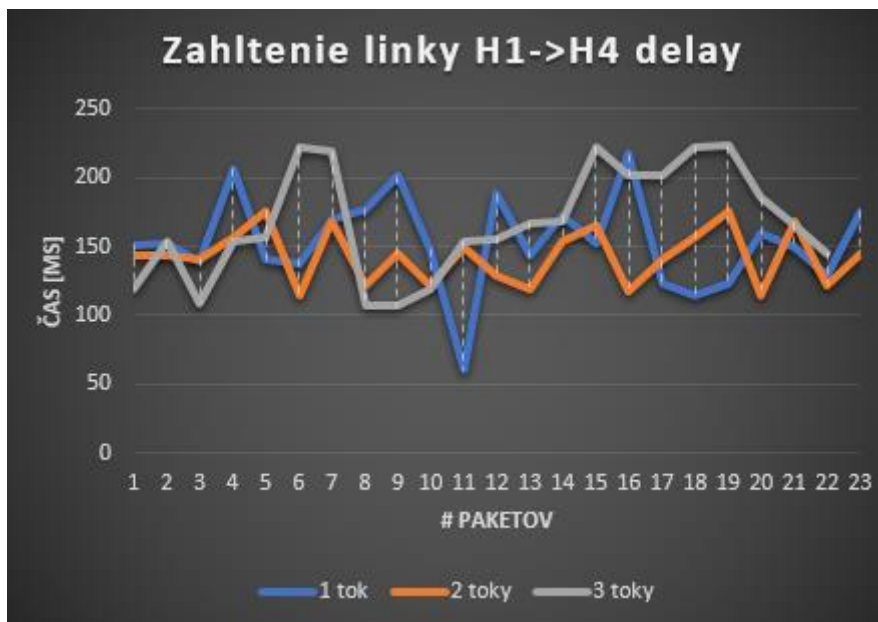
V poslednom testovacom scenári sa merala UDP komunikácia pre prenos videa. Aj v tomto prípade možno vidieť dodržanie maximálnych QoS parametrov.

V rámci testovania sa testovali aj viaceré toky súbežne. V tomto prípade sme testovanie rozdelili na 3 hlavné testovacie scenáre.

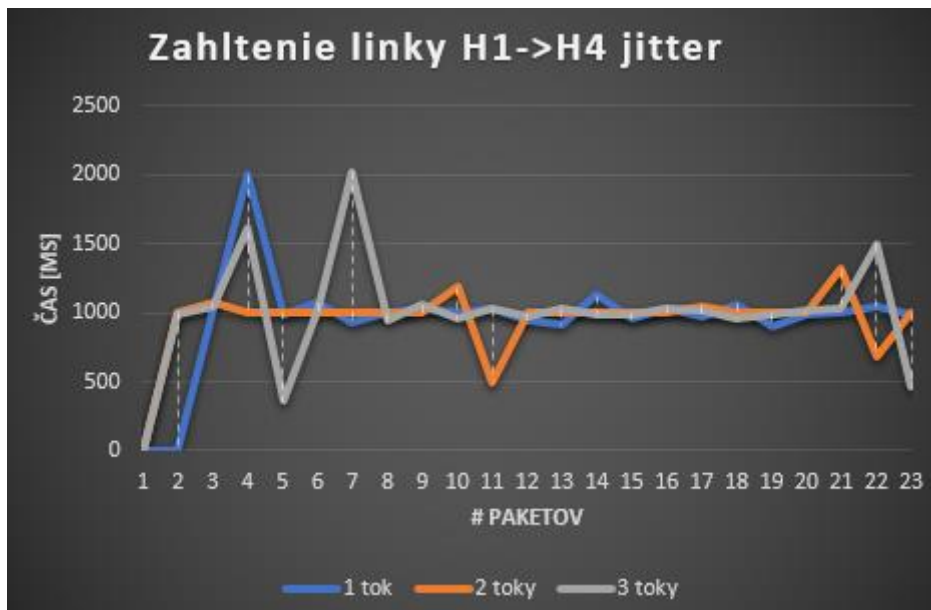
1. Zahľtenie linky – kedy sme testovali, ako sa kontrolér správa pri zahľtení linky.

V rámci testovacieho scenára bolo potreba upraviť kapacitu linky, aby v prípade pridania tretieho toku nastal prípad, že linka už z hľadiska kapacity by nemala povoliť komunikáciu tretieho toku.

Ako je možné vidieť na grafoch nižšie, oneskorenie po pridaní tretieho toku naďalej kolíše od 100 do 220ms. Avšak pri odchýlke je možné spozorovať väčšie množstvo anomálii, kedy by sa mala odchýlka udržiavať okolo 1000ms, ale je možné spozorovať aj vyššie či nižšie hodnoty.



Graf 59. zahľtenie linky oneskorenie



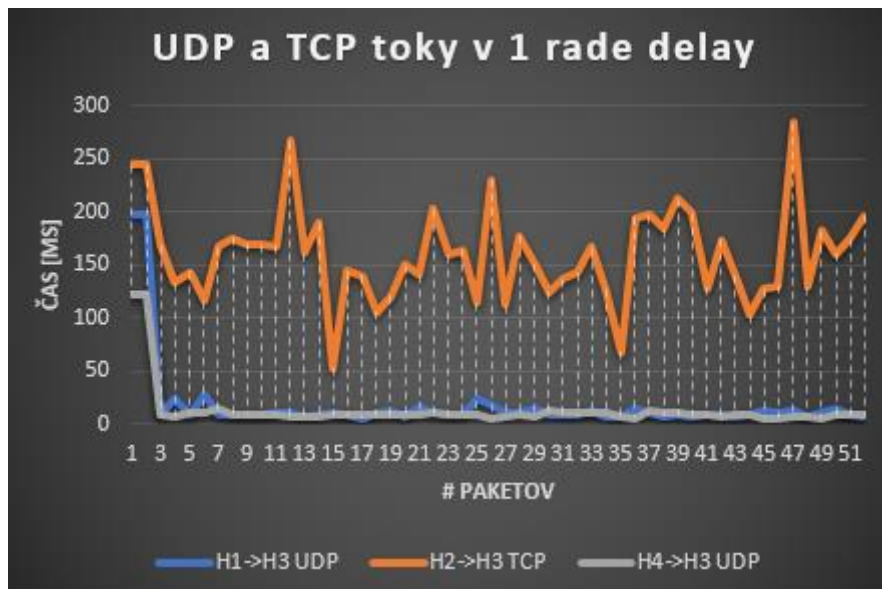
Graf 60. zahltenie linky odchýlka

2. Pridanie TCP toku do UDP tokov – kedy sme testovali, či kontrolér na základe QoS parametrov uprednostní TCP tok.

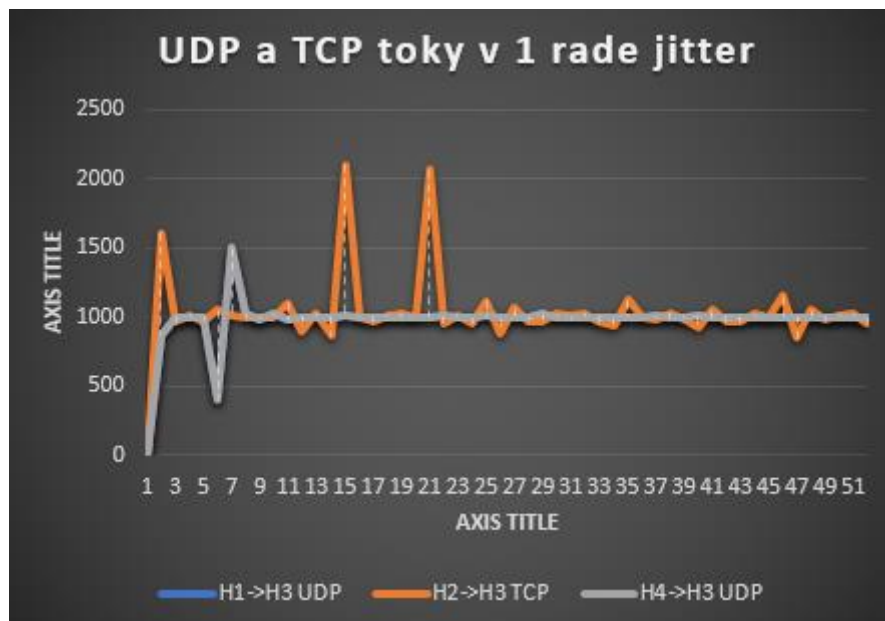
Kontrolér určuje priority jednotlivým radom a nie tokom. V tomto prípade sa testovalo v rámci jedného radu, kedy boli naraz spustené dva UDP toky a jeden TCP tok. V prípade viacerých radov, ak by boli rozdelené, že TCP tok je v jednom rade a UDP toky v druhom rade, kontrolér by mal uprednostniť TCP tok.

V prípade jedného radu, je možné spozorovať, že TCP tok má vyššie oneskorenie ako UDP toky. Ako už bolo spomínané vyššie vo všetkých testovaných scenároch dávala táto metóda nedostatočné výsledky ohľadom TCP tokov. Na nižšie zobrazenom grafe je možné vidieť, že oneskorenie sa pohybuje až do 300ms. Tiež je možné spozorovať, že UDP toky si stále udržiavajú konštantné oneskorenie okolo 0ms.

Pri odchýlke je možné vidieť, že UDP toky narozdiel od TCP toku nejavia žiadne anomálie, kedy pri TCP toku je odchýlka v dvoch prípadoch až 2000ms.



Graf 61. pridanie TCP toku oneskorenie



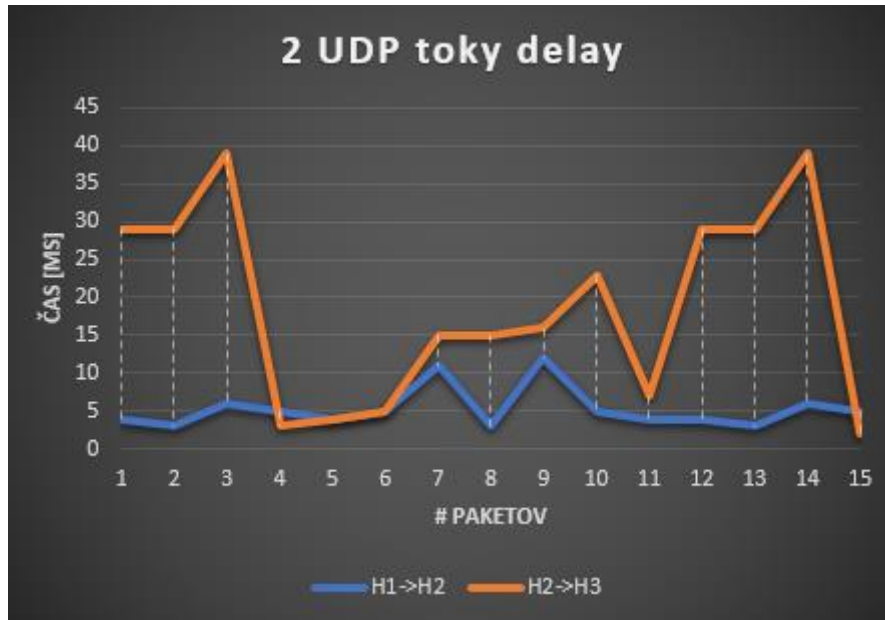
Graf 62. pridanie TCP toku odchýlka

3. Viaceré UDP toky – kedy sme testovali, či pri náraste UDP tokov, budú jednotlivé toky stále spĺňať QoS parametre.

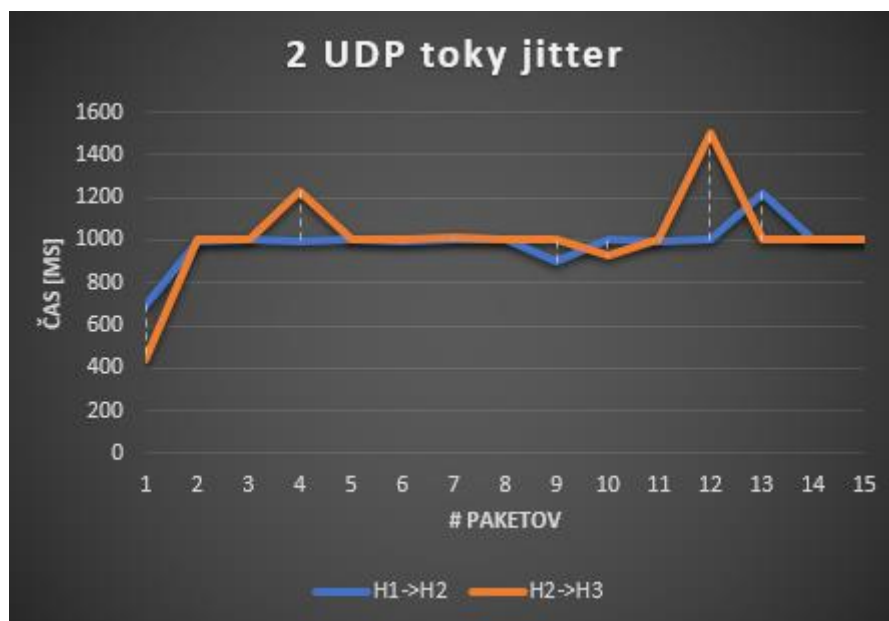
V rámci scenára sa testovali 2 rôzne typy UDP tokov – prenos videa a skype hovor.

Testovalo sa nasledovne: zo začiatku boli spustené len 2 UDP toky, pri ktorých sme si uložili výsledky (možno vidieť na grafoch). Následne sme pridali tretí UDP tok a postup opakovali. Nakoniec sme pridali posledný (štvrtý) UDP tok a namerané výsledky uložili.

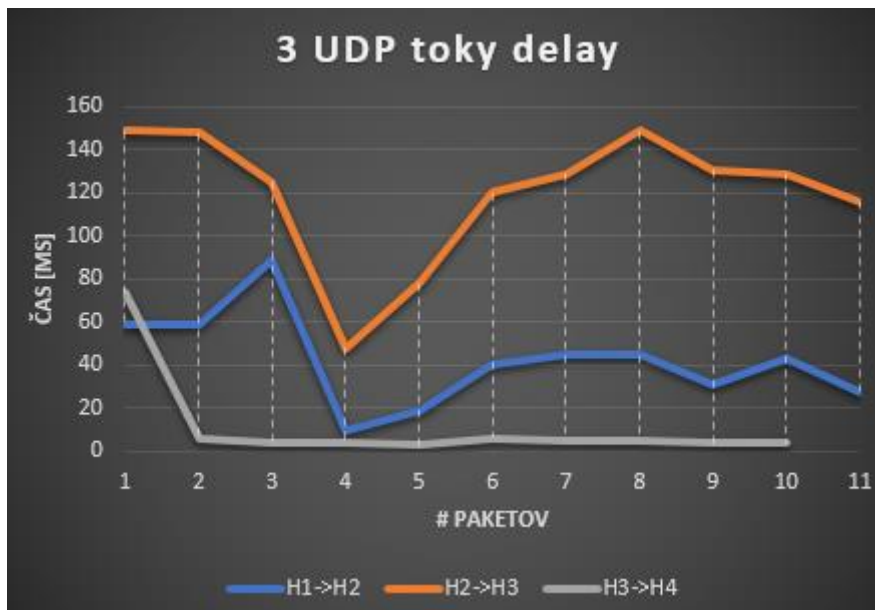
Na nižšie uvedených grafoch je možné vidieť oneskorenie a odchýlku od 2 UDP tokov po 4 UDP toky. Ako je možné spozorovať oneskorenie v rámci jednotlivých tokov má vplyv na ich počet len minimálne. Avšak nárast počtu tokov ovplyvňuje odchýlku, kedy sa vyskytuje s vyšším počtom tokov aj viac anomálii v rámci grafu (2000ms).



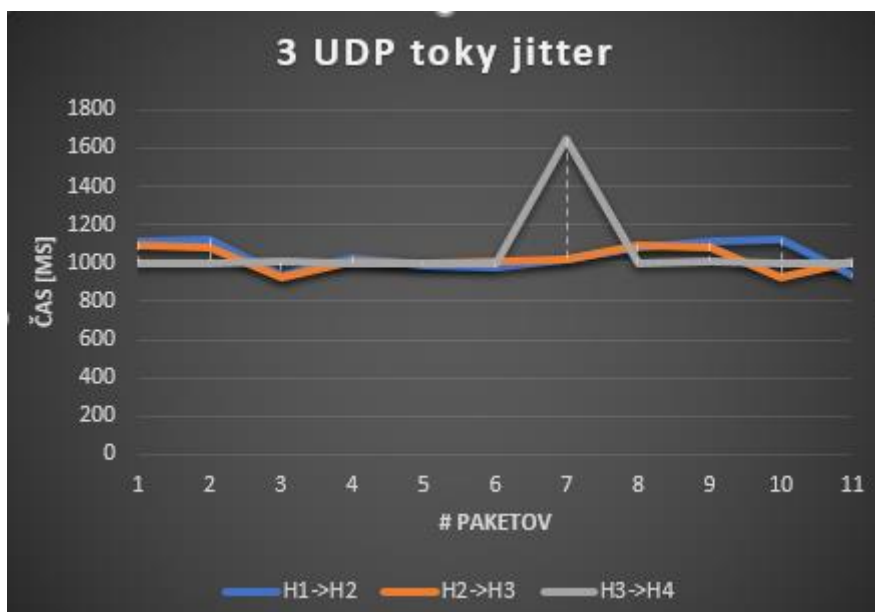
Graf 63. 2 UDP toky oneskorenie



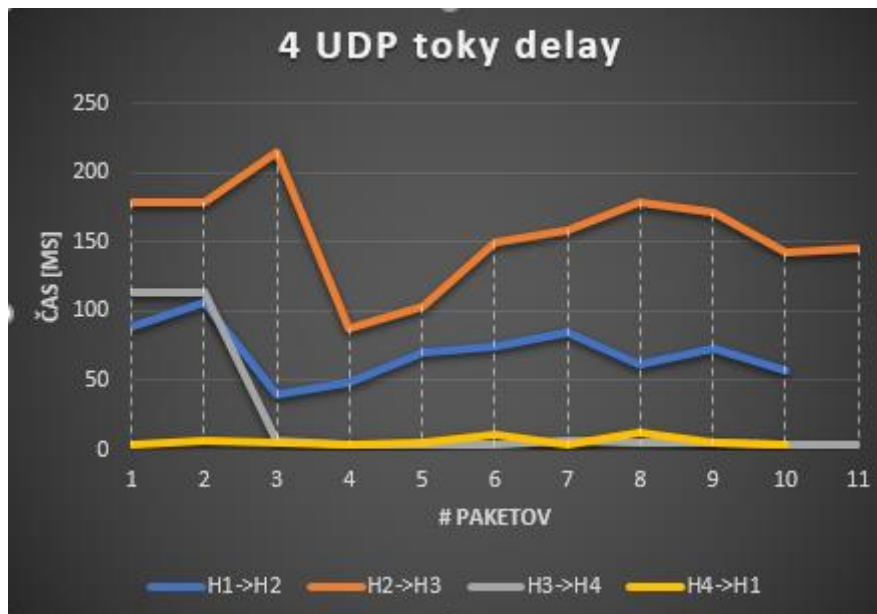
Graf 64. 2 UDP toky odchýlka



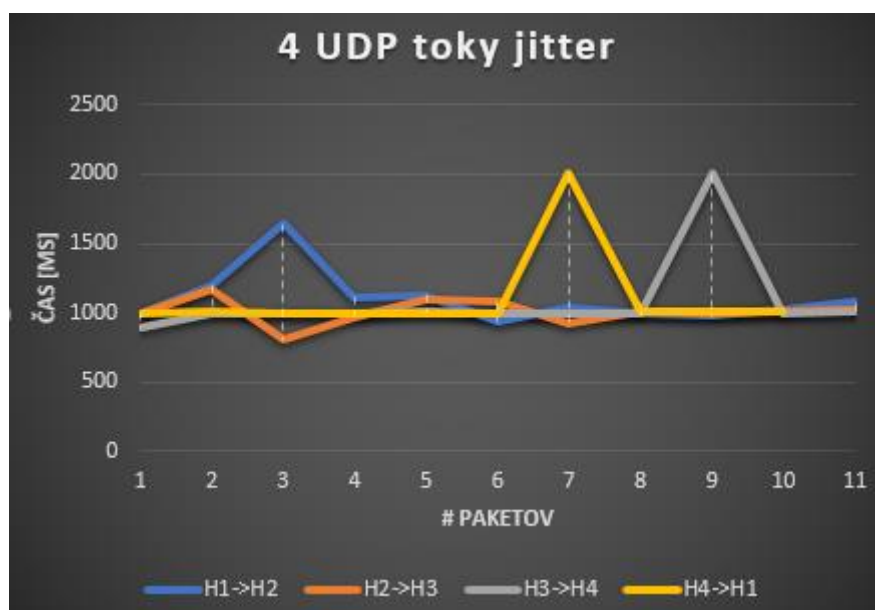
Graf 65. 3 UDP toky oneskorenie



Graf 66. 3 UDP toky odchylka



Graf 67. 4 UDP toky oneskorenie



Graf 68. 4 UDP toky odchylka

18. LLDP problém

Pri testovaní nami implementovaných metód sme objavili problém pri spracovávaní tokov. Problém spočíva v tom, že pri slučkových topológiách sa hromadia pakety v kontrolérovi, kedy nestíha spracovávať pakety, ktoré sa automaticky generujú a tak pakety, ktoré vznikajú nami vygenerovaných tokov nie sú spracovávané a na spracovanie čakáme aj niekoľko minút.

Z počiatku sme považovali problém pri STP protokolu, kedy bez STP protokolu nám vznikajú slučky alebo s STP protokolom nám vzniká len jedna cesta, avšak STP pakety zahlcovali kontrolér. Po bližšej analýze problému, sme zistili, že problém sa nachádza u LLDP protokolu a nie STP.

LLDP v pravidelných intervaloch posiela na kontrolér informácie o topológií, a tým pádom kontrolér zahlcuje. Avšak tento problém so zahltením kontroléra je najmä v slučkových topológiách, v topológiách bez slučiek sme nezaznamenali tento problém. Avšak tento jav sa nám nepodarilo objasniť.

18.1 LLDP protokol

LLDP (Link Layer Discovery Protocol) je protokol linkovej vrstvy, ktorý preposiela informácie o topológii. Ide o jednostranný protokol, ktorý iba vysiela informácie o nedochádza k žiadnemu potvrdeniu prijatej informácie alebo k nadviazaniu spojenia. Aktívny prvok odosiela cez svoje porty informácie o sebe ostatným LLDP zariadeniam v sieti. K odosielaniu dochádza periodicky alebo pri zmene aktívneho prvku. [42]

K šíreniu LLDP oznámenia sa používa multicast. Túto správu prijímajú najbližší susedia spolu s kontrolérom, informácie spracujú a ďalej neposielajú. Pomocou LLDP je teda možné zistiť iba priamo pripojených susedov. Keďže ale na kontrolér sa posielajú LLDP správy zo všetkých aktívnych prvkov, tak kontrolér vie zostaviť celú topológiu. [42]

Keď je aktívny prvok pripojený na kontrolér, kontrolér periodicky žiada všetky pripojené aktívne prvky o posielanie LLDP správ. [43]

LLDP paket vždy obsahuje [42]:

- Chassis ID
- Port ID
- TTL (Time to live) – po akú dobu si odosielajúci prvok myslí, že budú informácie platné.

- End of lldpdu

Ďalšie nepovinné informácie, ktoré môžu byť posielané, sú špecifikované v IEEE norme. Zariadenia často posielajú informácie aj o VLAN ID, VLAN name, Link aggregaci. V preddefinovanom stave aktívny prvok odosiela informácie na všetky svoje porty. Konfiguráciou je možné ovplyvniť čas odosielanie, vypršanie, oznámenie na všetkých portoch alebo aj samotné posielanie LLDP správ. [42]

18.2 Riešenie

Ako riešenie by sme navrhli, úplne zrušiť LLDP správy po naučení sa topológie. Keďže máme v metódach statický prvok (maximálny počet liniek v topológii) nemožno ju dynamicky meniť (za chodu odstraňovať a pridávať linky). Z tohto dôvodu nie je potreba LLDP správy stále posielat'. Preto po naučení sa liniek, kontrolér pošle všetkým aktívnym prvkom správu o zahadzovaní LLDP paketov.

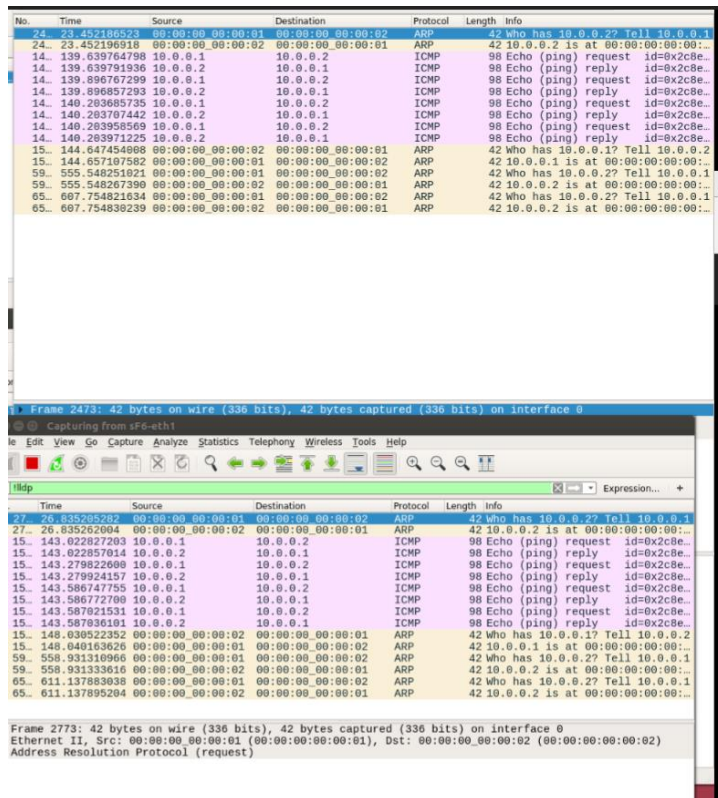
V rámci testovania sme už manuálne zrušili LLDP preposielanie pomocou príkazu:

```
sh ovs-ofctl add-flow sxx priority=65535, dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc,actions=drop
```

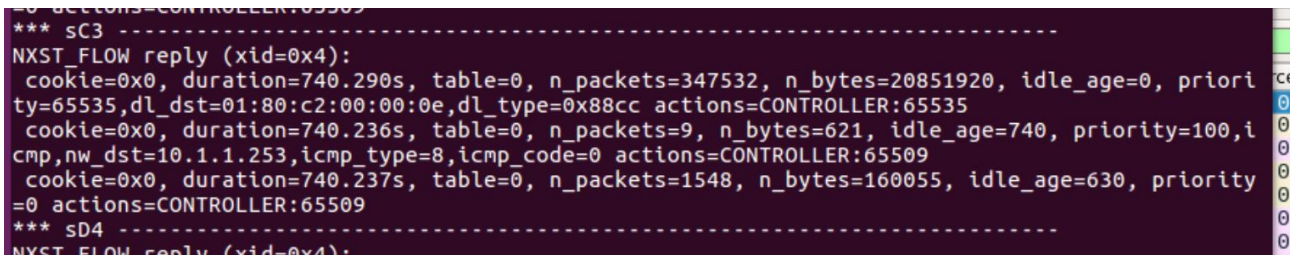
Po zadaní príkazu a následnom spustení toku, kontrolér v pomerne rýchlom čase našiel cestu a spustila sa komunikácia.

Dané riešenie je možné zostrojiť iba v prípade, že s topológiou sa nebude už po naučení liniek, nijako manipulovať. V prípade manipulácie (zrušení linky), kontrolér nebude schopný zistiť danú zmenu.

V prípade menenia topológie aj po naučení liniek, je potreba iné riešenie. V tomto prípade navrhujeme len upraviť čas odosielania LLDP paketov, kedy zo zopár ms to budú sekundy a možno dokonca minúty. (kontrolér každú minúty zažiada o LLDP správy). Avšak v tomto prípade navrhujeme počkať so spúšťaním iných tokov, dokým sa topológia neustáli a kontrolér sa nenaučí všetky aktuálne informácie. Tento prípad z vyššie uvedených dôvodov (statické zisťovanie maximálneho počtu liniek) sme neskúšali a tak nevieme overiť jeho funkcionality.



Obrázok 46. komunikácia pri spustenom LLDP



Obrázok 47. OpenFlow tabuľka pri spustenom LLDP

```
mininet> sh ovs-ofctl add-flow sF6 priority=65535,d_l_dst=01:80:c2:00:00:0e,d_l_type=0x88cc,actio
ns=drop
mininet> sh ovs-ofctl add-flow sE5 priority=65535,d_l_dst=01:80:c2:00:00:0e,d_l_type=0x88cc,actio
ns=drop
mininet> sh ovs-ofctl add-flow sD4 priority=65535,d_l_dst=01:80:c2:00:00:0e,d_l_type=0x88cc,actio
ns=drop
mininet> sh ovs-ofctl add-flow sC3 priority=65535,d_l_dst=01:80:c2:00:00:0e,d_l_type=0x88cc,actio
ns=drop
mininet> sh ovs-ofctl add-flow sB2 priority=65535,d_l_dst=01:80:c2:00:00:0e,d_l_type=0x88cc,actio
ns=drop
mininet> sh ovs-ofctl add-flow sA1 priority=65535,d_l_dst=01:80:c2:00:00:0e,d_l_type=0x88cc,actio
ns=drop
mininet>
mininet>
mininet>
mininet> dpctl dump-flows
*** sA1 -----
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=8.986s, table=0, n_packets=39, n_bytes=2340, idle_age=0, priority=65535,d
_l_dst=01:80:c2:00:00:0e,d_l_type=0x88cc actions=drop
*** sB2 -----
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=13.396s, table=0, n_packets=30, n_bytes=1800, idle_age=0, priority=65535,
d_l_dst=01:80:c2:00:00:0e,d_l_type=0x88cc actions=drop
*** sC3 -----
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=17.691s, table=0, n_packets=38, n_bytes=2280, idle_age=0, priority=65535,
d_l_dst=01:80:c2:00:00:0e,d_l_type=0x88cc actions=drop
*** sD4 -----
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=20.906s, table=0, n_packets=46, n_bytes=2760, idle_age=0, priority=65535,
d_l_dst=01:80:c2:00:00:0e,d_l_type=0x88cc actions=drop
*** sE5 -----
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=23.365s, table=0, n_packets=50, n_bytes=3000, idle_age=1, priority=65535,
d_l_dst=01:80:c2:00:00:0e,d_l_type=0x88cc actions=drop
*** sF6 -----
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=26.653s, table=0, n_packets=119, n_bytes=7140, idle_age=0, priority=65535
,d_l_dst=01:80:c2:00:00:0e,d_l_type=0x88cc actions=drop
mininet>
```

Obrázok 48. príkazy na zahodenie LLDP + OpenFlow tabuľka

19. Literatúra

- [1] SDxCentral. 2017. What is an OpenFlow Controller?
<https://www.sdxcentral.com/sdn/definitions/sdn-controllers/openflow-controller/>
- [2] SHALIMOV, F. et al.: Advanced study of SDN/OpenFlow controllers, Moscow, Russia: Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia, 2013 , ISBN 978-1-4503-2641-4
- [3] Thiago Paiva. 2013. OpenFLOW Controllers
https://www.ncc.unesp.br/its/projects/educloud/wiki/OpenFLOW_Controller/15
- [4] GUDE, N. et al.: NOX: Towards an Operating System for Networks, Yale University, New Haven CT, USA
- [5] David Erickson: The Beacon OpenFlow controller, Stanford University, CA, USA
- [6] KAUR, S., SINGH, J., GHUMMAN, N.S.: Network Programmability Using POX Controller, Ferozpur, India: International Conference on Communication, Computing & Systems, At SBS State Technical Campus , Ferozpur, Punjab , India, 2014
- [7] HARKAL, V.B., DESHMUKH, A.A: Software Defined Networking with Floodlight Controller: International Conference on Internet of Things, Next Generation Networks and Cloud Computing
- [8] SAIKIA, D., MALIK, N.: An Introduction to OpenMUL SDN Suite, 2014
- [9] CAI, Z., COX, A.L., EUGENE, T.S.: Maestro: A System for Scalable OpenFlow Control: Rice University
- [10] Sridhar Rao. 2015. SDN Series Part Eight: Comparison Of Open Source SDN Controllers
<https://thenewstack.io/sdn-series-part-eight-comparison-of-open-source-sdn-controllers/>
- [11] [<https://www.cisco.com/c/en/us/about/press/internet-protocol-journal/back-issues/table-contents-59/161-sdn.html>]
- [12] Programming and Communication Channel Issue - [NATARAJAN S., RAMAIAH A., MATHEN M.: "A software defined cloud-gateway automation system using OpenFlow", In: Proc. IEEE 2nd Int. Conf. CloudNet, Nov. 2013, pp. 219-226]

- [13] Potential single point of attack and failure - [BENTON K., CAMP J. L., SMALL CH.: OpenFlow Vulnerability Assessment, Indiana University Bloomington, Indiana, USA, pp. 1-6]
- [14] Mendoca M., Astuto B., Obraczka K., Turletti T.: Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks, 2013
- [15] Al-Somaidai M., Yahya E.: Survey of software components to emulate OpenFlow protocol as an SDN implementation, 2014
- [16] Open vswitch and ovs-controller. 2017. <http://openvswitch.org/>
- [17] OpenFlow1.3 Module for NS-3. 2017. <http://www.lrc.ic.unicamp.br/ofswitch13/>
- [18] Shankdhar, P.: 15 Best Free Packet Crafting Tools. 2017. <http://resources.infosecinstitute.com/15-best-free-packet-crafting-tools/#gref>
- [19] Oficiálna stránka hping. 2017. <http://www.hping.org/>
- [20] Ostinato Traffic generator. 2015. <https://sreeninet.wordpress.com/2015/04/24/ostinato-traffic-generator/>
- [21] Oficiálna stránka dokumentácie Scapy. 2017. <http://www.secdev.org/projects/scapy/>
- [22] Oficiálne úložisko Scapy. 2017. <https://github.com/secdev/scapy>
- [23] Oficiálna stránka PackETH. 2017. <http://packeth.sourceforge.net/packeth/Home.html>
- [24] Python Software Foundation. 2017. <https://pypi.python.org/pypi/ryu/4.18>
- [25] NTT Software Innovation Center. 2013. <https://osrg.github.io/ryu/slides/ONS2013-april-ryu-intro.pdf>
- [26] NTT Software Innovation Center. 2013. <https://osrg.github.io/ryu/slides/LinuxConJapan2013.pdf>
- [27] Tenarys, „QoS,“ 2 Februára 2016. <https://www.voip-info.org/wiki/view/QoS>
- [28] C. H. Tim Szigeti, Quality of Service Design Overview, 2004
- [29] V. Popeskić, „HOW DOES INTERNET WORK“. 2013. <https://howdoesinternetwork.com/2013/jitter>.

- [30] ONF „Software-Defined Networking (SDN)“. 2017. <https://www.opennetworking.org/sdn-definition/>
- [31] Big Switch Networks. 2017. <https://www.bigswitch.com>
- [32] RFC. 2015. <https://tools.ietf.org/html/rfc7426>
- [33] Guck J. W., Kellerer W., Achieving end-to-end real-time Quality of Service with Software Defined Networking, In: 2014 IEEE 3rd International Conference on Cloud Networking (CloudNet), Luxembourg (Luxembourg)
- [34] Palatinus M., Aplikácia SDN v bezdrôtových IoT sieťach, Diplomová práca, FIIT STU, 2017
- [35] L. F. C. projects, „Quality of Service (QoS)“, Linux Foundation, [Online]. Available: <http://docs.openvswitch.org/en/latest/faq/qos/>. [Cit. 11 December 2017].
- [36] „Measure UDP packet latency“, [Online]. Available: https://github.com/mrahtz/ultra_ping. [Cit. 11 December 2017].
- [37] Alpar Juttner, Balazs Szviatovszki, Ildiko Mecs, Zsolt Rajko, Lagrange Relaxation Based Method for the QoS Routing Problem, IEEE Transactions on Emerging Topics in Computing (Volume: 4, Issue: 2, April-June 2016)
- [38] Chienhung Lin, Kuochen Wang*, Guocin Deng, A QoS-aware routing in SDN hybrid networks, The 12th International Conference on Future Networks and Communications (FNC 2017)
- [39] Pop F., Dobre C., Comaneci D., Kolodziej J., V.: Adaptive Scheduling Algorithm for Media-Optimize, In: Journal Computing Volume 98 Issue 1-2, January 2016, Pages 147-168, New York, Inc. New York, NY, USA
- [40] Schepper T. D., Latre S., Famaey J., V.: A transparent load balancing algorithm for heterogeneous local area networks, In: Integrated Network and Service Management (IM), 2017 IFIP/IEEE Symposium on, Lisbon, Portugal
- [41] Oh S., Lee J., Lee K., Shin I.: RT-SDN: Adaptive Routing and Priority Ordering for Software-Defined Real-Time Networking, 2015, KAIST, South Korea
- [42] Petr Macek. 12.12.2013.

<https://www.samuraj-cz.com/clanek/zjistovani-sitoveho-okoli-protokolem-lldp-s-vyuzitim-php/>

[43] Volkan Yazici. 6.8.2013 <http://vlkan.com/blog/post/2013/08/06/sdn-discovery/>

Príloha A – Inštalčná príručka

Vzhľadom na veľmi zložitú inštaláciu a nastavovanie prostredia sa výsledný produkt nachádza vo virtuálnej mašine, ktorá je priložená v linku na Google Drive. Nie je potrebná žiadna inštalácia iba importovanie virtuálnej mašiny do Virtual boxu alebo obdobného programu.

Link na virtuálnu mašinu: <https://drive.google.com/drive/folders/1bKgEYu-CWHW5zbLdXGKqNLcrsdQqOusx>

Príloha B – Používateľská príručka

B.1 Mininet

Diagnostické príkazy

- **help**
 - zobrazenie dostupných príkazov
- **nodes**
 - zobrazenie uzlov zapojených v topológii
 - c0, h1, h2, s1
- **net**
 - zobrazenie informácie o linkách medzi uzlami
 - h1-eth0:s1-eth
 - s1-eth2:h2-eth0
- **dump**
 - zobrazenie informácie o všetkých zariadeniach v topológii
- **h1 ifconfig -a**
 - zobrazenie podrobnejších informácií o zariadení
- **h1 arp**
 - zobrazenie ARP tabuľky pre špecifikované zariadenie
- **h1 route**
 - zobrazenie smerovania pre špecifikované zariadenie
- **mn --v output**
 - zrušenie výpisov do konzoly pri vytváraní topológie
- **mn --v debug**
 - vynútenie všetkých výpisov do konzoly pri vytváraní topológie

Práca s topológiou

- **mn --topo minimal**
 - základná topológia
 - jeden kontrolér, jeden prepínač, dve koncové zariadenia
- **mn --topo linear, #**
 - sériové zapojenie prepínačov s jedným koncovým zariadením na prepínač
- **mn --topo single, #**
 - topológia s # koncovými zariadeniami na jeden prepínač
- **mn --switch ovs --controller ref --topo tree,depth=2,fanout=8 --test pingall**
 - Vytvorí stromovú topológiu kde na každý uzol napojí 8 hostov, dohrom. 64
 - Použije Open vSwitch a OpenSwitch/Stanford reference controller
 - Následne spustí ping medzi všetkými zariadeniami
- **link s1 h1 down / up**
 - zmena stavu linky medzi dvoma zariadeniami
- **mn --link tc, bw=10, delay=10ms**
 - nastavenie šírky pásma a oneskorenia medzi linkami
- **mn --mac**

- vynútenie nastavenia jednoduchých MAC adries pre rozhrania
- **mn --c**
 - odstránenie topológie
- **h1 python -m SimpleHTTPServer 80 &**
 - spustenie http serveru na špecifikovanom zariadení
 - (h1 python -m SimpleHTTPServer 80 >& /tmp/http.log &)
- **h2 wget -O - - h1**
 - vyžiadanie HTTP spojenia medzi špecifikovanými zariadeniami

Overenie funkčnosti topológie

- **h1 ps**
 - zobrazenie procesov celého zapojenia
- **h1 ping --c # h2**
 - ping medzi koncovými zariadeniami # krát
- **pingall**
 - ping na všetkých spojeniach
- **mn --test pingpair**
 - automatické vytvorenie topológie, overenie pomocou ping na všetkých linkách a zrušenie topológie

Vytvorenie vlastnej topológie

V aplikácii Mininet je možné vytvárať topológie podľa vlastných potrieb s požadovanými parametrami. Syntax vychádza z jazyka Python a je ľahko pochopiteľná a jednoznačná.



Obrázok 49. Vytvorená topológia

Nasledujúci kód vytvorí jednoduchú topológiu s dvoma prepínačmi, ktoré sú spoločne prepojené a každý má jedno koncové zariadenia. Taktiež linka medzi pravým prepínačom a jeho pripojeným zariadením je upravená nasledovne:

- šírka pásma 10 Mbps
- oneskorenie 5 ms (us, s)
- strata 2%
- maximálna dĺžka radu 1000 paketov
- s využitím princípu *Hierarchical Token Bucket*⁸

⁸ https://en.wikipedia.org/wiki/Token_bucket

```

from mininet.topo import Topo

class MyTopo( Topo ):
    "Simple topology example."

    def __init__( self ):
        "Create custom topo."

        # Initialize topology
        Topo.__init__( self )

        # Add hosts and switches
        leftHost = self.addHost( 'h1' )
        rightHost = self.addHost( 'h2' )
        leftSwitch = self.addSwitch( 's3' )
        rightSwitch = self.addSwitch( 's4' )

        # Add links
        self.addLink( leftHost, leftSwitch )
        self.addLink( leftSwitch, rightSwitch )
        self.addLink( rightSwitch, rightHost, bw=10, delay='5ms',
            loss=2, max_queue_size=1000, use_htb=True )

topos = { 'mytopo': ( lambda: MyTopo() ) }

```


Príloha C – Priebeh testovania

C.1 Testovací scenár č.1

1. over, či sú všetky linky zapojené tak, ako je potrebné //net
2. pingom over dostupnosť všetkých koncových zariadení //pingall
3. nájdi cestu z PC1 do PC5 //tracert -n IPaddress
4. zhoď linku na ceste, získanú z bodu 3 // ifconfig sX-ethX down
5. zopakuj krok 3 a over, či linka bola zmenená a sieť naďalej funguje
6. nájdi cestu z PC3 do PC7
7. zmeň šírku pásma na trase, získanej z bodu a)
8. zopakuj krok a) a over, či bola cesta zmenená
9. nájdi cestu z PC2 na PC4
10. na ceste získanej z bodu A. zmeň šírku pásma na niektorej linke
11. zopakuj krok A. a zisti, či bola cesta zmenená
12. vytvor http server //h1 python -m SimpleHTTPServer 80 &
13. pošli požiadavku na h2 //h2 wget -O - h1
14. vypni server //h1 kill %python

C.2 Testovací scenár č.2

1. Priradiť PC0 VLAN port a IP adresu (pri port 100 10.0.0.2/24, adresa 1.0.0.2/24)
2. Priradiť PC1 VLAN port a IP adresu (pri port 100 10.0.0.3/24, adresa 1.0.0.3/24)
3. Nastav Open vSwitch (Switch0) (priradiť porty s koncovými zariadeniami)
4. Vytvor Queue ⁹vo Switch0
5. Inštalácia toku pre PCP (Priority Code Point) do radu mapovania je zabezpečená RYU kontrolérom
6. Ping na PC1 (1.0.0.3) z PC0 – mal by fungovať
7. Ping na PC0 (1.0.0.2) z PC1 – mal by fungovať
8. Overenie toku tabuľky tokov pridaním VLAN tagu (PCP = 3)
9. Ping na PC1 (10.0.0.3) z PC0 – funguje
10. Ping na PC0 (10.0.0.2) z PC1 – funguje
11. Skontrolovanie šírka pásma pre tok bez VLAN tagu
12. Spustenie iperf servera na PC0 (iperf -s -p 8011 -u -f m -reportstyle C -i 1 //udp server)
13. Spustie iperf klienta na PC1 (iperf -c 1.0.0.3 -p 8011 -i 1 -f m -t 120 -u -b 10000000)
14. Overenie, že šírka pásma na PC1 je 10Mbps/s
15. Skontrolovanie šírky pásma pre tok s VLAN tagom (PCP = 3)
16. Spustenie iperf servera na PC0 (iperf -s -p 8011 -u -f m -reportstyle C -i 1 //udp server)
17. Spustie iperf klienta na PC1 (iperf -c 10.0.0.3 -p 8011 -i 1 -f m -t 120 -u -b 10000000)
18. Overenie, že výstup na PC1 je 5Mbps/s

⁹ <https://www.networkworld.com/article/2234323/cisco-subnet/quality-of-service--queuing.html>

C.3 Testovací scenár č.3

1. Nastavíme topológiu podľa obrázka 2.
2. Na s0 nastavíme, aby preniesol 50 % premávky cez ap1 a zvyšných 50 % cez ap2 do sta1
3. Otestujeme pomocou príkazov
4. Na sta1: `iperf -s -u -i 1`
5. Na h1: `iperf -c 192.168.10.10 -u -b 100M -t 5`
6. V prípade problémov konfigurácie kontroléra, je možné danú topológiu zostrojiť a otestovať aj bez neho.

C.4 Testovací scenár č.4

1. definuj kontrolér a sieť `//netMininet(controller=RemoteController)`
2. vytvor kontrolér s portom 6633 `//net.addController(name, port)`
3. vytvor 3 prepínače `//net.addSwitch(name)`
4. vytvor tri koncové zariadenia `//net.addHost(name,ip)`
5. vytvor linky podľa topológie `//net.addLink(src, dts)`
6. spusti sieť `//net.build(),xy.start(controller),et.startTerms()`
7. zapni CLI `//CLI(net)`
8. nastav OpenFlow verziu pre prepínače `//ovs-vsctl set Bridge sxp protocols=OpenFlow13`
9. na uzle root (Node: c0 (root)) spusti skript pre STP
10. `ryu-manager ryu.app.simple_switch_stp_13`
11. pre všetky uzly (nodes) nastav filtrovanie TCP/IP paketov
12. `tcpdump -i sx-eth2 arp`
13. pingom z host1 do host2 over, že paket nie loop-ovaný
14. zhod' eth linku s2 `// # ifconfig s2-eth2 down`
15. linka, ktorá bola v stave blocked, by sa mala prepnúť do stavu forwarding

Príloha D – Technická dokumentácia

D.1 Analýza kódu z diplomovej práce Michala Palatinusa

Programy potrebné pre spustenie kontroléra:

- Controller.py – samotný kontrolér
- QoS_linkDB.txt – súbor obsahujúci informácie pre QoS na jednotlivých linkách
- PathFinder.py – algoritmus pre vytvorenie cesty
- Globals.py – nastavenie parametrov pre jednotlivé linky

Program je navrhnutý na statické priradenie liniek a parametrov pre jednotlivé linky. Linky majú aj dopredu dané pomenovanie a aj z tohto dôvodu je potreba metódu prerobiť, keďže názov liniek sa môže líšiť v závislosti od simulačného prostredia alebo aj danej topológie.

Controller.py

Importuje PathFinder, ktorý vráti najlepšiu cestu, ktorá dosiahla najlepšie hodnoty QoS a Queues. Pri inicializácii topológie si vytvorí objekt jeho typu. V metóde createRoutingRules(self, ev, match, paket, parser, dp, ofp) volá funkciu pathfindra metódou findpath(QOS_LINK_PROPS, TOPOGRAPH, str(dp_id), str(end_dp), category) pričom

- QOS_LINK_PROPS: QoS hodnoty parametrov, ktoré daný tok vyžaduje
- TOPOGRAPH: Dátová štruktúra Slovník, uchováva v sebe graf topológie
- DP_id: ID inštancie
- End_dp: Cieľový port
- Category: Typ kategórie flowu

IP adresy, ktoré generuje kontrolér pri preposielaní paketov:

- Zdrojová adresa 10.1.1.252
- Cieľová adresa 10.1.1.253

def loadDatabases():

Zo súboru */home/mininet/ryu/ryu/app/QoS_linkDB.txt* sa načítavajú jednotlivé QoS parametre pre každú linku

```
def _icmp_send(dp, port_out, ip_src=DISCOVERY_IP_SRC, ip_dst=DISCOVERY_IP_DST, eth_src='02:b0:00:00:00:b5', eth_dst='02:bb:bb:bb:bb:bb', icmp_type=8, icmp_code=0):
```

- dp -- Datapath
- port_out – Port na prepínači pre preposlanie paketu
- ip_src -- IP adresa posielajúceho
- ip_dst -- IP adresa prijímateľa
- eth_src -- Ethernet adresa zdroja (štandardná je 02:b0:00:00:00:b5)
- eth_dst -- Ethernet cieľová adresa (pravdepodobne bude treba prerobiť)
- icmp_type -- ICMP typ, 8 – echo
- icmp_code -- ICMP kód, 0 – štandard

def on_inner_dp_join(self, dp):

Nový prepínač sa pridá do siete

Postup:

1. Vymažú sa všetky existujúce toky
2. Všetky pingy s cieľovou adresou = 10.1.1.253 sa presmerujú na kontrolér
3. Priradí sa k jednotlivým pingom akcia
4. Jednotlivé toky sa priradia do tabuľky tokov k jednotlivým switchom

def setStaticRoutesOnEdgeSwitch(self, dp, parser):

Staticky nastaví IP k jednotlivým koncovým zariadeniam. Kontroluje podľa datapath id.

def forwarder_state_changed(self, ev):

Kontroluje zmenu stavu prepínača. Môžu nastať dva stavy, pričom v metóde sa stavy kontrolujú nasledovne:

- ev.enter == True - nový prepínač je pripojený
- ev.enter == False - prepínač je odpojený

Postup:

1. Ak true
 - a) Špecifikujeme toky.
 - b) Nastavíme toky pre koncové zariadenia.
 - c) Spustíme funkciu on_inner_db_join .
 - d) Pre každý prepínač pošle ICMP pakety z každého jeho portu okrem toho, ktorým je pripojený na kontrolér.

2. Ak false
 - a) Vymaže daný prepínač.

def _packet_in(self, ev):

Metóda, ktorá sa stará o pakety, ktoré smerujú priamo ku kontrolérovi.

Postup:

1. V prípade vytvorenia novej linky medzi forwardermi.
2. Vytvorenie nového toku => treba vytvoriť nové pravidlá pre tok cez funkciu:
createRoutingRules(ev, match, paket, parser, dp, ofp).

def flow_removed_handler(self, ev):

Metóda zachytáva toky, ktoré sú odstránené. Vypíše sa dôvod zrušenia toku a potom je tok odstránený z databázy pomocou funkcie: *deleteFlowFromDB(str(msg.match), int(msg.cookie))*.

def createRoutingRules(self, ev, match, paket, parser, dp, ofp):

Jedna z hlavných funkcií pri určovaní path_finder algoritmu a konfigurovaní pravidiel.

- paket – paket, ktorý prišiel ku kontrolérovi, prepínač ho nemá vo svojej tabuľke tokov
- parser: OF 1.3 parser
- dp: datapath, z ktorého prišiel paket ku kontrolérovi

Postup:

1. Ak je zadaný protokol
2. Ak sa objaví nový tok
3. Ak zdrojová IP adresa nie je známa pri koncových zariadeniach (pole)
 - a) Pridaj zdrojovú IP adresu do koncových zariadení (pole)
 - b) Vytvor jednotlivé toky
4. Ak cieľová adresa je jeden z koncových zariadení (pole)
 - a) Zisti počiatočný port, cez ktorý pôjde paket von
 - b) Datapath id koncového zariadenia
 - c) Pomocou metódy vytvor nový tok *categorizeFlowAndDefineMatch(paket, src, dst, parser)*
 - d) Spusti path_finder algoritmus, ktorý nájde najlepšiu cestu podľa QoS
 - e) Nainštaluj toky

5. Ak cieľová adresa nie je jeden z koncových zariadení (pole)

a) Rozošli paket – pošli ho všetkým zariadeniam, okrem toho ktorým prišiel

def categorizeFlowAndDefineMatch(self, pkt, src, dst, parser):

Metóda, ktorá kategorizuje nový tok na základe charakteristík prijatého paketu.

- pkt: prijatý paket
- src: zdrojová IP
- dst: cieľová IP

1. Zistí o aký typ protokolu ide

a) ICMP – priradí prioritu a „match“

b) TCP

- (video streaming) - ak je zdrojová adresa 10.0.0.7 => priradí kategóriu a „match“
- (file server) – zdrojová adresa 10.0.0.6 => kategória + „match“
- (Skype) – zdrojová adresa 10.0.0.8 => kategória + tcp porty + „match“
- V opačnom prípade pridať prioritu a timeout, ostatné hodnoty sú null

c) UDP

- (video streaming) - ak je zdrojová adresa 10.0.0.7 => priradí kategóriu a „match“
- (file server) – zdrojová adresa 10.0.0.6 => kategória + „match“
- (Skype) – zdrojová adresa 10.0.0.8 => kategória + tcp porty + „match“
- V opačnom prípade pridať prioritu a timeout, ostatné hodnoty sú null

d) ARP – priradí prioritu a „match“ + timeout

e) Default routing

def addFlowToDB(self, match, cookie, node1, node2, category, queue):

Priradí nové toky to databázy a pomocou funkcie *decreaseLinkCapacity* zníži kapacitu linky.

- match: pravidlá toku
- cookie: číslo identifikácie toku (flow ID)
- node1: koncový bod pre identifikáciu toku
- node2: koncový bod pre identifikáciu linky
- category: kategória toku
- queue: rad priradený k toku

def deleteFlowFromDB(self, match, cookie):

Metóda, ktorá vymaže tok z databázy a zvýši kapacitu pomocou funkcie *increaseLinkCapacity*

- match: pravidlá toku
- cookie: číslo identifikácie toku (flow ID)

def decreaseLinkCapacity(self, node1, node2, category, queue):

- node1: koncový bod pre identifikáciu toku
- node2: koncový bod pre identifikáciu linky
- category: kategória toku
- queue: rad priradený k toku

def increaseLinkCapacity(self, node1, node2, category, queue):

- node1: koncový bod pre identifikáciu toku
- node2: koncový bod pre identifikáciu linky
- category: kategória toku
- queue: rad priradený k toku

def getIpAddresses(self, paket):

Funkcia vracia zdrojovú a cieľovú IP adresu a či sa jedná o ARP paket

def generateCookie(self):

Funkcia generuje náhodné 9 ciferné číslo ako identifikátor toku

def updateTopologyFromIcmp(self, paket, ev):

Discovery ping nesie v sebe informáciu o datapath.

Postup:

1. Rozparsujeme ICMP pomocou funkcie *_icmp_parse_payload*.
2. Zistí sa susedné datapath ID a port
3. Pridajú sa do topológie ako nové linky
4. Prebehne znova načítanie sa topológie
5. Kontroluje sa počet liniek v topológii a rady sú rovné false
 - a) Staticky sa kontroluje počet liniek.
 - b) Rady sa konfigurujú iba v prípade, že je nájdená celá topológia.

- c) Pre každý datapath a pre každý port v ňom nastav rady.
- d) Volá sa funkcia *set_queue*.
- e) Na konci sa zavolá funkcia *topoToSet*.

def add_flow(self, dp, priority, match, actions, cookie, table=0, idle_timeout=None):

Pridávanie nových tokov do topológie.

- dp: datapath (prepínač)
- priority: priorita nového toku
- match: pravidlá toku
- actions: akcie toku
- cookie: identifikátor toku
- table: tabuľka, ktorá nesie informácie
- idle_timeout: čas, po ktorom je tok vymazaný z tabuľky

def set_queue(self, datapath, port, ovsdb_bridge):

Metóda, ktorá nakonfiguruje jednotlivé rady pre porty.

- datapath: prepínač, na ktorom budú rady aplikované
 - port: port, na ktorom budú rady nakonfigurované
 - ovsdb_bridge
1. Ak sa nachádza port k danej linke
 - a) Ulož si ho
 2. Pre každú linku pre QoS
 - a) Nastav maxrate, minrate
 - b) rady sa konfigurujú len pre porty k ostatným prepínačom, preto <10000
 - c) ak je datapath id < 5 a počet portov je menší ako 10000
 - i. vytvor názov linky
 3. Ak názov nie je nulový priradiť jednotlivé QoS rady

def topoDictWithPorts(self, topo):

Metóda konvertuje topológiu na nejaký lepší formát – nie je implementované v tejto verzii

def topoToSet(self, topo):

Prekonvertuje topológiu do krajšieho formátu.

Napríklad.:

```
{  
'1': {'3', '2'},  
'11': {'2'},  
'13': {'2'},  
'2': {'1', '11', '13', '12', '5', '14'},  
}
```

PathFinder.py

Metóda `findpath(QOS_LINK_PROPS, TOPOGRAPH, str(dp_id), str(end_dp), category)`:

Jadro evolučného algoritmu. Proces prebieha v 10 iteráciách. Najskôr dostane 2 cesty z depth-first algoritmu. Potom prebieha 10 iterácií ak nezistí cestu s fv 0. Nájde sa spoločný stredný uzol. Ak existuje aplikuje sa kríženie a mutácia. Ak sa nenachádza aplikuje sa iba mutácia. Nakoniec sa všetky nové cesty priradia do zoznamu ciest, pokiaľ zoznam ciest ešte takú cestu nemá. Nakoniec sa vyberie najlepšia cesta a vráti sa.

Metóda `fitnessValue(nodes, category, QOS_LINK_PROPS)`

Vypočíta hodnotu cesty na základe jej QoS hodnôt.

Pomocné informácie

```
SERVICE_CLASS = {  
    #file_sharing: {'alpha': 0, 'beta': 0, 'gamma': 1, 'delay': 4.000, 'jitter': 0.01, 'thrput': 4000000}, #  
    Simulacia 1 #THRPT in bits # before 'thrput': 1000000  
  
    'file_sharing': {'alpha': 0, 'beta': 0, 'gamma': 1, 'delay': 4.000, 'jitter': 0.01, 'thrput': 5500000}, #  
    simulacia 2  
  
    'skype_audio': {'alpha': 1, 'beta': 0, 'gamma': 1, 'delay': 1.000, 'jitter': 0.01, 'thrput': 100000},  
  
    'mp4': {'alpha': 1, 'beta': 0, 'gamma': 1, 'delay': 4.000, 'jitter': 0.01, 'thrput': 4500000},  
  
    'management': {'alpha': 1, 'beta': 0, 'gamma': 1, 'delay': 1.000, 'jitter': 0.01, 'thrput': 500}  
}
```

V premennej SERVICE_CLASS, ktorá sa nachádza v súbore globals.py sú uložené potrebné QoS parametre, ktoré daná komunikácia vyžaduje.

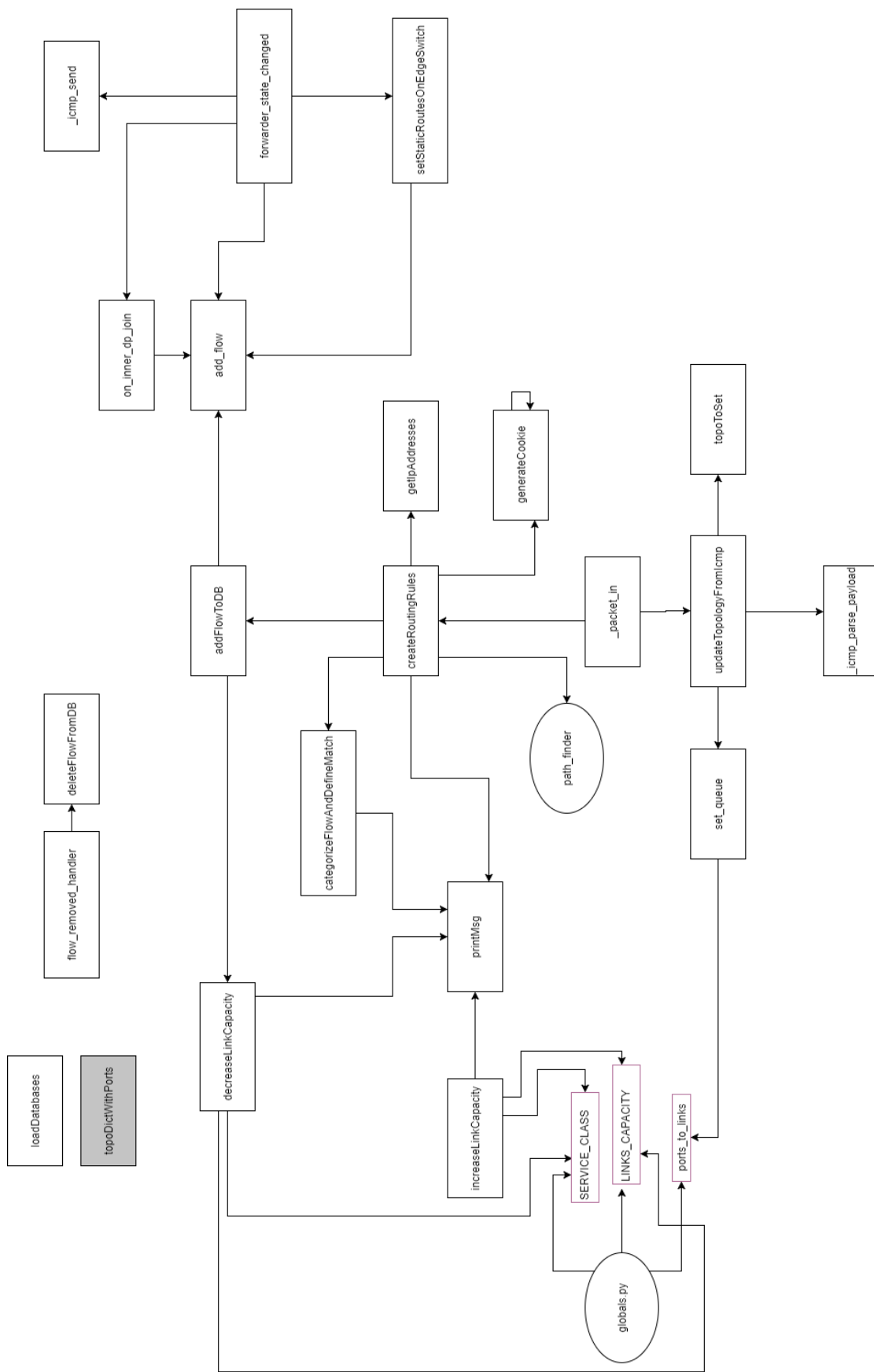
```
DELAY_TABLE = {  
  
    #1000000000: {'R1':7000000, 'R2':5500000, 'T1':0.12012, 'T2':0.5457}, #1x10^9bits =  
    125x10^6bytes s1-s2  
  
    100000000: {'R1':7000000, 'R2':5500000, 'T1':0.12012, 'T2':0.5457}, #1x10^8bits =  
    125x10^5bytes s1-serv1  
  
    10000000: {'R1':700000, 'R2':550000, 'T1':1.2012, 'T2':5.457}, #1x10^7bits = 125x10^4bytes s2-  
    s11  
  
    1000000: {'R1':70000, 'R2':55000, 'T1':12.012, 'T2':50.457} #1x10^6bits = 125x10^3bytes wifi  
    !!!! IN current implementation capacity of  
  
        # end links is not limited and links are not included in pathfinder. Default speed of  
    Mininet should be high enough:  
  
        #without setting the queues Jperf TCP bw was between 13000 and 17000 Mbits/sec  
  
}
```

V premennej DELAY_TABLE si uchováva hodnoty oneskorenia každého radu

```
LINKS_CAPACITY = { # one directional  
  
    '1-2': {0: {'max_cap': 5600000, 'remaining_cap': 5600000}, 1: {'max_cap': 4400000, 'remaining_cap':  
    4400000}},  
  
    '2-1': {0: {'max_cap': 5600000, 'remaining_cap': 5600000}, 1: {'max_cap': 4400000, 'remaining_cap':  
    4400000}}}
```

V premennej Links_CAPACITY má uložené údaje o kapacitách každej linky, maximálnu kapacitu a zostávajúcu kapacitu.

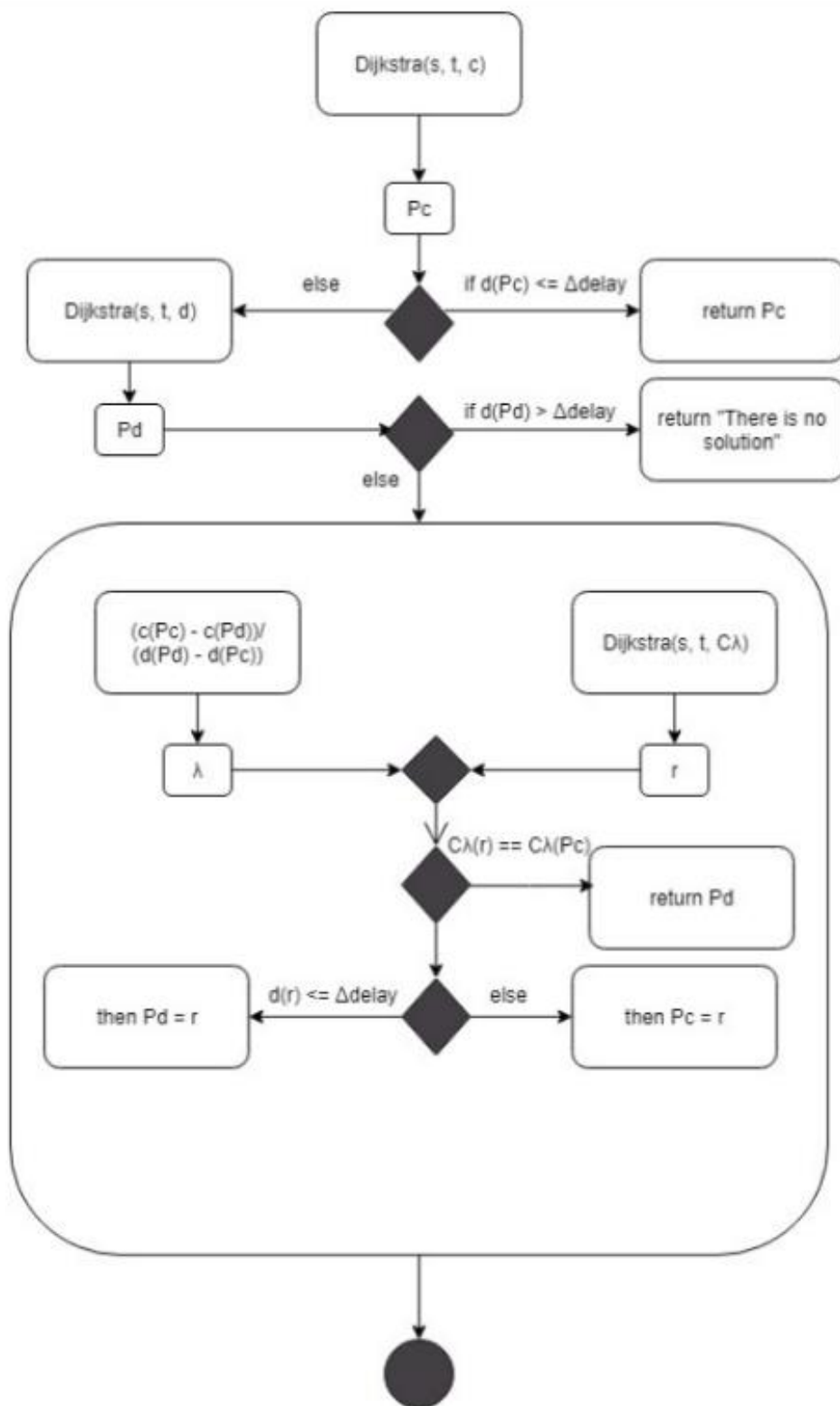
D.1.1 Diagram prepojenia funkcií



sivá farba predstavuje neimplementovanú funkciu v tejto verzii
 smer šípok predstavuje volanie funkcie - z akej počiatkovej funkcie bola konečná funkcia volaná

Obrázok 50. Diagram prepojenia funkcií

D.2 Diagram algoritmu LARAC



10

Obrázok 51. Algoritmus LARAC článok

¹⁰ Znáznomený diagram zámerne nespĺňa direktívy UML z podstaty efektívne demonštrovať priebeh algoritmu

D.3 Opis metódy LARAC

Oproti metóde z Palatinusa sa zmenil len obsah súboru `path_finder.py`, ktorý obsahuje algoritmus hľadania a výberu cesty. Opis je nasledovný:

`larac(self, src, dst, delay_graph, cost_graph):`

Hlavná funkcia algoritmu, vráti nájdenú cestu. Parametre: zdroj, cieľ, graf oneskorení liniek, graf kapacít liniek.

`calculate_lambda_graph(self, lambda_val, cost_graph, delay_graph):`

Vypočíta lambda graf od hodnoty lambda a oboch grafov

`calculate_path_cost(self, graph, path):`

Vypočíta veľkosť cesty v grafe. Parametre: graf, cesta

D.4 Opis metódy SAQR

Oproti metóde z Palatinusa sa zmenil len obsah súboru `path_finder.py`, ktorý obsahuje algoritmus hľadania a výberu cesty. Opis je nasledovný

`sacher(self, src, dst, graph, category):`

Funkcia je hlavnou funkciou súboru, volá ostatná výpočtové funkcie.

`shortest_path(graph.cost_graph, src, dst):`

Funkcia vypočíta najkratšiu cestu topológiou od zdroja po cieľ. Výslednú cestu vráti do premennej **P**.

`cal_path_cost(P, SERVICE_CLASS[category], graph):`

Funkcia vypočíta cenu cesty **P** na základe vzorcov zadaných v analýze metódy. Výslednú cenu cesty **P** vráti do premennej **CP**.

`gen_nei_path(P, graph, dst, src):`

Funkcia vypočíta novú cestu z náhodne vybraného prepínača z cesty **P**. Novú cestu vráti do premennej **N**.

Následne sa vždy prepočítava cena novej cesty a porovnáva sa s cenou predchádzajúcej. Algoritmus nakoniec vráti cestu s najlepšou cenou.

```
def sacher(self, src, dst, graph, category):
    P = shortest_path(graph.cost_graph, src, dst)
    CP = self.cal_path_cost(P, SERVICE_CLASS[category], graph)
    t = 1
    while t < len(graph.cost_graph):
        N = self.gen_nei_path(P, graph, dst, src)
        CN = self.cal_path_cost(N, SERVICE_CLASS[category], graph)
        if CP > CN:
            CP = CN
            P = N
        t = t + 1
    return P
```

Obrázok 52. Ukážka kódu k SAQR

Príloha E – testovanie treťou stranou

E.1 Úvod

Nakoľko je testovanie technicky náročné, pre účely testovania sme si vybrali osoby, ktoré už mali predošlé skúsenosti so softvérovo definovanými sieťami, ako aj s prostredím Mininetu. Používatelia vykonali testovanie podľa vopred pripraveného dokumentu testovacích scenárov.

E.2 Testovanie

Na začiatku boli používatelia oboznámení s dokumentom s testovacími scenármi. Následne nainštalovali virtuálny stroj, na ktorom následne prebiehalo testovanie jednotlivých metód. Inštalácia prebehla bez problémov.

E.2.1 Testovanie prostredia Mininetu

Na testovanie prostredia boli použité testovacie scenáre 1. – 4.

Testovanie prostredia prebehlo bez problémov pri scenároch 1., 2. a 4.. Pre testovací scenár číslo 3. bolo potrebné, aby testovacie prostredie disponovalo Wi-Fi, ktorú sme však neinštalovali, nakoľko sa nevyužíva ani vo zvyšných scenároch, ani v nami implementovanými metódach. Z časových a praktických dôvodov sme sa rozhodli tento testovací scenár vynechať.

E.2.2 Testovanie implementovaných metód

Testovanie implementovaných metód prebiehalo osobitne pre každú metódu. Aj keď sa všetky metódy zameriavajú na nájdenie cesty v sieti, každá metóda skúma iné prístupy a algoritmy, čomu sa prispôbili aj jednotlivé testovacie scenáre.

V ďalších podkapitolách budú zhrnuté pripomienky testerov pri testovaní jednotlivých metód, prípadne problémy, ktoré sa pri testovaní vyskytli. V poslednej podkapitole Zhrnutie budú uvedené konkrétne výsledky a ich porovnanie.

E.2.3 Testovanie upravenej metódy p. Palatinusa

Samotná metóda sa zameriava na dynamické priradovanie tokov do radov. Cesty sa hľadajú kombináciou genetického algoritmu a modifikovaného Network Calculus modelu, pričom sa pri hľadaní používajú tri metriky – oneskorenie, priepustnosť a odchýlka.

Testovanie metódy sa vykonávalo na topológii vytvorenej pre upravenú metódu p. Palatinusa, pomocou nástroja *iperf*, pričom sa testovali rôzne typy tokov – TCP, Skype hovor, prenos videa a manažment. Bližšia špecifikácia tokov je uvedená v dokumente testovacie scenáre. Postupovalo sa podľa pokynov v dokumente.

Výstupom bola cesta s najmenším počtom hopov a nie najrýchlejšia cesta.

Testerom prekážalo manuálne zadávanie počtu liniek do kontroléra.

E.2.4 Testovanie metódy LARAC

Metóda sa zameriava na dynamické priradovanie tokov do radov.

Testovanie prebiehalo na topológii vytvorenej pre metódu LARAC, ako aj na hlavnej topológii.

Testerom prekážali neskoré výpisy informácií o pingu, ktoré sa objavovali až po výpise ciest kontrolérom. Taktiež im prekážali informácie o cestách, ktoré nesúviseli s výslednou cestou. Za nedostatok sa opäť považovalo manuálne zadávanie počtu liniek do kontroléra.

E.2.5 Testovanie metódy SAQR

Metóda sa zameriava na dynamické priradovanie tokov do radov.

Testovanie prebiehalo na topológii vytvorenej pre metódu SAQR.

Kvôli nefunkčnosti pingu sa zmenili parametre liniek, kedy už metóda pracovala správne. Pre testovanie ďalších krokov sa však parametre vrátili do pôvodného stavu a metóda už ďalej pracovala správne.

E.3 Zhrnutie

Testovanie prebiehalo podľa inštrukcií v testovacích scenároch, pričom sa opakovali nasledujúce kroky:

- Nájdenie cesty z uzla h1 do uzla h2
- Zmena linky (vypnutie linky alebo zmena parametrov na linke)
- Nájdenie cesty z uhla h1 do uzla h3

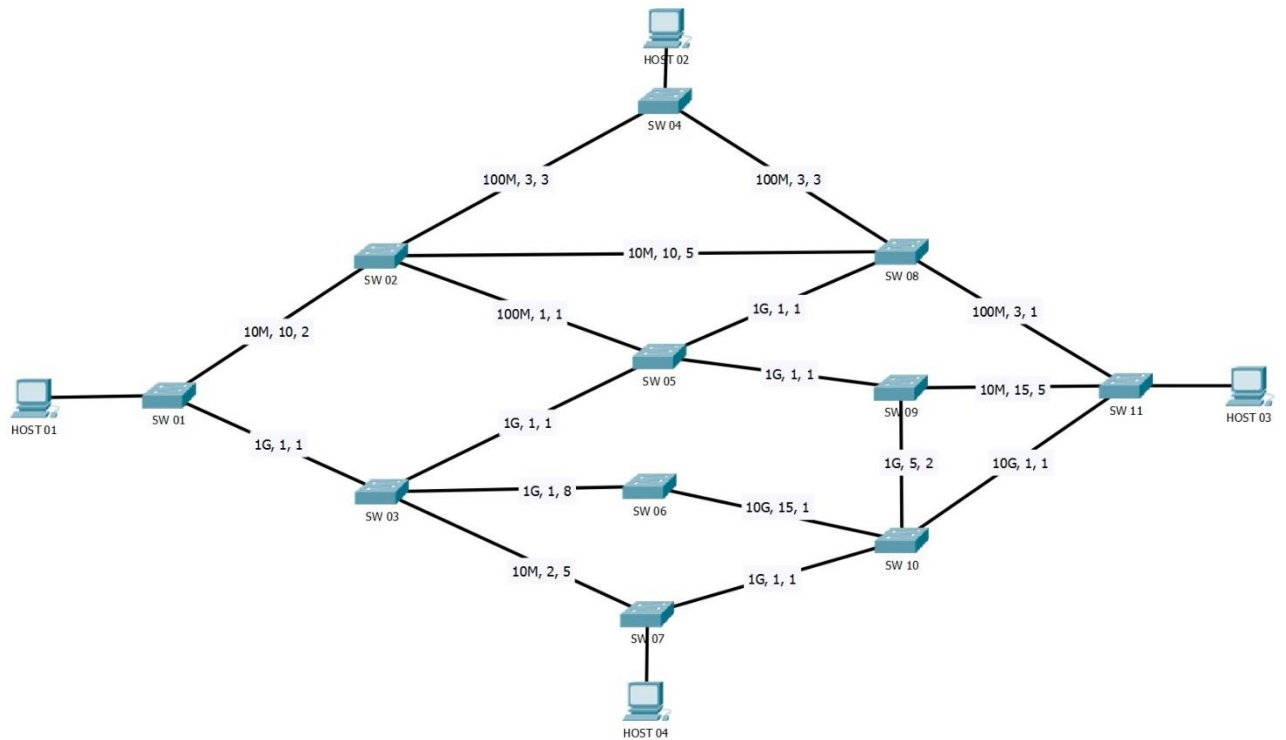
V tabuľke nižšie tak budú uvedené cesty tak, ako boli nájdené za pomoci jednotlivých metód a podľa testovacích scenárov. Uvedené budú 2 cesty pred zmenou parametrov liniek a cesta po zmene určitej linky, ako aj opis zmeny, ktorá bola na linke vykonaná.

| | h1 → h2 | h1 → h3 | zmena linky | po zmene linky |
|---------------------------------|-------------------|---------------------|---|--------------------|
| metóda p. Palatinusa | 1 → 2 → 4 | 1 → 3 → 6 → 10 → 11 | zrušenie linky medzi 2 a 8 | 1 → 3 → 5 → 9 → 11 |
| metóda LARAC (hlavná topológia) | 4 → 8 → 5 → 3 → 1 | 1 → 3 → 7 → 10 → 11 | | |
| metóda LARAC (topológia LARAC) | 1 → 3 → 6 | | zmena parametrov na linke sw3 ↔ sw6: <ul style="list-style-type: none"> • bw¹ = 20 na bw = 1 • delay² = 1ms na delay = 100000ms | 1 → 5 → 6 |
| metóda SAQR | 1 → 3 → 5 → 2 → 4 | 1 → 3 → 5 → 8 → 11 | zmena parametrov na linke sw2 ↔ sw5: <ul style="list-style-type: none"> • bw = 100 na bw = 1 • delay = 1ms na delay = 100000 • loss³ = 1 na loss = 10 | 1 → 3 → 5 → 8 → 4 |

¹ **bw** = bandwidth; priepustnosť linky
² **delay** = oneskorenie na linke
³ **loss** = packet loss; strata paketov

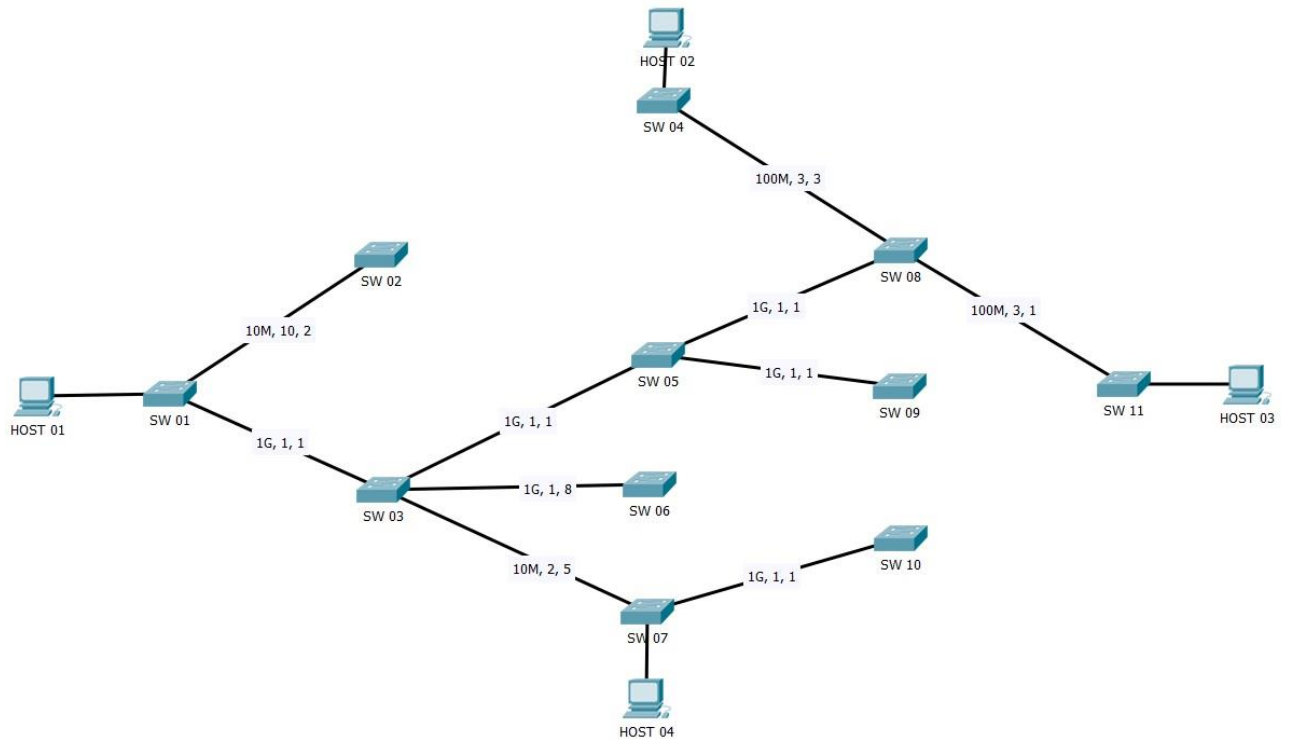
Obrázok 53. Testovanie treťou stranou

Príloha F – Testovacia topológia 1



Obrázok 54. Testovacia topológia 1

Príloha G – Testovacia topológia 2



Obrázok 53. Testovacia topológia 2

Príloha H – nesprávne testovanie

Metóda bola testovaná rovnako, ako v diplomovej práci Palatinusa. Využitá bola lineárna topológia o 8 koncových zariadeniach, 8 prepínačoch a 15 linkách. Nástroj na jednotlivé simulácie bol použitý *iperf*.

Jednotlivé toky, ktoré boli testované sú:

5. Prenos súborov

Protokol: TCP

Priepustnosť: 5,5 Mbit/s

Port: 5001

Veľkosť paketov: dynamicky dané TCP protokolom

Rýchlosť odosielania: dynamicky dané TCP protokolom

Odosielané dáta brané z existujúceho súboru (prepínač *-F /path/to/file.mp4*)

6. Skype hovor

Protokol: UDP (prepínač *-u*)

Priepustnosť: 100kbit/s

Port: 5002

Veľkosť paketov: 450B (prepínač *-l 450B*)

Rýchlosť odosielanie: 500 kbit/s (prepínač *-b 500k*)

7. Prenos videa

Protokol: UDP (prepínač *-u*)

Priepustnosť: 4,5Mbit/s

Port: 5003

Veľkosť paketov: 1460B (prepínač *-l 1460B*)

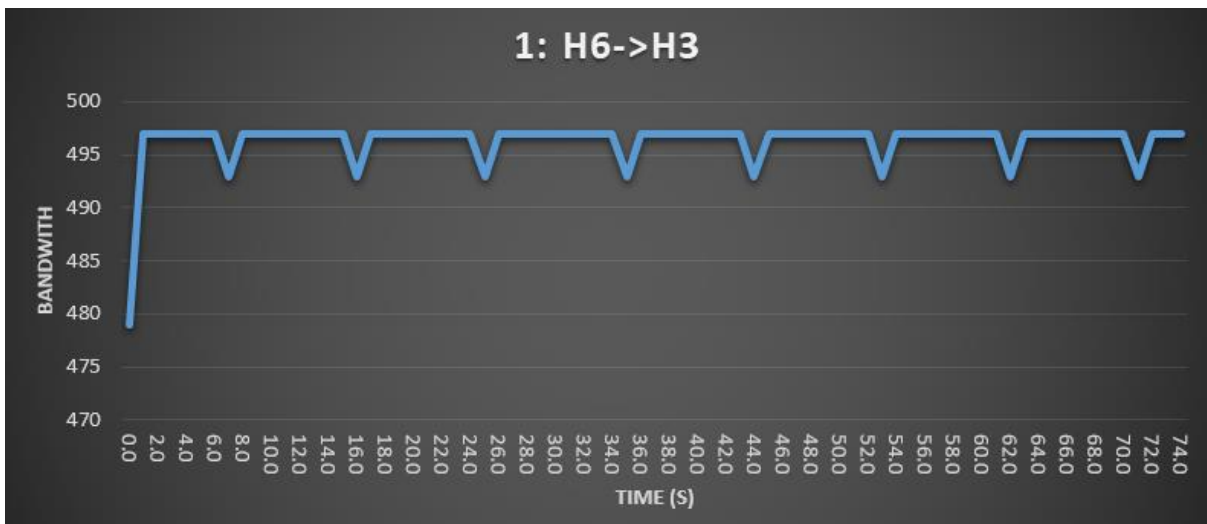
Rýchlosť odosielania: 4500 kbits/s (prepínač *-b 4500k*)

8. Manažment

Manažment predstavuje premávku ako ICMP, ARP a ostatnú IP komunikáciu, ktorá nespadá do ostatných troch kategórií tokov.

Toky:

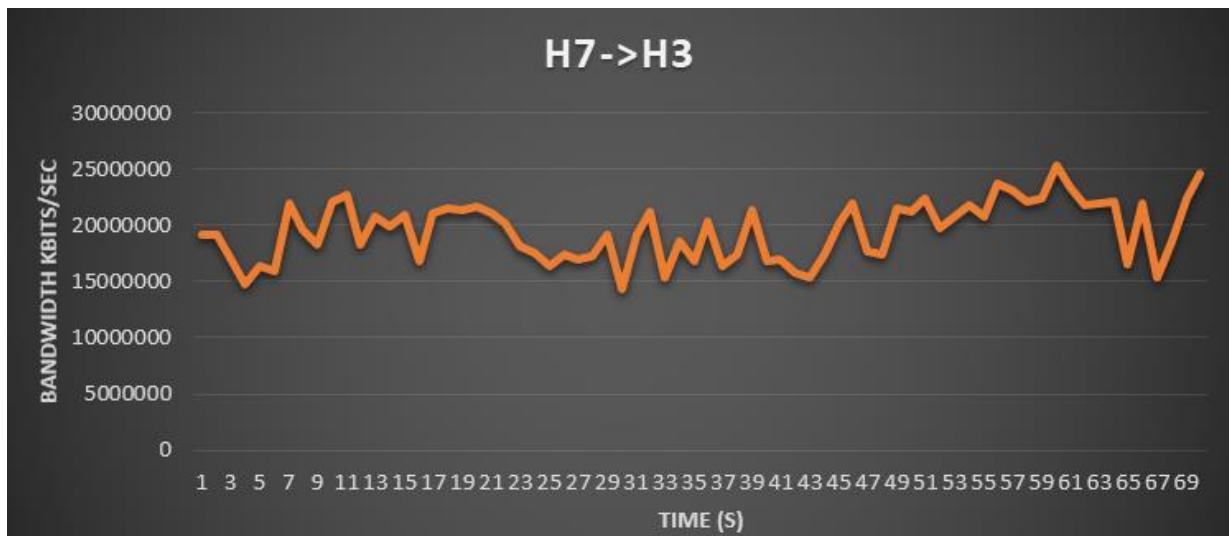
1. `iperf -c 10.0.0.6 -u -P 1 -i 1 -p 5002 -l 450B -f k -b 500k -t 75`



Graf 69.

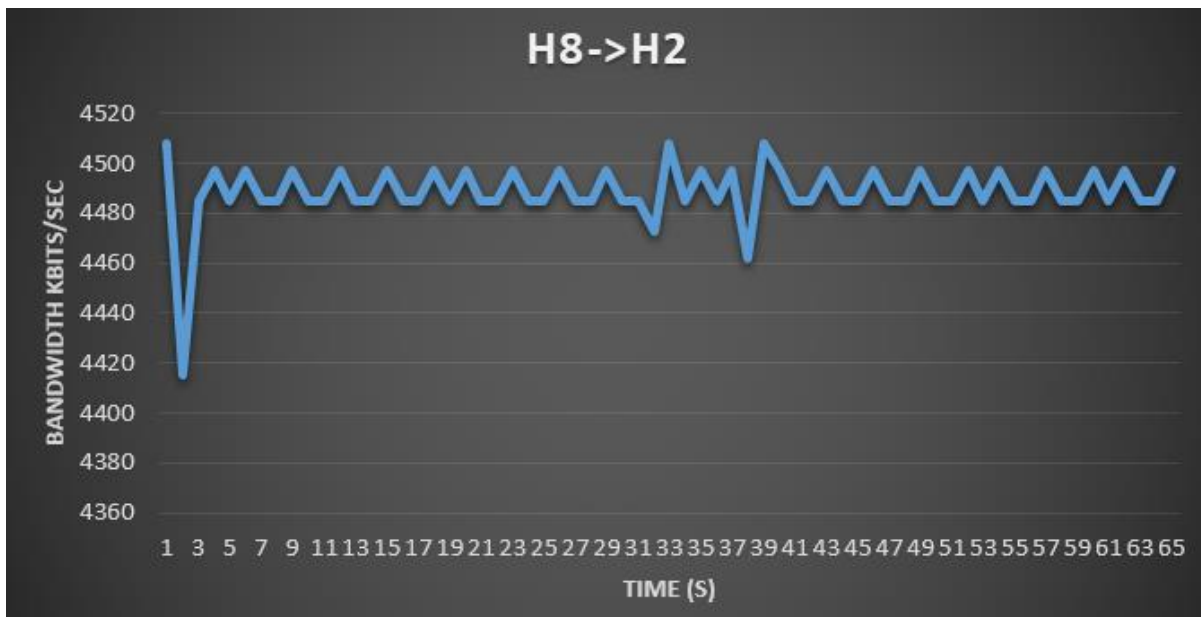
Na grafe 69 možno vidieť bandwidth v kbit/s, ako je možné vidieť, priepustnosť neklesla ani ku stanovenej hranici 100kbit/s a po celý čas sa drží konštantnej prepustnosti.

2. `iperf -c 10.0.0.7 -P 1 -i 1 -p 5001 -f k -t 70 -F /home/bakalarka/Downloads/`



Graf 70.

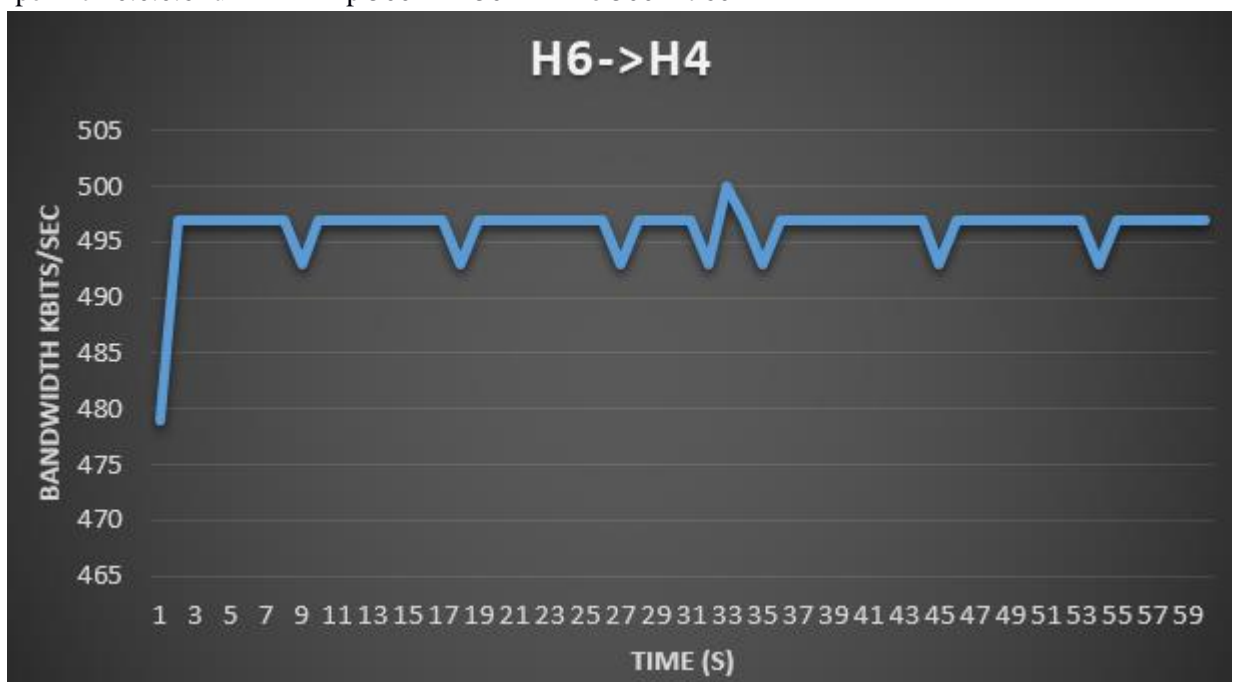
3. `iperf -c 10.0.0.8 -u -P 1 -i 1 -p 5003 -l 1460B -f k -b 4500k -t 65 -T 1`



Graf 71.

Na grafe 71 možno vidieť prenos videa. Minimálna priepustnosť je stanovená na 4,5Mbit/s, ale ako je možné vidieť, kontrolér si udržiava priepustnosť v rozmedzí 4480 – 4500 kbits/s.

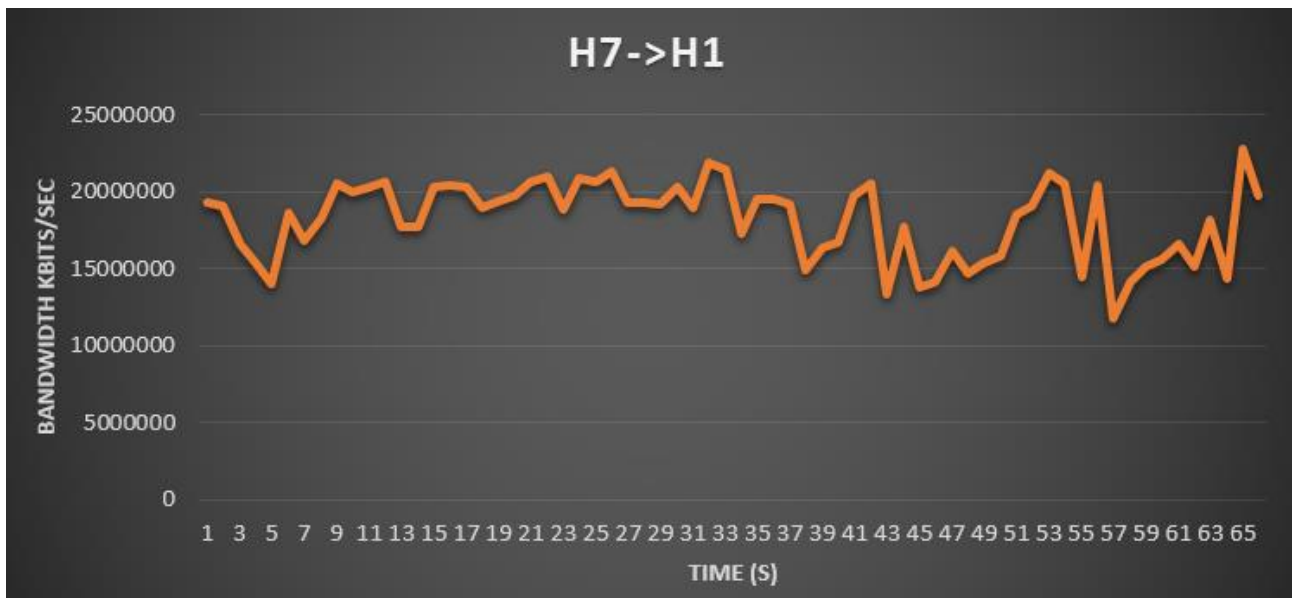
- iperf -c 10.0.0.6 -u -P 1 -i 1 -p 5002 -l 450B -f k -b 500k -t 60



Graf 72.

Na grafe 72 je opäť znázornený skype hovor, a ako môžeme vidieť požadovaná priepustnosť ďaleko prevyšuje stanovenú 100kbit/s.

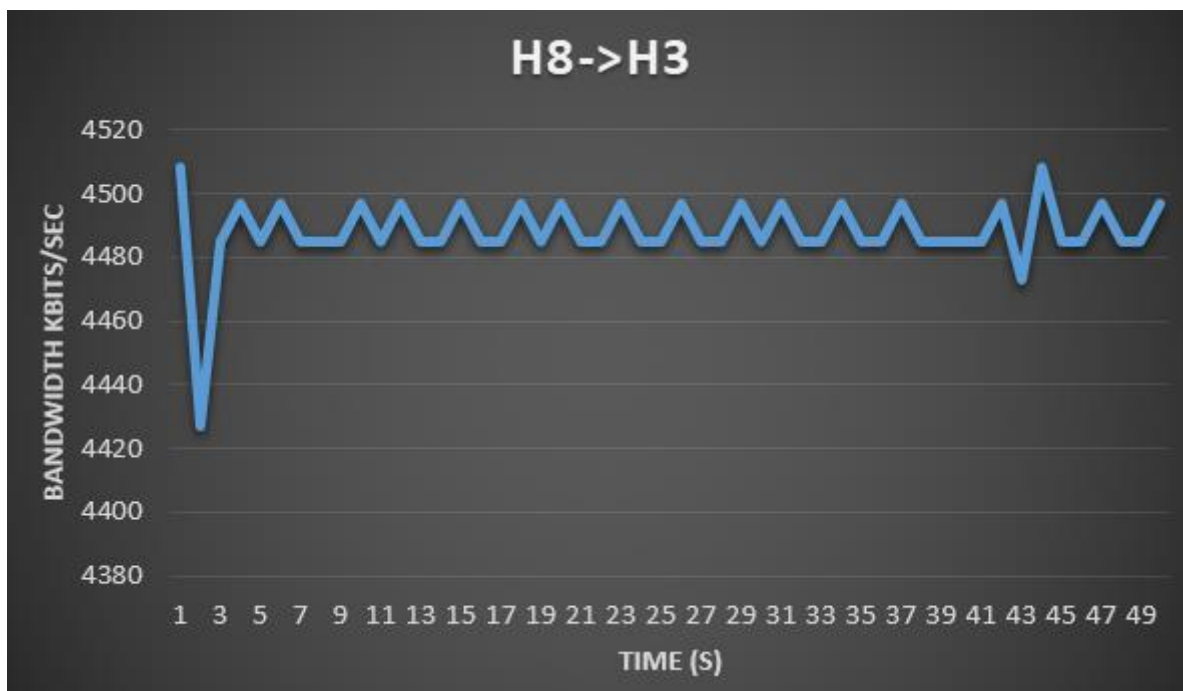
- iperf -c 10.0.0.7 -P 1 -i 1 -p 5001 -f k -t 70 -F /home/bakalarka/Downloads/



Graf 73.

Ako je možné vidieť z grafu 74, priepustnosť v rámci posielania súborov je vskutku vysoká. Dôvodom je, že nebola nastavená priepustnosť liniek a tak sa zobrala najväčšia možná akú systém (virtuálny stroj) povolil. Tento scenár sa opakuje vo všetkých prípadoch s posielaním súborov.

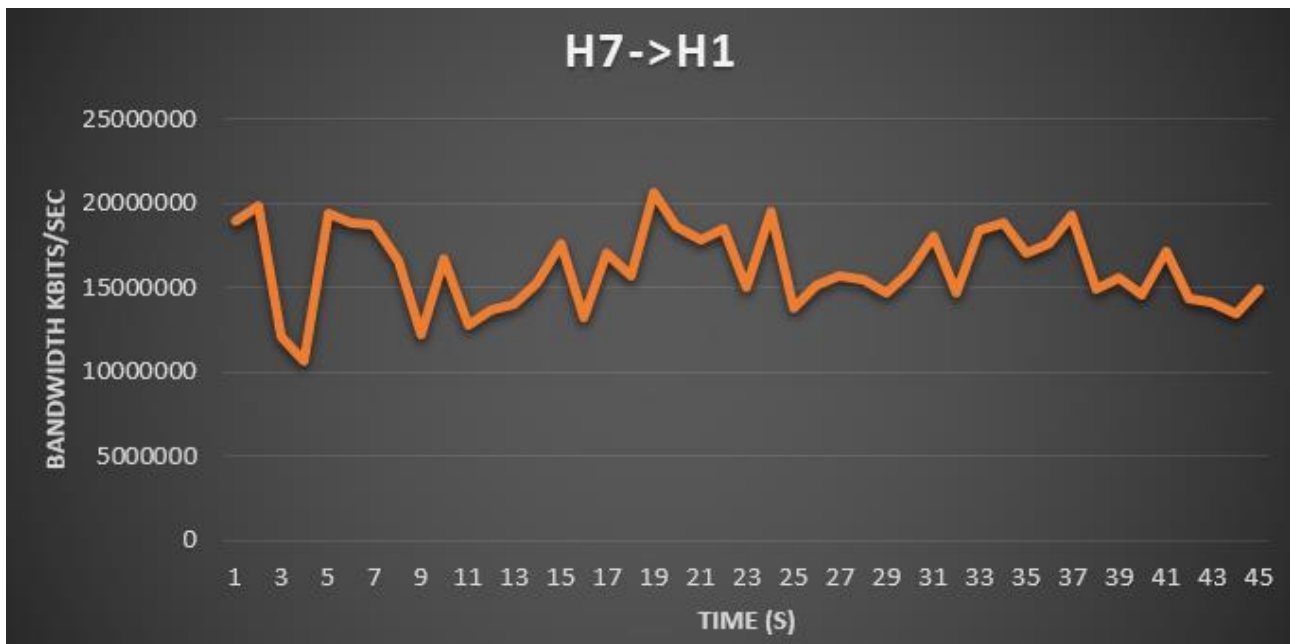
6. `iperf -c 10.0.0.8 -u -P 1 -i 1 -p 5003 -l 1460B -f k -b 4500k -t 50 -T 1`



Graf 74.

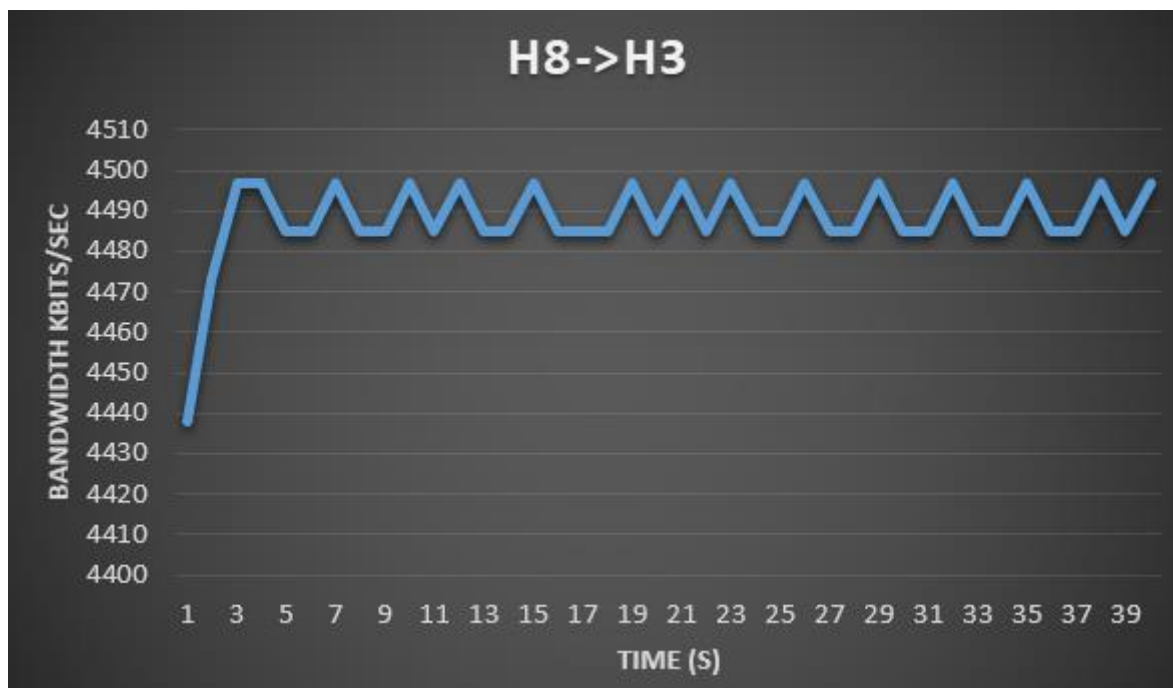
Graf 75 znázorňuje prenos videa, avšak minimálna priepustnosť nie je splnená.

7. `iperf -c 10.0.0.7 -P 1 -i 1 -p 5001 -f k -t 45 -F /home/bakalarka/Downloads/`



Graf 75.

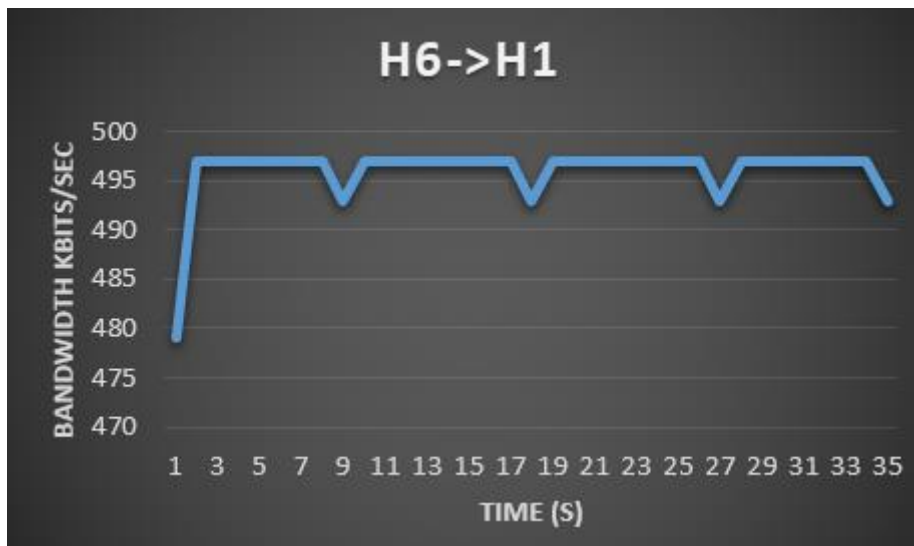
1. `iperf -c 10.0.0.8 -u -P 1 -i 1 -p 5003 -l 1460B -f k -b 4500k -t 40 -T 1`



Graf 76.

Graf 76 znázorňuje prenos videa, avšak minimálna priepustnosť nie je splnená.

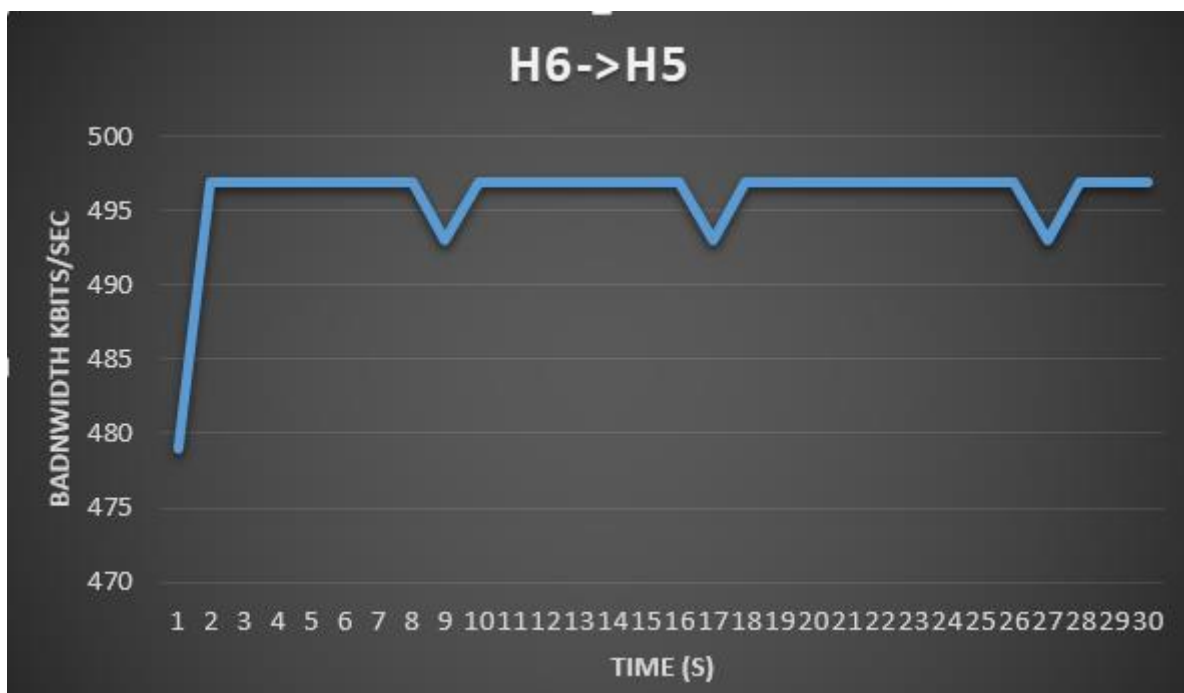
2. `iperf -c 10.0.0.6 -u -P 1 -i 1 -p 5002 -l 450B -f k -b 500k -t 35`



Graf 77.

Graf 77 znázorňuje skype hovor, kde minimálna priepustnosť je dodržaná.

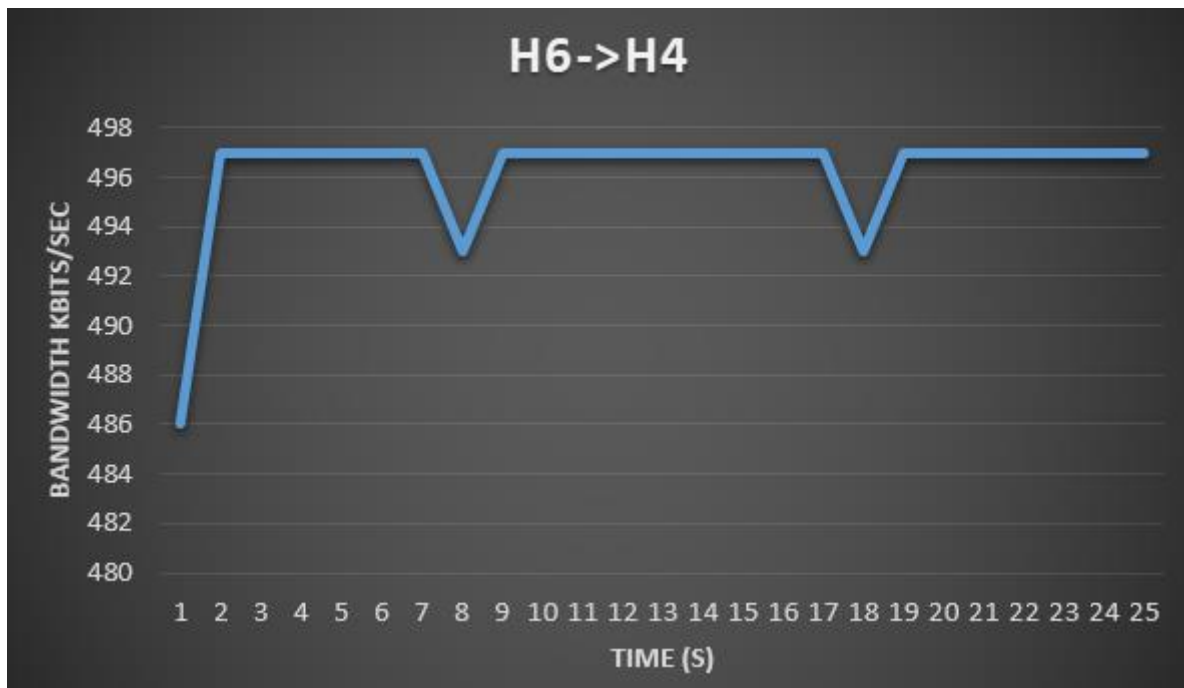
- iperf -c 10.0.0.6 -u -P 1 -i 1 -p 5002 -l 450B -f k -b 500k -t 30



Graf 78.

Graf 77 znázorňuje skype hovor, kde minimálna priepustnosť je dodržaná.

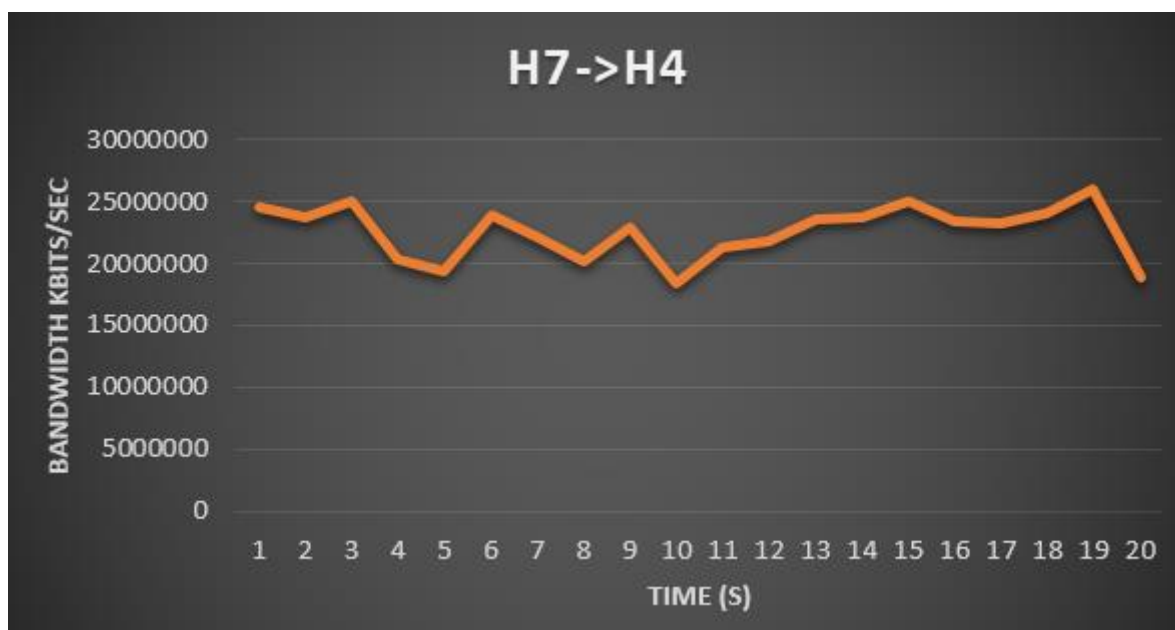
- iperf -c 10.0.0.6 -u -P 1 -i 1 -p 5002 -l 450B -f k -b 500k -t 25



Graf 79.

Graf 77 znázorňuje skype hovor, kde minimálna priepustnosť je dodržaná.

5. `iperf -c 10.0.0.7 -P 1 -i 1 -p 5001 -f k -t 20 -F /home/bakalarka/Downloads/`



Graf 80.

6. Pustené 4 rôzne toky súčasne:

Z dôvodu lepšieho zobrazenia na grafoch, sme sa rozhodli jednotlivé toky rozdeliť do dvoch grafov. H1 a H5 predstavujú toky v grafe 82 a H2 a H4 zas toky v grafe 81. Avšak aj keď sú toky zobrazené v rôznych grafoch, boli spustené v rovnakom čase. Na grafe 81 možno vidieť dvakrát

prenos súboru. Ako už bolo spomínané, na linkách nebolo spustené žiadne obmedzenie veľkosti a preto jednotlivé toky spĺňajú minimálnu priepustnosť 5,5Mbit/s.

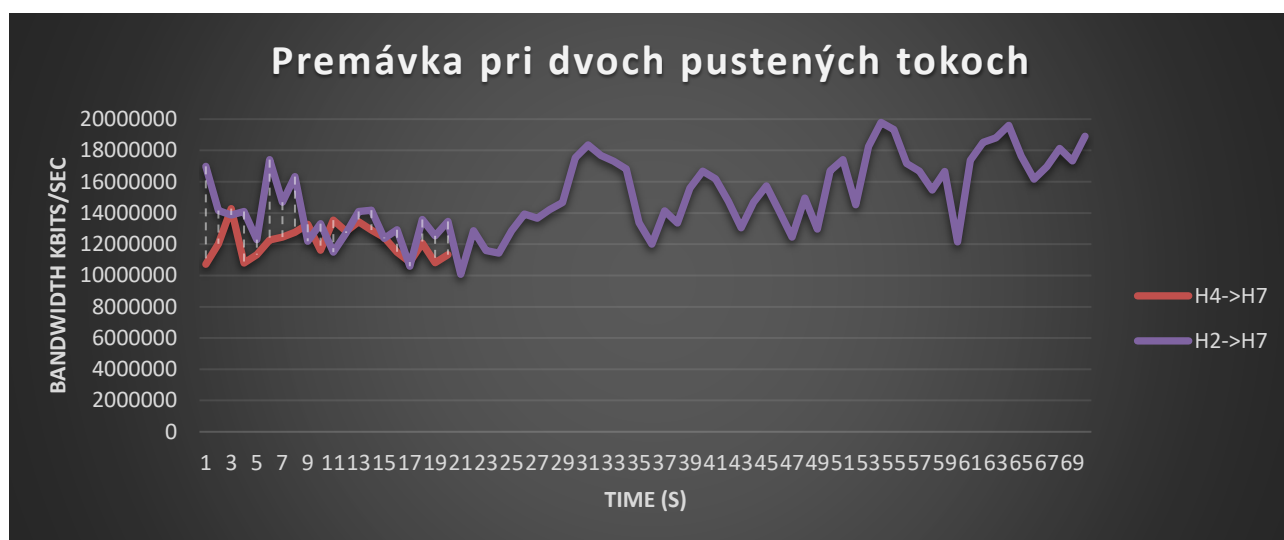
Na grafe 81 možno vidieť UDP komunikáciu – skype hovor. Aj keď zároveň prebiehali dva prenosi súborov, je možné vidieť, že priepustnosť sa ani len nepriblížila minimálnej priepustnosti 100kbit/s.

H1: iperf -c 10.0.0.6 -u -P 1 -i 1 -p 5002 -l 450B -f k -b 500k -t 35

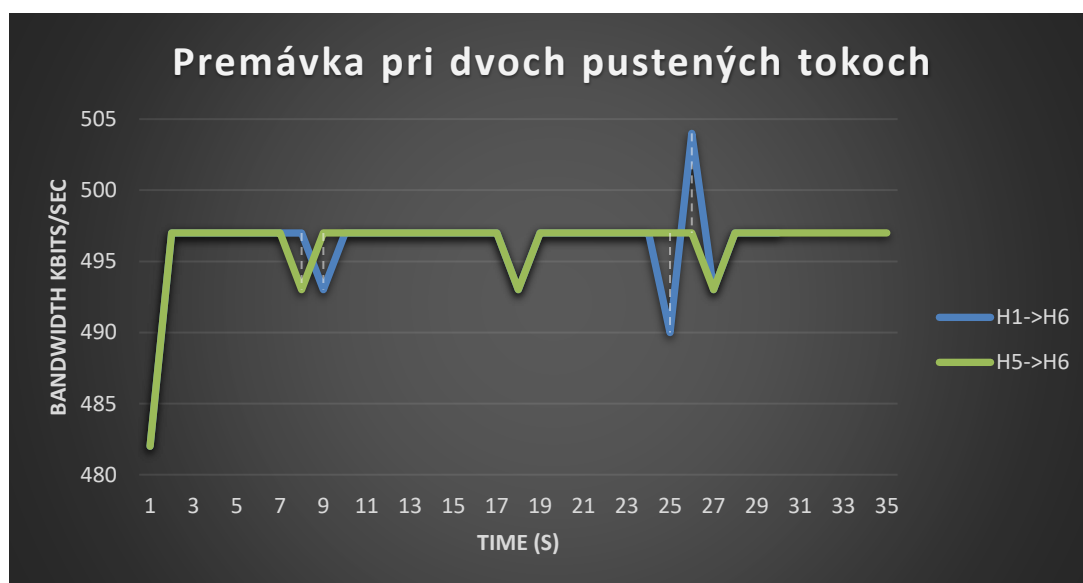
H2:iperf -c 10.0.0.7 -P 1 -i 1 -p 5001 -f k -t 70 -F /home/bakalarka/Downloads/

H4: iperf -c 10.0.0.7 -P 1 -i 1 -p 5001 -f k -t 20 -F /home/bakalarka/Downloads/

H5:iperf -c 10.0.0.6 -u -P 1 -i 1 -p 5002 -l 450B -f k -b 500k -t 30



Graf 81.



Graf 82.