UNIVERZITET U BEOGRADU

ELEKTROTEHNIČKI FAKULTET

Nataša D. Maksić

OPTIMIZACIJA I IMPLEMENTACIJA NAPREDNIH PROTOKOLA ZA RUTIRANJE

Doktorska disertacija

Beograd, 2014.

UNIVERSITY OF BELGRADE
SCHOOL OF ELECTRICAL ENGINEERING

Nataša D. Maksić

OPTIMIZATION AND IMPLEMENTATION OF THE ADVANCED ROUTING PROTOCOLS

Doctoral Dissertation

Belgrade, 2014

MENTOR:

dr Aleksandra Smiljanić, vanredni profesor, Univerzitet u Beogradu, Elektrotehnički fakultet

ČLANOVI KOMISIJE:

dr Irini Reljin, redovni profesor, Univerzitet u Beogradu, Elektrotehnički fakultet

dr Zoran Čiča, docent, Univerzitet u Beogradu, Elektrotehnički fakultet

dr Branislav Todorović, viši naučni savetnik, Istraživačko-razvojni institut RT-RK

DATUM ODBRANE

.

Apstrakt

Ova disertacija u prvom delu analizira protokole rutiranja u topologijama data centara. Posebna pažnja je posvećena dvofaznim protokolima rutiranja koji u pravilnim topologijama data centara sa velikim brojem alternativnih putanja omogućavaju izbegavanje zagušenja u mreži. U disertaciji je predložen novi algoritam za optimizovano dvofazno balansiranje, LB-ECR, koji omogućava bolje iskorišćenje komunikacionih mreža data centra. Korišćenjem metoda linearnog programiranja LB-ECR maksimizuje protok bez gubitaka za date saobraćajne zahteve svičeva. LB-ECR se oslanja na ECMP rutiranje koje je uobičajeno u data centrima jer koristi alternativne putanje iste cene. Dvofazno balansiranje omogućava pojednostavljenje linearnog modela eliminacijom saobraćajnih tokova i smanjuje mogućnost zagušenja raspoređivanjem saobraćaja po manje iskorišćenim linkovima mreže. Pojednostavljenje linearnog modela omogućava njegovo rešavanje za mreže velikih data centara. Disertacija sadrži pregled topologija i načina rutiranja u data centrima i rezultate poređenja različitih algoritama rutiranja u tipičnim topologijama data centara.

Pored optimizacije rutiranja, disertacija razmatra algoritme ažuriranja lukap tabela rutera. Drugi deo disertacije sadrži pregled lukap algoritama i algoritama ažuriranja, i ispitivane su performanse ažuriranja dva napredna lukap algoritma. Izvedene su formule za proračun najgoreg slučaja zauzeća memorija lukap bloka, dok rezultati simulacije pokazuju zauzeće memorije za tipične tabele rutiranja. Prikazani su i rezultati broja pristupa memorijama lukap bloka u toku ažuriranja, kao i kompleksnost algoritama ažuriranja i vreme izvršavanja za tipične tabele rutiranja.

Treći deo disertacije obuhvata opis implementacija algoritama rutiranja sa balansiranjem i algoritama ažuriranja lukap tabela, kao i načina integracije ovih implementacija u okviru rutera.

KLJUČNE REČI:

ruter, data centar, optimalno rutiranje, ažuriranje IP lukap tabela, implementacija

NAUČNA OBLAST:

Telekomunikacije i elektronika

UŽA NAUČNA OBLAST:

Arhitektura svičeva i rutera

UDK BROJ:

621.3

Abstract

The first part of this dissertation presents analysis of the routing protocols in data center topologies. Emphasis is put on two-phase routing protocols which enable congestion avoidance in data center topologies, which are regular and have significant number of alternative paths. This dissertation proposes new two-phase routing algorithm, LB-ECR, which enables better utilization of data center networks. Using the method of linear programming, LB-ECR maximises the loss-free throughput for the given required traffic of the switches. LB-ECR is based on ECMP routing which is common in data centers due to its utilization of alternative equal-cost paths. Two-phase balancing enables simplification of the linear model by eliminating traffic flows and decreases the possibility of congestion by distributing traffic among less used links. Simplification of the linear model simplifies its solution for large data centers. The first part of this dissertation gives an overview of the network topologies and the routing in data centers, and performance comparison of different routing algorithms in typical data center topologies.

In addition to the routing optimization, this dissertation examines the algorithms for updating the lookup tables of Internet routers. The second part of this dissertation provides an overview of the lookup and update algorithms. Updating performance of two advanced lookup algorithms is examined. We develop formulas for the worst-case memory requirements for two fast lookup algorithms, while we show through simulations the memory requirements for typical routing tables. We also evaluate the number of memory accesses to the lookup modules during updates, the complexity of the updating algorithms, as well as their execution time for typical routing tables.

The third part of this dissertation encompasses description of the implementations of the two-phase routing algorithms and of the lookup update algorithms, as well as the integration of these components inside the router.

KEYWORDS:

router, data center, optimal routing, IP lookup table updating, implementation

SCIENTIFIC AREA:

Telecommunications and electronics

SCIENTIFIC SUBAREA:

Switch and router architecture

UDC NUMBER:

621.3

Sadržaj

1 Uvod 1

1.1 Rutiranje u data centrima 2

1.2 Algoritmi za ažuriranje lukap tabela 5

1.3 Implementacija protokola za rutiranje i algoritama za ažuriranje 6

1.4 Organizacija teze 7

2 Napredni protokol za rutiranje u data centrima 8

2.1 Mrežne topologije u data centrima 8

2.1.1 *N*-dimenziona meš topologija 8

2.1.2 Torus 9

2.1.3 Hiperkocka 10

2.1.4 Topologija stabla sa više korena 11

2.1.5 *Fat-tree* topologija 12

2.1.6 *Flattened butterfly* 15

2.1.7 *Dragonfly* 17

2.1.8 BCube 18

2.1.9 DCell 19

2.2 Protokoli za rutiranje u data centrima 21

2.2.1 Rutiranje u meš topologiji i topologijama torusa i hiperkocke 22

2.2.2 Rutiranje u *fat-tree* topologiji i topologiji stabla sa više korenova 24

2.2.3 Rutiranje u *flattened butterfly* topologiji 25

2.2.4 Rutiranje u *dragonfly* topologiji 25

2.2.5 Rutiranje u BCube i DCell topologijama 26

2.2.6 *Multipath* TCP i *Hedera* 27

2.2.7 Adaptivni protokoli rutiranja 28

2.3 Optimizacija iskorišćenja linkova korišćenjem dvofaznih protokola za rutiranje ... 29

2.3.1 Karakteristike Valiant rutiranja ... 31

2.3.2 Optimizacija dvofaznog balansiranja ... 34

2.3.3 Dvofazno balansiranje u mreži sa rutiranjem po najkraćim putanjama ... 36

2.3.4 Dvofazno balansiranje u mreži sa ECMP rutiranjem ... 38

2.3.5 SPRm i LB-SPRm rutiranja ... 39

2.4 Poređenje algoritama rutiranja u datacentrima ... 39

2.4.1 Analiza najgoreg slučaja opterećenja linka ... 40

2.4.1.1 Zavisnost kvaliteta dvofaznog balansiranja od osnovnog rutiranja ... 44

2.4.2 Simulacija rutiranja u različitim topologijama data centara ... 45

2.4.2.1 Formiranje simulacije u mrežnom simulatoru *ns-3* ... 45

2.4.2.2 Simulacija rutiranja sa balansiranjem u mrežnom simulatoru *ns*-3 ... 47

2.4.2.3 Poređenje protokola za rutiranje u data centrima kroz simulacije ... 49

2.4.2.4 Analiza simulacionih krivih u oblasti gubitaka paketa ... 53

3 Ažuriranje lukap tabela ... 55

3.1 Lukap procedure i ažuriranje ... 56

3.1.1 TCAM lukap procedura ... 57

3.1.1.1 Struktura TCAM lukap modula ... 57

3.1.1.2 Ažuriranje TCAM lukap modula ... 59

3.1.2 Lukap procedure sa bitskim stablima ... 59

3.1.2.1 Jednobitsko stablo ... 60

3.1.2.2 Multibitsko stablo ... 61

3.1.2.3 Kompresija putanja u stablu ... 62

3.1.2.4 Potiskivanje ruta u listove stabla ... 62

3.1.2.5 Formiranje skupa nepreklapajućih prefiksa ... 63

3.1.2.6 *Lulea* lukap procedura 63
3.1.2.6.1 Ažuriranje *Lulea* lukap modula 64
3.1.2.7 *Tree Bitmap* lukap procedura 66
3.1.2.7.1 Ažuriranje *Tree Bitmap* lukap modula 67
3.1.2.8 Pretraga sa prioritetnim stablom 69
3.1.2.8.1 Ažuriranje prioritetnog stabla 70
3.1.2.9 POLP lukap procedura 72
3.1.2.9.1 Procedura ažuriranja za POLP lukap algoritam 74
3.1.2.10 BPFL lukap procedura 76
3.1.2.10.1 Procedura ažuriranja za BPFL lukap algoritam 78
3.1.3 Pretraga po dužinama prefiksa 82
3.1.3.1 Ažuriranje modula za pretragu po dužinama prefiksa 84
3.1.3.2 Ubrzanje pristupa heš tabelama korišćenjem Blumovih filtara 85
3.1.3.2.1 Ažuriranje Blumovih filtara 86
3.1.4 Pretraga po granicama opsega prefiksa 87
3.1.4.1 Ažuriranje granica opsega prefiksa 88
3.2 Analiza performansi 89
3.2.1 Analiza najgoreg slučaja memorijskih zahteva za IPv6 tabele rutiranja podrazumevajući raspodelu prefiksa na Internetu 89
3.2.2 Analiza najgoreg slučaja za memorijske zahteve za slučaj POLP lukap algoritma 90
3.2.3 Analiza najgoreg slučaja za memorijske zahteve za slučaj BPFL lukap algoritma 94
3.2.4 Memorijski zahtevi 95
3.2.5 Broj upisa 98
3.2.6 Vreme izvršavanja 101
4 Implementacija za prototip internet rutera 103

4.1 Implementacija protokola za rutiranje 103
4.1.1 Organizacija implementacije rutiranja sa balansiranjem 104
4.1.2 Organizacija programskog koda *ospfd* implementacije 105
4.1.3 Razmena informacija između rutera 107
4.1.4 Optimizacioni modul 111
4.1.4.1 Proračun putanja paketa kroz mrežu 112
4.1.4.2 Proračun opterećenosti linkova 115
4.1.4.3 Formiranje i rešavanje optimizacionog modela 120
4.1.5 Usmeravanje paketa u mreži sa balansiranjem 124
4.1.6 Organizacija programskog koda 128
4.1.6.1 Implementacija rutiranja sa balansiranjem u okviru *ospfd* programa 128
4.2 Implementacija algoritama ažuriranja 131
4.2.1 Algoritam ažuriranja za BPFL lukap tabele 132
4.2.1.1 Brisanje rute 133
4.2.1.2 Dodavanje rute 135
4.2.1.3 Premeštanje čvorova između balansiranih stabala 139
4.2.1.4 Organizacija programskog koda 140
4.2.2 Algoritam ažuriranja za POLP lukap tabele 141
4.2.2.1 Brisanje rute 142
4.2.2.2 Dodavanje rute 145
4.2.2.3 Proračun celog pajplajna 148
4.2.2.4 Organizacija programskog koda 150
4.2.3 Algoritam ažuriranja lukap bloka sa prioritetnim stablom 151
4.2.3.1 Organizacija programskog koda 154
4.3 Integracija softverskih komponenti u okviru rutera 155

4.3.1 Integracija softvera za rutiranje i softvera za ažuriranje 155

4.3.2 Povezivanje ravni podataka i kontrolne ravni 157

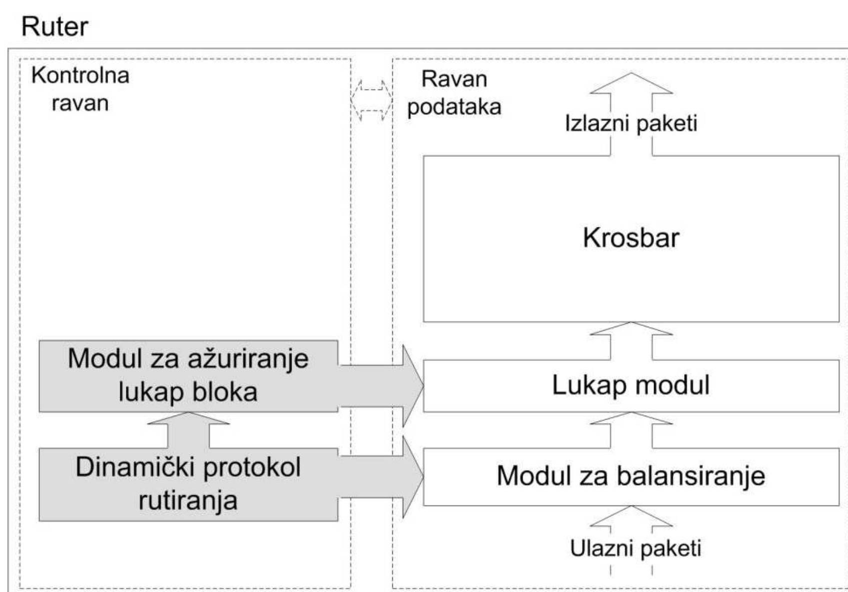
5 Zaključak 159

Reference 161

Biografija kandidata 166

1 Uvod

Ruteri visokih performansi koriste najnovija tehnička rešenja da bi ostvarili što veći protok, sa stalnom težnjom za smanjenjem cene rutera i utroška energije. Dva važna elementa kontrolne ravni rutera su protokol rutiranja i algoritam ažuriranja lukap bloka. Položaj blokova protokola rutiranja i ažuriranja lukap bloka u okviru rutera prikazani su na Slici 1.



Slika 1 Implementacija algoritama rutiranja i ažuriranja lukap modula u okviru rutera

Protokol rutiranja utiče na raspodelu saobraćaja u mreži i time i na mogućnosti izbegavanja zagušenja u mreži, koja ključno utiču na sposobnost mreže da prenese potrebnu količinu podataka u zahtevanom vremenu. U toku razvoja paketskih telekomunikacionih mreža različite osobine mreža dovele su do formiranja različitih metoda rutiranja. Evoluciono formiranje Internet mreža i predvidivost njihovog saobraćaja rezultovalo je skupom protokola rutiranja u kojima je raspodela saobraćaja u mreži značajnim delom posledica planiranja i usmeravanja od strane operatera mreže. U savremenim telekomunikacijama, potreba za naprednim protokolima rutiranja javlja se u mrežama data centara, koje sadrže veliki broj računara sa velikim i brzo promenljivim saobraćajnim zahtevima i rutere sa velikim brojem brzih linkova.

Prosleđivanje paketa u ruterima sa velikim brojem brzih linkova zahteva korišćenje specijalizovanih lukap modula za brzo pronalaženje izlaznog porta paketa koji prolaze kroz ruter. Da bi omogućili brzo pronalaženje izlaznog porta i za tabele koje sadrže po nekoliko stotina hiljada prefiksa, zadržavajući pri tome male memorijske zahteve i ekonomičnost, lukap moduli koriste optimizovane strukture podataka. Algoritmi ažuriranja lukap bloka imaju zadatak da tabelu rutiranja transformišu u strukture podataka lukap bloka. Upisi u memorije lukap bloka mogu dovesti do privremenog prekida lukap procesa, pa u brzim ruterima, koji imaju velike paketske protoke, algoritam ažuriranja treba da minimizuje broj upisa u lukap blok. Ako su u okviru lukap bloka mogući različiti zapisi tabele rutiranja, algoritam ažuriranja treba da formira strukture podataka lukap bloka tako da se, u mogućoj meri, izbegnu kasnije promene zapisa pri dodavanju novih ruta. Algoritam ažuriranja treba i da bude efikasan i da ažuriranje lukap tabele izvrši u potrebnom vremenu uz što manje opterećenje procesora.

Ova disertacija se bavi optimizacijom i implementacijom naprednih protokola rutiranja. Ona obuhvata analizu postojećih protokola rutiranja u data centrima i predlog novog protokola koji omogućava optimalno rutiranje u data centrima, prilagođeno osobinama topologija, broja elemenata i osobinama saobraćaja data centara. Disertacija takođe obuhvata analizu osobina funkcija ažuriranja različitih tipova lukap bloka rutera visokih performansi. Uticaj funkcije ažuriranja na dizajn rutera visokih performansi analiziran je kroz definisanje najgoreg slučaja zauzeća memorije lukap bloka, i analizu broja pristupa memoriji lukap bloka, vremena izvršavanja ažuriranja i zauzeća memorije lukap bloka. U disertaciji su opisane implementacije izabranih algoritama rutiranja i algoritma ažuriranja lukap bloka rutera visokih performansi.

1.1 Rutiranje u data centrima

Oblast primene rutera koja brzo napreduje i postavlja nove ciljeve u unapređenju performansi rutera su data centri. Koncentracija velikog broja računara u data centru sa ciljem postizanja zahtevanih performansi obrade podataka zahteva formiranje telekomunikacione mreže koja povezuje te računare i omogućava njihovu međusobnu komunikaciju sa velikim protocima podataka i malim kašnjenjem. Broj servera u

jednom data centru dostiže stotine hiljada i grade se sve veći data centri. Unapređenje rutiranja u data centrima je važan zadatak savremenih telekomunikacija.

Broj servera u data centru i njihovi komunikacioni zahtevi rezultuju visokom cenom telekomunikacione mreže data centra. Ova mreža treba da sadrži linkove i svičeve velikih protoka da bi omogućila potrebne performanse. Sa druge strane, postojeći algoritmi rutiranja saobraćaja u data centrima nemaju mogućnost iskorišćenja svih resursa mreže i dešava se da zbog lokalnih zagušenja serveri ne mogu komunicirati potrebnom brzinom iako bi preusmeravanjem dela saobraćaja na druge putanje u mreži ta komunikacija bila moguća. Unapređenje rutiranja koje bi omogućilo bolje iskorišćenje telekomunikacione mreže donelo bi značajno smanjenje cene telekomunikacione mreže i poboljšanje performansi data centara.

Serveri i telekomunikaciona mreža data centara su locirani na malom prostoru i iz tog razloga performanse mreže se poboljšavaju dodavanjem velikog broja linkova između svičeva. Na taj način se formiraju alternativne putanje sa jednakom cenom. Ove putanje su pogodne za paralelni prenos podataka.

Postojanje velikog broja alternativnih putanja u topologijama data centara omogućava izbegavanje zagušenja u mreži korišćenjem dvofaznog balansiranja. Istorija dvofaznog rutiranja počinje Valiant rutiranjem [1], koje je predloženo sa ciljem ograničenja zagušenja pri rutiranju, nezavisno od rasporeda komunikacionih parova. Valiantovo rutiranje bira sa jednakom verovatnoćom jedan od rutera u mreži kao odredište prve faze balansiranja. Na taj način Valiantovo rutiranje postiže ravnomeran raspored saobraćaja u mreži i izbegavanje tačaka zagušenja koje mogu nastati nepovoljnim rasporedom komunikacionih parova. U cilju unapređenja performansi dvofaznog balansiranja predložen je optimizacioni model baziran na dvofaznom balansiranju za mreže u kojima su definisani maksimalni izlazni i ulazni saobraćaj rutera [2]. U ovom modelu, rezultat optimizacije su putanje između krajnjih tačaka prve i druge faze rutiranja, kao i verovatnoće izbora međurutera. U ovoj disertaciji su analizirane performanse standardnih i dvofaznih protokola za rutiranje u tipičnim topologijama data centara.

Ova disertacija predlaže, opisuje i analizira algoritam rutiranja LB-ECR (eng. *Load Balanced Equal-Cost Routing*), koji omogućava optimalno iskorišćenje linkova

telekomunikacione mreže sa alternativnim putanjama jednake cene, korišćenjem dvofaznog balansiranja. LB-ECR optimizacija vrši se metodom linearnog programiranja. Dvofazno balansiranje omogućava eliminaciju zavisnosti linearnog modela od saobraćajnih tokova i na taj način omogućava formiranje praktično rešivog matematičkog modela za optimizovano rutiranje u mrežama data centara. U okviru tog modela, zavisnost od liste tokova ili zavisnost od matrice saobraćaja, koja definiše intenzitet saobraćaja svih komunikacionih parova, svodi se na zavisnost od dva vektora saobraćaja, koji definišu ukupni generisani i terminirani intenzitet saobraćaja čvorova mreže. Saobraćaj data centara karakteriše veliki broj brzo promenljivih tokova, i optimizacija saobraćaja bazirana na tokovima zahteva proračune i operacije koji se sa današnjom tehnologijom ne mogu izvršiti u realnom vremenu. Korišćenjem dvofaznog balansiranja, optimizacija se vrši na osnovu topologije mreže, ECMP putanja paketa i ukupnih saobraćajnih zahteva čvorova mreže, i rezultujuće formule modela su jednostavne i sastoje se od jedne nejednačine za svaki link i jedne dodatne jednačine koja definiše zbir verovatnoća izbora balansirajućih rutera. ECMP rutiranje na kome se LB-ECR bazira dominantno se koristi u data centrima pošto omogućava korišćenje alternativnih putanja koje su brojne u data centrima. U kombinaciji sa optimizovanim dvofaznim balansiranjem postiže se preusmeravanje saobraćaja preko manje iskorišćenih linkova i za date saobraćajne zahteve svičeva postiže se maksimizacija protoka bez gubitaka u mreži data centra. U okviru ove disertacije, LB-ECR rutiranje je implementirano i predstavljeni su rezultati simulacije rutiranja u topologijama data centara koji potvrđuju rezultate analize i daju detaljniji uvid u ponašanje saobraćaja u mreži.

U okviru prve celine ove disertacije opisane su i razmotrene topologije data centara kao i algoritmi rutiranja korišćeni u data centrima. Razmotrene su meš topologija, torus, hiperkocka, stablo sa više korena, *fat-tree*, *flattened butterfly*, *dragonfly*, BCube i DCell. Performanse LB-ECR rutiranja poređene su sa performansama dve varijacije direktnog rutiranja najkraćim putanjama, direktnim rutiranjem putanjama jednakim cenama, LB-SPR rutiranjem, kao i Valiant balansiranjem. Ispitivanja su vršena na *fat-tree*, *flattened butterfly* i *dragonfly* topologijama. U svim merenim slučajevima analitički je pronađena matrica saobraćaja koja pri najmanjem protoku dovodi do zagušenja u mreži, i LB-ECR je u svim merenim

slučajevima ili imao najveću vrednost protoka pri kome nastaje zagušenje, ili je delio najveću vrednost sa još nekim od protokola. Merenja protoka za ove matrice saobraćaja su potvrdila proračunate vrednosti protoka pri kojima nastupaju gubici paketa, i dala uvid u ponašanje protoka u oblasti gubitaka paketa.

1.2 Algoritmi za ažuriranje lukap tabela

Druga celina ove disertacije obuhvata algoritme ažuriranja lukap tabela rutera. Ova celina sadrži pregled značajnih lukap algoritama i njihovih algoritama ažuriranja. Pregledom su obuhvaćeni TCAM lukap, *Lulea*, *Tree Bitmap*, prioritetno stablo, POLP, BPFL, pretraga po dužinama prefiksa i po granicama opsega prefiksa. Rad sadrži ispitivanje memorijskih zahteva, broja pristupa lukap modulu i vremena izvršavanja ažuriranja za BPFL i POLP lukap algoritme, koji su predstavnici paralelizovanih lukap algoritama visokih performansi.

U okviru analize memorijskih zahteva definisane su formule za proračun najgoreg slučaja memorijskih zahteva BPFL i POLP algoritama. Ove formule omogućavaju određivanje veličina potrebnih memorija za najgori slučaj vrednosti prefiksa za zadatu raspodelu dužina prefiksa. Simulacije memorijskih zahteva izvršene su za IPv4 i IPv6 tabele rutiranja. Za IPv4 rutiranje simulacije su izvršene sa tabelama rutiranja koje postoje u Internet mreži. IPv6 tabele rutiranja na Internetu još se uvećavaju, pa su pri simulaciji IPv6 tabele rutiranja generisane na osnovu postojećih velikih IPv4 tabela rutiranja. Pored generisanja IPv6 tabela rutiranja koje odgovaraju budućim praktičnim slučajevima, generisne su i tabele koje za istu raspodelu prefiksa zauzimaju maksimalnu količinu memorije, i izvršeno je poređenje očekivanog i maksimalnog zauzeća memorije.

Ažuriranje informacija o rutama u memorijama lukap modula utiče na pristup tim memorijama u toku prosleđivanja paketa, i može prouzrokovati pauze u prosleđivanju paketa. Ruteri velikih brzina prosleđuju veliki broj paketa u jedinici vremena i pauze u prosleđivanju moraju biti ograničenog trajanja i učestanosti, pa je merenje broja upisa u memoriju lukap modula pri ažuriranju značajno. U radu su predstavljeni podaci o merenom broju upisa u memorije za BPFL i POLP module.

Algoritmi ažuriranja izvršavaju se na procesoru ugrađenom u ruter, i vreme izvršavanja algoritma zavisi od kompleksnosti algoritma i od performansi procesora. Zahtevano vreme izvršavanja ovih algoritama određeno je brzinom promena tabela rutiranja u mreži, i treba da bude kraće od brzine propagacije rute da bi ažuriranje moglo da prati uzastopne promene ruta. Da bi ovaj uslov bio ispunjen, algoritmi veće kompleksnosti treba da se izvršavaju na bržim procesorima, što utiče na cenu rutera, i iz tog razloga je u radu predstavljena kompleksnost algoritama ažuriranja BPFL i POLP memorija i rezultati merenja vremena izvršavanja ovih algoritama.

1.3 Implementacija protokola za rutiranje i algoritama za ažuriranje

Treća celina u okviru rada je opis implementacije algoritama rutiranja sa balansiranjem i algoritama ažuriranja lukap tabela. Ove implementacije se mogu koristiti kao važan deo softverskih ili hardverskih rutera.

Opisani su algoritmi za formiranje i rešavanje optimizacionog modela za rutiranja sa balansiranjem. Proračun rutiranja sa balansiranjem počinje proračunom putanja paketa korišćenjem osnovnog rutiranja u mreži, koje određuje putanje paketa od izvorišta ka balansirajućim ruterima i od balansirajućih rutera ka odredištima. Opisani su algoritmi za proračun putanja kroz mrežu za rutiranje najkraćom putanjom i rutiranje putanjama sa jednakim cenama. Rezultati ovih algoritama su ulazni podaci za proračun opterećenja linkova koji generiše opterećenja linkova za sve moguće komunikacione parove. Opterećenja linkova omogućavaju generisanje formula optimizacionog modela, koji rezultuje verovatnoćama izbora rutera za balansiranje.

Pored verovatnoća izbora balansirajućih rutera, za balansiranje su svakom ruteru potrebne i adrese balansirajućih rutera na koje treba slati pakete. Ove adrese generiše algoritam koji proračunava putanje kroz mrežu korišćenjem osnovnog rutiranja. Algoritam beleži adrese portova na proračunatim putanjama u slučaju kada je balansirajući ruter jedan od rutera na putanji od izvorišnog rutera. Ove adrese se koriste kao adrese balansiranja, obezbeđujući tako da paket do balansirajućeg rutera dolazi putem koji je predviđen proračunom verovatnoća balansiranja. Verovatnoće balansiranja i adrese balansiranja su dovoljni podaci za rutiranje sa balansiranjem.

Pored algoritama rutiranja sa balansiranjem implementirani su i algoritmi ažuriranja lukap modula, za BPFL i POLP lukap module i za lukap modul sa priroritenim stablom. Za svaki od lukap modula opisani su algoritmi brisanja i dodavanja ruta. Zbog složenijih struktura lukap modula, BFPL i POLP algoritmi zahtevaju komplikovanije algoritme ažuriranja, i preraspodelu podataka u okviru lukap modula po potrebi. Implementacije ovih preraspodela podataka predstavljaju celine u okviru algoritama brisanja i dodavanja ruta i opisane su u posebnim odeljcima.

Integracija algoritama rutiranja, algoritama ažuriranja lukap tabela i samih lukap tabela omogućava funkcionisanje opisanih algoritama u okviru rutera. U okviru ove disertacije opisan je način integracija ovih komponenti u okviru kontrolne ravni zasnovane na *Linux* operativnom sistemu. Položaj opisanih implementacija u okviru kontrolne ravni rutera prikazan je na Slici 1. Rutiranje sa balansiranjem predstavlja nadogradnju dinamičkog protokola rutiranja koji obezbeđuje osnovno rutiranje u mreži. Od osnovnih rutiranja razmatrana su rutiranje najkraćom putanjom i rutiranje putanjama sa jednakim cenama. Rezultat proračuna osnovnog rutiranja je tabela rutiranja koju modul za ažuriranje lukap bloka pretvara u strukture podataka lukap bloka rutera. Sa druge strane, na osnovu putanja proračunatih osnovnim protokolom rutiranja proračunavaju se opterećenosti linkova i formira optimizacioni model za proračun verovatnoća izbora balansirajućih rutera. Podaci za balansiranje u ravni podataka utiču na usmeravanje paketa prvo ka balansirajućem ruteru, a zatim ka odredištu.

1.4 Organizacija teze

Ova disertacija sadrži pet poglavlja i tri navedene celine disertacije predstavljene su u drugom, trećem i četvrtom poglavlju. Prvo poglavlje ove disertacije sadrži uvod. Dugo poglavlje obuhvata opis topologija i rutiranja u data centrima, predstavljanje LB-ECR algoritma rutiranja i analitičkih i simulacionih rezultata poređenja protokola rutiranja u izabranim topologijama. Treće poglavlje sadrži opis lukap algoritama, algoritama ažuriranja lukap tabela i analitičke i simulacione rezultate performansi algoritama ažuriranja za BPFL i POLP lukap algoritme. Četvrto poglavlje obuhvata opis implementacija algoritama rutiranja sa balansiranjem, opis implementacija algoritama ažuriranja i opis integracije ovih algoritama u okviru rutera. Peto poglavlje sadrži zaključak.

2 Napredni protokol za rutiranje u data centrima

Data centri se odlikuju stalnim rastom broja servera. Savremeni data centri sadrže do nekoliko stotina hiljada računara i stalno se grade novi od kojih mnogi imaju više servera od svakog prethodnog [1]. Uz veliki broj računara, data centri sadrže i veliki broj kablova i svičeva, kao i sisteme za napajanje i hlađenje. Pored visoke cene ugrađene infrastrukture, data centri imaju i visoku cenu korišćenja koja je značajnim delom posledica velike potrošnje električne energije, čija se cena meri milijardama dolara godišnje [5].

Ogromna moć obrade i skladištenja podataka opravdavaju visoku cenu data centara. Svrha mrežne infrastrukture u data centrima koju čine kablovi i svičevi je da omogući komunikaciju među serverima data centra, sa zahtevanim protokom i kašnjenjem [6]. Sami serveri komuniciraju na složen način što u nekim slučajevima dovodi do formiranja zagušenja u nekim delovima mrežne infrastrukture, dok su drugi delovi nedovoljno iskorišćeni. Jedno rešenje za prevazilaženje zagušenja u mreži je instalacija mrežne infrastrukture većeg kapaciteta da bi se zadovoljili saobraćajni zahtevi servera. Drugo rešenje je poboljšanje rutiranja saobraćaja u mreži, tako da se mrežna infrastruktura manjeg kapaciteta bolje iskoristi drugačijim rutiranjem saobraćaja u mreži. S obzirom na veliku ukupnu cenu data centara, poboljšanja u rutiranju omogućavaju velike uštede u ceni formiranja i korišćenja data centara.

U ovom poglavlju biće predstavljene najznačajnije topologije data centara i načini rutiranja u njima. Zatim će biti predstavljen optimizacioni model koji omogućava usmeravanje saobraćaja u mreži data centra sa ciljem ravnomernog iskorišćenja saobraćajne infrastrukture i izbegavanja tačaka zagušenja mreže.

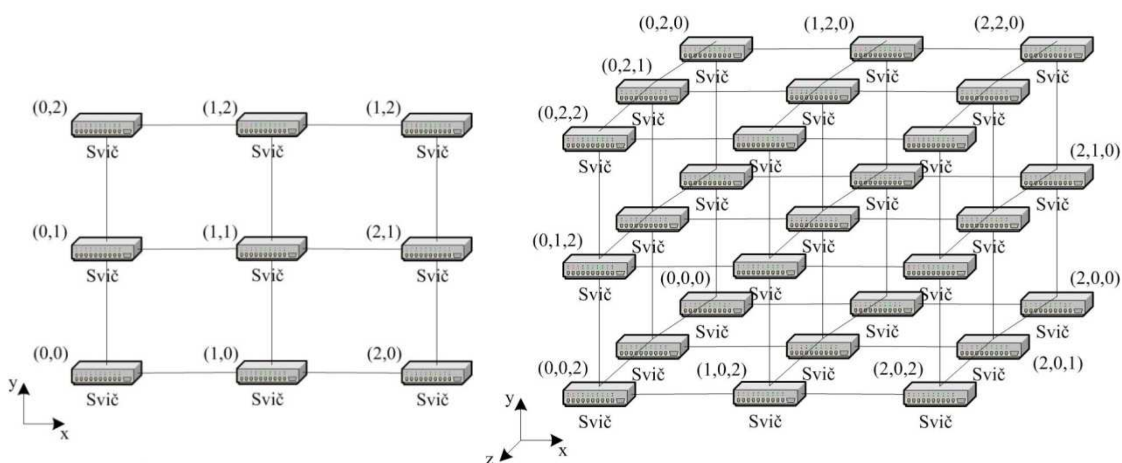
2.1 Mrežne topologije u data centrima

2.1.1 *N*-dimenziona meš topologija

Meš (eng. *mesh*) topologija korišćena u data centarima [1] definiše se brojem dimenzija i brojem elemenata u svakoj od dimenzija. Na Slici 2-1 prikazani su primeri

dvodimenzione i trodimenzione meš topologije, sa po tri elementa u svakoj dimenziji. U meš topologiji je svaki svič u svakoj dimenziji povezan sa svojim susedima.

Svičevi u meš topologiji sa dve dimenzije imaju do četiri veze sa drugim svičevima, dok svičevi u meš topologiji sa tri dimenzije imaju do šest veza. Mali broj veza između svičeva čini meš topologiju pogodnim kandidatom ukoliko korišćeni svičevi imaju mali broj portova. Svičevi su imali mali broj portova u početku razvoja data centara. Međutim, i tada je topologija torusa, nastala nadogradnjom meš topologije, bila zastupljenija, zato što smanjuje maksimalno rastojanje između svičeva u mreži.

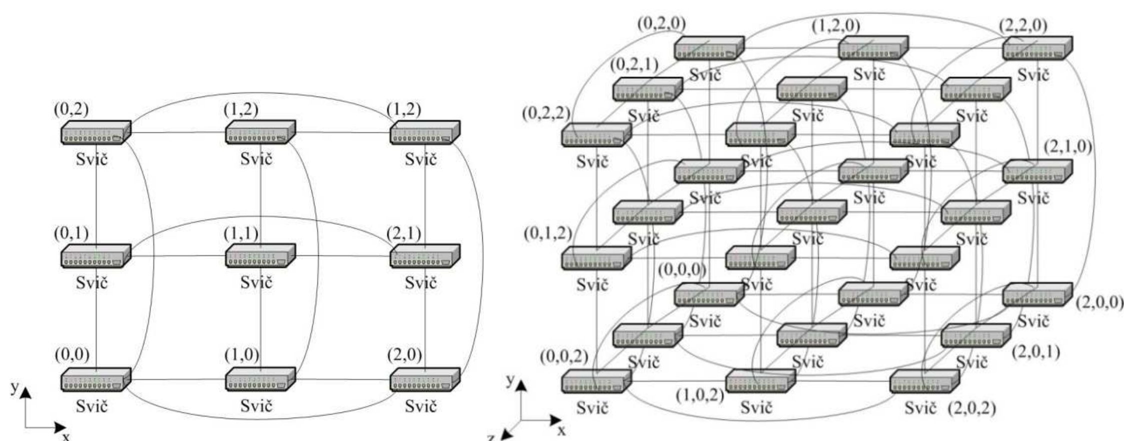


Slika 2-1 Dvodimenziona i trodimenziona meš topologija sa tri elementa u svakoj dimenziji

2.1.2 Torus

Topologija torusa [1] je slična meš topologiji opisanoj u prethodnom odeljku. Kao u meš mreži, i u torusu je svaki svič povezan sa svojim susedima u svakoj dimenziji. U odnosu na meš, torus ima dodatne veze koje povezuju i krajnje svičeve u svakoj dimenziji. Povezivanjem krajnjih svičeva smanjuje se maksimalno rastojanje između dva sviča u mreži.

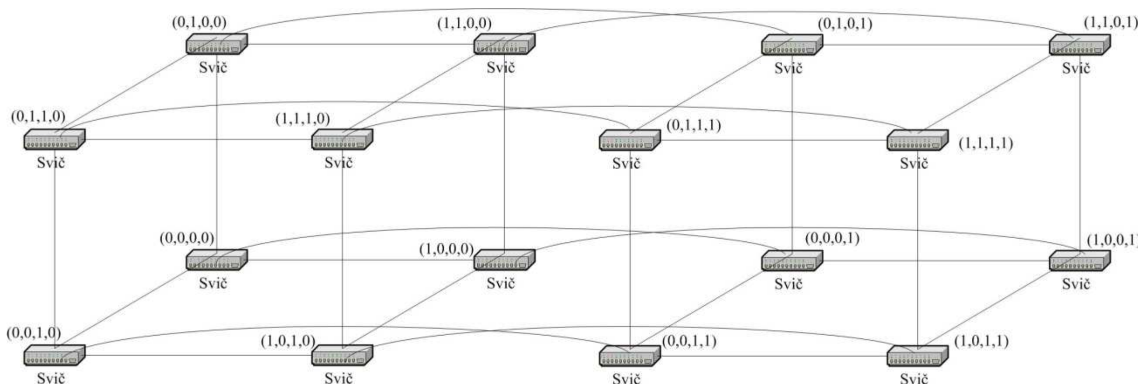
Dvodimenziona i trodimenziona torus topologija sa po tri elementa u svakoj dimenziji prikazane su na Slici 2-2. U topologiji torusa maksimalno rastojanje između svičeva u bilo kojoj od dimenzija jednako je celebrojnoj vrednosti polovine broja svičeva u toj dimenziji. Ovo rastojanje je duplo manje od maksimalnog rastojanja u meš mreži.



Slika 2-2 Dvodimenziona i trodimenziona torus topologija sa tri elementa u svakoj dimenziji

Torus u dve i tri dimenzije je pored *fat-tree* topologije i hiperkocke jedna od topologija koje se tradicionalno koriste u data centrima [7]. Ova topologija bila je popularnija dok su svičevi imali relativno mali broj portova.

2.1.3 Hiperkocka



Slika 2-3 Hiperkocka sa četiri dimenzije

Pravilna topologija koja omogućava korišćenje svičeva sa većim brojem portova je hiperkocka [1]. Za razliku od torusa koji se koristi sa dve i tri dimenzije sa svičevima koji imaju manji broj portova, hiperkocka se koristi sa više dimenzija i sa svičevima koji imaju više portova.

Na Slici 2-3 je prikazana hiperkocka sa četiri dimenzije. Svičevi hiperkocke se povezuju tako što se svako teme jedne hiperkocke niže dimenzije povezuje sa temenom druge hiperkocke niže dimenzije na istoj poziciji. Rezultat je da su povezani svi svičevi

čije se adrese razlikuju u jednoj cifri identifikatora sviča predstavljenog u brojnom sistemu sa osnovom dva. Topologija generalizovane hiperkocke predstavlja uopštenje topologije hiperkocke u kome brojni sistem korišćen za numerisanje svičeva može imati proizvoljnu osnovu.

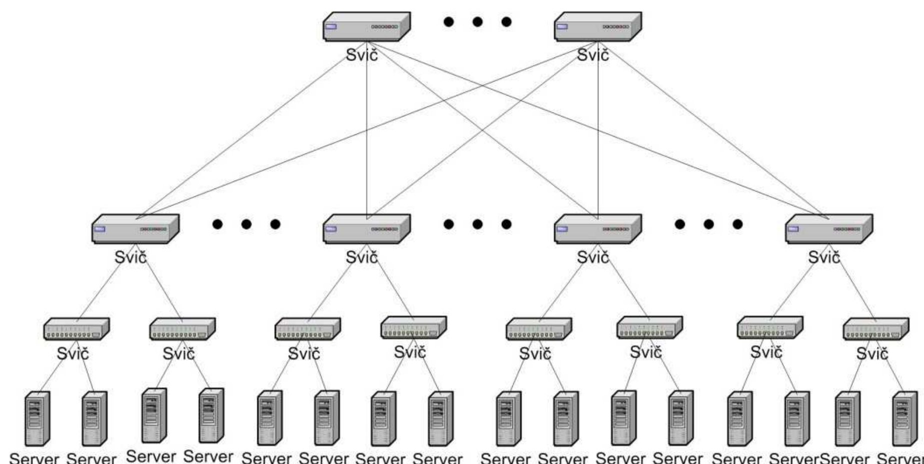
Hiperkocka, meš sa dva elementa po dimenziji i torus sa dva elementa po dimenziji su ekvivalentne topologije za zadati broj dimenzija. Ove tri topologije se generišu u okviru koordinatnog sistema, koji određuje i adrese svičeva. Adresiranje svičeva u ovim topologijama bazirano na koordinatnom sistemu prikazano je na Slikama 2-1, 2-2 i 2-3.

2.1.4 Topologija stabla sa više korena

Veliki broj servera u data centrima zahteva veliki broj svičeva na koje te servere treba povezati. Sledeći korak je međusobno povezivanje svičeva na koje su povezani serveri. Zbog velikog broja ovih svičeva, oni ne mogu biti povezani svaki sa svakim, zato što nemaju dovoljno slobodnih portova. Jedan način za njihovo međusobno povezivanje mogao bi biti formiranje stabla, u kome svičevi sa serverima čine najniži nivo. Više svičeva nižeg nivoa bi mogli biti povezani na jedan svič višeg nivoa, i na taj način bi svi svičevi u strukturi stabla mogli biti povezani sa raspoloživim brojem portova. Problem sa opisanom strukturom stabla je što svaki viši nivo u stablu zbog koncentracije saobraćaja treba da ima sve veću brzinu, i brzine linkova u višim nivoima tehnički ne bi bile ostvarive. Ovaj problem je moguće prevazići uvođenjem više korena u stablo [8], čime se saobraćaj umesto koncentracije u jednom korenu stabla raspoređuje na više korena. Struktura stabla sa više korena prikazana je na Slici 2-4 [8].

U topologiji stabla sa više korena svaki koren je povezan sa svim svičevima u redu ispod. Svaki od ovih svičeva je koren svog podstabla ispod koga se nalaze drugi svičevi i serveri. Pošto se u ovim podstablima vrši koncentracija saobraćaja prema korenu, u praksi topologija stabla sa više korena se koristi sa tri nivoa svičeva, kao što je prikazano na Slici 2-4. Ograničavanjem broja nivoa se izbegava preopterećenje linkova prema korenu podstabla.

U stablu sa više korena moguće je biranjem broja korena i brzina linkova određivati koncentraciju saobraćaja, čime se može postići željeni odnos cene i performansi.



Slika 2-4 Topologija stabla sa više korena

Topologija stabla sa više korena je uobičajena topologija u data centrima koji su se vremenom proširivali i u kojima je broj svičeva rastao uključivanjem novih servera [8]. Ova topologija ima slabiji odnos cene i performanse od *fat-tree* toplogije, i za broj čvorova veći od 10000 ima značajno veću cenu od *fat-tree* topologije čak i kada se u njoj koristi velika koncentracija saobraćaja koja iznosi 7:1, a *fat-tree* topologija se koristi bez koncentracije [8].

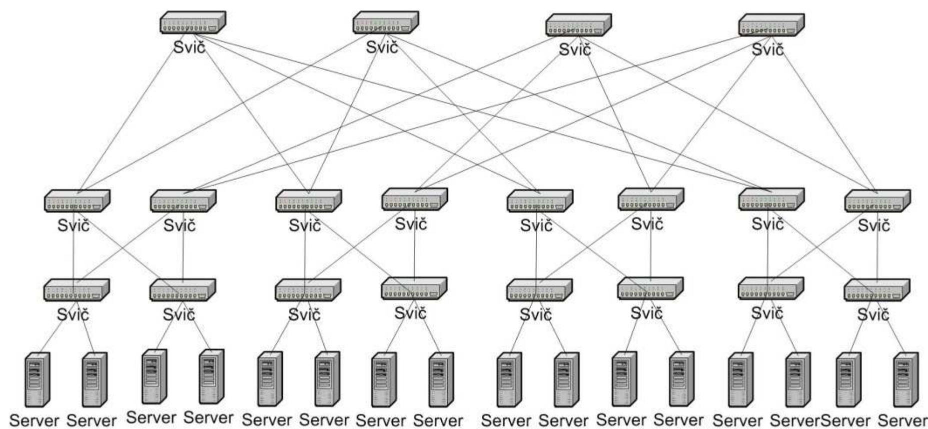
2.1.5 *Fat-tree* topologija

Fat-tree topologija je jedna od najčešće korišćenih topologija u data centrima [8]. Kod ove topologije koriste se svičevi istog tipa što smanjuje cenu u poređenju sa topologijom stabla sa više korenova. U *fat-tree* strukturi sa tri nivoa prikazanoj na Slici 2-5, svičevi iz najnižeg nivoa označavaju se kao ivični svičevi, iz višeg nivoa agregacioni i iz poslednjeg nivoa centralni svičevi.

Jedna od razlika između *fat-tree* topologije i topologije stabla sa više korenova opisane u odeljku 2.1.4 je što se u *fat-tree* topologiji svaki svič višeg nivoa povezuje sa svakom grupom svičeva nižeg nivoa, dok se u topologiji stabla sa više korenova svaki koren povezuje sa svakim svičem u nivou ispod. Iz ove razlike sledi mogućnost da se u *fat-tree* topologiji koriste svičevi sa više portova i manjom brzinom portova umesto svičeva sa velikim brzinama portova koji su skuplji, što rezultuje nižom cenom *fat-tree* topologije [8], koja je navedena u odeljku 2.1.4.

U okviru *fat-tree* strukture postoji m grupa. U *fat-tree* strukturi sa tri nivoa, svaka grupa sadrži $m/2$ agregacionih svičeva i $m/2$ ivičnih svičeva. Svaki ivični svič ima

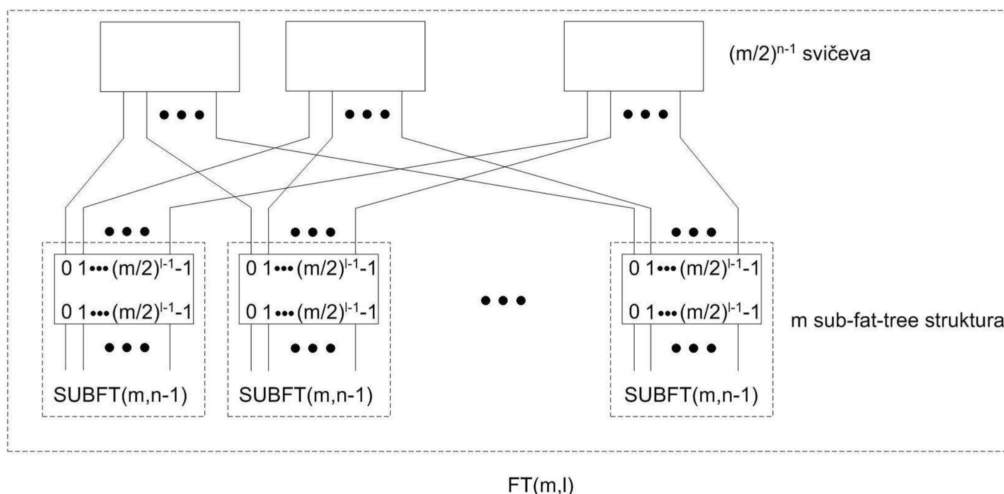
m portova, pri čemu je sa *m/*2 portova povezan na hostove a sa *m/*2 portova na agregacione svičeve u istoj grupi. Centralni svičevi takođe imaju po *m* portova, i *m/*2 portova svakog agregacionog sviča je povezano na centralne svičeve. Agregacioni svičevi iz iste grupe se vezuju na različite centralne svičeve. Ukupan broj centralnih svičeva je $(m/2)^2$. Svaki centralni svič ima po jedan port vezan na svaku grupu. Kako u okviru trostepene *fat-tree* strukture postoji *m* grupa, a u svakoj ima *m/*2 ivičnih svičeva, to je ukupno $m^2/2$ ivičnih svičeva. Svaki ivični svič je sa po *m/*2 portova povezan na hostove, pa je ukupan broj hostova je $m^3/4$ [8].



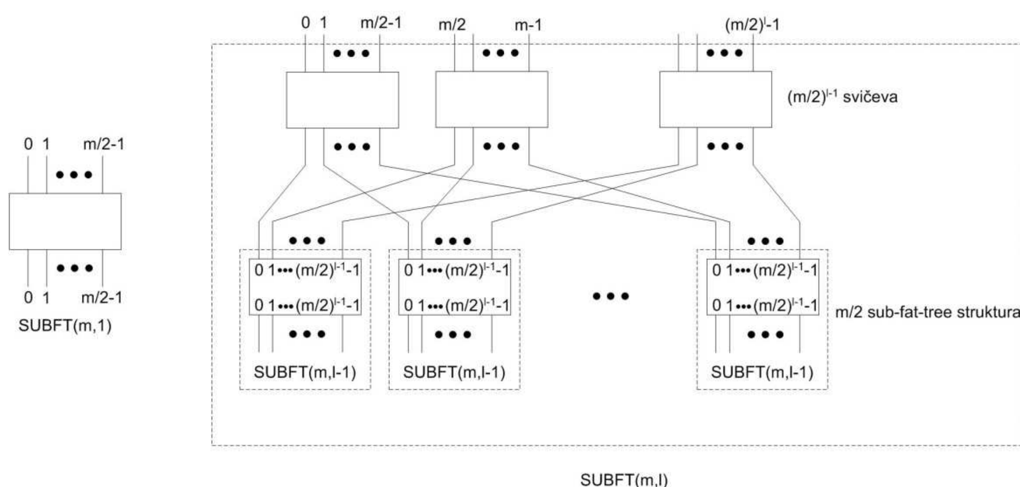
Slika 2-5 *Fat-tree* topologija

Na Slikama 2-6 i 2-7 prikazana je formalna rekurzivna definicija *fat-tree* topologije, koja u potpunosti opisuje formiranje *fat-tree* strukture proizvoljne veličine. *Fat-tree* topologija FT(*m*,*n*) koja se sastoji od *n* nivoa svičeva sa *m* portova definiše se preko struktura SUBFT(*m,n*-1) [9]. Struktura FT(*m,n*) formira se povezivanjem svičeva sa *m* portova sa *m* SUBFT(*m,n-1*) struktura, kao što je prikazano na Slici 2-6. Strukture FT(*m,l*) i SUBFT(*m,l*) se razlikuju po tome što struktura SUBFT(*m,l*) ima linkove ka višem nivou dok ih FT(*m,l*) nema. Struktura SUBFT(*m,l*) definiše se rekurzivno na način prikazan na Slici 2-7.

Za $l = 1$ SUBFT(m,1) sadrži 1 svič sa *m* portova, pri čemu su *m/*2 portova povezani na hostove, a ostalih *m/*2 portova ostaju nepovezani, i biće iskorišćeni za povezivanje na svičeve višeg nivoa. U cilju izračunavanja ukupnog broja hostova povezanih na FT(*m,n*) strukturu, označimo broj linkova SUBFT(*m,l*) povezanih na svičeve višeg nivoa sa $br(m, l)$. Tada je $br(m, 1) = m/2$.



Slika 2-6 Definicija *fat-tree* strukture



Slika 2-7 Definicija SUBFT strukture

Lema 2.1: Broj linkova strukture SUBFT(*m*,*l*) povezanih na svičeve višeg nivoa je $br(m,l) = (m/2)^l$.

Dokaz:

Struktura SUBFT(*m*,*l*) se formira od *m/*2 SUBFT(*m*,*l*-1) struktura i svičeva sa *m* portova, kao što je prikazano na Slici 2-7, i to tako što se svaki svič višeg nivoa povezuje sa svakom od SUBFT(*m*,*l*-1) struktura. Port *i* svake od SUBFT(*m*,*l*-1) struktura povezan je na *i*-ti svič višeg nivoa. Kako je broj izlaznih linkova SUBFT(*m*,*l*-1) strukture ka višem nivou $br(m,l-1)$ to je $br(m,l) = br(m,l-1) \cdot (m/2)$. Kako je $br(m,1) = m/2$, matematičkom indukcijom zaključujemo da je $br(m,l) = (m/2)^l$.

■

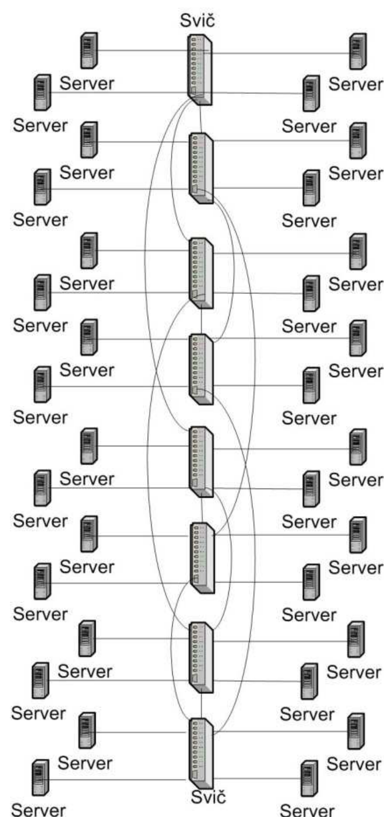
Dakle, svaka SUBFT(*m,l*) struktura ima $(m/2)^l$ linkova ka višem nivou i $(m/2)^l$ veza ka hostovima.

Struktura FT(*m,n*) se formira od $br(m, n-1) = (m/2)^{n-1}$ svičeva koji su povezani na m SUBFT(*m,n*-1) struktura, kao što je prikazano na Slici 2-6. Ukupan broj servera FT(*m,n*) strukture je $m \cdot (m/2)^{n-1}$. Najviši nivo FT(*m,n*) strukture sadrži $(m/2)^{n-1}$ svičeva, a svaki od preostalih *n*-1 nivoa sadrži $m \cdot (m/2)^{n-2}$ svičeva. Naime, nivo *n*-1 FT(*m,n*) strukture sadrži m SUBFT(*m,n*-1) struktura, a svaka od SUBFT(*m,n*-1) struktura sadrži $(m/2)^{n-2}$ svičeva na najvišem nivou, pa je njihov ukupan broj u FT(*m,n*) strukturi $m \cdot (m/2)^{n-2}$. Broj svičeva proizvoljnog nižeg nivoa i FT(*m,n*) strukture se može proračunati tako što se broj svičeva najvišeg nivoa SUBFT(*m,i*) pomnoži brojem SUBFT(*m,i*) struktura koje formiraju SUBFT(*m,i*+1) strukturu, a zatim se taj broj pomnoži brojem SUBFT(*m,i*+1) struktura koje formiraju SUBFT(*m,i*+2) strukturu i tako redom dok se ne pomnoži brojem SUBFT(*m,n*-1) struktura koje formiraju FT(*m,n*) strukturu. Odnosno, za proizvoljan nivo i, taj broj je $(m/2)^{i-1} \cdot m/2 \cdot m/2 \cdot \cdots \cdot m$ pri čemu je broj množioca $m/2$ *n*-*i*-1, pa je broj svičeva u proizvoljnom nivou i FT(*m,n*) strukture jednak $m \cdot (m/2)^{n-2}$. Ukupan broj svičeva FT(*m,n*) strukture dobija se sabiranjem $(m/2)^{n-1}$ svičeva iz najvišeg nivoa strukture, i po $m \cdot (m/2)^{n-2} = 2 \cdot (m/2)^{n-1}$ svičeva iz svakog od nižih *n*-1 nivoa, i jednak je $(2 \cdot n - 1) \cdot (m/2)^{n-1}$.

2.1.6 *Flattened butterfly*

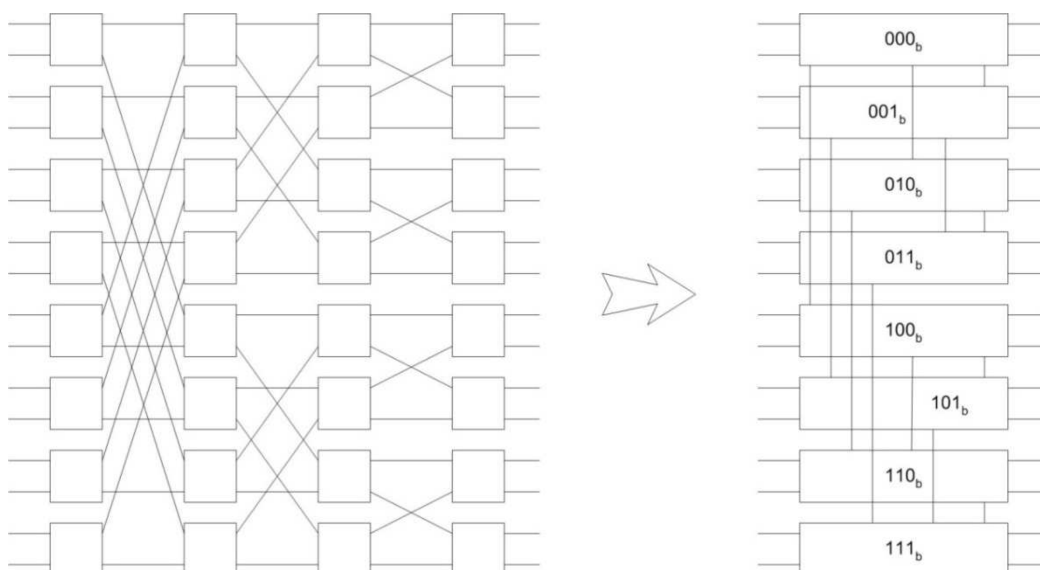
Flattened butterfly [10] topologija sa osam svičeva prikazana je na Slici 2-8. U ovoj topologiji svičevi se povezuju međusobno tako što se formiraju veze između svičeva čije se vrednosti binarnih indentifikatora razlikuju u jednom bitu. Na preostale portove svičeva povezuju se serveri.

Flattened butterfly se formira od *butterfly* topologije spajanjem svih svičeva *butterfly* topologije koji pripadaju jednoj vrsti u jedan svič, pri čemu veze između vrsta postaju veze između svičeva [10]. Na Slici 2-9 je prikazana *flattened butterfly* topologija nastala od *butterfly* topologije sa četiri kolone i svičevima sa dva porta.

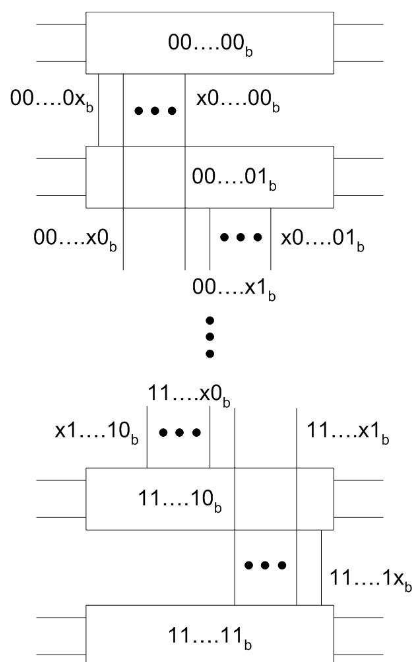


Slika 2-8 Primer *flattened butterfly* topologije

Spajanjem horizontalne grupe *butterfly* topologije u jedan svič *flattened butterfly* topologije horizontalni linkovi *butterfly* topologije postaju suvišni. Za razliku od *butterfly* topologije, *flattened butterfly* ima različite putanje jednakih cena. Rutiranje u *flattened butterfly* mreži opisano je u potpoglavlju 2.2.3.



Slika 2-9 Formiranje *flattened butterfly* topologije od *butterfly* topologije



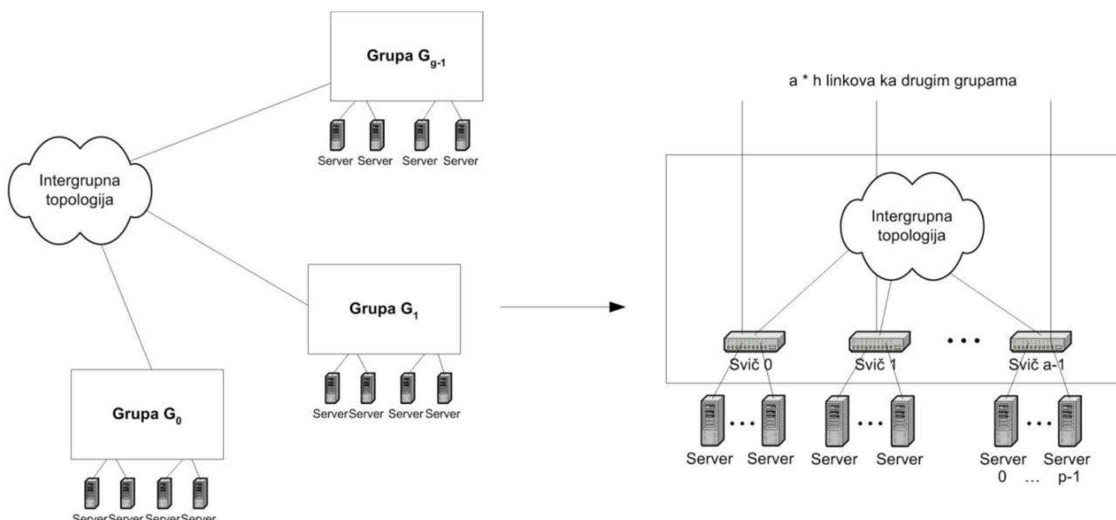
Slika 2-10 *Flattened butterfly* topologija

Opšti slučaj povezivanja svičeva *flattened butterfly* topologije prikazan je na Slici 2-10. Identifikatori svičeva prikazani su u binarnom obliku, i uz svaki link je predstavljen binarni broj sa vrednošću *x* koja može biti 0 ili 1. Vredost *x* predstavlja poziciju na kojoj se razlikuju indentifikatori svičeva koje link povezuje.

2.1.7 *Dragonfly*

Dragonfly topologija [11], prikazana na Slici 2-11, se zasniva na podeli računara u data centrima u grupe koje imaju intragrupnu topologiju koja moze npr. biti *fat-tree* ili torus. Svaka od grupa je povezana sa svim ostalim grupama preko intergrupne topologije. Za intergrupnu topologiju i intragrupnu topologiju mogu se koristiti proizvoljne topologije. Na primer, za intergrupnu topologiju je moguće koristiti potpuno povezanu topologiju, a za intragrupnu topologiju *flattened butterfly* [10].

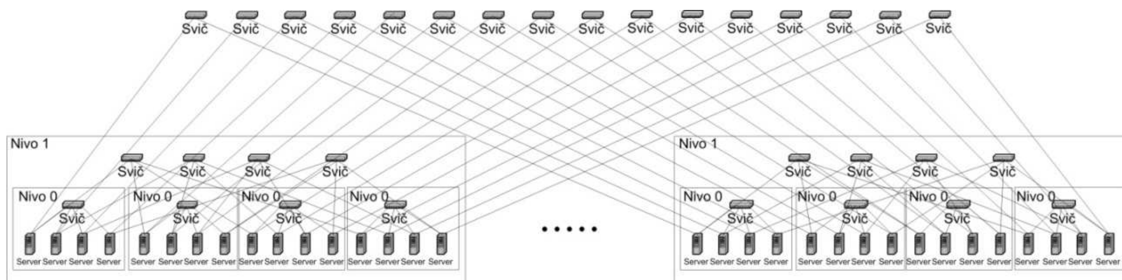
Na Slici 2-11 prikazana je *dragonfly* topologija. Svaki svič u okviru grupe ima *p* veza do servera, *a-1* vezu do svičeva u okviru iste grupe (u slučaju potpuno povezane intragrupne topologije) i *h* veza do svičeva u drugim grupama, pa je broj portova sviča $k'=p+a+h-1$. Svaka grupa ima *a* svičeva na koje se povezuju serveri. Svi svičevi u okviru grupe se mogu posmatrati kao virtuelni svič sa $a(h+p)$ portova. Maksimalni broj grupa, ukoliko su sve međusobno povezane, je $ah+1$, zato što je svaka grupa preko *ah* linkova povezana sa svim ostalim grupama.



Slika 2-11 *Dragonfly* topologija

2.1.8 BCube

BCube topologija [12] je namenjena modularnom građenju data centara od kontejnera koji u sebi sadrže određeni broj računara i svičeva. Pošto je komponente smeštene u kontejnerima komplikovano popraviti ili zameniti po zatvaranju kontejnera, jedan od ciljeva BCube topologije je blaži pad performansi u slučaju kvarova od ostalih topologija. Takođe, BCube koristi jeftinije svičeve koji se uobičajeno koriste u modularnim data centrima.



Slika 2-12 BCube topologija

BCube predstavlja rekurzivno definisanu strukturu. Primer BCube topologije prikazan je na Slici 2-12. $BCube_0$ predstavlja n servera povezanih na n-portni svič. $BCube_1$ se sastoji od n $BCube_0$ i n n-portnih svičeva. Dakle, nivo 1 ima n^2 servera. Po jedansever iz svakog $BCube_0$ povezan je na jedan svič iz nivoa 1. U opštem slučaju, $BCube_k$ se sastoji od n $BCube_{k-1}$ i n^k n-portnih svičeva. Svaki server u $BCube_k$ ima k+1 port, odnosno do svakog nivoa po jednu vezu. $BCube_k$ ima n^{k+1} servera. Svičevi nisu

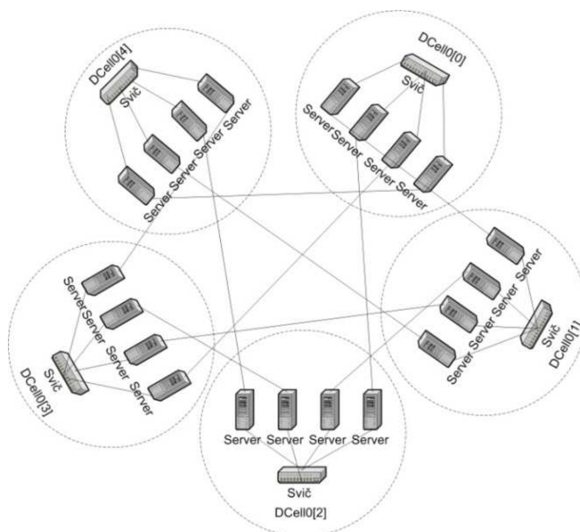
međusobno direktno povezani. $BCube_k$ ima k+1 nivoa svičeva a u svakom nivou ima n^k n-portnih svičeva, pa je ukupan broj svičeva $(k+1) \cdot n^k$.

BCube mreža se može preslikati u generalizovanu hiperkocku tako što se svaki svič i njegovi linkovi zamene vezama koje direktno povezuju sve servere koji su bili povezani na svič, svaki sa svakim. Uvođenjem sviča umesto potpuno povezane topologije smanjuje se broj linkova i broj potrebnih portova na serverima.

Adrese servera u BCube strukturi formiraju se kao nizovi cifara koji sadrže poziciju sviča u $BCube_0$ strukturi, zatim poziciju $BCube_0$ u okviru $BCube_1$, poziciju $BCube_1$ u okviru $BCube_2$ i tako dalje do pozicije $BCube_{k-1}$ u okviru $BCube_k$.

2.1.9 DCell

DCell je topologija namenjena efikasnom povezivanju velikog broja servera [13]. U DCell svičevi se koriste samo u okviru osnovnog nivoa topologije, dok se dalja spajanja vrše korišćenjem servera.

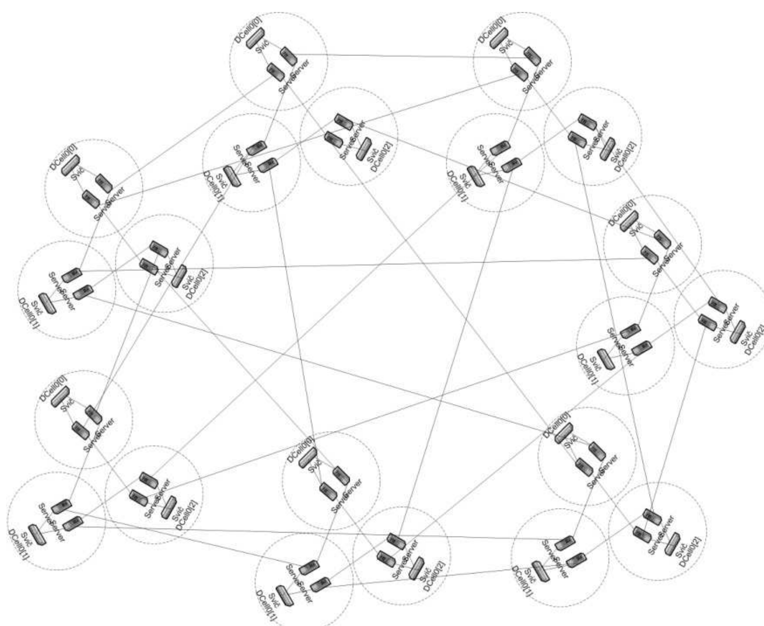


Slika 2-13 Formiranje $Dcell_1$ sa četriri servera u okvoru $Dcell_0$

DCell je rekurzivno definisana struktura u kojoj DCell višeg nivoa sadrži DCell-ove nižeg nivoa i DCell-ovi na istom nivou su međusobno potpuno povezani. Svičevi povezuju računare u okviru $DCell_0$. Na Slici 2-13 je prikazan način formiranja $DCell_1$ od $DCell_0$. Ukoliko svaki server označimo uređenim parom $[a_1, a_0]$, gde su a_1 i a_0 identifikatori u okviru nivoa 1 i nivoa 0, tada je povezan svaki par servera $[i, j-1]$ i $[j, i]$ za svako i i svako $j > i$.

Na Slikama 2-12 i 2-13 može se uočiti da BCube i DCell topologije imaju istu osnovnu strukturu BCube_0, odnosno DCell_0.

Na Slici 2-14 je prikazano rekurzivno formiranje DCell_2 od DCell_1 i DCell_0 struktura pri čemu ima dva servera u okviru DCell_0 . Procedura formiranja DCell_k strukture od DCell_{k-1} izvodi se potpunim povezivanjem DCell_{k-1} topologija tako što povezujemo po jedan par servera u različitim DCell_{k-1} podmrežama. Ako DCell_{k-1} topologija ima t_{k-1}servera onda se DCell_k topologija gradi od $t_{k-1}+1$ DCell_{k-1} topologija, tako što se svake dve DCell_{k-1} topologije povežu linkom između para servera koji pripadaju tim topologijama.



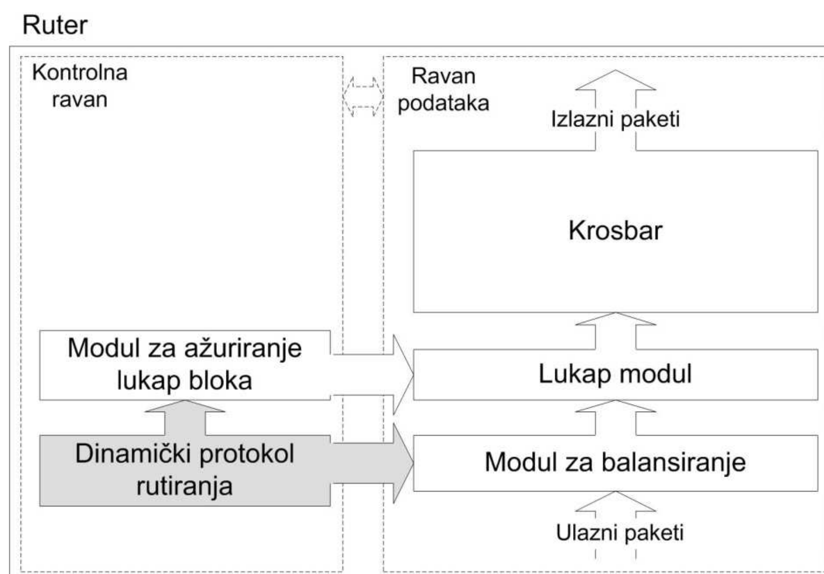
Slika 2-14 Rekurzivno formiranje Dcell_2 sa dva servera u okviru Dcell_0

Broj servera t_k u DCell_k topologiji koja ima n servera u osnovnoj DCell_0 topologiji uzima vrednosti između $(n+1/2)^{2^k}-1/2$ i $(n+1)^{2^k}-1$ [13]. Iz ovih formula sledi da broj servera u DCell mreži raste veoma brzo sa nivoom DCell topologije, odnosno sa brojem portova svakog servera.

Slično adresama u BCube mrezi, i u DCell mreži adrese oslikavaju pozicije servera u mreži. Pozicije servera opisane su nizom brojeva koji sadrže poziciju servera u DCell_0, zatim poziciju DCell_0 u okviru DCell_1, zatim DCell_1 u okviru DCell_2 i tako dalje do pozicije DCell_{k-1} u okviru DCell_k.

2.2 Protokoli za rutiranje u data centrima

Topologije data centara sadrže više putanja između svaka dva servera. Cilj većine protokola za rutiranje u data centrima je iskorišćavanje ovih putanja u cilju ravnomernog iskoriščenja prenosnog kapaciteta i izbegavanja nastanaka zagušenja u situacijama kada postoji kapacitet na alternativnim putanjama. Veza protokola rutiranja sa ostalim modulima rutera prikazana je na Slici 2-15.



Slika 2-15 Dinamički protokol rutiranja u okviru rutera

Zbog postojanja više putanja između servera data centra, osnovni protokol za rutiranje u mrežama data centara je rutiranje po putanjama jednakih cena, tj. ECMP rutiranje (eng. *Equal-Cost Multipath Routing*) [14]. ECMP rutiranje ravnomerno raspoređuje saobraćaj po putanjama istih cena, i na taj način koristi raspoložive kapacitete mreže data centra. Pošto koristi alternativne minimalne putanje, ECMP je minimalno rutiranje.

Pošto vrši slučajan izbor između alternativnih minimalnih putanja ECMP obezbeđuje ravnomeran raspored saobraćaja po alternativnim putanjama u srednjem. Pri slučajom biranju moguće je da u nekom intervalu vremena jednom alternativnom putanjom krene više paketa nego drugom. Privremeni viškovi paketa nastali zbog ovakvih neravnomernosti raspodele paketa smeštaju se u bafere dok se opterećenje na toj putanji ne smanji. Pored rutiranja po alternativnim minimalnim putanjama, postoje i neminimalni protokoli za rutiranje po alternativnim putanjama. Predstavnik ove klase

protokola je Valiant rutiranje u kome se slučajno bira međusvič kroz koga paket mora da prođe [1]. Ovaj međusvič može biti bilo koji svič u mreži, uključujući i svičeve koji nisu na jednom od minimalnih puteva između krajnjih tačaka, što Valiant rutiranje čini neminimalnim. Pri slučajnom biranju putanja u Valiant rutiranju moguće je, kao i pri ECMP rutiranju da privremeno postoji neravnomeran raspored paketa po putanjama.

Neravnomeran raspored paketa po putanjama pri ECMP i Valiant rutiranju dovodi do opterećenja pojedinih linkova i sakupljanja paketa u baferima. Pošto je u srednjem rapodela paketa po putanjama ravnomerna, ako su baferi dovoljno veliki, paketi će biti sačuvani u njima dok se ne smanji broj paketa usmeren preko tih linkova. Adaptivni protokoli rutiranja su posebna klasa protokola koji pri rutiranju uzimaju u obrzir i stanja bafera i vrše rutiranje sa ciljem da se izbegne nagomilavanje paketa 2.2.7. Adaptivni protokoli rutiranja mogu omogućiti ravnomernije opterećenje bafera u mreži, ali mogu dovesti i do nestabilnosti u mreži u kojoj se koriste i drugi protokoli koji reaguju na zagušenje poput transportnih TCP protokola.

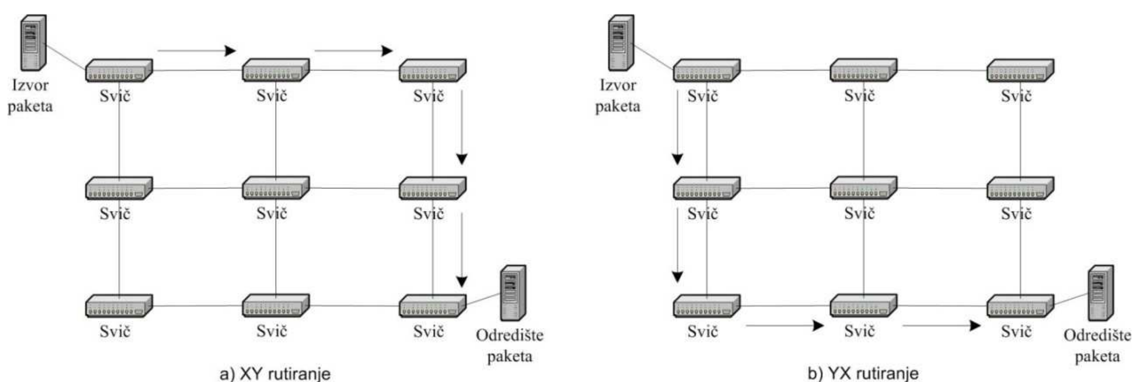
Pored ECMP i Valiant rutiranja koja su značajna u svim topologijama, u narednim odeljcima opisana su i rutiranja koja koriste specifičnosti pojedinih topologija. U odeljku 2.2.1 opisana su rutiranja koja koriste pravilnu matematičku strukturu svičeva meš topologije. U odeljku 2.2.2 opisana su rutiranja u *fat-tree* topologiji i topologiji stabla sa više korenova. U odeljku 2.2.3 opisana su rutiranja korišćena u *flattened buterfly* topologiji, dok su u odeljku 2.2.4 opisana rutiranja u *dragonfly* topologiji. U odeljku 2.2.5 opisani načini rutiranja koje koriste specifičnosti BCube i DCell topologija.

2.2.1 Rutiranje u meš topologiji i topologijama torusa i hiperkocke

Najjednostavnije rutiranje u pravilnim topologijama kao što su meš, torus i hiperkocka je rutiranje po dimenzijama (eng. DOR – *Dimension Order Routing*) [16]. Rutiranje po dimenzijama je omogućeno time što se u svakoj od dimenzija ovih struktura svičevima može dodeliti pozicija. Rutiranje se vrši tako što se prvo po jednoj od dimenzija dođe do vrednosti koordinate odredišta u toj dimenziji. Zatim se u drugoj dimenziji vrši pomeranje paketa kroz svičeve dok se ne dođe do sviča čija je koordinata jednaka koordinati odredišta. Pomeranje se na ovaj način obavlja i kroz preostale dimenzije, čime paket dolazi do odredišnog sviča.

U slučaju dvodimenzionih struktura sa X i Y osama, rutiranje po dimenzijama se naziva XY rutiranje u slučaju da se pomeranje paketa vrši prvo po X osi a zatim po Y. U slučaju da se pomeranje vrši prvo po Y osi ovo rutiranje se naziva YX rutiranje. XY i YX rutiranja u dvodimenzionoj meš topologiji prikazana su na Slici 2-16.

Rutiranje po dimenzijama, čiju su predstavnici XY i YX rutiranja, je jednostavno, ali ne iskorišćava postojanje različitih minimalnih putanja između izvorišnog i odredišnog sviča. U cilju boljeg iskorišćenja resursa mreže predloženi su različiti algoritmi rutiranja.



Slika 2-16 XY i YX rutiranje u dvodimenzionoj meš topologiji

O1TURN (eng. *Orthogonal One Turn*) rutiranje [17] poboljšava iskorišćenje mreže tako što se za svaki paket slučajno bira da li će biti rutiran YX ili XY rutiranjem. U radu [17], autori pokazuju de se sa O1TURN postiže iskorišćenje mreže blisko optimalnom. O1TURN rutiranje je dobilo ime na osnovu činjenice da paket samo jednom skreće.

Algoritam rutiranja U2TURN [19] koristi do dva skretanja na putu paketa. Skup U2TURN ruta obuhvata YX i XY ruta iz O1TURN rutiranja i dodaje XYX i YXY rute. Pri XYX slučajno se bira koordinata između X koordinata krajnih svičeva, pa se prvo rutira do nje duž X ose, pa se po Y osi rutira do Y kooridnate odredišta, i na kraju se put do odredišta završava po X koordinati. YXY rutiranje funkcioniše slično XYX rutiranju, sa tim što se slučajno izabrana međutačka bira na Y osi.

ROMM algoritam [18] koristi veći broj skretanja, i funkcioniše tako što se bira *p*-1 tačaka koje se nalaze između kranjih tačaka, i paket pri rutiranju prolazi kroz svaku od tih tačaka. ROMM ima lošiji protok najgoreg slučaja u odnosu na O1TURN i U2TURN rutiranja. ROMM, kao i O1TURN, U2TURN i rutiranje po dimenzijama

predstavljaju minimalno rutiranje, jer paket prolazi kroz minimalani mogući broj svičeva u mreži.

Još jedan način za poboljšanje iskorišćenja meš topologije i topologija torusa i hiperkocke je Valiant rutiranje, u kome se međusvič kroz koji paket treba da prođe bira slučajno između svih svičeva u topologiji.

Meš topologija i topologije torusa i hiperkocke imaju više minimalnih putanja sa istim cenama i kod njih bi ECMP rutiranje omogućilo raspodelu saobraćaja po tim minimalnim putanjama. U ECMP rutiranju bi se u svakom koraku birao jedan od raspoloživih puteva sa istom cenom do odredišta.

2.2.2 Rutiranje u *fat-tree* topologiji i topologiji stabla sa više korenova

Minimalno rutiranje u *fat-tree* topologiji i topologiji stabla sa više korenova sastoji se od kretanja od izvorišnog servera ka višim nivoima strukture sve dok se ne dođe do sviča ispod koga se nalaze i izvorišni i odredišni server, i zatim spuštanje do odredišnog servera. Pri kretanju ka višim nivoima strukture, rutiranje će biti minimalno bez obzira kojim od mogućih puteva se paket kreće. U *fat-tree* rutiranju paket može izabrati bilo koji od linkova iznad sviča, dok u topologiji sa više korenova paket može ići preko bilo kog od korenova ako je najviši zajednički nivo za izvor i odredište nivo koji sadrži korenove.

U *fat-tree* topologiji za svaka dva servera koji nisu povezani na isti svič postoji više minimalnih putanja sa istom cenom, dok kod topologije stabla sa više korenova više putanja postoji samo ako paket treba da pređe preko nivoa u kome se nalaze korenovi stabla, i u tom slučaju alternativnih minimalnih putanja ima koliko ima i korenova.

Pošto obe topologije imaju više minimalnih putanja, ECMP rutiranje može ravnomerno rasporediti saobraćaj po alternativnim minimalnim putanjama.

Valiant rutiranje nije pogodno za korišćenje u *fat-tree* topologiji i topologiji stabla sa više korenova, jer može dovesti do kretanja paketa od izvorišnog čvora preko viših nivoa topologije ka balansirajućem sviču, i opet ka višim nivoima topologije na putu od balansirajućeg sviča do odredišta.

2.2.3 Rutiranje u *flattened butterfly* topologiji

U *flattened butterfly* topologiji minimalno rutiranje zahteva onoliko koraka u koliko bita se razlikuju pozicije izvorišnog i odredišnog sviča. Ove korake je moguće napraviti u različitom redosledu, pa ako se pozicije izvorišnog i odredišnog sviča razlikuju u b bita, tada postroji $b!$ različitih minimalnih putanja u *flattened butterfly* topologiji. Pošto postoje različite minimalne putanje, korišćenje ECMP rutiranja može omogućiti ravnomernije opterećenje linkova mreže.

U *flattened butterfly* topologiji se koristi i Valiant rutiranje, kojim mreža dostiže performanse ECMP rutiranja u *fat-tree* topologiji [10]. Time se saobraćaj ravnomerno raspoređuje po linkovima *flattened butterfly* topologije i izbegava se zagušenje bez obzira na sadržaj matrice saobraćaja.

2.2.4 Rutiranje u *dragonfly* topologiji

Minimalno rutiranje u *dragonfly* topologiji u slučaju kada krajnji svičevi pripadaju različitim grupama vrši se preko jednog linka između grupa. Pored tog linka, minimalna putanja sadrži i svičeve unutar izvorišne grupe na putu od izvorišnog sviča do sviča na koga je povezan link, i svičeve unutar odredišne grupe na putu od sviča na koga je povezan link do odredišnog sviča. Ako su kranje tačke unutar iste grupe, minimalno rutiranje se odvija po najkraćoj putanji između tih svičeva, zavisno od topologije unutar grupa.

U cilju ravnomernije raspodele saobraćaja po linkovima *dragonfly* topologije predloženo je korišćenje modifikacije Valiant rutiranja [11]. Modifikacija se sastoji u primeni slučajnog izbora grupe kroz koju paket treba da prođe na putu do odredišta, umesto slučajnog izbora sviča u originalnom Valiant rutiranju. Pri Valiant rutiranju, zavisno od položaja slučajno izabrane međugrupe kroz koji paket treba da prođe, koriste se jedan ili dva linka između grupa. Ako su izvorišna grupa, međugrupa i odredišna grupa različite, tada se rutiranje odvija od izvorišnog sviča do sviča na koga je povezan link ka grupi međusviča, zatim kroz taj link do grupe međusviča, pa u toj grupi do međusviča pa do sviča na linku ka odredišnoj grupi, pa kroz taj link i kroz svičeve odredišne grupe do odredišnog sviča. U ovom slučaju se rasterećuje direktan link između izvorišne i odredišne grupe i na slučajan način se biraju dva druga međugrupna

linka za prenos paketa. Time se eliminiše zagušenje koga bi prouzrokovale matrice saobraćaja koje bi izazvale zagušenje na malom broju međugrupnih linkova.

2.2.5 Rutiranje u BCube i DCell topologijama

BCube i DCell topologije imaju istu osnovnu strukturu koja se sastoji iz servera povezanih svičem, i viši nivoi strukture se formiraju dodavanjem novih konekcija serverima. U BCube mreži se viši nivoi strukture formiraju dodavanjem svičeva svičeva i njihovim povezivanjem sa serverima osnovne strukture (2.1.8), dok se u DCell mreži viši nivoi strukture dobijaju direktnim povezivanjem servera (2.1.9).

Pošto se BCube mreža može preslikati u generalizovanu hiperkocku, i rutiranje u BCube mreži može biti slično rutiranju u hiperkocki. U BCube topologiji, adrese dva servera povezana preko sviča *i*-tog nivoa se razlikuju samo u cifri na poziciji *i*. Preko sviča *i*-tog nivoa od jednog servera povezanog na njega moguće doći do ostalih servera čije se adrese u odnosu na posmatrani serverrazlikuju samo na *i*-toj poziciji (2.1.8). Sledi da je rutiranje moguće tako što se u svakom koraku po jedna cifra tekućeg sviča koja je različita od odredišne adrese izjednači sa odredišnom adresom. Maksimalni broj koraka pri rutiranju u BCube mreži jednak je broju cifara u adresi, odnosno broju ugnježdenih BCube struktura.

Između svaka dva servera u $BCube_k$ postoji $k+1$ putanja koje nemaju zajedničke tačke [12], što omogućava raspoređivanje saobraćaja u cilju ravnomernog iskorišćenja resursa mreže. BCube rad [12] predlaže i BCube izvorno rutiranje u kome se pri započinjanju toka prvo slanjem kontrolnog paketa različitim putanjama do odredišta. Pošto od međuservera posredstvom kontrolnog paketa dobije informacije o stanju putanje, izvorišni server šalje novi tok najpovoljnijom putanjom, npr. putanjom koja ima najviše slobodnog kapaciteta.

Slično rutiranju u BCube mreži koje se oslanja na vezu BCube adrese i pozicije servera u mreži, i u DCell mreži je moguće rutirati na osnovu pozicije servera u mreži opisane njegovom adresom [12]. Zavisno od pozicija krajnjih čvorova u DCell topologiji, DCell adrese ovih čvorova će imati različitu dužinu zajedničkog prefiksa. Krajnji serveri će pripadati jednoj $DCell_i$ strukturi koja je najmanja struktura kojoj oba krajnja servera pripadaju. Pronalaženje linkova koji povezuju krajnje tačke odvija se tako što se prvo pronađe link koji povezuje $DCell_{i-1}$ strukture u okviru $DCell_i$ strukture.

Pronađeni link je deo tražene putanje, ali nije povezan sa krajnjim serverima. Povezivanje je potrebno izvršiti u okviru $DCell_{i-1}$ struktura. Ovo povezivanje se vrši na nivou $DCell_{i-2}$ struktura. Ako se tačke u okviru $DCell_{i-1}$ struktura nalaze u različitim $DCell_{i-2}$ tada se pronalazi link koji povezuje te $DCell_{i-2}$ strukture, i taj link predstavlja deo tražene putanje. Posle povezivanja $DCell_{i-2}$ struktura prelazi se na povezivanje $DCell_{i-3}$ struktura koje se vrši na isti način. Ova procedura se nastavlja i završava se kada se dođe do $DCell_0$ strukture, u okviru koje su svi serveri povezani na svič. Ovako dobijena putanja nije najkraća putanja ali je u radu pokazano da je statistički veoma bliska najkraćoj putanji.

2.2.6 Multipath TCP i *Hedera*

Osnovni problem prenosa TCP tokova u data centrima je mogućnost pojava velikih tokova koji mogu ugroziti druge tokove prenošene istom putanjom. Dva predstavnika protokola koji omogućavaju bolji prenos TCP tokova kroz mreže data centara su *Multipath* TCP i *Hedera*.

Ideja *Multipath* TCP-ja je da TCP ne ide jednom putanjom već da se formiraju podtokovi koji idu različitim putanjama i imaju nezavisnu regulaciju protoka [14]. Na taj način tok može iskoristiti slobodni kapacitet na putanji gde on postoji.

U data centrima, *Multipath* TCP podtokovi se usmeravaju slučajnim izborom po različitim ECMP putanjama ili se, u slučaju kada server ima više mrežnih interfejsa, različite putanje podtokova postižu izborom različitih mrežnih interfejsa servera.

Hedera algoritam [22] podrazumeva postojanje centralnog računara koji prati stanje mreže, i usmerava tokove različitim putanjam. Praćenje i premeštanje tokova vrši se korišćenjem *OpenFlow* standarda [23], koji omogućava daljinski pristup svičevima, praćenje njihovog rada i podešavanje parametara rutiranja.

U *Hedera* algoritmu centralni računar reaguje kada brzina toka dostigne vrednost koju tekuća putanja ne može prihvatiti. Centralni računar korišćenjem *Global First Fit* algoritma premešta taj tok na prvu putanju koju pronađe i koja može da prihvati taj tok, ili traži približno optimalan raspored tokova algoritmom *Simulated Annealing* [24]. *Simulated Annealing* algoritam je inspirisan metalurškim postupkom kojim se materijal kontrolisano hladi i greje u cilju poboljšanja strukture i smanjenja

defekata. U okviru ovog algoritma se u okviru analogije sa metalurškim postupkom definišu stanja, verovatnoće prelaza između susednih stanja, funkcija energije u stanjima i temperatura. Putanje tokova i saobraćaj u mreži se preslikavaju u elemente *Simulated Annealing* algoritama u cilju proračuna putanja velikih tokova [22]. Algoritam dozvoljava prelazak u stanje više energije, omogućavajući na taj način izbegavanje lokalnih minimuma. Temperatura predstavlja broj preostalih iteracija *Simulated Annealing* algoritma, i algoritam se završava kada temperatura dostigne vrednost nula. Jedno stanje algoritma je skup uređenih parova (odredišni server, svič), pri čemu se svaki odredišni server pojavljuje tačno jednom. Saobraćaj namenjen odredišnom serveru će prolaziti kroz svič iz uređenog para koji sadrži odredišni server. Susedna stanja se definišu kao stanja u kojima su za dva uređena para međusobno zamenjeni svičevi. Funkcija energije je zbir prekoračenja kapaciteta linkova za sve linkove u mreži za trenutnu matricu saobraćaja za koju *Hedera* pronalazi rutiranje. Verovatnoća prelaza između stanja zavisi od energija susednog stanja, energije tekućeg stanja i temperature. Algoritam dozvoljava i prelazak u stanje više energije, omogućavajući na taj način izbegavanje lokalnih minimuma. Od svih stanja koja su posećena u toku rada algoritma kao rezultat se bira stanje sa najmanjom energijom.

Kada se tok startuje, njegova početna putanja se bira slučajno. Centralni računar reaguje samo kada tok dostigne određenu veličinu, i iz tog razloga *Hedera* rutiranje nije efikasno u slučaju rutiranja velikog broja malih tokova.

2.2.7 Adaptivni protokoli rutiranja

Adaptivni protokoli rutiranja prilagođavaju putanje paketa trenutnoj zauzetosti redova za čekanje u mreži. Primer adaptivnog protokola rutiranja korišćenog u mrežama data centara je UGAL (*Universal Global Adaptive Load Balancing*) [20], u kome se u prvom sviču na putu paketa bira da li se rutira najkraćim putem ili se balansira Valiant balansiranjem. Ova odluka se donosi na osnovu zauzetosti redova za čekanje na linkovima za balansiranje najkraćim putem. Odluka se donosi samo u prvom sviču na putu paketa jer bi mogućnost izbora izmedju najkraćeg puta i balansiranja u svakom sviču mogla da dovede do lutanja paketa u mreži.

Slično UGAL rutiranju je PAR adaptivno rutiranje (*Progressive Adaptive Routing*) [21]. U PAR rutiranju se takođe bira između minimalnog rutiranja i Valiant

rutiranja, pri čemu je ovu odluku za prelazak na Valiant rutiranje moguće doneti u bilo kom sviču na putu paketa, ali samo jednom za jedan paket. Kada se za paket pređe na Valiant rutiranje, to rutiranje ostaje do pristizanja paketa na odredište.

Pored odlučivanja na osnovu zauzetosti redova za čekanje u lokalnom sviču, pri adaptivnom rutiranju je moguće odlučivati o putanji i na osnovu prikupljenih informacija o zagušenjima u drugim svičevima. Ove infomacije moguće je prikupiti merenjem trajanja prenosa paketa ili zapisivanjem informacija o zagušenju u pakete.

Adaptivni protokoli rutiranja u data centrima mogu rezultovati većim protocima od neadaptivnih protokola. Međutim, ovi protokoli ne garantuju bolje rutiranje, i pošto deluju reaktivno na saobraćaj u mreži, njihova stabilnost zavisi od brzine menjanja saobraćaja koji ulazi u mrežu. Takođe, njihova interakcija sa TCP protokolom koji je takođe reaktivan može dovesti do problema sa stabilnošću.

2.3 Optimizacija iskorišćenja linkova korišćenjem dvofaznih protokola za rutiranje

Zajednička osobina data centara je postojanje više alternativnih putanja između servera. Osnovni cilj rutiranja u data centrima je iskorišćenje ovih alternativnih putanja u cilju izbegavanja zagušenja u mreži u situacijama kada postoje alternativne putanje na kojima postoji slobodni kapacitet.

Kada se koristi ECMP rutiranje, neki link može prenositi komunikaciju većeg broja servera i biti preopterećen, dok neki drugi linkovi mogu prenositi manji broj tokova. Za proračunate minimalne putanje, postojaće matrice saobraćaja koje neravnomerno opterećuju linkove i izazivaju zagušenja koja bi se mogla izbeći drugačijim rutiranjem.

U cilju ravnomernog raspoređivanja saobraćaja u mreži za različite matrice saobraćaja predložena su različita rešenja.

Adaptivna rutiranja izbor putanja vrše na osnovu trenutnog iskorišćenja resursa mreže. Adaptivna rutiranja u data centrima, opisana u odeljku 2.2.7, usmeravanje vrše na osnovu informacija o zauzetosti redova za čekanje u lokalnom sviču, ili na osnovu informacija prenetih do njih od tačke zagušenja. Opisani protokoli adaptivnog rutiranja odluku donose bez poznavanja stanja u celoj mreži, što dovodi do mogućnosti

donošenja pogrešnih odluka o rutiranju i nestabilnog rada, posebno u stanju većeg opterećenja mreže. Rutiranje na osnovu potpunog uvida u trenutni saobraćaj u mreži takođe nosi probleme kao što su prikupljanje informacija u slučaju postojanja zagušenja i usklađeno usmeravanje saobraćaja u različitim ruterima. Zadatak postavljen pred adaptivno rutiranje je složen i dodatno je otežan brojem elemenata i količinom saobraćaja u data centrima. Iz tog razloga u data centrima se koriste jednostavni protokoli poput UGAL i PAR protokola opisanih u odeljku 2.2.7, koji se zasnivaju na izboru između minimalnog i Valiant rutiranja za svaki paket na osnovu zauzetosti lokalnih redova za čekanje u sviču.

Algoritmi *Hedera* i *Multipath* TCP se ograničavaju na poboljšavanje rutiranja TCP tokova i prevazilaženje problema pojave velikih TCP tokova (eng. *elephant TCP flows*). *Multipath* TCP, opisan u odeljku 2.2.6, deli tok na više podtokova koji se usmeravaju različitim putanjama, tako da u slučaju pojave velikog TCP toka ostaju alternativne putanje za prenos podataka. *Hedera* opisana u odeljku 2.2.6 se zasniva na praćenju stanja svih TCP tokova i prebacivanja tokova sa jedne putanje na drugu u slučaju zagušenja neke od putanja. Ova dva rešenja su ograničena na TCP tokove, pri čemu *Multipath* TCP, kao ECMP koristi samo minimalne putanje i time ima ograničenu mogućnost iskorišćenja linkova. *Hedera* zahteva praćenje velikog broja TCP tokova što zahteva specijalizovana rešenja u svičevima.

Jedno rešenje za ravnomerniju raspodelu saobraćaja u mreži data centara koje ne zahteva praćenje saobraćaja je Valiant balansiranje, opisano u odeljku 2.3.1, koje u cilju ravnomernog iskorišćenja linkova šalje paket preko međusviča, pri čemu međusvič može biti bilo koji drugi svič u mreži sa jednakom verovatnoćom. Eliminisanje potrebe za stalnim praćenjem saobraćaja i proračuna rutiranja na osnovu prikupljenih podataka čini Valiant balansiranje efikasnim rešenjem. Valiant balansiranje je povoljno u dobro povezanim topologijama kao što su topologije data centara, ali biranje svakog međusviča sa jednakom verovatnoćom nije optimalno u većini topologija.

Optimizacija izbora međusvičeva pri dvofaznom balansiranju omogućila bi usmeravanje saobraćaja u cilju iskorišćenja raspoloživih resursa data centra bez potrebe za komplikovanim algoritmima zasnovanim na praćenju saobraćaja i reakciji na zagušenja. Opšti optimizacioni model rutiranja sa dvofaznim balansiranjem predstavljen

je u odeljku 2.3.2. Optimizacija dvofaznog balansiranja zasnovanog na rutiranju najkraćim putanjama predstavljena je u odeljku 2.3.3. Rutiranje putanjama sa jednakim cenama omogućava iskorišćenje alternativnih putanja u topologijama data centara, i optimizacija dvofaznog balansiranja zasnovana na njemu predstavljena je u odeljku 2.3.4. Odeljak 2.3.5 uvodi optimizaciju dvofazog balansiranja zasnovanog na modifikaciji rutiranja najkraćim putanjama prilagođenoj mrežama sa alternativnim putanjama iste cene, u koje spadaju i mreže data centara.

2.3.1 Karakteristike Valiant rutiranja

Valiant rutiranje [1] je prvo dvofazno rutiranje i u njemu se paket prvo rutira od izvora do slučajno izabranog međusviča, a zatim do odredišta. Valiant rutiranje omogućava smanjenje zagušenja u topologijama sa većim brojem putanja.

U cilju poređenja Valiant balansiranja sa rutiranjem u kome se koristi jedna putanja, za procenu opterećenja linka pri rutiranju sa jednom putanjom moguće je koristiti Borodin-Hopkroftovu donju granicu [25]-[26]. Borodin-Hopkroftova donja granica određuje minimalni broj tokova koji će za neku od matrica saobraćaja proći kroz bar jedan čvor mreže. Borodin-Hopkroftova donja granica definisana sledećom teoremom se odnosi na matrice saobraćaja u kojima je svaki čvor izvor jednog toka i odredište jednog toka.

Teorema 2.1: (Borodin-Hopkroftova donja granica): Za svaki graf *G* sa *n* čvorova i maksimalnim stepenom *d*, za svako neadaptivno rutiranje sa jednom putanjom između para krajnjih tačaka postoji permutacija komunikacionih parova π u kojoj kroz neki čvor mreže prolazi $\sqrt{n/d}$ putanja.

Dokaz Borodin-Hopkroftove donje granice predstavljen je u [25] i [26].

Vrednost Borodin-Hopkroftove donje granice je nepovoljna za mreže u kojima postoji veliki broj svičeva, što je slučaj u mrežama data centara. Prednost Valiant rutiranja u odnosu na rutiranje po jednoj putanji može se ilustrovati na primeru hiperkocke, koji je razmatran u [26].

Teorema 2.2: U hiperkocki, donja granica opterećenja lika korišćenjem Valant rutiranja iznosi $2^d/(2^d-1)$, dok korišćenjem proizvoljnog direktnog rutiranja iznosi $\sqrt{2^d/d}\,/d$.

Dokaz:

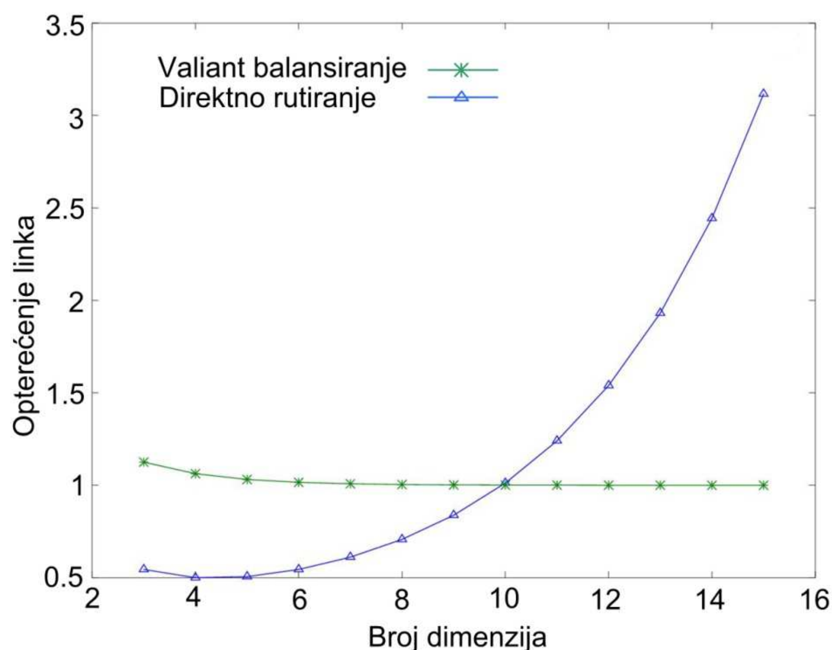
U hiperkocki broj čvorova raste eksponencijalno sa dimenzijom d i jednak je 2^d, dok stepen čvora raste linearno i jednak je d. Iz Borodin-Hopkroftove donje granice sledi da u hiperkocki postoji čvor kroz koga prolazi bar $\sqrt{2^d/d}$ tokova. Pošto je dimenzija čvora jednaka d, sledi da taj čvor ima bar jedan link kroz koji prolazi bar $\sqrt{2^d/d}\,/d$ tokova.

U slučaju Valiant balansiranja u hiperkocki dimenzije d i matrice saobraćaja korišćene u formulaciji Borodin-Hopkroftove donje granice, u kojoj je svaki čvor izvor i odredište tačno jednog toka, protok koji prolazi kroz svaki link približno je jednak protoku jednog toka. Ovaj rezultat za opterećenje linkova u hiperkocki može se dobiti polazeći od broja putanja koje prolaze kroz svaki link [26]. Svaki link u hiperkocki povezuje dva čvora čije se adrese razlikuju u tačno jednoj dimenziji, koju označavamo sa i. U slučaju rutiranja u kome se redom izjednačuje jedna po jedna različita dimenzija izvora i odredišta, ovaj link će predstavljati vezu između 2^i tačaka sa jedne strane i 2^{d-i-1} tačaka sa druge strane. Ako se pri rutiranju izjednačava jedna po jedna različita dimenzija izvora i odredišta, tada će kroz posmatrani link prolaziti paketi čije izvorište ima prvih i-1 bita proizvoljnih, a sledećih d-i bita jednakih odgovarajućim bitima početnog čvora na posmatranom linku, i čije odredište ima prvih i bita jednakih odgovarajućim bitima adrese krajnjeg čvoa posmatranog linka i preostalih d-i-1 bita proizvoljnih. Početni i krajnji čvor linka se posmatraju u definisom smeru prolaska paketa kroz link. U drugom smeru ovaj link će predstavljati vezu između jednog od 2^{d-i-1} izvorišnih čvorova i jednog od 2^i odredišnih čvorova. U oba slučaja kroz link će prolaziti $2^i \cdot 2^{d-i-1} = 2^{d-1}$ veza, što je rezultat predstavljen u [26].

Pri Valiant balansiranju, između dve tačke prolazi saobraćaj u slučaju kada je prva tačka izvor a druga tačka balansirajući čvor, i u slučaju kada je prva tačka balansirajući čvor a druga tačka odredište. Ako svaki tok ima jedinični intenzitet, i svaki čvor ima izbor od 2^d -1 balansirajućih rutera, to je intenzitet saobraćaja između krajnjeg čvora i balansirajućeg rutera $1/(2^d-1)$. Za svaki uređeni par čvorova pri Valiant balansiranju moguća su dva slučaja komunikacije tih čvorova. U prvom slučaju prvi čvor je izvorišni a drugi balansirajući, dok je u drugom slučaju prvi čvor balansirajući a drugi odredišni. Pošto je intenzitet saobraćaja između krajnjeg čvora i balansirajućeg

rutera $1/(2^d - 1)$, sledi da je intenzitet saobraćaja koji potiče od bilo kog uređenog para čvorova jednak $2/(2^d - 1)$. Uzimajući u obzir broj parova koji komuniciraju moguće je izračunati saobraćaj po linku kao $2^{d-1} \cdot 2/(2^d - 1) = 2^d/(2^d - 1)$. Ovaj broj je nešto veći od 1, i za veći broj dimenzija, zanemarivanjem jedinice u imeniocu dobija se vrednost 1. ■

Na Slici 2-17 prikazani su grafici opterećenosti linkova u topologiji hiperkocke za rutiranje jednom putanjom i za Valiant rutiranje. Kao što je opisano u ovom odeljku, grafici pokazuju garantovano opterećenje za bar jednu permutaciju tokova u kojoj svaki čvor komunicira sa jednim drugim čvorom saobraćajem jediničnog intenziteta. U slučaju Valiant rutiranja opisanom u ovom odeljku, svi linkovi su jednako opterećeni saobraćajem intenziteta $2^d/(2^d - 1)$. Grafik direktnog rutiranja prikazuje kritično opterećenje linka proračunatog korišćenjem Borodin-Hopkroftove donje granice, koje iznosi $\sqrt{2^d/d}\,/d$. Na Slici 2-17 je prikazano da pri Valiant rutiranju opterećenje ostaje približno konstantno sa promenom dimenzije hiperkocke, dok u slučaju rutiranja jednom putanjom garantovano opterećenje na bar jednom od linkova mreže eksponencijalno raste. U hiperkocki dimenzije veće od 10 rutiranja jednom putanjom će imati opterećenije linkove od Valiant rutiranja.



Slika 2-17 Garantovano opterećenje bar jednog linka u topologiji hiperkocke

Predstavljeno poređenje na primeru matrice saobraćaja u kome svaki čvor prima jedan jedinični tok i šalje jedan jedinični tok pokazuje da u topologiji hiperkocke sa više od deset dimenzija Valiant balansiranje omogućuje znatno poboljšanje u odnosu na rutiranje sa jednom putanjom između dva sviča. Valiant balansiranje je pogodno u topogijama sa većim brojem alternativnih putanja, kao što su topologije data centara. U tim topologijama međutim balansiranje preko svakog sviča u mreži sa jednakom verovatnoćom može dati loše rezultate i u tom cilju je moguće prilagoditi Valiant rutiranje izborom verovatnoća odabira balansirajućeg sviča koje nisu jednake za svaki svič u mreži, kao i izborom putanja kojima se paketi usmeravaju između krajnjih svičeva i balansirajućeg sviča.

Valiant balansiranje podrazumeva dvofazno balansiranje u kome se saobraćaj prvo rutira do slučajno izabranog balansirajućeg rutera, pa zatim do odredišta. Rutiranje do slučajno izabranog balansirajućeg rutera rezultuje eliminisanjem zavisnosti saobraćaja u mreži od komunikacionih parova, odnosno matrice saobraćaja, i preostaje zavisnost saobraćaja od ukupnog saobraćaja koga svičevi generišu i terminiraju. Ova osobina je povoljna u data centrima u kojima se matrice saobraćaja brzo menjaju.

2.3.2 Optimizacija dvofaznog balansiranja

Rad [2] predstavlja optimizacione modele za direktno rutiranje i rutiranje sa dvofaznim balansiranjem. U oba modela se optimizuju putanje paketa, a pri dvofaznom balansiranju se optimizuju i verovatnoće usmeravanja paketa ka balansirajućim ruterima i same putanje paketa. Rad razmatra posebne verovatnoće balansiranja paketa za svaki par čvorova. Optimizacione jednačine predstavljene u radu [2] date su formulama (15)-(19).

min Q

$$\forall[t_{ij}] \in \mathrm{T}(\vec{R},\vec{C}), \forall a,b \in N: \tau_{ab} \geq \sum_{m\in N} k_b^{am} \cdot t_{am} + \sum_{m\in N} k_a^{mb} \cdot t_{mb} \tag{1.1}$$

$$\forall i,j,k \in N: \sum_{l\in OUT(k)} x_l^{ij} - \sum_{l\in IN(k)} x_l^{ij} = \begin{cases} +\tau_{ij}, \forall k = i \\ -\tau_{ij}, \forall k = j \\ 0, ina\check{c}e \end{cases} \tag{1.2}$$

$$\forall l \in E: \sum_{i,j\in N} x_l^{ij} \leq Q \cdot C^l \tag{1.3}$$

$$\forall m, a, b \in N: k_m^{ab} \geq 0 \tag{1.4}$$

$$\forall l \in E, \forall i, j \in N: x_l^{ij} \geq 0 \tag{1.5}$$

Model opisan jednačinama (1.1)-(1.5) koristi različite koeficijente balansiranja k_m^{ab} za svaki komunikacioni par. Koeficijenti balansiranja k_m^{ab} se u radu [2] nazivaju generalizovani koeficijenti balansiranja, i za komunikacioni par a i b označavaju koji deo saobraćaja se usmereva preko čvora *m*. U jednačinama linearnog modela oznake t_{ab} predstavljaju saobraćaj između svičeva a i b i čine elemente matrice saobraćaja *T*. Oznake τ_{ab} predstavljaju ukupni saobraćajni kapacitet potreban između svičeva a i b. Ovaj saobraćajni kapacitet u mreži sa balansiranjem obuhvata saobraćaj od čvora a do balansirajućeg čvora b i saobraćaj od balansirajućeg čvora a do čvora b, i predstavljen je formulom (1.6).

$$\tau_{ab} = \max_{[t_{ij}] \in T(\vec{R}, \vec{C})} \left[\sum_{m \in N} k_b^{am} \cdot t_{am} + \sum_{m \in N} k_a^{mb} \cdot t_{mb} \right] \tag{1.6}$$

Jednačina linearnog modela (1.1) sledi iz jednačine (1.6). U jednačini (1.2) linearnog modela x_l^{ab} označava saobraćaj komunikacionog para svičeva a i b koji prolazi kroz link *l*. Oznake $OUT(i)$ i $IN(i)$ predstavljaju skup izlaznih i ulaznih linkova za čvor *i*. Jednačina (1.2) predstavlja uslov održanja saobraćaja i odnosi se na saobraćaj svakog komunikacionog para *i* i *j* i za svaki svič *k* u mreži. Jednačina (1.2) tvrdi da je za svaki svič k različit od i i j razlika suma saobraćaja po ulaznim i izlaznim linkovima čvora *k* jednaka nuli. U slučaju kad je *k* jednak i tada je ta razlika jednaka τ_{ij}. Razlika suma saobraćaja po ulaznim i izlaznim linkovima sviča *k* jednaka je $-\tau_{ij}$ u slučaju kada je *k* jednako *j*.

U jednačini (1.3) oznaka C^l predstavlja kapacitet linka *l*. Oznaka Q predstavlja veličinu koja se minimizuje. U jednačini (1.3) ova veličina označava korišćeni kapacitet linka i pokazuje koliko je zbirni saobraćaj po linku manji od kapaciteta linka. Uslovi (1.4) i (1.5) obezbeđuju da su koeficijenti balansiranja i saobraćaji linkova pozitivne vrednosti.

Promenjive linearnog modela predstavljenog formulama (1.1)-(1.5) su generalizovani koeficijenti balansiranja k_m^{ab}, saobraćaji komunikacionih parova čvorova a i b za svaki link l x_l^{ab} i potrebni saobraćajni kapaciteti između čvorova τ_{ab}.

Nejednakosti (1.1) treba da budu ispunjene za svaku matricu saobraćaja i ima ih beskonačno mnogo, pa se u radu [2] ovaj linearni model dalje transformiše u cilju dobijanja modela sa ograničenim brojem formula. Transformacija ovog modela se vrši na osnovu linearnog modela, u kome su elementi matrice saobraćaja promenljive. Ciljna funkcija ovog modela je maksimizovanje izraza na desnoj strani nejednakosti (1.1) i on pronalaženjem matrice saobraćaja za koju je ispunjena nejednakost (1.1) omogućava proveru ove nejednakosti. Proračunom njegovog dualnog modela, elementi matrice saobraćaja, koji su promenljive primarnog modela nestaju. Transformacija linearnog modela predstavljenog formulama (1.1)-(1.5) vrši se zamenom nejednakosti (1.1), kojih ima beskonačno, sa dobijenim dualnim modelom, koji ima konačan broj formula.

Model predstavljen u radu [2] je opšti model koji pronalazi optimalno balansiranje sa slobodnim izborom načina rutiranja, i sa generalizovanim koeficijentima balansiranja koji se proračunavaju za svaki komunikacioni par. U radu je pokazano da je protok sa optimalnim balansiranjem najmanje jednak polovini protoka sa optimalnim direktnim rutiranjem. Simulacijom praktičnih mreža Internet provajdera u ovom radu utvrđeno je da su protoci pri optimizovanom direktnom rutiranju i optimalnom dvofaznom rutiranju u tim mrežama slični.

2.3.3 Dvofazno balansiranje u mreži sa rutiranjem po najkraćim putanjama

U mrežama Internet provajdera za unutardomensko rutiranje uobičajeno se koristi OSPF protokol i rutiranje najkraćim putanjama. Rad [28] definiše linearni model za određivanje optimalnih koeficijenata dvofaznog balansiranja u slučaju kada se kao osnovno rutiranje koristi rutiranje najkraćim putanjama. Ova vrsta rutiranja nazvana je LB-SPR (eng. *Load Balanced Shortest Path Routing*). Jednačine linearnog modela predstavljenog u radu [28] prikazane su formulama (1.7)-(1.9).

$$\min Q$$

$$\sum_{i=1}^{N} k_i = 1 \tag{1.7}$$

$$\forall l \in E: \frac{\sum_{(i,m)} F_{im}^{l}(k_i r_m + k_m s_i)}{C^l} \leq Q \tag{1.8}$$

$$\forall n \in V: \sum_{l \in IN(n)} L^l - \sum_{l \in OUT(n)} L^l = r_n - s_n \tag{1.9}$$

U jednačini (1.7) oznaka k_i predstavlja koeficijent balansiranja za čvor i. U svakom sviču mreže, izvorišni paketi se usmeravaju ka balansirajućem ruteru i sa verovatnoćom jednakom k_i. Jednačina (1.7) definiše da je suma svih koeficijenata balansiranja jednaka 1.

Jednačina (1.8) predstavlja uslov da je zauzetost linka manja ili jednaka od maksimalne zauzetosti Q koja se minimizira. U jednačini (1.8) koeficijent F_{im}^{l} ima vrednost 0 ili 1 i označava da li saobraćaj komunikacionog para *i* i *m* koristi link *l*. Koeficijent F_{im}^{l} je rezultat proračuna najkraćih putanja u mreži za koji se koristi Dijkstra algoritam [31]. Oznaka r_m predstavlja ukupni saobraćaj terminiran u sviču *m*, dok oznaka s_i predstavlja ukupni saobraćaj generisan u čvoru *i*. Suma u brojiocu na levoj strani formule (1.8) predstavlja saobraćaj linka *l* koji obuhvata saobraćaj svih komunikacionih parova usmeren od izvorišnog rutera *i* do balansirajućeg rutera *m*, predstavljen formulom $F_{im}^{l}k_m s_i$, i saobraćaj svih komunikacionih parova od balansirajućeg rutera *i* do odredišnog rutera *m*, predstavljen formulom $F_{im}^{l}k_i r_m$. U jednačini (1.8) oznaka C^l predstavlja kapacitet linka *l*.

Jednačina (1.9) predstavlja uslov održanja saobraćaja, koji tvrdi da je razlika saobraćaja po ulaznim i izlaznim linkovima jednaka razlici terminiranog i generisanog saobraćaja u sviču. Oznaka $IN(n)$ predstavlja skup ulaznih linkova čvora *n*, dok oznaka $OUT(n)$ predstavlja skup izlaznih linkova čvora *n*. Oznaka L^l predstavlja saobraćaj linka *l* koji se proračunava formulom (1.10).

$$L^l = \sum_{(i,m)} F_{im}^{l}(k_i r_m + k_m s_i) \qquad (1.10)$$

Promenljive linearnog modela predstavljenog formulama (1.7)-(1.10) su koeficijenti balansiranja k_i čiji je broj jednak broju svičeva u mreži. Rad [28] opisuje implementaciju LB-SPR rutiranja i prikazuje rezultate emulacije LB-SPR rutiranja i direktnog rutiranja najkraćim putanjama. Ovi rezultati ilustruju zagušenje prouzrokovano matricom saobraćaja najgoreg slučaja pri direktnom rutiranju najkraćim putanjama. U radu je prikazan i saobraćaj za tu matricu saobraćaja pri LB-SPR rutiranju, koje minimizira maksimalno zagušenje linka, saobraćaj ravnomernije raspoređuje po linkovima mreže i eliminiše preopterećenje linka koje se javlja pri rutiranju najkraćim putanjama. Emulirana mreži se sastojala od 22 virtuelna rutera

realizovana uz pomoć *Xen* [29] virtuelizacione tehnike. Mreža je predstavljala uprošćenje *Exodus* mrežne topologije [30] tako što su svi čvorovi u jednom gradu ekvivalentirani jednim čvorom. Za kritičnu matricu saobraćaja za OSPF rutiranje, pri rutiranju sa balansiranjem kritični link je bio opterećen 43.3%. Takođe, u mreži je pri balansiranju bio opterećen veći broj linkova naspram broja opterećenih linkova pri OSPF rutiranju.

2.3.4 Dvofazno balansiranje u mreži sa ECMP rutiranjem

U data centrima se koristi ECMP rutiranje koje ravnomerno raspoređuje saobraćaj između dve krajnje tačke različitim putanjama sa jednakim cenama (potpoglavlje 2.2). Međutim ECMP nema mogućnost preusmeravanja saobraćaja po neminimalnim putanjama, pa za određene matrice saobraćaja delovi mreže mogu biti preopterećeni, iako postoje manje opterećene putanje kojima bi bilo moguće preneti saobraćaj. LB-ECR rutiranje [32] (eng. *Load Balanced Equal-Cost Routing*) vrši optimizovano dvofazno balansiranje saobraćaja bazirano na ECMP rutiranju i omogućava omogućava maksimizaciju protoka bez gubitaka paketa i unapređenje iskorišćenosti telekomunikacionih mreža data centara.

Korišćenjem dvofaznog rutiranja, u LB-ECR se eliminiše zavisnost raspodele saobraćaja od matrice saobraćaja i zadržava se samo zavisnost od ukupnog generisanog i terminiranog saobraćaja u svakom čvoru. Ova osobina je pogodna u data centrima koji se odlikuju brzo promenljivim matricama saobraćaja. LB-ECR linearni model obuhvata formule (1.11) i (1.12).

$$\min Q$$

$$\sum_{i=1}^{N} k_i = 1 \tag{1.11}$$

$$\forall l \in E: \frac{\sum_{(i,m)} F_{im}^{l} (k_i r_m + k_m s_i)}{C^l} \leq Q \tag{1.12}$$

Linearni model minimizuje maksimalno opterećenje u mreži Q, čime maksimizira protok u mreži. Promenljive modela su koeficijenti balansiranja k_i, koji predstavljaju verovatnoću balansiranja paketa preko sviča i. Jednačina (1.11) definiše da je suma koeficijenata balansiranja jednaka 1. Nejednačine (1.12) postavljaju uslov za saobraćaj svakog linka l u mreži. Brojilac razlomka na levoj strani nejednakosti predstavlja ukupni saobraćaj linka l, koga čini suma doprinosa saobraćaja tokova koji se

kreću od izvorišnih do balansirajućih čvorova, i od balansirajućih do odredišnih čvorova. Koeficijent F_{im}^{l} je realni broj koji uzima vrednosti između 0 i 1, i označava deo saobraćaja između čvorova i i m koji se prenosi linkom l. Ovaj koeficijent se dobija na osnovu proračuna ECMP putanja. Par čvorova i i m doprinose opterećenju linka l saobraćajem od izvorišta i do balansirajućeg rutera m i saobraćajem od balansirajućeg rutera i do odredišnog rutera m. Oznaka C^{l} predstavlja kapacitet linka l. Linearni model ne sadrži jednačine održanja saobraćaja koje bi odgovarale jednačinama (1.9) i (1.10) iz odeljka 2.3.3, pošto putanje paketa određene osnovnim rutiranjem obezbeđuju da ove jednačine budu ispunjene. LB-ECR rutiranje predstavljeno je u radu [32].

2.3.5 SPRm i LB-SPRm rutiranja

SPRm rutiranje [32] je minimalno rutiranje jednom putanjom čiji je cilj ravnomernije raspoređivanje saobraćaja po linkovima u topologijama sa više alternativnih putanja jednake cene. SPRm proračun putanja je baziran na Dijkstra algoritmu [31], pri čemu se na stablo povezuje onaj kandidat čiji čvor-roditelj ima najmanji broj čvorova dece. Dijkstra algoritam originalno ne definiše način na koji se bira koji će kandidat biti povezan na stablu u slučaju da u jednom koraku ima više kandidata koji se mogu povezati sa istom cenom. U slučaju postojanja više ravnopravnih kandidata konzistentan izbor je moguće obezbediti korišćenjem algoritma koji bira kandidata sa najmanjim identifikatorom, što se tipično čini. Izborom kandidata čiji čvor-roditelj ima najmanji broj čvorova dece smanjuju se stepeni čvorova u stablu i saobraćaj se raspoređuje između alternativnih putanja sa jednakom cenom, što SPRm rutiranje čini kandidatom za primenu u data centrima, bilo samostalno, ili sa optimizovanim dvofaznim balansiranjem LB-SPRm. U slučaju više ravnopravnih kandidata bira se kandidat sa najmanjim identifikatorom. Optimizacioni linearni model LB-SPRm algoritma ima iste formule kao i model LB-ECR algoritma, opisan u poglavlju 2.3.4, pri čemu se koeficijenti F_{im}^{l} proračunavaju u skladu sa SPRm rutiranjem.

2.4 Poređenje algoritama rutiranja u datacentrima

U ovom potpoglavlju biće predstavljeni rezultati analize i simulacije rutiranja jednom minimalnom putanjom, rutiranja svim minimalnim putanjama, kao i rutiranja sa

dvofaznim balansiranjem koje se oslanja na rutiranja jednom i svim minimalnim putanjama. U okviru rutiranja jednom minimalnom putanjom biće analizirano rutiranje Dijkstra algoritmom [31], označeno skraćenicom SPR (*Shortest Path Routing*), i modifikaciju ovog rutiranja, SPRm [32], čiji je cilj ravnomernija raspodela putanja po linkovima mreže. SPRm rutiranje je opisano u odeljku 2.3.5. Za navedena rutiranja jednom minimalnom putanjom, SPR i SPRm, biće razmotreni izvedeni dvofazni protokoli sa rutiranjem, odnosno algoritmi LB-SPR i LB-SPRm. Ovi algoritmi rutiranja biće poređeni sa algoritmom za rutiranje svim minimalnim putanjama, tj. ECMP algoritmom i dvofaznim balansiranjem baziranom na ovom algoritmu, tj. LB-ECR algoritmom rutiranja.

Ravnopravno poređenje algoritama rutiranja moguće je pronalaženjem matrice saobraćaja koja dovodi do maksimalnog zauzeća nekog od linkova mreže. U ovom radu za svaki od navedenih algoritama rutiranja biće izračunata kritična matrica saobraćaja, pod uslovom da su saobraćajni zahtevi svih računara u mreži jednaki. Uslov za jednake saobraćajne zahteve je realan zato što predstavlja slučaj ravnopravnih računara. Rezultat proračuna kritične matrice saobraćaja je i kritično zauzeće linka iz koga sledi intenzitet saobraćaja u mreži pri kome će doći do preopterećenja kritičnog linka i gubitka paketa.

Pored teoretske analize čiji je jedan od rezultata granica saobraćaja pri kojoj dolazi do gubitka paketa u mreži, biće predstavljeni rezultati simulacije različitih intenziteta saobraćaja čija je matrica jednaka kritičnoj matrici date mreže, za proračunate koeficijente balansiranja kritičnog saobraćaja. Simulacija obuhvata više vrednosti protoka i daje uvid u stanje mreže i za vrednosti protoka koje su manje i veće od proračunate vrednosti pri kojoj počinju gubici paketa u mreži.

2.4.1 Analiza najgoreg slučaja opterećenja linka

Analiza najgoreg slučaja opterećenja linka omogućava pronalaženje matrice saobraćaja koja dovodi do maksimalnog zagušenja u mreži. Način pronalaženja kritične matrice saobraćaja opisan je u radu [33], i zasniva se na algoritmu za uparivanje maksimalne težine u bipartitnom grafu (eng. *maximum bipartite matching algorithm*). Pri pronalaženju kritične matrice saobraćaja, za svaki link u mreži se formira jedan bipartitni graf. Bipartitni graf se formira tako da čvorovi grafa u obe particije predstavljaju svičeve mreže, dok težine grana između particija predstavljaju opterećenja

linka pri komunikaciji od čvora u prvoj particiji grafa ka čvoru u drugoj particiji. U tako formiranom bipartitnom grafu postoje grane od svakog čvora u prvoj particiji grafa do svakog čvora u drugoj particiji. Težina grane će biti jednaka nuli u slučaju da komunikacioni par koga grana predstavlja ne komunicira preko linka za koga je formiran bipartitni graf. U slučaju rutiranja najkraćom putanjom težine grana mogu imati samo vrednosti 0 i 1, odnosno ili link neće biti na izabranom najkraćem putu, ili će link biti na najkraćem putu u kom slučaju će težina grane grafa biti jednaka 1. U slučaju rutiranja po najkraćim putanjama, saobraćaj će se deliti po tim putanjama i težine grana grafa mogu biti između 0 i 1, zavisno od dela saobraćaja koji prolazi kroz link za koga se formira bipartitni graf.

Cilj algoritma uparivanja maksimalne težine je formiranje bipartitnog grafa u kome svaki čvor ima samo jednu incidentnu granu. Poznato rešenje problema uparivanja je takozvani mađarski algoritam (eng. *Hungarian algorithm*) [34]. Ovaj algoritam se može formulisati u grafovskoj formi i u matričnoj formi. Elementi koji se uparuju predstavljeni su čvorovima bipartinog grafa, odnosno vrstama i kolonama matrice. Težine grana grafa i vrednosti elemenata matrice označavaju cene uparivanja. Algoritam iterativno vrši transformacije grafa, odnosno matrice, koje ne utiču na način uparivanja, ali menjaju strukturu u oblik koji definiše traženo uparivanje.

Primenom algoritma uparivanja dobija se bipartitni graf čije grane imaju maksimalnu težinu u odnosu na sve ostale skupove grana takve da svaki čvor ima jednu incidentnu granu. Na ovaj način dobija se maksimalno opterećenje linka u slučaju u kome svaki čvor šalje saobraćaj jednom drugom čvoru i svaki čvor prima saobraćaj od jednog drugog čvora.

Osnova za izvršenje algoritma uparivanja maksimalne težine je proračun težina grana bipartitnog grafa. Pošto težina grane bipartitnog grafa predstavlja intenzitet saobraćaja po linku koji je posledica komunikacije čvorova na krajevima grane bipartitnog grafa, potrebno je proračunati putanje saobraćaja svakog para čvorova kroz mrežu. Ovaj proračun se vrši na različite načine za direktno rutiranje i rutiranje sa balansiranjem, i detaljno je opisan u pododeljku o implementaciji 4.1.4.2. Pošto se putanje direktnih rutiranja najkraćom putanjom proračunavaju korišćenjem Dijkstra algoritma i njegovih modifikacija, putanje kroz mrežu se proračunavaju na osnovu

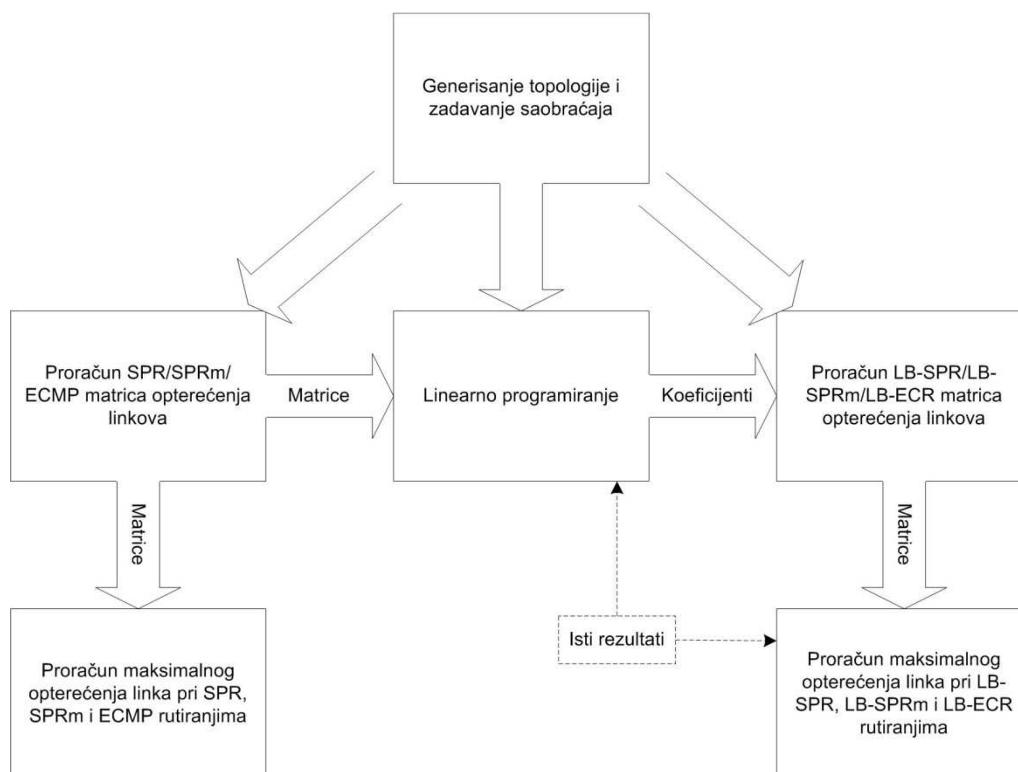
Dijsktra stabala za svaki čvor. U slučaju direktnog rutiranja po svim putanjama iste cene, Dijkstra stablo se pretvara u graf koji može imati zatvorene konture, pošto svaki čvor koji se povezuje na stablo može imati više čvorova-roditelja. I u ovom slučaju svaki čvor ima svoj graf ruta, i na osnovu tih grafova, putanje saobraćaja za sve komunikacione parove moguće je proračunati na način opisan u pododeljku 4.1.4.2. Pri rutiranju minimalnom putanjom sav saobraćaj koji se prenosi između dve krajnje tačke ide jednom putanjom. U tom slučaju, za svaki link u mreži moguće je da link ili prenosi kompletan saobraćaj komunikacionog para ili da uopšte ne prenosi saobraćaj. U slučaju rutiranja po svim minimalnim putanjama, ako postoji više od jedne minimalne putanje između krajnjih tačaka, saobraćaj će biti prenošen po svim minimalnim putanjama. U tom slučaju, svaka putanja će prenosti jednak deo saobraćaja.

Pri balansiranju, između početnog i balansirajućeg sviča, kao i između balansirajućeg i odredišnog sviča, koristi se direktno rutiranje, i put saobraćaja u mreži zavisi od izbora komunikacionih čvorova i od verovatnoća izbora balansirajućih rutera. Pri proračunu saobraćaj se deli srazmerno verovatnoćama izbora balansirajućih rutera, pa se od izvorišnog čvora do balansirajućeg rutera i od balansirajućeg rutera do odredišnog čvora opterećenosti linkova proračunavaju kao za direktno rutiranje.

Na osnovu izračunatog bipartitnog grafa sa čvorovima koji su upareni granama čiji je zbir težina maksimalan, opterećenje linka se izračunava sabiranjem težina grana. Pošto se procedurom uparivanja maksimalne težine za svaki link pronađu komunikacioni parovi koji dovode do maksimalnog opterećenja tog linka, za kritičnu komunikacionu matricu za celu mrežu se bira ona koja izaziva maksimalno opterećenje linka.

Blok šema prikazana na 2-18 predstavlja redosled proračuna potrebnih za dobijanje kritičnog opterećenja linka u mreži za ispitivane algoritme rutiranja. Prvi korak je generisanje topologije i zadavanje matrice saobraćaja. Zatim se na osnovu tih podataka vrši proračun SPR i SPRm matrica opterećenja linkova i proračun ECMP matrica opterećenja linkova. Sledeći korak je pronalaženje koeficijenata balansiranja za LB-SPR, LB-SPRm i LB-ECR koje se vrši korišćenjem linearnog programiranja. Na osnovu dobijenih koeficijenata, topologije i matrice saobraćaja vrši se proračun i matrica opterećenja linkova pri LB-SPR, LB-SPRm i LB-ECR balansiranjima. Na

osnovu matrica opterećenosti linkova proračunavaju se maksimalna opterećenja linkova u mreži za sve ispitivane algoritme rutiranja. Maksimalna opterećenja linkova za LB-SPR, LB-SPRm i LB-ECR se poklapaju sa proračunatim vrednostima maksimalnog zagušenja *Q* iz njihovih linearnih modela, što je naznačeno na Slici 2-18.



Slika 2-18 Dijagram proračuna

Na opisani način se analitičkim putem dobija optećenje kritičnog linka u mreži i matrica saobraćaja koja generiše to opterećenje. Pročun je vršen za (10,3) *fat-tree* topologiju, koja se sastoji iz tri nivoa svičeva sa po deset portova. Ova topologija sadrži 75 svičeva i 125 servera. Korišćena *dragonfly* topologija ima *g*=17 grupa sa po *a*=4 sviča u grupi, i po *h*=4 veza svakog sviča prema drugim grupama. *Dragonfly* topologija sa navedenim parametrima ima 68 svičeva sa po devet portova i 136 servera. *Flattened butterfly* topologija ima dimeziju *d*=6, pa sadrži 64 osmoportna sviča i 128 servera. Kapaciteti svih linkova pri analizi i simulaciji su bili 1*Gb*/*s*. Tabela 2-1 prikazuje maksimalni ukupni protok bez gubitaka normalizovan kapacitetom linka.

Rezultati predstavljeni u Tabeli 2-1 pokazuju da LB-ECR daje bolje rezultate od ostalih ispitivanih algoritama rutiranja za *dragonfly* topologiju. U *fat-tree* topologiji svi balansirajući protokoli sem Valiant rutiranja daju isti rezultat maksimalnog protoka bez

gubitaka saobraćaja, dok u *flattened butterfly* topologiji LB-ECR i Valiant rutiranje daju isti rezultat koji je bolji od rezultata ostalih algoritama balansiranja.

Tabela 2-1 Maksimalni normalizovani protoci za različite algoritme rutiranja i date toplologije

Algoritam rutiranja	SPR	LB-SPR	SPRm	LB-SPRm	ECMP	LB-ECR	VBR
Fat-tree	5	125	25	125	125	125	89.825
Dragonfly	11.333	69.515	11.333	72.009	17	76.81	60.176
Flattened butterfly	5.33	40.58	4.571	43.63	29.538	64	64

2.4.1.1 Zavisnost kvaliteta dvofaznog balansiranja od osnovnog rutiranja

Rezultati prikazani u Tabeli 2-1 pokazuju da je za mereni slučaj LB-ECR bolji ili jednak od LB-SPR i LB-SPRm balansiranja. Sledeći dokaz pokazuje da je u svakoj topologiji u kojoj ECMP daje bolji ili jednak rezultat u odnosu na SPR (SPRm), LB-ECR takođe daje bolji ili jednak rezultat u odnosu na LB-SPR (LB-SPRm).

Lema 2.2: Ukoliko je u mreži ECMP bolji ili jednak od SPR, LB-ECR je bolji ili jednak od LB-SPR ukoliko LB-ECR i LB-SPR koriste iste koeficijente balansiranja.

Dokaz:

Posmatramo mrežu u kojoj se vrši balansiranje sa zadatim koeficijentima i zadatom matricom saobraćaja. Saobraćaj u mreži sa balansiranjem jednak je saobraćaju u istoj mreži sa direktnim rutiranjem za matricu saobaćaja u kojoj svi čvorovi komuniciraju sa balansirajućim svičevima, istim intenzitetom kao u mreži sa balansiranjem. U tom slučaju, LB-SPR rutiranje sa prvom matricom saobraćaja, ekvivalentno je SPR rutiranju sa drugom matricom saobraćaja, dok je LB-ECR rutiranje sa prvom matricom saobraćaja jednako ECMP rutiranju sa drugom matricom saobraćaja. Kako je na osnovu prepostavke ECMP bolji ili jednak SPR rutiranju za datu matricu saobraćaja, to je na osnovu prethodne rečenice i LB-ECR bolji ili jednak od LB-SPR za rutiranje u mreži sa datim skupom koeficijenata.

Teorema 2.3: U svakoj topologiji u kojoj ECMP daje bolji ili jednak rezultat u odnosu na SPR, LB-ECR takođe daje bolji ili jednak rezultat u odnosu na LB-SPR.

Dokaz:

Za datu mrežu i matricu saobraćaja, razmatramo LB-SPR rutiranje sa svojim skupom koeficijenata i LB-ECR rutiranje sa svojim skupom koeficijanata. Na osnovu leme 2.1 LB-ECR daje jednake ili bolje rezultate u odnosu na LB-SPR rutiranje za LB-SPR skup koeficijenata. Na osnovu postupka dobijanja LB-ECR koeficijenata LB-ECR daje bolje ili jednake rezultate pri rutiranju sa LB-ECR koeficijentima nego sa LB-SPR koeficijenatima. Iz prethodne dve tvrdnje sledi da LB-ECR balansiranje za istu matricu saobraćaja daje bolje ili jednake rezultate u odnosu na LB-SPR balansiranje, pod pretpostavkom da ECMP rutiranje u toj mreži daje bolji ili jednak rezultat od SPR rutiranja. ■

2.4.2 Simulacija rutiranja u različitim topologijama data centara

U cilju ispitivanja saobraćaja u mrežama data centara za protoke koji su manji i veći od proračunatih maksimalnih vrednosti saobraćaja bez gubitaka izvršene su simulacije rada ovih mreža u mrežnom simulatoru *ns*-3 [35]. U pododeljku 2.4.2.1 opisan je način formiranja simulacije u mrežnom simulatoru *ns*-3. Pododeljak 2.4.2.2 opisuje način simulacije rutiranja sa dvofaznim balansiranjem u *ns*-3. Rezultati simulacije predstavljeni su u pododeljku 2.4.2.3, dok pododeljak 2.4.2.4 analizira ponašanje simulacionih krivih u oblasti gubitaka paketa.

2.4.2.1 Formiranje simulacije u mrežnom simulatoru *ns-3*

Mrežni simulator *ns*-3 omogućava simulacije različitih tipova telekomunikacionih mreža. Ovaj simulacioni softver napisan je u C++ programskom jeziku i formiranje simulacije vrši se korišćem *ns*-3 klasa. Na početku generisanja topologije generišu se svičevi, koji su predstavljeni *ns*-3 objektima klase *Node*. U *ns*-3 simulatoru, više objekata tipa *Node* čuva se u objektu tipa *NodeContainer*. Objekat tipa *NodeContainer* koristi se i za kreiranje objekata svičeva, korišćenjem njegove metode *Create*(*uint32_t n*) čiji je argument broj svičeva koje treba kreirati. Potom se kreiranim svičevima dodaje IP protokol stek korišćenjem objekta *ns*-3 klase *InternetStackHelper*. Za dodavanje IP protokol steka koristi se funkcija klase *InternetStackHelper* pod nazivom *Install*(*NodeContainer c*). Argument ove funkcije je objekat *ns*-3 kontejnera čvorova, i IP protokol stek se dodaje svim čvorovima u kontejneru.

Linkovi topologije se kreiraju korišćenjem objekta *ns*-3 klase *PointToPointHelper*. Objekti klase imaju metod *Install*(*NodeContainer c*), čiji je argument kontejner koji sadrži čvorove izmedu kojih treba formirati link. Pre formiranja linka, u objektu klase *PointToPointHelper* podešavaju se brzina i kašnjenje linka. Brzina linka se podešava korišćenjem funkcije *SetDeviceAttribute*(*std*::*string name*, *const AttributeValue &value*) čiji su argumenti ime i vrednost atributa sviča. Atribut koji predstavlja brzinu linka naziva se "*DataRate*", dok je vrednost atributa objekat tipa *AttributeValue*, i u slučaju da je brzina linka 10Gbit/s on se može zadati kao *StringValue*("10*Gbps*"). Klasa *StringValue* je nasleđena iz klase *AttributeValue* i omogućava zadavanje vrednosti atributa korišćenjem stringa. Kašnjenje linka se podešava korišćenjem funkcije *SetChannelAttribute*(*std::string name*, *const AttributeValue &value*) u kome ime atributa ima vrednost "*Delay*" dok se vrednost atributa može zadati npr. u formi *StringValue*("2*us*").

Adrese se interfejsima dodeljuju korišćenjem *ns*-3 klase *Ipv4AddressHelper*. Skup iz koga se dodeljuju adrese specificira se korišćenjem funkcije *SetBase*(*Ipv4Address network*, *Ipv4Mask mask*, *Ipv4Address base=*"0.0.0.1"). Argumenti ove funkcije su mrežna adresa i maska, kao i početna adresa hosta. Adrese specificirane argumentima funkcije dodeljuju se svičevima sadržanim u kontejneru *c* korišćenjem funkcije *Assign*(*const NetDeviceContainer &c*).

Mreže data centara imaju pravilne topologije i sadrže veliki broj svičeva i servera. Pravilne topologije ovih mreža omogućavaju algoritamsko generisanje topologija proizvoljne veličine. U okviru simulacionog ispitivanja osobina rutiranja u data centrima implementirani su generatori topologija za *fat-tree, flattened butterfy* i *dragonfly* topologije, u skladu sa definicijama ovih topologija opisanih u odeljcima 2.1.5, 2.1.6 i 2.1.7 respektivno.

Pošto je u čvorovima simulacione mreže instaliran IP stek i, po potrebi, sloj za dvofazno balansiranje, formirani linkovi i dodeljene adrese, simulacioni saobraćaj se zadaje tako što se u čvorovima formiraju aplikacije koje komuniciraju. UDP saobraćaj moguće je generisati formiranjem generatora paketa u jednom čvoru korišćenjem *OnOffHelper* klase i aplikacije koja prihvata paketa u drugom čvoru korišćenjem *PacketSinkHelper* klase. Objekat tipa *ApplicationContainer* koji sadrži generatorsku

aplikaciju formira se korišćenjem metoda *Install*(*Ptr<Node> node*) čiji je argument čvor koji treba da sadrži generator UDP saobraćaja. Trenutak startovanja i zaustavljanja aplikacije zadaje se korišćenjem metoda *Start*(*Time start*) i *Stop*(*Time stop*) objekta klase *ApplicationContainer* koji sadrži aplikaciju.

Klasa *OnOffHelper* omogućava formiranje aplikacije koja ima stanje u kome generiše pakete i stanje pauze. Pri formiranju aplikacije potrebno je zadati atribute koji određuju trajanje ova dva stanja, brzinu generisanja paketa u stanju kada se paketi generišu, veličinu generisanih paketa, i ukupni broj bajtova koje treba preneti. Ovi parametri podešavaju se korišćenjem funkcije *SetAttribute*(*std::string name, const AttributeValue &value*). Pored zadavanja ovih atributa zadaju se i vremena početka i kraja rada aplikacije, i aplikacija se instalira na željeni čvor na isti način kao u slučaju aplikacije koja prima pakete.

Detaljne informacije o korišćenju mrežnog simulatora *ns*-3 nalaze se u dokumentaciji *ns*-3 projekta [36].

2.4.2.2 Simulacija rutiranja sa balansiranjem u mrežnom simulatoru *ns*-3

U odeljku 4.1.5 opisana je implementacija balansiranja u okviru koje se za usmeravanje paketa preko balansirajućeg sviča koristi izvorno rutiranje sa delimičnim spiskom svičeva. Pošto izvorno rutiranje nije implementirano u mrežnom simulatoru *ns*-3, u okviru ovog rada je za potrebe usmeravanja paketa preko balansirajućih svičeva, u *ns*-3 implementirano dodatno zaglavlje između mrežnog sloja i sloja transporta. Ovo zaglavlje omogućava dvofazno balansiranje paketa i sadrži informaciju o krajnjem odredištu paketa i informaciju o korišćenom protokolu transportnog sloja. Pri balansiranju, u polje odredišta mrežnog sloja upisuje se adresa sviča za balansiranje, dok se u dodatno zaglavlje upisuje krajnje odredište paketa. Po pristizanju u balansirajući svič, posle obrade u mrežnom sloju biće analizirano dodatno zaglavlje, pri čemu se odredišna adresa paketa poredi sa adresom sačuvanom u dodatnom zaglavlju. Ako su te adrese različite, odredišna adresa paketa postaje adresa sačuvana u dodatnom zaglavlju i paket se šalje ka krajnjem odredištu. Po pristizanju u krajnje odredište, odredišna adresa paketa biće ista kao adresa sačuvana u dodatnom zaglavlju, i u tom slučaju se pri obradi dodatnog zaglavlja paket prosleđuje odgovarajućoj funkciji transportnog sloja na osnovu informacije o protokolu transportnog sloja sačuvane u

dodatnom zaglavlju. Rutiranje sa balansiranjem pomoću dodatnog zaglavlja može se koristiti nezavisno od algoritma proračuna putanja i od toga da li se paketi prosleđuju po jednoj ili više putanja, i ono je korišćeno u svim simulacijama sa balansiranjem u mrežnom simulatoru *ns*-3.

Funkcionalnost sloja za dvofazno balansiranje sadržana je u klasi *LbRouting*. Pri slanju paketa poziva se metod *Send*(*Ptr<Packet> packet*, *Ipv4Address source*, *Ipv4Address destination*, *uint8_t protocol*, *Ptr<Ipv4Route> route*) u kome se određuje adresa za balansiranje i formira zaglavlje za dvofazno rutiranje. Pri prijemu paketa poziva se metod *Receive*(*Ptr<Packet> p*, *Ipv4Header const &ip*, *Ptr<Ipv4Interface> incomingInterface*) u kome se proverava da li je paket primljen u balansirajućem ruteru ili odredišnom ruteru.

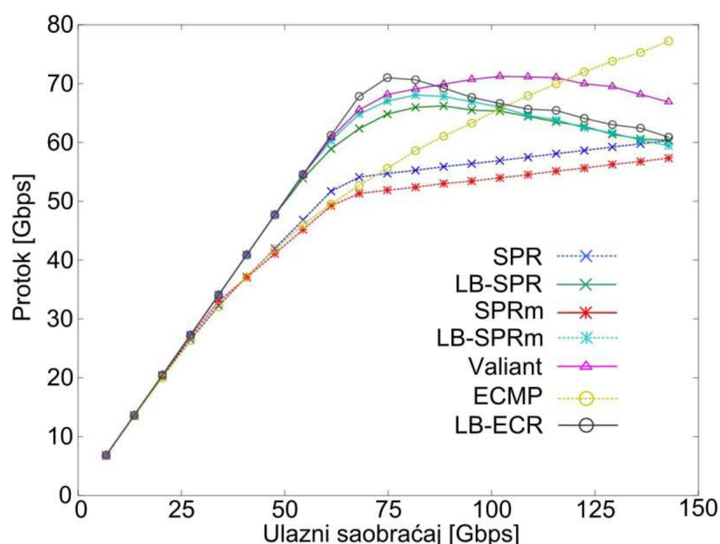
Klasa *LbRouting* koristi klasu *LbOptionHeader* koja sadrži metode za serijalizaciju i deserijalizaciju sadržaja zaglavlja. Objekat ove klase pri slanju paketa formira zaglavlje za dvofazno balansiranje na osnovu podataka o korišćenom protokolu transportnog sloja i o odredištu paketa. Pri prijemu paketa objekat ove klase se koristi za čitanje informacija (o protokolu transportnog sloja i o odredištu paketa) iz zaglavlja za dvofazno balansiranje. Klase *LbRouting* i *LbOptionHeader* nalaze se u folderu *src*/*lb*/*model* u okviru osnovnog foldera *ns*-3 simulatora.

Dodavanje sloja za dvofazno balansiranje omogućeno je klasama *LbHelper* i *LbMainHelper*. Klasa *LbHelper* omogućava dodavanje sloja za dvofazno balansiranje simulacionom čvoru korišćenjem metoda *Create*(*Ptr<Node> node*, *uint32_t *switches*, *uint32_t swcount*, *double *coeficients*). Ova metoda instalira sloj za dvofazno balansiranje u čvor *node*, i čvoru predaje i adrese i broj balansirajućih svičeva kao i koeficijente balansiranja. Klasa *LbMainHelper* omogućava dodavanje sloja za dvofazno balansiranje čvorovima sadržanim u kontejneru korišćenjem funkcije *Install*(*LbHelper &lbHelper*, *NodeContainer nodes*, *uint32_t *switches*, *uint32_t swcount*, *double *coeficients*). Prvi argument ove funkcije je objekat klase *LbHelper*. U okviru ove funkcije, pozivi se za svaki čvor sadržan u skupu simulacionih čvorova *nodes* prosleđuju objektu tipa *LbHelper*. Objekat tipa *LbHelper* izvršava instalaciju sloja za dvofazno balansiranje u čvor. Klase *LbHelper* i *LbMainHelper* nalaze se u folderu *src*/*lb*/*helper* u okviru osnovnog foldera *ns*-3 simulatora.

2.4.2.3 Poređenje protokola za rutiranje u data centrima kroz simulacije

Saobraćaj u topologijima *flattened butterfly*, *dragonfly* i *fat-tree* u slučajevima rutiranja SPR, SPRm, ECMP, Valiant balansiranje, LB-SPR, LB-SPRm, LB-ECR poređen je korišćenjem mrežnog simulatora *ns*-3. U okviru simulacije mreže data centra su formirane u skladu sa definicijama topologija predstavljenim u odeljcima 2.1.5, 2.1.6 i 2.1.7 respektivno. Za usmeravanje paketa pri dvofaznom rutiranju korišćena je implementacija dodatnog zaglavlja opisana u odeljku 2.4.2.2. Poređeni su ukupni protoci, pri kritičnoj matrici saobraćaja koja dovodi do gubitaka paketa pri najmanjem ukupnom protoku, proračunatog na način opisan u odeljku 2.4.1.

Pri simulacijama je korišćen UDP protokol, koji omogućava merenje protoka koji zavisi samo od načina rutiranja, ne i od regulacionih mehanizama samog transportnog protokola. TCP protokol raspolaže mehanizmima regulacije protoka koji dovode do usporenja toka u slučajevima zagušenja, pa mereni protok u mreži ne bi zavisio samo od načina rutiranja već i od algoritma i parametara regulacije TCP protoka. Pri simulaciji su korišćeni parametri topologija opisani u odeljku 2.4.1.

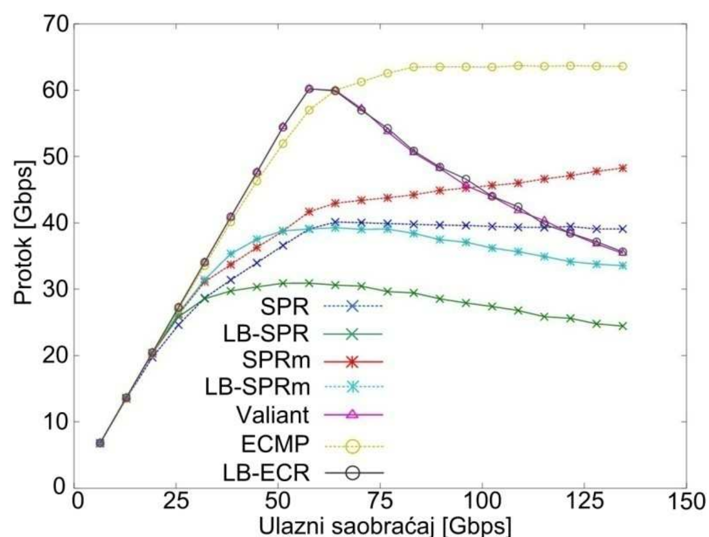


Slika 2-19 Protok u *dragonfly* topologiji

Rezultati simulacije za *dragonfly* topologiju prikazani su na Slici 2-19. Uz ovu sliku, kao i ostale slike protoka, potrebno je koristiti informacije iz Tabele 2-1 koja je prikazana u odeljku 2.4.1. Tabela 2-1 pokazuje da je maksimalni protok bez gubitaka za kritičnu matricu saobraćaja u slučaju *dragonfly* topologije najveći za LB-ECR rutiranje i da iznosi 76.81Gbit/s. Na Slici 2-19 oblast protoka u kojoj krive linearno rastu sa

nagibom jednakim 1 označava komunikaciju bez gubitaka paketa. Sa Slike 2-19 se vidi da LB-ECR dostiže veći protok bez gubitaka paketa od ostalih ispitivanih protokola.

Protok saobraćaja u *flattened butterfly* topologiji prikazan je na Slici 2-20. U *flattened butterfly* topologiji, najbolji rezultat daju LB-ECR rutiranje i Valiant rutiranje. Ova dva algoritma rutiranja daju isti rezultat zato što LB-ECR proračun koeficijenata rezultuje jednakim koeficijentima balansiranja za sve svičeve u *flattened butterfy* mreži. Jednaki koeficijenti balansiranja LB-ECR i Valiant rutiranja su posledica simetrije *flattened butterfy* topologije, u kojoj su direktno povezani svičevi čiji se identifikatori razlikuju u jednom bitu. Zbog ove osobine svi svičevi u *flattened butterfy* mreži imaju isti broj direktnih suseda, suseda na rastojanju dva itd. Uz to, broj različitih putanja do svih suseda na istom rastojanju je jednak. Navedene osobine rezultuju simetričnom raspodelom ruta u mreži pri ECMP rutiranju, pa su svi balansirajući ruteri pri LB-ECR rutiranju ravnopravni, odnosno imaju jednake koeficijente balansiranja. Valiant rutiranje u *flattened butterfly* dostiže performanse ECMP rutiranja u *fat-tree* topologiji [10].

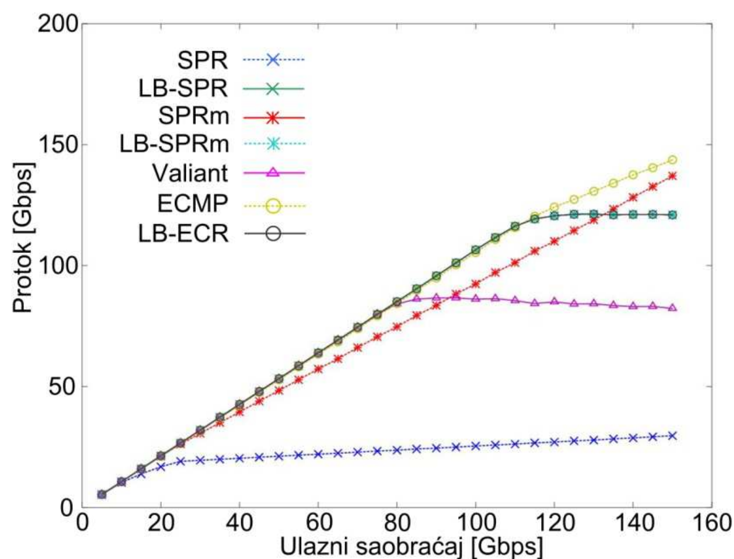


Slika 2-20 Protok u *flattened butterfly* topologiji

Maksimalni protok bez gubitaka za kritičnu matricu saobraćaja za LB-ECR i Valiant protokole u *flattened butterfly* topologiji naveden u Tabeli 2-1 je 64Gbit/s. Ovaj rezultat je u skladu sa Slikom 2-20. Posle ovog maksimuma, protok pri LB-ECR i Valiant rutiranju opada. Sa druge strane, na Slici 2-20 je primetan rast prijemnog protoka pri ECMP rutiranju. Iz Tabele 2-1 se može očitati da gubici pri ECMP rutiranju počinju već od 29.5Gbit/s, pa sledi da je rast prijemnog protoka pri ECMP rutiranju

iznad ove vrednosti praćen gubicima paketa. Rast protoka praćen gubicima paketa je moguć jer zagušenje ne nastaje istovremeno u svim delovima mreže, i protok može rasti u delovima u kojima zagušenje nije nastupilo.

Zavisnost prijemnog protoka od predajnog protoka u *fat-tree* topologiji prikazana je na Slici 2-21. Tabele 2-1 pokazuje da je maksimalni protok bez gubitaka za kritičnu matricu saobraćaja u *fat-tree* topologiji jednak za LB-SPR, LB-SPRm i LB-ECR i da ima veću vrednost u odnosu na preostala analizirana rutiranja.



Slika 2-21 Protok u *fat-tree* topologiji

Iako je Valiant balansiranje dvofazno balansiranje kao LB-SPR, LB-SPRm i LB-ECR, maksimalna vrednost saobraćaja bez gubitaka za Valiant balansiranje je niža. Ovo je posledica efekta da u *fat-tree* topologiji, prikazanoj na Slikama 2-5, 2-6 i 2-7, jednaki koeficijenti balansiranja za sve svičeve mogu dovesti do nepotrebnog prenosa paketa iz gornjeg nivoa topologije ka svičevima u nižim nivoima, pa zatim opet ka gornjem nivou, pa tek onda ka odredišnom serveru. Navedeni slučaj se dešava kada se balansirajući svič nalazi u različitoj SUBFT podmreži ispod gornjeg nivoa svičeva (Slika 2-6), u odnosu na svičeve povezane na izvorišni i odredišni server. Tada paket putuje od izvorišnog server do gornjeg nivoa, pa zatim do balansirajućeg sviča, pa opet do gornjeg nivoa da bi preko njega mogao doći do odredišnog servera.

LB-SPR, LB-SPRm i LB-ECR rutiranja sa balansiranjem na osnovu optimizacionih proračuna koriste samo gornji nivo svičeva za balansiranje, pa ne vrše

opisano nepotrebno prosleđivanje paketa nadole i nagore unutar *fat-tree* topologije. Sva tri protokola imaju iste koeficijente balansiranja za svičeve u gornjem nivou.

Na Slici 2-21 su primetne dobre performanse ECMP rutiranja, koje ima isti maksimalni porotok bez gubitaka kao LB-SPR, LB-SPRm i LB-ECR rutiranja. Ovo je posledica činjenice da pri kretanju paketa između nivoa *fat-tree* topologije ova rutiranja koriste sve raspoložive linkove. Sa druge strane, rutiranja najkraćim putem SPR i SPRm ne koriste sve raspoložive linkove topologije između nivoa, pa su im performanse znatno lošije. Zahvaljujući ravnomernijem raspoređivanju najkraćih putanja po linkovima topologije SPRm rutiranje u *fat-tree* topologiji ostvaruje pet puta bolji rezultat od SPR rutiranja. SPRm rezultuje stablom sa manjim stepenima čvorova, pa se saobraćaj čvora ka različitim odredištima ravnomernije raspoređuje po raspoloživim linkovima povezanim na taj čvor. Svaki čvor u mreži izračunava svoje stablo i postojanje istog čvora većeg stepena u više stabala dovodi do preklapanja putanja između različitih komunikacionih parova. Ovo rezultuje zagušenjem u tom čvoru. To pokazuje primer SPR rutiranja u *fat-tree* topologiji u kome zagušenje nastaje u najvišem nivou topologije. U slučajevima *dragonfly* i *flattened butterfly* topologije uticaj mogućnosti biranja čvora roditelja sa najmanjom cenom znatno manje utiče na rezultujuće stepene čvorova stabla u odnosu na *fat-tree* topologiju, pa su kritični protoci SPR i SPRm rutiranja bliski.

LB-ECR rutiranje u topologijama data centara omogućava maksimizaciju protoka ravnomernim raspoređivanjem saobraćaja po linkovima mreže, čime se postižu dobre performanse prenosa nepovoljnih matrica saobraćaja. LB-ECR korišćenjem osnovnog ECMP rutiranja omogućava iskorišćenje alternativnih putanja sa istom cenom, dok dvofazno balansiranje omogućava ravnomerno raspoređivanje saobraćaja po putanjama sa različitim cenama. U topologijama data centara SPR i SPRm rutiranja najkraćom putanjom ne iskorišćavaju alternativne putanje iste cene kao ECMP, pa predstavljeni rezultati pokazuju da LB-SPR i LB-SPRm imaju u *flattened butterfly* i *dragonfly* topologiji manji garantovani protok od LB-ECR rutiranja. U fat-tree topologiji LB-SPR, LB-SPRm i LB-ECR imaju isti garantovani protok pošto svi ravnomerno balasiraju saobraćaj preko rutera u gornjem nivou topologije, a od servera ka balansirajućem ruteru u gornjem nivou, kao i od balansirajućeg rutera u gornjem

nivou do servera postoji po jedna putanja, pa rutiranje po putanjama sa istim cenama između balansirajućih rutera i krajnjih tačaka veze ne dolazi do izražaja.

2.4.2.4 Analiza simulacionih krivih u oblasti gubitaka paketa

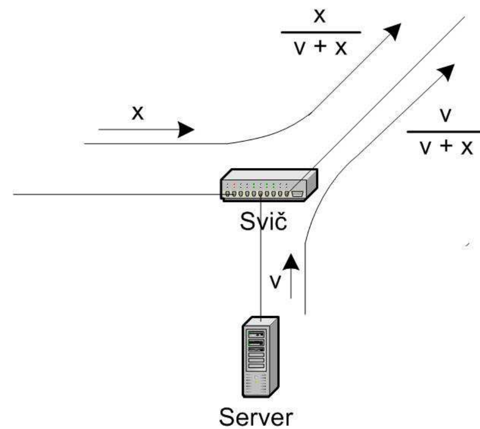
Za male ulazne protoke svi protokoli rutiranja rezultuju linearnom zavisnošću prijemnog protoka od predajnog protoka, sa nagibom jednakim 1. To znači da sav poslati saobraćaj stiže do odredišta i da nema izgubljenih paketa. Kada odbacivanje paketa počne, nagib krive se smanjuje. U oblasti gubitaka paketa na Slikama 2-19, 2-20 i 2-21 primetno je različito ponašanje krivih za direktna rutiranja u koja spadaju SPR, SPRm i ECMP, i krivih za rutiranje sa dvofaznim balansiranjem u koje spadaju LB-SPR, LB-SPRm i LB-ECR. Kod krivih sa direktnim rutiranjem rast prijemnog protoka se smanjenim intenzitetnom nastavlja i u oblasti sa gubicima, dok pri rutiranju sa balansiranjem u oblasti sa gubicima prijemni protok počinje da opada sa rastom predajnog protoka.

Mrežni simulator *ns*-3 u simulaciji IP mreža koristi izlazne redove za čekanje. Svaki port koristi FIFO redove za čekanje sa odbacivanjem paketa na kraju reda za čekanje. Pojava pada protoka u oblasti sa gubicima paketa može se objasniti produženim putem paketa pri dvofaznom balansiranju, pri čemu u izlaznim redovima za čekanje novi saobraćaj potiskuje saobraćaj koji je već u mreži i prolazi kroz svič. U skladu sa principom jednakog tretiranja svih paketa, novi saobraćaj koji u mrežu ulazi u svičevima na putu toka u svakom sviču će sve više smanjivati protok toka. Pri rutiranju sa balansiranjem broj svičeva kroz koji paket prolazi je uvećan pa efekat potiskivanja toka brzo dovodi do smanjenja protoka na većem broju tokova i do smanjenja ukupnog protoka.

Na Slici 2-22 predstavljen je primer jednog sviča iz mreže u kome se mešaju saobraćaj u mreži i ulazni saobraćaj. U delu mreže sa Slike 2-22 zbir tokova x i v je veći od kapaciteta linka na desnoj strani sviča. Kapacitet linka je normalizovan na 1.

Sa x je označen saobraćaj koji prolazi kroz svič. Za kapacitet na desnoj strani prikazanog sviča konkurišu tok protoka x i tok protoka v koga šalje prikazani server. Kako ovi tokovi ravnopravno konkurišu za kapacitet oni će dobiti delove kapaciteta linka koji su proporcionalni njihovim brzinama, odnostno $x/(x+v)$ i $v/(x+v)$. Ako je kapacitet linka normalizovan na 1, to će tokovi x i v zauzeti $x/(x+v)$ i $v/(x+v)$ delove

kapaciteta linka, i protok kroz svič će biti smanjen sa *x* na *x*/(*x*+*v*). U slučaju rutiranja sa balansiranjem tokovi prolaze kroz više svičeva i njihov protok će biti značajno smanjen, što dovodi i do ukupnog smanjenja protoka u mreži.



Slika 2-22 Multipleksiranje paketskog saobraćaja

Efekat pada protoka je najizraženiji u *flattened butterfly* i *dragonfly* mreži zato što u njima u svim svičevima saobraćaj ulazi u mrežu. Pošto je pri rutiranjima sa balansiranjem cilj da mreža radi u oblasti protoka u kojoj nema gubitaka paketa, efekat pada protoka nije značajan. Rad u oblasti gubitaka paketa nije pogodan za data centre zbog nemogućnosti prenosa tokova u zadatom vremenu usled gubitka paketa, što je preduslov rada većine aplikacija u data centrima.

3 Ažuriranje lukap tabela

Internet ima sve više korisnika, i svaki korisnik ima mogućnost korišćenja sve brže veze ka Internetu. Ovaj trend rezultuje stalnim povećanjem količine paketskog saobraćaja koje je omogućeno povećanjem brzine linkova zahvaljujući optičkim vlaknima i povećanjem brzine rutera korišćenjem sve bržih čipova i boljih algoritama za obradu i usmeravanje paketa. Važna operacija pri prosleđivanju paketa kroz ruter je lukap, u toku koga se za odredišnu adresu paketa pronalazi link na koga treba proslediti paket. Paket se u ruteru korišćenjem krosbara prenosi sa porta na koga je pristigao na odlazni port proračunat lukap procedurom.

Lukap koristi tabelu rutiranja, koja sadrži informacije o odredišnim adresama koje su dostupne sa rutera i odlaznim linkovima na koje treba prosleđivati pakete. Informacije o odredišnim adresama se čuvaju u parovima koji sadrže adresu i masku. Svaki takav par definiše grupu Internet adresa, koje se na bitima na kojima maska ima vrednost 1 poklapaju da adresom sadržanom u tom paru. Za svaki par je definisan izlazni link na koga ruter treba da prosledi paket na putu ka odredištu. Paket se pridružuje paru po principu najdužeg poklapanja, tj. adresa se pridružuje onom paru sa kojim ima poklapanje na najvećem broju bita. Naime, u tabeli rutiranja može biti više prefiksa koji se poklapaju sa odredišnom adresom paketa.

Tabela rutiranja može imati veliki broj unosa. Sa druge strane, lukap je potrebno brzo obavljati zbog velike brzine pristizanja paketa po savremenim linkovima velikih protoka. Iz ovog razloga se za funkciju lukapa koriste optimizovani logički moduli u okviru čipova ugrađenih u ruter. Ubrzavanje pronalaženja najdužeg poklapanja je osnovni cilj pri dizajnu ovih modula, da bi oni mogli funkcionisati sa linkovima velikih protoka. Druge važne osobine koje ovi moduli treba da zadovolje su što manje korišćenje skupe memorije koje podiže cenu ovih modula pa samim tim i rutera, kao i što manji uticaj ažuriranja u toku rada rutera na proces lukapa.

Da bi se moglo ostvariti brzo pronalaženje najdužeg poklapanja, te odredišnih linkova za pakete koji pristižu velikim brzinama, savremeni lukap moduli koriste složene strukture podataka u koje smeštaju informacije iz lukap tabele [37]. Ove

strukture podataka omogućavaju pronalaženje velikog broja najdužih poklapanja u jedinici vremena.

U toku rada rutera topologija mreže se menja i ruteri neprekidno razmenjuju informacije prema dinamičkim protokolima rutiranja i proračunavaju tabele rutiranja koje odgovaraju tekućem stanju mreže. Pošto se usmeravanje paketa u ruterima vrši u lukap modulu, u cilju ispravnog usmeravanja paketa u mreži neophodno je posle svake promene u tabeli rutiranja izvršiti i odgovarajuću promenu u strukturama podataka lukap modula, i ovaj proces je zadatak algoritma ažuriranja lukap modula.

Algoritam ažuriranja lukap modula ima složen zadatak, koji je posledica složenosti struktura podataka u lukap modulu. Ove strukture podataka su dizajnirane sa primarnim ciljem brzog pronalaženja najdužeg poklapanja odredišne adrese dolaznog paketa sa unosima tabele rutiranja, i sekundarnim ciljem što manjeg utroška resursa čipa. Zbog navedenih zahteva, strukture podataka lukap modula mogu biti veoma složene, pa i algoritam ažuriranja ima složen zadatak. Pored potrebe za popunjavanjem struktura podataka lukap modula u skladu sa lukap algoritmom, algoritam ažuriranja treba i da vodi računa o uticaju upisa u lukap modul na lukap proces koji je u toku, kao i na popunjavanje struktura podataka na način koji će optimalno iskoristiti resurse lukap modula.

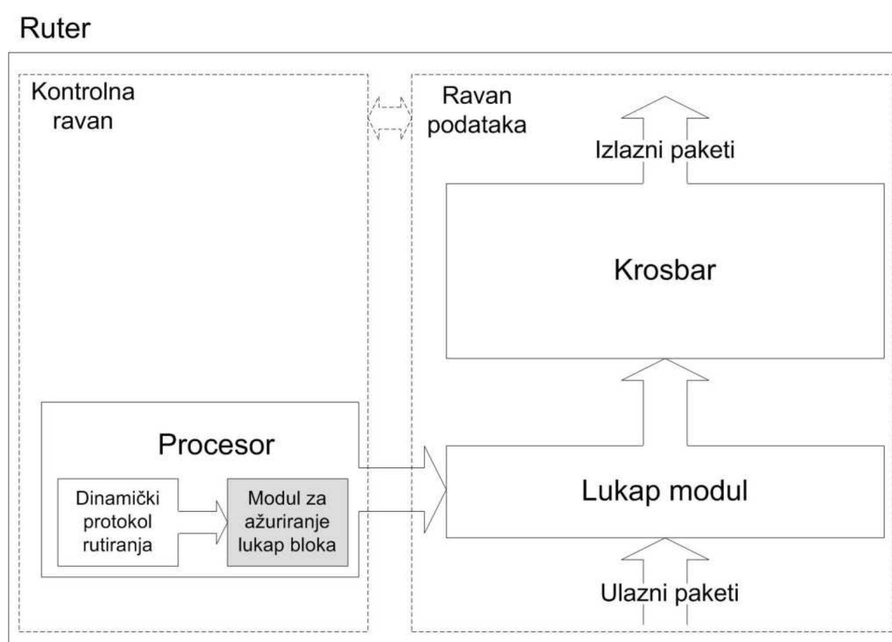
3.1 Lukap procedure i ažuriranje

Osnovne funkcionalne celine rutera prikazane su na Slici 3-1. Prikazan je procesor koji izvršava dinamički protokol rutiranja i modul za ažuriranje, koji na osnovu promena u tabeli rutiranja ažurira strukture podataka u lukap modulu. Proračun putanja i ažuriranje podataka za prosleđivanje paketa u lukap modulu čine deo kontrolne ravni paketske mreže. Ove komponente određuju putanje paketa kroz mrežu.

Na Slici 3-1 su prikazani i lukap modul, koji za svaki ulazni paket na osnovu njegove odredišne adrese određuje izlazni port, i krosbar koji prosleđuje pakete između portova rutera. Ove komponente čine ravan podataka paketeske mreže. One učestvuju u prosleđivanju podataka sadržanih u paketima kroz paketsku mrežu.

Na Slici 3-1 je prikazano da modul za ažuriranje lukap bloka predstavlja vezu između kontrolne ravni i ravni podataka. Sa jedne strane, cilj kontrolne ravni je što bolji

proračun putanja i raspodela saobraćaja u mreži. Sa druge strane, cilj ravni podataka je postizanje što većih brzina prosleđivanja i količine prosleđenog saobraćaja. Cilj algoritma za ažuriranje je da sve promene u načinu prosleđivanja paketa koje generiše kontrolna ravan prenese u ravan podataka i da pri tome što manje utiče na proces prenosa paketa u ravni podataka. Promene u načinu prosleđivanja paketa predstavljene su operacijama dodavanja i brisanja unosa tabele rutiranja i operacijom promene izlaznog porta postojećih unosa. U razmatranim algoritmima ažuriranja operacija promene izlaznog porta postojećih unosa obuhvaćena je algoritmom dodavanja rute.



Slika 3-1 Blok šema rutera

U ovom poglavlju biće razmotreni karakteristični primeri lukap modula i načini ažuriranja njihovih struktura podataka koje nose informacije potrebne za pronalaženje izlaznog porta paketa. Kao različite klase lukap procedura izdvajaju se TCAM, lukap procedure sa bitskim stablima, pretraga po dužinama prefiksa i pretraga po granicama opsega prefiksa.

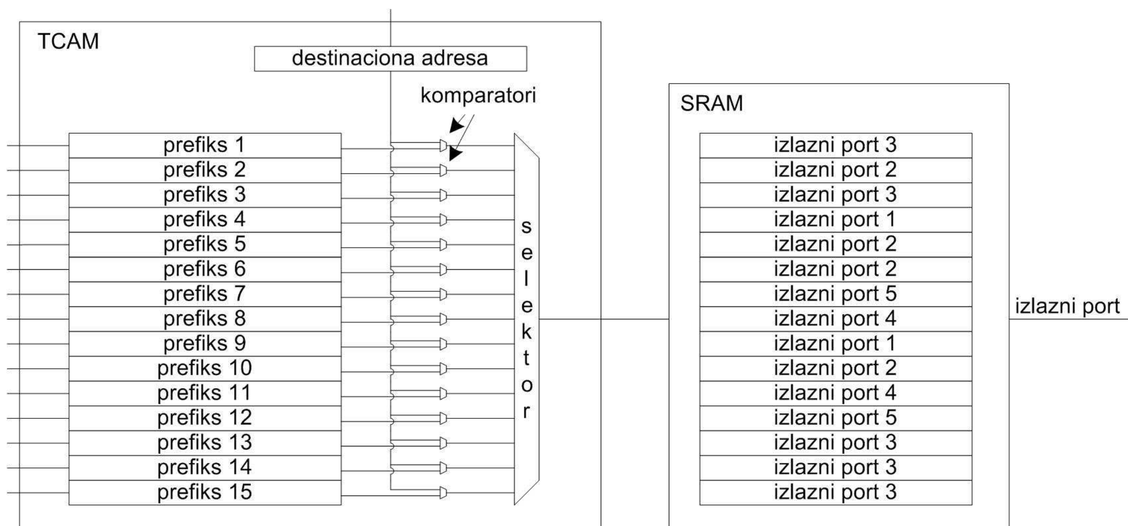
3.1.1 TCAM lukap procedura

3.1.1.1 Struktura TCAM lukap modula

Pronalaženje prefiksa moguće je izvršiti poređenjem svakog prefiksa sa odredišnom adresom paketa. Ovakav postupak bi u slučaju izvršavanja na procesoru

poredio jedan po jedan prefiks sa odredišnom adresom i u proseku bi trajao proporcionalno broju unosa u tabeli rutiranja, koji može biti i nekoliko stotina hiljada. Ukoliko se lukap procedura ne izvršava u procesoru, već u specijalizovanim logičkim modulima, tada je svaki prefiks tabele rutiranja moguće smestiti u jedan blok koji poredi prefiks sa odredišnom adresom paketa. Ukoliko se svaki prefiks smesti u odvojen blok, i ako se poređenje vrši paralelno u svim blokovima, tada bi vreme traženja bilo jednako vremenu poređenja u jednom bloku, uključujući i poređenje dužina prefiksa koji odgovaraju odredišnoj adresi i izbor najdužeg prefiksa.

Primer implementacije u kojoj logički blokovi vrše istovremeno poređenje svih prefiksa tabele rutiranja sa dolaznom adresom su TCAM (*Ternary Content-Addressable Memory*) memorije [38], čiji se naziv može prevesti kao ternarna memorija adresirana sadržajem. Memorija adresirana sadržajem omogućava pronalaženje memorijske lokacije na osnovu sadržaja lokacije. Memorija adresirana sadržajem je ternarna ukoliko pored vrednosti bita 0 i 1 ima i treću vrednost, koja označava da vrednost bita nije bitna pri pronalaženju memorijske lokacije na osnovu adrese. Struktura TCAM memorije prikazana je na Slici 3-2.



Slika 3-2 TCAM lukap modul

Ako se uvede ograničenje da zapisi u TCAM memoriji budu sortirani po dužini prefiksa, selektor se može implementirati kao prioritetni enkoder koji je jednostavan i brz. Zahtev da zapisi budu sortirani dovodi do mogućeg velikog broja pomeranja u toku ažuriranja, što može rezultovati predugim pauzama u prosleđivanju u ruterima velikih

brzina. Iz tog razloga, predložene su kompleksnije implementacije selektora koje ne zahtevaju da zapisi u TCAM memoriji budu sortirani [39].

Formiranje velikog broja logičkih blokova od kojih svaki sadrži po jedan prefiks tabele rutiranja i vrši poređenje sa dolaznom adresom zahteva veliki broj veza na čipu i elemenata za poređenje i time značajno poskupljuje implementaciju na čipu. Iz tog razloga, razvijene su druge procedure za traženje najdužeg prefiksa u tabeli rutiranja.

3.1.1.2 Ažuriranje TCAM lukap modula

Pri ažuriranju TCAM lukap modula prefiks tabele rutiranja se direktno upisuje u TCAM memoriju, sa tim da se biti koji ne pripadaju adresi mreže postave u stanje koje označava da se vrednosti tih bita neće koristiti pri poređenju sa odredišnom IP adresom. Oznaka izlaznog porta na koga treba slati pakete ukoliko se poklapaju sa prefiksom upisuje se u SRAM memoriju.

U slučaju kada dizajn selektora zahteva da zapisi u TCAM memoriji budu sortirani, algoritam ažuriranja vrši pomeranje potrebnog broja zapisa da bi se oslobodilo mesto za zapis prefiksa nove rute.

3.1.2 Lukap procedure sa bitskim stablima

Većina lukap procedura predloženih u literaturi baziraju se na bitskim stablima. U sledećim odeljcima biće opisane strukture jednobitskog i multibitskog stabla. Zatim će biti opisane tri modifikacije bitskih stabala koje se koriste u okviru lukap procedura: kompresija putanja u stablu, formiranje skupa nepreklapajućih prefiksa i potiskivanje ruta u listove stabla.

Posle opšteg opisa bitskih stabala biće predstavljeni značajni lukap algoritmi bazirani na ovim stablima. Prvo će biti opisan *Lulea* [40] i *Tree Bitmap* [41] lukap algoritmi koji postižu mali memorijski utrošak i omogućavaju efikasno korišćenje keš memorija pri procesorski izvršavanom lukapu. Sledeća će biti opisana lukap procedura zasnovana na prioritetnom stablu, transformaciji jednobitskog stabla koje eliminiše čvorove koji ne sadrže prefikse tabele rutiranja. Potom će biti opisani POLP [42] i BPFL [43]-[44], dva lukap algoritma visokih performansi, koji nisu namenjeni procesorskom izvršavanju već implementaciji u okviru specijalizovanih logičkih modula.

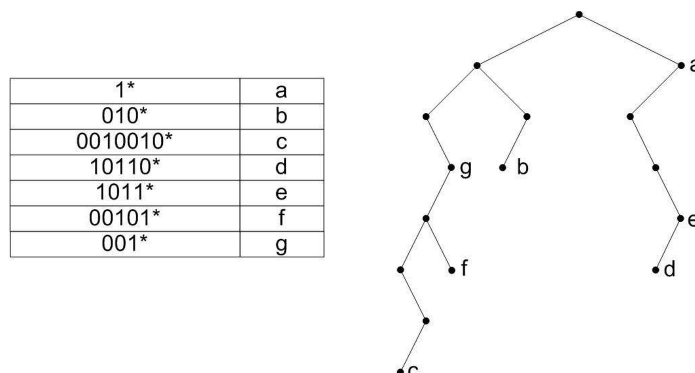
3.1.2.1 Jednobitsko stablo

Moguće je formirati stablo u kome grane odgovaraju bitima prefiksa. U slučaju kada svaka grana stabla odgovara jednom bitu prefiksa formira se jednobitsko stablo. Pri formiranju stabla, za svaki prefiks kreće se od korena stabla. Ako je prvi bit prefiksa jednak nuli, tada se formira levi čvor-dete, ukoliko već nije formiran. Ako je prvi bit prefiksa jednak jedinici, formira se desni čvor-dete, ukoliko već nije formiran. Posle prvog koraka, u stablu se stiglo do jednog od čvorova-dece korena stabla i tada se posmatra drugi bit prefiksa. Ako je drugi bit jednak nuli formira se levi čvor-dete čvora do koga se stiglo ukoliko nije već formiran, inače se formira desni čvor-dete ukoliko nije formiran, i tekući čvor postaje jedan od ta dva čvora. Potom se posmatra treći bit prefiksa i na osnovu njega se iz tekućeg čvora prelazi u jedan od njegovih čvorova-dece koji se prethodno formira ako nije postojao. Ova procedura se nastavlja dok se ne dođe do poslednjeg bita prefiksa, i u čvoru koji odgovara poslednjem bitu zapisuje se vrednost izlaznog porta koja odgovora prefiksu.

Jednobitsko stablo se gradi tako što se u stablo dodaju svi prefiksi tabele rutiranja, i svaki prefiks napravi čvorove koji nedostaju u njegovom prolazu i upiše vrednost izlaznog porta u poslednji čvor u prolazu kroz stablo. Ovako dobijeno jednobitsko stablo koristi se za pronalaženje najdužeg prefiksa koji odgovara odredišnoj IP adresi tako što se prolazi kroz stablo na osnovu bita odredišne adrese sve dok postoje čvorovi stabla na putanji. U toku prolaza pamti se poslednja vrednost izlaznog porta na koju se naišlo. Kada se prolaz kroz stablo završi dolaskom do poslednjeg napravljenog čvora na putanji ili prolaskom kroz sve bite odredišne adrese, zapamćena vrednost je ujedno i najduži prefiks koji odgovara odredišnoj adresi. Ukoliko se u toku prolaza ne naiđe ni na jednu vrednost izlaznog porta, tada ni jedan prefiks ne odgovara odredišnoj adresi i paket se rutira na izlazni port koji odgovara podrazumevanom (eng. *default*) ruteru. Struktura jednobitskog stabla prikazana je na Slici 3-3.

Vreme pronalaženja izlaznog porta u jednobitskom stablu ne zavisi od veličine tabele rutiranja, već od dubine jednobitskog stabla koja je maksimalno jednaka broju bita u IP adresi. Ipak, jednobitsko stablo pored čvorova koji sadrže informacije o izlaznom portu može sadržati i značajan broj praznih čvorova, koji su formirani zato što su na putu prolaza prefiksa kroz stablo, što zahteva dodatnu memoriju i vreme za

očitavanje vrednosti tih čvorova u toku prolaska kroz stablo. Iz tog razloga, predložene su brojne modifikacije jednobitskog stabla u cilju poboljšanja njegovih perofmansi.

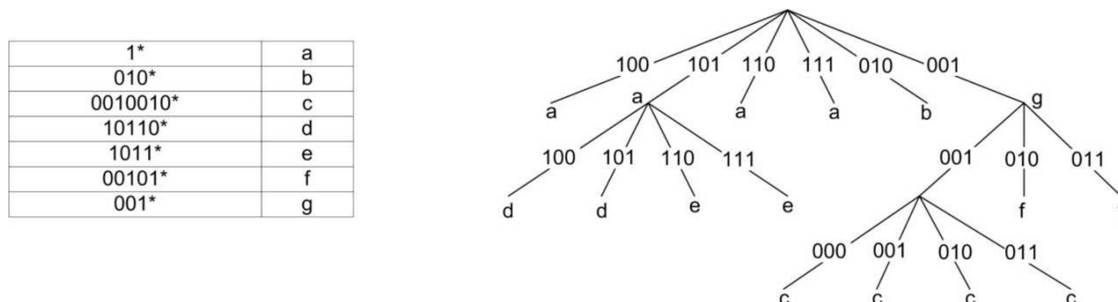


1*	a
010*	b
0010010*	c
10110*	d
1011*	e
00101*	f
001*	g

Slika 3-3 Jednobitsko stablo

3.1.2.2 Multibitsko stablo

Jedna mogućnost za ubrzanje pronalaska rute je modifikacija jednobitskog stabla u multibitsko stablo [37]. Za razliku od jednobitskog stabla u kome jedna grana stabla odgovara jednom bitu adrese, u multibitskom stablu jedna grana stabla odgovara većem broju bita adrese. Broj bita adrese koji odgovara grani stabla naziva se korak (eng. *stride*). U opštem slučaju stablo se može sastojati iz niza različitih koraka, prilagođenih gustini prefiksa u okviru koraka. Struktura multibitskog stabla prikazana je na Slici 3-4.



1*	a
010*	b
0010010*	c
10110*	d
1011*	e
00101*	f
001*	g

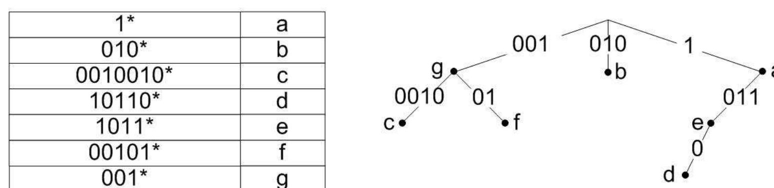
Slika 3-4 Multibitsko stablo sa korakom 3

Kao u jednobitskom stablu, i u multibitskom stablu vrednosti izlaznih portova se čuvaju u čvorovima stabla. Pošto u multibitskom stablu ne postoje čvorovi za sve dužine prefiksa, potrebno je prefikse tabele modifikovati tako da njihove dužine budu jednake jednoj od vrednosti dužine prefiksa koja postoji u multibitskom stablu. Ovo se postiže tako što se svi prefiksi čija dužina ne odgovara nekoj od postojećih dužina u stablu produžavaju do prve postojeće dužine koja je veća od dužine prefiksa. Prilikom produžavanja prefiksa, jedan kraći prefiks će formirati 2^i prefiksa, gde i predstavlja broj

bita za koji se vrši produžavanje prefiksa. Prilikom produžavanja moguće je da se veći broj prefiksa produži do istog čvora stabla, i tada se u taj čvor upisuje vrednost izlaznog porta koja odgovara najdužem od tih prefiksa.

3.1.2.3 Kompresija putanja u stablu

Modifikacija jednobitskog stabla koja je slična multibitskom stablu je kompresija putanja. Primer stabla sa komprimovanim putanjama prikazan je na Slici 3-5. Prilikom kompresije putanja, nizovi čvorova koji u sebi ne sadrže rute i koji imaju samo po jedan čvor-dete se komprimuju u jednu granu stabla, koja u toku pretrage odgovara nizu bita koji je odgovarao granama pre kompresije. Po kompresiji, pretraga se može vršiti brže pošto se izbegava očitavanje komprimovanih čvorova. Kompresija ne daje dobre rezultate u slučaju razgranatog stabla koje je tipično u slučaju većih tabela rutiranja, pri čemu zahteva složeniju implementaciju pošto koraci kroz stablo zahtevaju poređenje sa nizovima bita različitih dužina.



1*	a
010*	b
0010010*	c
10110*	d
1011*	e
00101*	f
001*	g

Slika 3-5 Komprimovano stablo

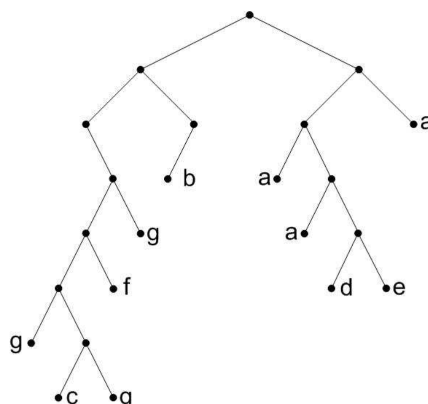
3.1.2.4 Potiskivanje ruta u listove stabla

Još jedna korišćena modifikacija opisanih stabala je potiskivanje ruta u listove stabla (eng. *leaf pushing*). Procedura potiskivanja ruta u listove stabla omogućava da čvorovi koji nisu listovi sadrže samo pokazivače na njihove čvorove-decu, a da čvorovi-listovi sadrže vrednost izlaznog porta. Ovo omogućava da svi čvorovi imaju kompaktan zapis iste veličine, što je važno za optimizaciju.

Na Slici 3-6 prikazano je stablo koje je rezultat potiskivanja ruta u listove u jednobitskom stablu prikazanom na Slici 3-3. Pri potiskivanju ruta u listove, kreće se od čvora koji sadrži rutu i nije list, i prolazi se kroz podstablo čiji je taj čvor koren do listova tog stabla. U svakom listu se proverava da li tekuća ruta u listu ima kraći prefiks od rute koja se potiskuje, i ako ima, vrednost izlaznog porta rute koja se potiskuje se upisuje u list. U slučaju da se u prolazu kroz podstablo naiđe na čvor koji ima samo

jedno dete-čvor, potrebno je napraviti drugo dete-čvor i u njega upisati vrednost izlaznog porta rute koja se potiskuje.

1*	a
010*	b
0010010*	c
10110*	d
1011*	e
00101*	f
001*	g



Slika 3-6 Potiskivanje ruta ka listovima jednobitskog stabla

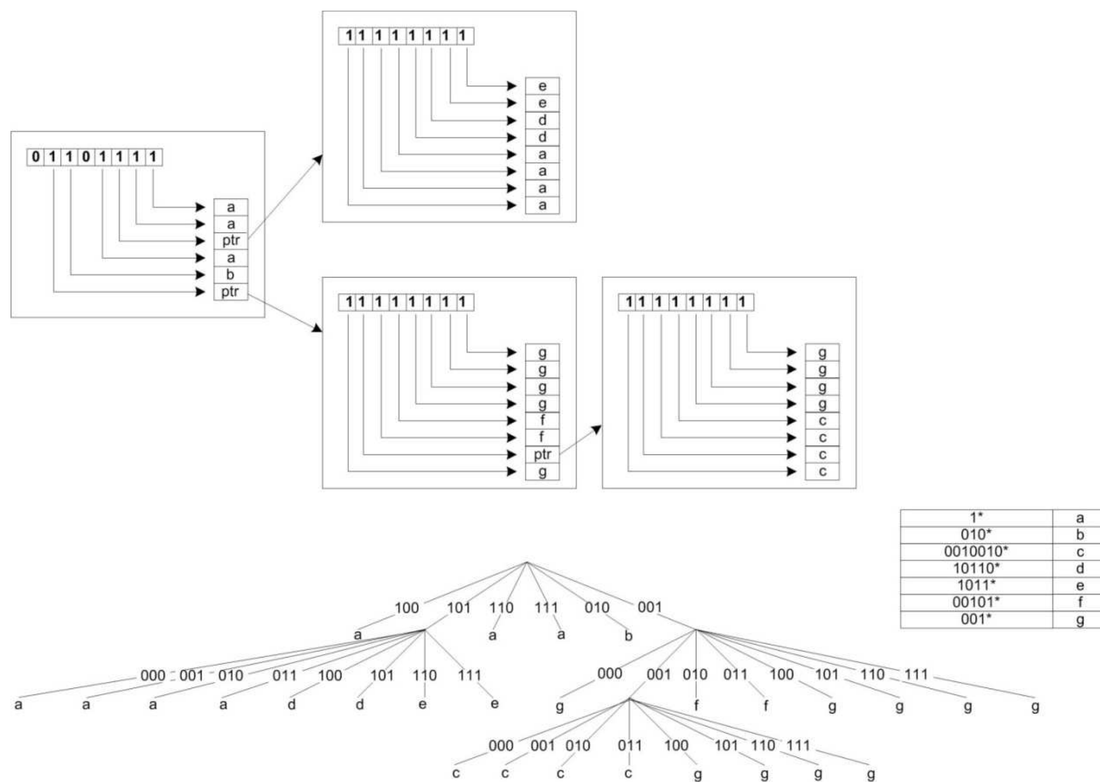
3.1.2.5 Formiranje skupa nepreklapajućih prefiksa

Važna modifikacija skupa prefiksa koji čine tabelu rutiranja je formiranje skupa nepreklapajućih prefiksa (eng. *disjoint prefixes*). U ovom procesu svi prefiksi koji su obuhvaćeni drugim prefiksima tabele rutiranja se produžuju tako da ni jedan prefiks u tabeli ne bude obuhvaćen drugim prefiksom. Prednost nepreklapajućeg skupa prefiksa je što odredišna adresa paketa čiji se izlazni port traži ne može imati poklapanje sa više od jednim prefiksom iz tog skupa, što omogućava optimizaciju procesa lukapa. Stablo sa nepreklapajućim skupom prefiksa može se dobiti izvršavanjem potiskivanja ruta u unutrašnjim čvorovima stabla ka listovima.

3.1.2.6 *Lulea* lukap procedura

Pored navedenih transformacija jednobitskog stabla, performanse pretrage jednobitskog stabla moguće je poboljšati i korišćenjem optimizovanih metoda čuvanja i očitavanja stabla. Primer lukap procedure koja postiže izuzetno nizak stepen utroška memorije je *Lulea* procedura [40], koja postiže utrošak od prosečno 4-5 bajtova po ruti za velike tabele rutiranja. *Lulea* ovo postiže korišćenjem stabla koje predstavlja kombinaciju opisanih tehnika. *Lulea* koristi multibitsko stablo sa tri nivoa čiji koraci iznose 16 bita za prvi nivo i po 8 bita za drugi i treći nivo. Pri dodavanju u stablo svakog prefiksa vrši se njegovo potiskivanje u listove, pa prefiksi u stablu formiraju skup nepreklapajućih prefiksa. Korišćenjem navedenih tehnika i kompaktnog bitskog zapisa opisanog u radu [40] postiže se veoma mali utrošak memorije iako se za

predstavljanje tabele rutiranja koristi struktura stabla. Veoma kompaktan bitski zapis je i mana *Lulea* algoritma, jer promena bilo koje rute praktično zahteva ponovni proračun kompletne bitske predstave korišćenog multibitskog stabla, što zahteva veliki broj operacija prilikom ažuriranja *Lulea* lukap modula.



Slika 3-7 *Lulea* lukap procedura

3.1.2.6.1 Ažuriranje *Lulea* lukap modula

Primer *Lulea* lukap procedure predstavljen je na Slici 3-7. Zbog ograničene veličine slike, nije korišćeno *Lulea* multibitsko stablo čiji su koraci 16, 8 i 8, već su korišćeni koraci veličine 3. Na Slici 3-7, prikazana je tabela rutiranja, i multibitsko stablo koraka 3 koja odgovara toj tabeli rutiranja. U gornjem delu slike prikazane su *Lulea* predstave delova multibitskog stabla. Prvi korak multibitskog stabla je ceo predstavljen blokom prikazanim u levom delu slike, a svako podstablo u ostalim koracima multibitskog stabla je predstavljeno posebnim blokom, i na Slici 3-7 su prikazana tri takva bloka.

Svaki od blokova prikazanih u gornjem delu Slike 3-7 sastoji se od niza bita i niza pokazivača. Niz bita predstavlja čvorove podstabla, i jedinice označavaju da čvor podstabla koji odgovara poziciji jedinice sadrži pokazivač na sledeće podstablo ili na

memoriju u kojoj se čuvaju informacije o izlaznim portovima. Na primer, prvi korak multibitskog stabla predstavljen je blokom *Lulea* strukture na levom kraju slike. Niz bita u tom bloku ima nule na prvoj i četvrtoj poziciji, i te pozicije odgovaraju čvorovima 000 i 011, koje ne postoje u stablu. U nizu pokazivača, pokazivači na podstabla predstavljeni su oznakom ptr, i postavljeni su na drugoj i šestoj poziciji koje odgovaraju čvorovima 001 i 101. Na slici multibitskog stabla se vidi da ovi čvorovi imaju čvorove-decu u multibitskom stablu. Preostali biti u nizu bita *Lulea* bloka koji odgovara prvom koraku stabla odgovaraju čvorovima koji u sebi imaju pokazivače na identifikator odredišnog porta paketa.

U svakom od *Lulea* blokova predstavljenih na Slici 3-7, niz pokazivača ima po jedan element za svaku jedinicu u nizu bita tog bloka. Sledi da svaka promena vrednosti bita u nizu bita odgovara dodavanju ili brisanju pokazivača iz niza pokazivača. *Lulea* lukap procedura definiše kompaktan zapis ovog niza pokazivača, u kome brisanje ili dodavanje elementa zahteva promenu velikog dela memorijskog bloka koji čini niz pokazivača. Za razliku od Slike 3-7, u kojoj prvi korak multibitskog stabla ima dužinu 3, u *Lulea* algoritmu prvi korak ima dužinu 16, pa broj bita u nizu bita iznosi 2^{16}=65536, koliko iznosi i maksimalni broj pokazivača u nizu pokazivača. U *Lulea* algoritmu, svaki pokazivač zauzima 16 bita, od kojih prva dva bita označavaju tip pokazivača, dok preostalih 14 bita predstavljaju adresu *Lulea* bloka u kome se nalaze čvorovi-deca tekućeg čvora, ili adresu u memoriji identifikatora izlaznih portova. Tip pokazivača označava da li pokazivač sadrži adresu *Lulea* bloka u kome se nalaze čvorovi-deca tekućeg čvora, ili memorijsku adresu na kojoj se nalazi identifikator izlaznog porta. Ukupno, pokazivači *Lulea* bloka koji odgovara prvom koraku multibitskog stabla mogu zauzimati do jednog megabita memorije, i dodavanje ili brisanje pokazivača u nizu pokazivača ovog bloka može prouzrokovati promenu velikog dela memorijskih lokacija ovog dela memorije. Ovaj veliki broj promena nije povoljan za ažuriranje pošto zahteva veliki broj upisa u memoriju, i produženo vreme izvršavanja.

Drugi i treći korak *Lulea* multibitskog stabla su dužine 8, i sadržani su u *Lulea* blokovima. Dužina niza bita u ovim blokovima je 256, koliko iznosi i maksimalni broj pokazivača u nizu pokazivača tih blokova. Ažuriranje blokova dužine osam je značajno efikasnije od ažuriranja bloka koji odgovara prvom koraku multibitskog stabla.

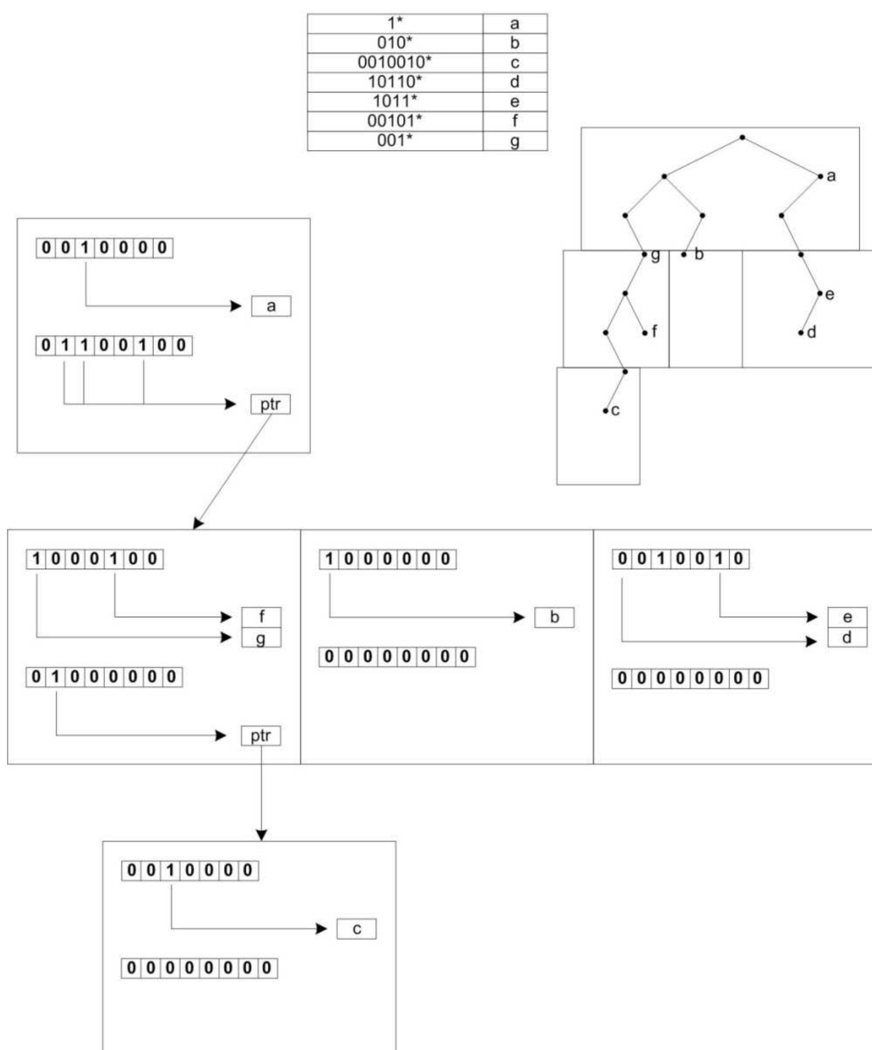
3.1.2.7 *Tree Bitmap* lukap procedura

Lukap procedura koja postiže memorijske zahteve slične *Lulea* algoritmu je *Tree Bitmap* [41]. *Tree bitmap* se zasniva na multibitskom stablu. Za razliku od *Lulea* algoritma koji koristi potiskivanje ruta u listove i nivoe čiji koraci iznose $m_1 = 16$, $m_2 = 8$ i $m_3 = 8$ bita, autori *Tree Bitmap* algoritma su izabrali da ne koriste potiskivanje u listove i da svi koraci imaju istu veličinu od $m = 8$ bita. Pošto svi koraci imaju istu veličinu od $m = 8$ bita, multibitsko stablo se može predstaviti čvorovima koji interno sadrže jednobitska podstabla dubine $r = 8$. Jednobitsko stablo dubine r, na nivou i ima maksimalno 2^{i-1} čvorova, dok celo stablo čiji su svi nivoi popunjeni sadrži $2^r - 1$ čvorova. Na osnovu toga sledi da jednobitsko stablo dubine $r = 8$ ima 511 čvorova. *Tree Bitmap* algoritam predviđa da svaki čvor multibitskog stabla sadrži niz bita u kome svaki bit odgovara čvoru podstabla i nosi informaciju da li taj čvor podstabla sadrži rutu. Za čvorove koji sadrže rute vrednost izlaznog porta je smeštena u poseban niz, u kome pozicija *i* odgovara *i*-toj jedinici u nizu bita.

Pošto podstablo koje čini čvor multibitskog stabla sadrži deo jednobitskog stabla, čvorovi u poslednjem nivou podstabla mogu sadržati čvorove-decu koja su pridružena čvorovima multibitskog stabla u sledećem nivou. Iz tog razloga je u svakom čvoru multibitskog stabla formiran niz bita u kome svaki bit definiše da li čvor-dete čvora u poslednjem nivou podstabla postoji, i takvih bita ima 512. U *Tree Bitmap* algoritmu se svi čvorovi-deca u multibitskom stablu koji imaju zajedničkog roditelja smeštaju jedan pored drugog. Na taj način se smanjuje broj potrebnih pokazivača, jer je dovoljno da čvor roditelj multibitskog stabla sadrži pokazivač na prvi čvor-dete. Ostali čvorovi-deca su poređani iza prvog u redosledu koji odgovara redosledu jedinica u nizu bita koji definiše da li čvorovi-deca postoje. Ilustracija *Tree Bitmap* stabla prikazana je na Slici 3-8.

Lulea i *Tree Bitmap* lukap procedure omogućavaju malo zauzeće memorije i grupisanje memorijskih blokova koji se u toku lukapa često koristi, tako da omogućavaju efikasan rad sa keš memorijama koje imaju veliku brzinu pristupa ali ograničen kapacitet. Brzi pristup keš memoriji omogućava i brzo pronalaženje izlaznog porta za paket koji prolazi kroz ruter. Pored ubrzanja pronalaženja izlaznog porta za paket, veći protok kroz ruter je moguće postići i izvršavanjem većeg broja pretraga

izlaznog porta istovremeno, što se može postići raspoređivanjem informacija u lukap modulu u više memorija kojima se može pristupati istovremeno. Primer lukap procedure koja postiže visok stepen paralelizacije pretraga izlaznog porta je POLP lukap procedura.



Slika 3-8 *Tree Bitmap* lukap procedura

3.1.2.7.1 Ažuriranje *Tree Bitmap* lukap modula

Slika 3-8 predstavlja sadržaj *Tree Bitmap* lukap modula za dati primer tabele rutiranja. Zbog ograničene veličine slike, korak multibitskog stabla je 3 umesto 8 koliko iznosi u *Tree Bitmap* lukap modulu za IPv4 protokol rutiranja. Tabela rutiranja prikazana je u gornjem delu Slike 3-8. Ispod tabele rutiranja sa desne strane je prikazano jednobitsko stablo koje odgovara tabeli rutiranja, i čvorovi tog stabla su grupisani tako da formiraju čvorove multibitskog stabla. Multibitsko stablo ima pet

čvorova, i svakom čvoru odgovara jedan *Tree Bitmap* blok koji sadrži dva niza bita i dva niza pokazivača.

U *Tree Bitmap* strukturi je predstavljen kompletan sadržaj čvora multibitskog stabla, koji u sebi sadrži podstablo jednobitskog stabla. Pošto se u *Tree Bitmap* stablu ne vrši potiskivanje ruta u listove, u okviru podstabla jednobitskog stabla sadržani su čvorovi koji mogu sadržati pokazivače na identifikatore izlaznog porta. U poslednjem nivou jednobitskog podstabla koje pripada čvoru mogu se nalaziti pokazivači na sledeće čvorove multibitskog stabla.

U okviru *Tree bitmap* bloka, jedan niz bita se koristi za označavanje pozicija čvorova jednobitskog podstabla koji predstavljaju prefikse tabele rutiranja, tj. sadrže rute. Jedinica u nizu bita označava da čvor sadrži rutu. Na primer, na Slici 3-8 gornji *Tree Bitmap* blok odgovara korenu multibitskog stabla i u nizu bita koji označava čvorove sa rutama on ima samo jednu jedinicu, na trećoj poziciji. Pošto se čvorovi podstabala numerišu odozgo na dole i sleva na desno, trećoj poziciji odgovara desni čvor dete korena podstabla, koji u sebi sadrži rutu označenu slovom *a*. U nizu pokazivača koji sadrži pokazivače na identifikatore izlaznih portova upisana je vrednost koja odgovara ruti *a*. Ovaj niz pokazivača sadrži po jedan pokazivač za svaku jedinicu u nizu bita koji označava čvorove sa rutom.

Prilikom dodavanja ili brisanja rute u okviru jednobitskog podstabla potrebno je promeniti vrednost bita koji odgovara čvoru u kome se ruta dodaje ili briše. Takođe, potrebno je i dodati ili obrisati vrednost pokazivača na izlazni port, što može rezultovati pomeranjem većeg broja pokazivača u nizu pokazivača da bi se napravilo mesta za novi pokazivač, ili da bi se popunilo mesto nastalo brisanjem pokazivača.

Tree Bitmap blok sadrži i niz bita koji definiše veze između čvorova multibitskog stabla. Ovaj niz bita sadrži po jednu jedinicu za svaki čvor-dete čvora predstavljenog *Tree Bitmap* blokom. Uz ovaj niz bita, *Tree Bitmap* blok sadrži i pokazivač na čvor-dete koji odgovara prvoj jedinici u nizu bita. Pokazivači na ostale čvorove-decu nisu potrebni jer se svi *Tree Bitmap* blokovi koji predstavljaju čvorove-decu jednog čvora-roditelja smeštaju u memoriji jedan pored drugog, u redosledu koji odgovara jedinicama u nizu bita čvora-roditelja. Na ovaj način se štedi memorija. Niz bita koji definiše veze između čvorova multibitskog stabla u primeru sa slike sadrži

osam bita, dok u *Tree Bitmap* stablu sa korakom dužine 8 on iznosi 256, koliko iznosi i maksimalni broj čvorova-dece jednog čvora-roditelja u multibitskom stablu sa korakom 8.

Dodavanje i brisanje čvora-deteta u čvoru roditelju je efikasno jer se menja samo jedan bit u nizu bita, i eventualno vrednost pokazivača, ako se dodaje ili briše čvor-dete koji odgovara prvoj jedinici u nizu bita. Dodavanje i brisanje čvora međutim može rezultovati velikim brojem operacija u memoriji koja sadrži niz *Tree Bitmap* struktura koji predstavljaju čvorove-decu jednog čvora roditelja. Ove operacije vrše se prilikom pomeranja *Tree Bitmap* struktura da bi se napravilo mesta za novi čvor-dete ili da bi se popunilo upražnjeno mesto pri brisanju čvora-deteta.

3.1.2.8 Pretraga sa prioritetnim stablom

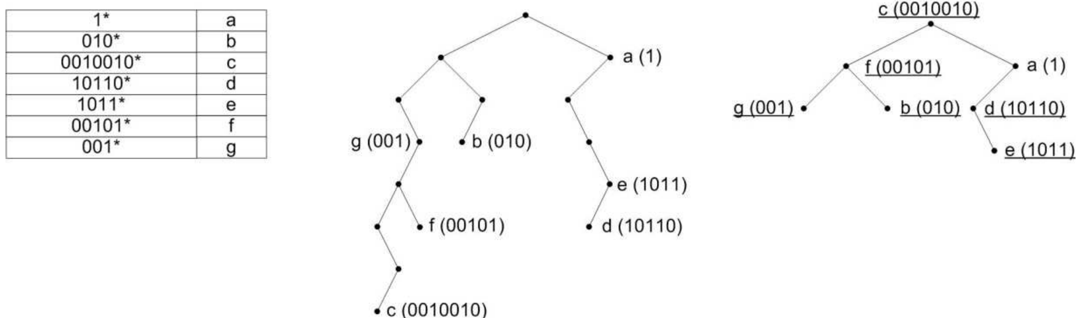
Prioritetno stablo [44] se formira od jednobitskog stabla premeštanjem prefiksa u čvorove u gornjim nivoima koji ne sadrže prefikse. Na Slici 3-9 je prikazana tabela rutiranja sa pripadajućim jednobitskim stablom i prioritetnim stablom.

U prioritetnom stablu postoje dve vrste čvorova: prioritetni čvorovi i neprioritetni čvorovi. Na Slici 3-9, prioritetni čvorovi su podvučeni. Prioritetni čvor se nalazi na dubini koja je manja od dužine njegovog prefiksa, i njegov prefiks ima osobinu da ne postoji prefiks koji ga sadrži ni u jednom čvoru ispod tog prioritetnog čvora. Iz tog razloga pretraga se može završiti kada dođe do poklapanja sa prefiksom prioritetnog čvora. Čvorovi koji nisu prioritetni su neprioritetni.

Prilikom pretrage prioritetnog stabla kreće se od korena stabla. Kretanje kroz stablo se vrši kao u jednobitskom stablu. Polazi se od prvog bita IP adrese za koju se traži izlazni port i na osnovu vrednosti bita se bira sledeći korak. Ukoliko je vrednost bita jednaka nuli bira se levi čvor-dete, a ako je jednaka jedinici bira se desni čvor-dete. U svakom koraku prelazi se na sledeći bit adrese.

U svakom čvoru stabla vrši se poređenje IP adrese sa prefiksom sadržanim u čvoru prioritetnog stabla. Ukoliko IP adresa sadrži prefiks iz prioritetnog čvora stabla, pretraga se završava i unos tabele rutiranja koji određuje izlazni port je unos pridružen prioritetnom čvoru. Ukoliko postoji poklapanje sa neprioritetnim čvorom, pamti se poslednji neprioritetni čvor sa kojim je bilo poklapanja. Ovaj zapamćeni čvor će biti

rezultat pretrage ukoliko se do kraja prolaska kroz stablo ne naiđe na novo poklapanje. Novo poklapanje može biti prioritetni čvor, u kom slučaju se pretraga završava ili neprioritetni čvor, koji će postati novi zapamćeni kandidat za rezultat pretrage.



Slika 3-9 Lukap sa prioritetnim stablom

Prioritetno stablo eliminiše čvorove jednobitskog stabla koji ne sadrže prefike i koji su napravljeni samo da bi se moglo doći do čvorova koji sadrže prefikse. Na ovaj način štedi se memorija potrebna za čuvanje čvorova koji ne sadrže prefikse. Drugo poboljšanje prioritetnog stabla u odnosu na jednobitsko stablo je manja dubina stabla. Manja dubina stabla omogućava bržu pretragu jer je potrebno izvršiti manje operacija u toku prolaska kroz stablo. Pretraga je dodano ubrzana postojanjem prioritetnih čvorova, pošto poklapanje sa prefiksom u prioritetom čvoru omogućava završetak pretrage pre završetka prolaska kroz stablo.

3.1.2.8.1 Ažuriranje prioritetnog stabla

U procesu formiranja prioritetnog stabla od jednobitskog stabla, prolazi se kroz sve čvorove jednobitskog stabla odozgo na dole i sa leva na desno. Kada se naiđe na čvor koji ne sadrži prefiks, tada se iz njegovog podstabla uzima najduži prefiks i smešta u tekući čvor. Čvor koji je pripadao premeštenom prefiksu i svi čvorovi koji su vodili do njega, a nisu sadržali prefikse, brišu se iz stabla. Čvor u koga je prefiks premešten označava se kao prioriteni čvor, pošto u podstablu ispod njega više ne postoji prefiks duži od njegovog prefiksa, koji sadrži njegov prefiks. U primeru na Slici 3-8, u koren stabla je smešten prefiks *c* koji je najduži prefiks ispod korena stabla. Čvor koji je sadržao prefiksu *c* i dva čvora koja su do njega vodila potom su obrisana. Zatim je posećen levi čvor-dete korena stabla i u njega je smešten prefiks *f*, koji je najduži prefiks u podstablu ispod levog čvora-deteta. Levi čvor-dete je poglašen za prioriteni čvor i čvor *f* i jedan čvor iznad njega su obrisani. Potom je posećen desni čvor-dete

korena stabla. Desni-čvor dete već sadrži prefiks tako da se njegov sadržaj ne menja. Ovaj proces se nastavlja posećivanjem svih čvorova u stablu odozgo na dole i sleva na desno.

U toku rada rutera, dinamički prokol rutiranja menja tabelu rutiranja i nije pogodno prilikom svakog dodavanja ili brisanja unosa u tabeli rutiranja formirati celo prioritetno stablo iz početka. Novi prefiks se može dodati u prioritetno stablo tako što se traži mesto za novi prefiks kretanjem kroz prioritetno stablo u skladu sa bitima novog prefiksa, na isti način kao kad se vrši pretraga stabla. U svakom čvoru se vrši poređenje dužine novog prefiksa sa prefiksom sačuvanim u čvoru, i završava se ako se naiđe na prioritetni čvor sa kraćim prefiksom ili ako se dođe do dubine prioritetnog stabla koja je jednaka dužini novog prefiksa. U prvom slučaju, nova ruta se smešta u tekući čvor i čvor ostaje prioritetni. Prolazak kroz stablo se nastavlja sa rutom koja je bila sačuvana u tom čvoru, i na isti način se traži mesto za njeno dodavanje u prioritetno stablo. U slučaju kada se pri dodavanju prefiksa dođe do dubine koja je jednaka dužini prefiksa koji se dodaje, prefiks se smešta u tekući čvor kao neprioriteni prefiks, a dodavanje se se nastavlja sa rutom koja je bila u tom čvoru.

U slučaju brisanja unosa iz tabele rutiranja, na mesto obrisanog čvora premešta se jedan od njegove dece-čvorova. Zatim se na upražnjeno mesto premeštenog čvora-deteta premešta jedan od njegovih čvorova-dece i tako redom dok se ne stigne do kraja stabla. Prilikom premeštanja čuva se osobina prioritetnosti čvora, zato što neprioriteni čvor ne može postati prioritetni zbog mogućnosti da ispod njega ima dužih prefiksa koji sadrže njegov prefiks.

Procedure iterativnog dodavanja i brisanja čvorova iz prioritetnog stabla su efikasne jer se završavaju u toku jednog prolaska kroz prioritetno stablo. Prioritetno stablo je kraće od jednobitskog stabla pa je ovaj broj operacija dodatno umanjen.

Pošto se pri brisanju ruta pri podizanju čvora-deteta na mesto čvora-roditelja zadržava prioritetnost čvora, posle većeg broja brisanja je moguće povećanje broja neprioritetnih čvorova u stablu u odnosu na broj neprioritetnih čvorova koji bi postojali u stablu formiranom za istu tabelu rutiranja počev od praznog stabla bez brisanja.

Primer povećanja broja neprioritetnih čvorova koji se sastoji od dodavanja rute, pa zatim njenog brisanja, biće opisan u ovom paragrafu. U primeru posmatramo prefiks

u čvoru koji je bio prioritetni pre dodavanja nove rute. Prilikom dodavanja nove rute posmatrani prefiks može biti premešten u neprioritetni čvor ako na njegovo mesto upiše drugi prefiks, a on se usled smeštanja nove rute na njegovo mesto spusti do dubine koja odgovara njegovoj dužini. Posle brisanja dodate rute, prefiks koji je zbog dodavanja nove rute bio pomeren u čvor na nižem nivou može biti ponovo premešten u čvor na višem nivou. Pri premeštanju na viši nivo, u skladu sa algoritmom brisanja ruta u prioritetnom stablu opisanim u ovom pododeljku, čvor u koji se prefils smesti biće neprioritetan. Na početku ovog primera, prefiks se nalazio u prioritetnom čvoru na dubini manjoj njegove dužine, dok se na kraju primera prefiks nalazio u čvoru koji je takođe na dubini manjoj od njegove dužine, ali je bio neprioritetni. Dodavanje i brisanje ruta sa kraćim prefiksima je premestilo posmatrani prefiks iz prioritetnog čvora u neprioritetni.

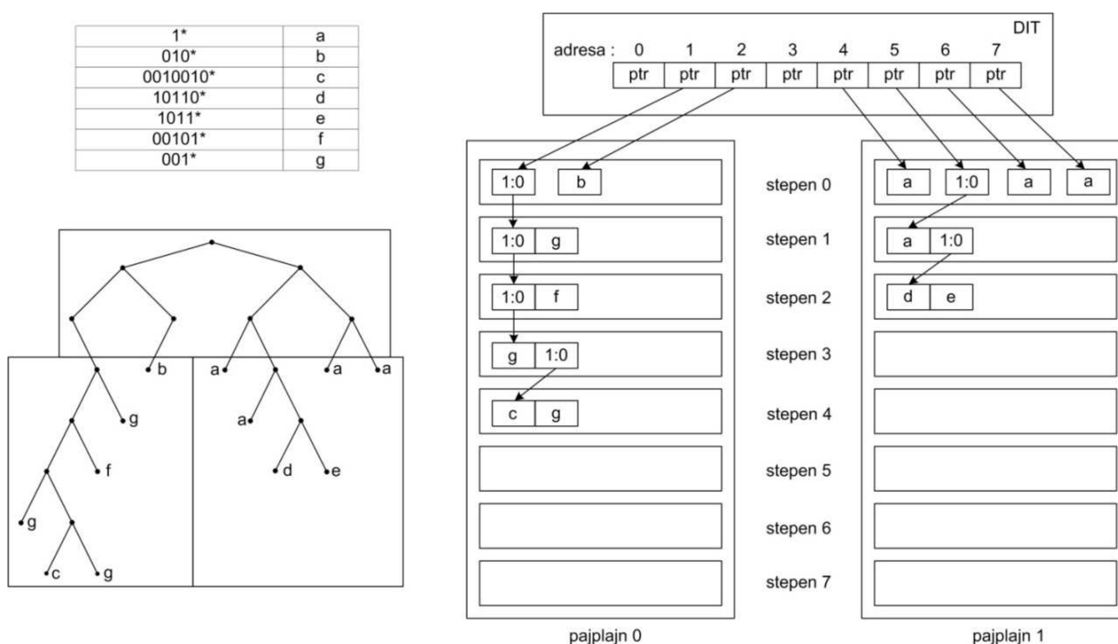
Povećanje broja neprioritetnih čvorova može uticati na povećanje vremena pretrage, pošto će u više pretraga biti potrebno doći do kraja stabla, umesto završetka pretrage nailaskom na prioriteni čvor. Ovo produženje vremena pretrage nije značajno, a može se prevazići izvršavanjem proračuna celog prioritetnog stabla, čime se može uvećati broj prioritenih čvorova.

3.1.2.9 POLP lukap procedura

POLP (*Parallel Optimized Linear Pipeline*) lukap procedura [42] koristi tehnike paralelizacije i pajplajna u cilju postizanja velike lukap brzine. POLP se zasniva na jednobitskom binarnom stablu i tehnici potiskivanja ruta u listove. POLP algoritam čvorove jednobitskog stabla smešta u memorije koje su grupisane u pajplajnove. Više IP adresa se može istovremeno pretraživati u svakom od pajplajna, i korišćenjem više pajplajna se dodatno povećava mogućnost paralelnog pretraživanja, a time i broj izvršenih pretraga u jedinici vremena. POLP stablo je na dubini I podeljeno na podstabla. Dubina I naziva se dubina deljenja stabla (eng. *splitting depth*). Deo adrese koji čine prvih I bita se koristi kao identifikator podstabla. Identifikator podstabla se koristi kao adresa u DIT memoriji (eng. *Destination Index Table*), koja sadrži adrese podstabala u prvim stepenima pajplajnova. Svako od podstabala se pridružuje jednom od pajplajnova. Sami pajplajnovi se sastoje od niza stepeni, odnosno memorija, i broj memorija treba da bude jednak ili veći maksimalnoj dubini podstabla. Na Slici 3-10

desni deo slike predstavlja sadržaj POLP lukap modula za tabelu rutiranja prikazanu u gornjem levom delu slike. U donjem levom delu slike je prikazano jednobitsko stablo koje odgovara korišćenoj tabeli rutiranja.

POLP strukture podataka prikazane na Slici 3-10, formirane su za vrednost dubine deljenja podstabla *I*=3, u cilju formiranja pregledne slike. Prefiksi dužine 3 imaju vrednosti od 0 do 7, i DIT memorija ima osam lokacija u kojima su zapisane adrese podstabala u prvim stepenima pajplajnova. Čvorovi podstabla se raspoređuju u memorijama pajplajna poštujući pravilo da čvor-dete treba da bude smešten u memoriji koja se u pajplajnu nalazi iza memorije u koju je smešten čvor-roditelj. U čvor-roditelj se upisuje rastojanje od memorije čvora-roditelja do memorije čvorova-dece i pozicija čvorova-dece u okviru njihove memorije. Zapis čvorova-dece jednog pored drugog omogućava da čvor-roditelj umesto dva pokazivača na čvorove-decu sadrži jedan pokazivač. Na primer, na Slici 3-10, oznaka 1:0 označava da se čvorovi-deca nalaze na pozicijama 0 i 1 u sledećoj memoriji ispod memorije čvora-roditelja. Prvi broj u oznaci 1:0 označava rastojanje od memorije čvora-roditelja do memorije čvorova-dece, dok drugi broj označava poziciju levog čvora-deteta u njegovoj memoriji. Pretraga se pomera iz memorije u memoriju niz pajplajn, i u jednom trenutku pristupa se jednoj od memorija u okviru svake pretrage, dok ostale memorije mogu biti korišćene od strane drugih pretraga u tom pajplajnu.



Slika 3-10 POLP lukap procedura

3.1.2.9.1 Procedura ažuriranja za POLP lukap algoritam

Autori POLP lukap procedure su predložili dva algoritma [46] za inkrementalno dodavanje ruta, agresivan i konzervativan. Kod konzervativnog postupka, čvorovi se raspoređuju u najpraznije stepene od mogućih stepeni, pri čemu su mogući stepeni određuju tako da preostane mesta za dodavanje sledećih ruta. Dakle, ukoliko je ostalo još m bita prefiksa, odnosno m čvorova da se mapira na stepene, tada se pronalazi m stepena memorijski najmanje iskorišćenih, pri čemu iza poslednjeg stepena treba da ostane onoliko mesta koliko je razlika između najdužeg prefiksa i prefiksa koji se smešta . U agresivnoj metodi, čvorovi se raspoređuju slobodno, pa se vrši remapiranje, odnosno pomeranje čvorova ka početnim stepenima ukoliko nova ruta ne može da bude dodata. Obe metode su analizirane kako za jednake veličine memorije stepena, tako i za slučaj kada su memorije stepeni oko 16. stepena znatno veći da bi bili prilagođeni većem broju prefiksa.

U okviru ovog rada predlaže se inkrementalna procedura u okviru koje su veličine memorija u okviru pajpaljnova proporcionalne raspodeli prefiksa na Internetu. Novi čvor stabla dodaje se u prvi raspoloživi slot u stepenu ispod stepena u koji je smešten čvor roditelj. Ukoliko takav stepen ne postoji, tada se postojeći prethodnici čvora koji se dodaje pomeraju ka praznim slotovima u prethodnim stepenima ukoliko je moguće. Ukoliko takvo pomeranje nije moguće realizovati, potrebno je ponovo uraditi proračun za ceo pajplajn, kako bi se mogao dodati novi čvor. U tom slučaju, nodovi se pridružuju memorijama u opadajućem redosledu prema njihovim visinama (najveća udaljenost od lista). Ukoliko ni nakon procesa ponovnog proračuna za pajplajn, novi čvor ne može da bude upisan u memoriju, tada se vrši proračun za celu lukap tabelu, koji uključuje ponovnu preraspodelu podstabala po pajplajnovima.

Kada se vrši ponovni proračun za lukap tabelu, jednobitsko stablo se deli na podstabla čiji su koreni na nivou deljenja (I). Ova podstabla se sortiraju u opadajućem redosledu prema broju njihovih čvorova, i u tom redosledu se pridružuju pajplajnovima. U slučaju ponovnog proračuna kompletne lukap tabele, podstablo se dodaje u onaj pajplajn koji u tom trenutku ima najmanji broj nodova.

U toku procesa dodavanja čvorova u stepene pri ponovnom proračunu pajplajna, stepen i se popunjava do N_i čvorova, pri čemu je

$$N_i \leq S_i \cdot N_r / \sum_{j=i+1}^{N_{stages}} S_j, \tag{2.1}$$

gde je S_i veličina stepena i N_r je broj preostalih čvorova koje treba dodati u pajplajn. Ograničavajući broj nodova u stepenu i pajlajna na N_i, čuva se memorija za prefikse koje će biti potrebno dodati u toku sledećih promena tabele rutiranja. Cilj je pridružiti čvorove iz istog nivoa stabla istom stepenu, čime bi se smanjio broj konflikata u okviru pajplajna. Konflikti se događaju kada dva lukap procesa pokušaju da pristupe istom stepenu.

Dodavanje novog prefiksa je zasnovano na prolasku kroz jednobitsko stablo prema bitima prefiksa. Ukoliko se prilikom pomeranja kroz stablo (u skladu sa bitima prefiksa) naiđe na list, potrebno ga je zapamtiti u cilju potiskivanja u listove nakon dodavanja novog prefiksa. Novi čvorovi stabla se kreiraju u skladu sa bitima prefiksa. Ako je dubina novog kreiranog čvora I novo podstablo se kreira i pridružuje pajplajnu sa najmanjim brojem nodova. Kada se formiraju svi novi nodovi, potrebno je izvršiti potiskivanje zapamćenog noda u list. U suprotnom, ukoliko su svi čvorovi za bite koji odgovaraju novom prefiksu već bili formirani, tada je potrebno izvršiti potiskivanje poslednjeg čvora sa putanje novog prefiksa u list.

Prilikom postupka dodavanja novog čvora u pajplajn, čuva se informacija o jednom praznom slotu u okviru stepena ukoliko postoji, u cilju efikasnijeg dodavanja novog čvora (detaljniji opis će biti dat u pododeljku 4.2.2.3 o implementaciji). Čvor se pridružuje prvom slobodnom slotu koji je u stepenu iza stepena u kome je smešten čvor-roditelj. Nakon smeštanja čvora u stepen, informacije o njegovoj lokaciji upisuju se u čvor-roditelj. Takođe, potrebno je i uvećati brojač koji čuva informacije o broju čvorova u pajplajnu u koji je dodat novi čvor.

Postupak brisanja rute vrši se tako što se na osnovu bita prefiksa koji se želi obrisati pronalaze čvorovi u jednobitskom stablu, i brišu se svi prefiksi iz listova koji odgovaraju toj ruti. Ukoliko u stablu postoji ruta sa kraćim prefiksom koji je istovremeno i prefiks rute koja se briše, tada će izlazni port rute sa kraćim prefiksom u listovima zameniti izlazni port rute koja se briše. Ukoliko ruta sa takvim kraćim prefiksom ne postoji, tada ne postoji ruta koju bi bilo potrebno potisnuti u listove. U tom slučaju listovi koji sadrže izlazni port rute koja se briše uklanjaju iz stabla. Algoritam brisanja rute u POLP lukap modulu opisan je u pododeljku 4.2.2.1.

Zamena izlaznog porta prefiksa predstavlja slučaj kada se ruta ponovo dodaje, sa novom vrednošću izlaznog porta. Algoritam dodavanja rute obuhvata i ovaj slučaj, i njegov rezultat će biti zamena izlaznog porta za tu rutu u lukap modulu. I za ostale lukap module važi da algoritam dodavanja rute vrši i zamenu izlaznog porta rute.

3.1.2.10 BPFL lukap procedura

Lukap algoritam koji omogućuje paralelizaciju uz efikasno korišćenje memorije i mali broj pristupa memorijama u toku lukapa je BPFL (*Balanced Parallel Frugal Lookup*) [42]-[44]. BPFL algoritam je razvijen na Elektrotehničkom fakultetu u Beogradu.

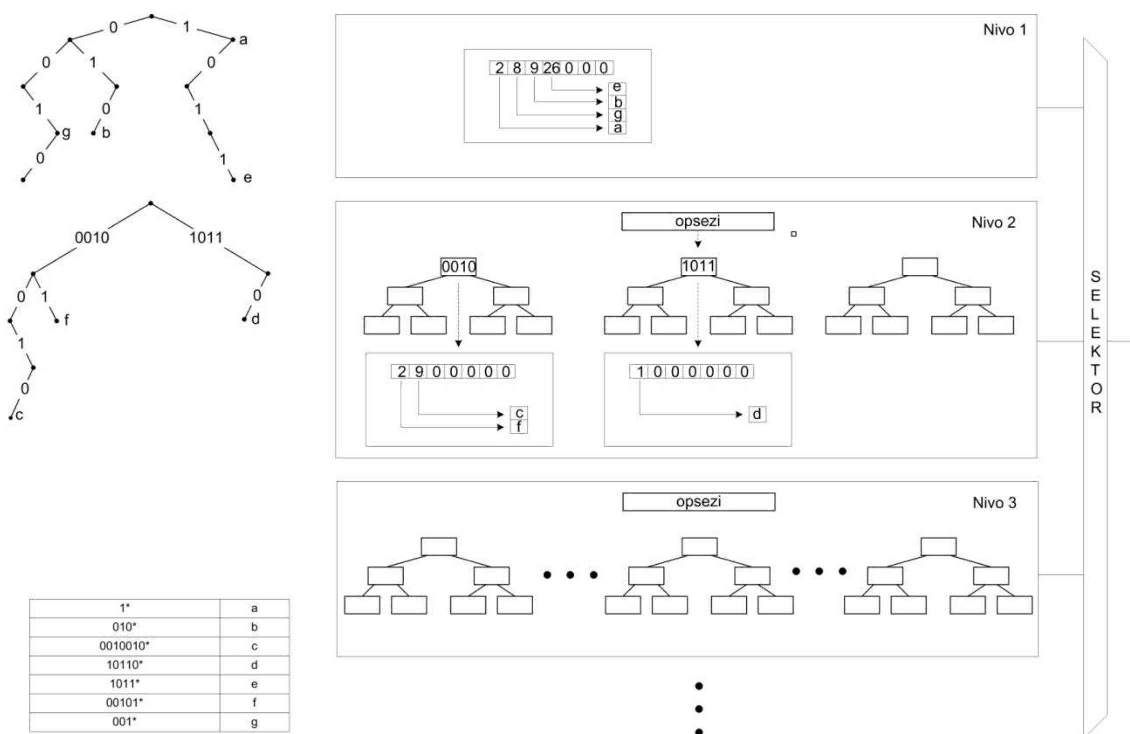
BPFL je baziran na više multibitskih stabala, od kojih svako obuhvata jedan opseg prefiksa. Struktura BPFL lukap tabele je prikazana na Slici 3-11. Prefiksi su podeljeni na L nivoa, u zavisnosti od njihovih dužina. Svaki nivo i sadrži prefikse čije su dužine u opsegu $(i-1)\cdot D_S$ i $i\cdot D_S$. $D_S = L_a/L$, pri čemu je L_a označava broj bita u IP adresi. Na svakom od nivoa, informacije o prefiksima se čuvaju u okviru balansiranih stabala i u podstablima.

U okviru nivoa, pretraga se prvo vrši u okviru prvih $p_i = (i-1)\cdot D_S$ bita adrese, koji čine prvi korak multibitskog stabla tog nivoa. Sve vrednosti koje ovi biti uzimaju u okviru tabele rutiranja smeštene su u balansirana stabla.. Da bi se izbeglo pretraživanje svih balansiranih stabala u okviru nivoa, stabla se popunjavaju tako da svako sadrži prefikse iz određenog opsega, i granice opsega se čuvaju u posebnim registrima. Prvi korak u pretrazi u okviru nivoa je pretraga opsega i pronalaženje stabla u okviru čijih opsega se nalazi prefiks IP adrese. Pretragom balansiranog stabala pronalazi se poklapanje prvog koraka multibitskog stabla sa prvih p_i bita IP adrese čiji se izlazni port traži. Kada se u okviru balansiranog stabla pronađe čvor koji odgovara IP adresi, tada se prelazi na pretragu sledećih D_S bita IP adrese.

Delovi jednobitskog stabla dubine D_S koji se nalaze ispod čvorova koji odgovaraju vrednostima balansiranog stabla čuvaju se u podstablima. Svakom čvoru balansiranog stabla odgovara jedno podstablo koje je smešteno na memorijskoj lokaciji određenoj čvorom balansiranog stabla. Ukoliko se u podstablu pronađe prefiks tabele rutiranja koji se poklapa sa IP adresom, tada taj prefiks postaje najduže poklapanje u

okviru nivoa. Ukoliko podstablo sadrži više prefiksa tabele rutiranja koji se poklapaju sa IP adresom, bira se najduži od njih. U okviru BPFL algoritma predviđena su dva načina zapisa prisustva prefiksa u čvorovima podstabala. Prvi način predviđa korišćenje jednog bita za svaki čvor podstabla, i vrednost bita 1 označava da u tom čvoru postoji prefiks tabele rutiranja. Ovaj način zapisa naziva se zapis gusto popunjenog podstabla. Drugi način zapisa naziva se zapis retko popunjenog podstabla i u njemu se ne beleže informacije za sve čvorove podstabla već se samo beleže indeksi čvorova koji sadrži prefikse. Zapis retko popunjenog podstabla koristi se za podstabla koja sadrže mali broj prefiksa zato što tada zauzima manje mesta u memoriji od gustog zapisa. Podstabla zapisana u zapisu retko popunjenih podstabala menjaju zapis u zapis gusto popunjenih podstabala kada broj prefiksa u podstablu dostigne predefinisanu vrednost.

Ukoliko poklapaje postoji na više nivoa, rezultat pretrage je najduži prefiks, odnosno prefiks na najdubljem nivou (nivou sa najdužim prvim korakom multibitskog stabla). Izlazni port je određen nivoom, čvorom balansiranog stabla, i čvorom podstabla koji odgovaraju IP adresi.



Slika 3-11 BPFL lukap procedura

Pošto se u okviru nivoa poređenje sa IP adresom prvo vrši na prvih $(i-1) \cdot D_S$ bita odjednom a potom na sledećih D_S bita bit po bit, pretraga u okviru nivoa se može

predstaviti kao pretraga multibitskog stabla u kome prvi korak ima $(i-1)\cdot D_S$, a sledećih D_S koraka po jedan bit. Na Slici 3-11 u desnom delu slike predstavljena je struktura multibitskih stabla koja odgovaraju BPFL nivoima. Nivo 1 ne sadrži balansirana stabla, već samo podstablo, koje je na primeru sa Slike 3-11 predstavljeno u zapisu retko popunjenog podstabla. Ostali nivoi na Slici 3-11 sadrže opsege i balansirana stabla, a čvorovima balansiranih stabala pridružena su podstabla. Na Slici 3-11 u okviru nivoa 1 predstavljena su dva podstabla pridružena čvorovima balansiranih stabala sa vrednostima 0010 i 1011.

3.1.2.10.1 Procedura ažuriranja za BPFL lukap algoritam

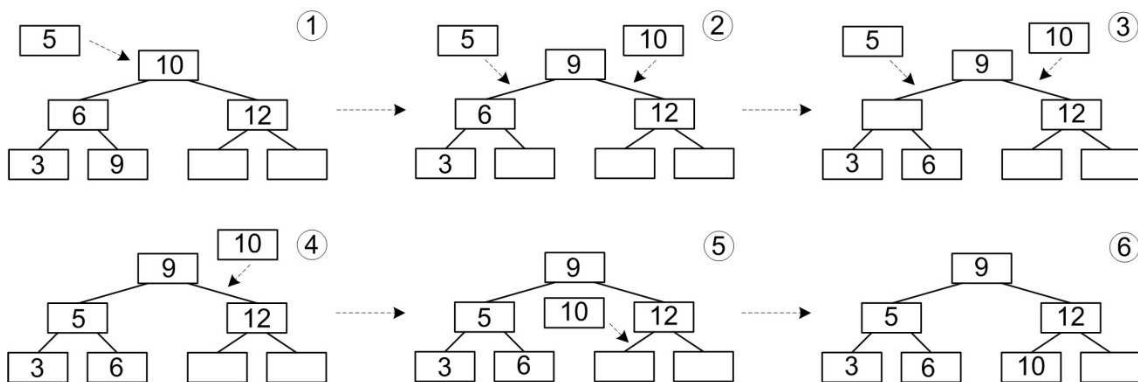
U BPFL-u, inicijalno, vrednosti donjih i gornjih granica opsega koji odgovaraju balansiranim stablima nivoa i su jednaki $(i-1)\cdot D_S$ delu prefiksa dolazećih ruta, sve dok je broj ruta manji ili jednak broju balansiranih stabala na posmatranom nivou. Nakon postavljanja inicijalnih vrednosti za opsege balansiranih stabala, prefiks se dodaje ili u najbliži opsegili u opseg kome pripada. Ukoliko se prefiks doda u opseg kome pripada granice opsega se ne menjaju. Ukoliko se prefiks doda u najbliži opseg tada on postaje jedna od granica opsega. Granice opsega će se takođe promeniti ukoliko dođe do pomeranja prefiksa iz jednog stabla u drugo.

Prilikom dodavanja rute u BPFL lukap tabelu, najpre se binarnom pretragom pronalazi balansirano stablo u koje nova ruta treba da bude dodata Odgovarajuće balansirano stablo se određuje na osnovu opsega kome pripada prefiks koji treba dodati. Binarna pretraga se izvršava na nizu granica opsega. Balansirano stablo ili pripada opsegu kojem pripada i prefiks koji se dodaje, ili je to balansirano stablo koje pripada opsegu koji je najbliži prefiksu koji se dodaje. Ukoliko odabrano stablo nije puno, novi prefiks se dodaje u njega. Potom, ukoliko neko od susednih balansiranih stabala ima bar dva nepopunjena mesta više od odabranog stabla u koje se dodaje prefiks, odgovarajući granični čvor iz posmatranog stabla se premešta u susedno stablo, u cilju održavanja što ravnomernije raspodele čvorova u balansiranim stablima. Ukoliko je balansirano stablo čijem opsegu pripada prefiks koji dodajemo puno, jedan od njegovih graničnih čvorova se pomera u susedno stablo. Susedno stablo određeno je onom stranom koja je manje popunjena. Na taj način čvorovi se ravnomernije raspoređuju po balansiranim stablima, i smanjuje se potreba za kasnijim preuređivanjima balansiranih stabala.

Balansirana stabla predstavljaju kompletna stabla, odnosno svi nivoi balansiranog stabla su kompletno popunjeni izuzev poslednjeg nivoa koji se popunjava sa leva na desno. Iz tog razloga brisanje i dodavanje rute u balansirano stablo može uključivati pomeranje čvorova kako bi bio ispunjen uslov kompletne popunjenosti. Nakon dodavanja čvora u balansirano stablo, odgovarajuće podstablo na lokaciji određenoj dodatim čvorom se ažurira, a informacija o izlaznom portu se memoriše. Takođe na kraju procesa dodavanja nove rute, potrebno je postaviti nove granice za opseg balansiranog stabla.

Kako bi se dodala nova ruta u balansirano stablo, potrebno je znati za svako balansirano stablo koji se nivo trenutno popunjava i broj zauzetih čvorova u nepopunjenom nivou. Kada želimo dodati novi čvor u stablo, polazimo od korena stabla i pomeramo se na levu stranu ukoliko je vrednost rute koju želimo da dodamo manja u odnosu na vrednost u čvoru sa kojim vršimo poređenje, odnosno na desnu stranu u suprotnom slučaju. Sa ciljem održavanja kompletnosti balansiranog stabla, ukoliko je odabrana izlazna grana na putanji ka prvom slobodnom čvoru u poslednjem nivou, nastavljamo poređenje rute koju želimo da dodamo sa vrednošću čvora do koga smo došli. Ukoliko se kretanje ne bi nastavilo ka prvom slobodnom čvoru u poslednjem nivou, pamti se vrednost čvora do kog smo došli, i briše se iz tog čvora. Potom se vrednost sa kojom se do tada prolazilo kroz stablo dodaje u podstablo koje nije na putu prvog slobodnog mesta u poslednjem nivou stabla. Način dodavanja ove tekuće vrednosti zavisi od toga da li se dodaje u levo ili desno podastablo. Ako se dodaje u levo podstablo, tekuća vrednost se poredi sa najvećom vrednošću u tom podstablu i, ukoliko je veća od najveće vrednosti, tekuća vrednost se odmah smešta u upražnjeno mesto do koga se stiglo. Najveća vrednost u podstablu se pronalazi pretragom podstabla, koja se vrši prolaskom kroz podstablo i pamćenjem najveće vrednosti na koju se pri prolasku naišlo. Ukoliko tekuća vrednost nije veća od najveće vrednosti levog podstabla, tada se najveća vrednost levog podstabla premešta u upražnjeno mesto, zatim se prva sledeća manja vrednost iz podstabla premešta na mesto premeštene vrednosti, pa se premešta sledeća manja vrednost, i tako redom dok se ne stigne do tekuće vrednosti koja se upisuje u podstablo. Kada se tekuća vrednost upiše na poziciju koja je pripadala prvom elementu podstabla većem od nje, moguće je nastaviti prolaz kroz stablo sa zapamćenom vrednošću. Ukoliko je tekući čvor potrebno upisati u desno podstablo,

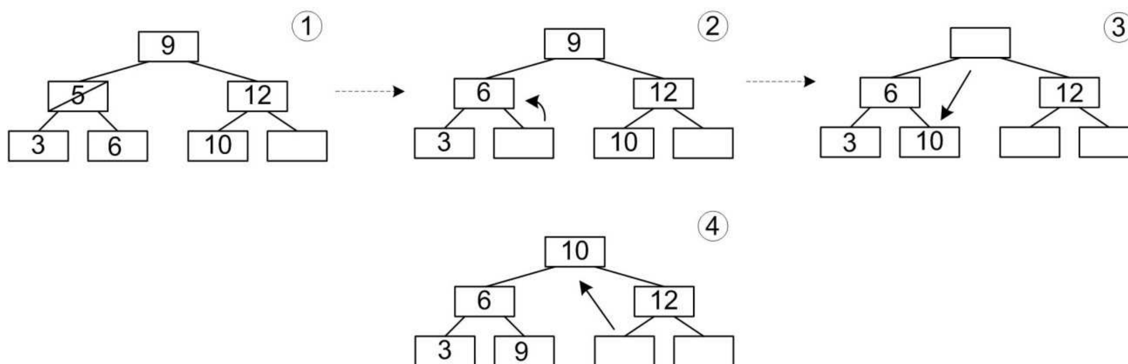
postupak je sličan sa razlikom da se tekući čvor poredi sa najmanjim elementom desnog podstabla, pa se, ako je veći od njega, prvo premešta najmanji čvor desnog podstabla, pa prvi veći i tako redom dok se ne stigne do vrednosti tekućeg čvora. Ovaj način premeštanja obezbeđuje da stablo ostane kompletno i balansirano. Postupak prolaska kroz stablo se nastavlja na opisani način dok se ne popuni prvo slobodno mestu u poslednjem nivou stabla.



Slika 3-12 Dodavanje čvora u kompletno balansirano stablo

Primer dodavanja čvora u balansirano stablo prikazano je na Slici 3-12. Na Slici 3-12 prikazano je dodavanje vrednosti 5 u kompletno balansirano stablo. U prvom koraku dodavanja vrednosti 5 u stablo, nova vrednost se poredi sa vrednošću korena stabla. Da bi stigao do prve slobodne pozicije u poslednjem nivou stabla prolaz kroz stablo bi trebalo da krene desno od korena stabla, a vrednost 5 je manja od vrednosti 10, pa bi na osnovu toga trebalo da krene levo. Da bi se moglo nastaviti kretanje ka prvom slobodnom mestu u stablu, vrednost 10 se uzima iz korena stabla da bi se sa njim nastavio prolaz kroz stablo i pamti se. U koren stabla se u drugom koraku dodavanja čvora prikazanom na Slici 3-12 smešta vrednost 9, najveća vrednost iz levog podstabla. Na mesto vrednosti 9 smešta se prva sledeća manja vrednost iz levog podstabla, i to je vrednost 6. Prva sledeća vrednost iz levog podstabla je manja od vrednosti 5, pa se ona ne pomera, već se u trećem koraku vrednost 5 smešta na upražnjeno mesto na kome je bila vrednost 6. Pošto je vrednost 5 dodata u levo podstablo, nastavlja se sa dodavanjem vrednosti 10 u desno podstablo. U četvrtom koraku algoritma, vrednost 10 se poredi sa vrednošću 12, i pošto je manja od 12, prolazak kroz stablo se nastavlja u levo podstablo čvora 12 koji sadrži prvo slobodno mesto u poslednjem nivou stabla. U petom koraku vrednost 10 se upisuje u novi čvor u poslednjem nivou stabla, čime se završava operacija dodavanja čvora u stablo.

Prilikom brisanja rute iz BPFL lukap tabele potrebno je najpre odrediti čvor balansiranog stabla koji odgovara ruti koju je potrebno obrisati. Ukoliko to nije poslednja ruta koja odgovara tom jednom čvoru balansiranog stabla potrebno je samo obrisati informaciju iz podstabla o postojanju odgovarajuće rute. Ukoliko je to jedina ruta koja odgovara tom čvoru balansiranog stabla potrebno je obrisati čvor balansiranog stabla. Prilikom brisanja čvora treba održati uslov kompletne popunjenosti balansiranog stabla. Ako se briše poslednji čvor iz poslednjeg nivoa (najdubljeg nivoa) balansiranog stabla, tada stablo posle brisanja tog čvora ostaje kompletno. Ukoliko je potrebno obrisati neki drugi čvor balansiranog stabla tada je nakon brisanja željenog čvora potrebno izvršiti premeštanje čvorova. To se vrši tako što se na mesto obrisanog čvora premešta vrednost prefiksa koja je između vrednosti poslednjeg čvora iz poslednjeg nepopunjenog nivoa i vrednosti prefiksa koji se briše. Na mesto obrisanog čvora se premešta prefiks najbliži obrisanom prefiksu, zatim se na njegovo mesto premešta sledeći najbliži prefiks i tako redom dok se ne premesti prefiks iz poslednjeg čvora u poslednjem nivou.



Slika 3-13 Brisanje čvora iz kompletnog balansiranog stabla

Primer brisanja čvora iz balansiranog stabla prikazan je na Slici 3-13. U primeru prikazanom na Slici 3-13, vrednost 5 se briše iz stabla. Pošto stablo treba da zadrži osobinu kompletnosti, na kraju brisanja treba da bude uklonjen poslednji čvor iz poslednjeg nivoa stabla. U primeru sa Slike 3-13, vrednost u tom čvoru je 10, i to je vrednost koja je veća od vrednosti koja se briše. Pošto je vrednost zapisana u poslednjem čvoru u poslednjem nivou stabla veća od vrednosti koja se briše, brisanje se vrši tako što se na mesto obrisanog čvora smešta prvi sledeći veći čvor iz stabla, pa se na mesto tog čvora smešta prvi sledeći veći od njega, i tako dok se ne obriše poslednji čvor u poslednjem nivou stabla. U primeru sa Slike 3-13, u prvom koraku se briše

vrednost 5 iz čvora u kome se nalazi, pa se u drugom koraku na njegovo mesto upisuje prva sledeća veća vrednost koja postoji u stablu a to je vrednost 6. Vrednost 6 se briše iz čvora u kome se nalazila, i u trećem koraku algoritma, na njeno mesto se upisuje prva sledeća veća vrednost u stablu, i to je vrednost 9 koja se nalazi u korenu stabla. Vrednost 9 se briše iz korena stabla, i na njeno mesto se upisuje prva sledeća veća vrednost koja postoji u stablu, a to je vrednost 10. Vrednost 10 se nalazila u posledenjem čvoru u poslednjem nivou stabla, i postupak brisanja vrednosti iz stabla se završava brisanjem poslednjeg čvora u poslednjem nivou.

Modifikacija rute u BPFL lukap tabeli slična je procesu lukap pretrage. Najpre se na osnovu prefiksa određuje nivo, a potom lokacija na kojoj je sačuvana informacija o izlaznom portu koju je potrebno ažurirati.

3.1.3 Pretraga po dužinama prefiksa

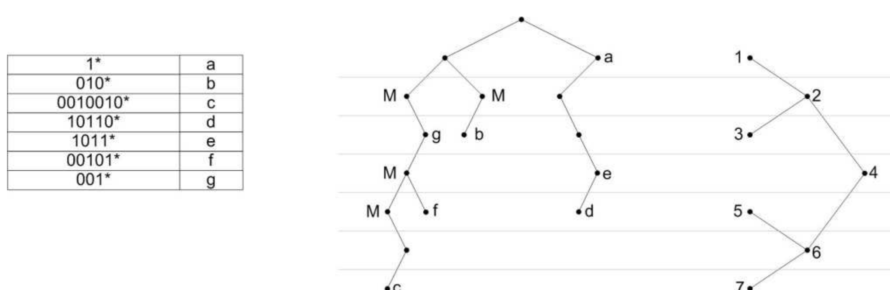
U jednobitskim stablima svaki korak prolaska kroz stablo vrši ne na osnovu jednog bita IP adrese. U multibitskim stablima pretraga svaki korak pretrage izvršava se na osnovu dela IP adrese koji može sadržati jedan ili više bita. Broj koraka u pretrazi odgovara dubini stabla. Duži koraci u multibitskom stablu mogu smanjiti dubinu stabla, uz cenu usložnjavanja algoritma pretrage zbog postojanja prefiksa različitih dužina u svakom koraku ili povećanja potrebne memorije zbog potiskivanja prefiksa na dužine određene koracima multibitskog stabla.

Pretraga po dužinama prefiksa [47] predstavlja alternativni način pronalaženja rute sa najdužim poklapanjem prefiksa. Za pretragu po dužinama, svi prefiksi jednake dužine se grupišu i smeštaju u strukturu pogodnu za brzu pretragu, koja je najčešće heš tabela. Broj heš tabela jednak je broju bita IP adrese. Da bi se izbeglo pretraživanje svih heš tabela, koristi se binarna pretraga, u kojoj se prvo predražuje heš tabela sa prefiksima čija je dužina jednaka polovini dužine IP adrese. U zavisnosti od rezultata pretrage bira se opseg dužina iznad ili ispod ove dužine i proverava se heš tabela na polovini tog opsega i tako redom dok se ne pronađe heš tabela sa najdužim prefiksom koji ima poklapanje sa odredišnom IP adresom.

U svakom koraku binarne pretrage preostali interval dužina prefiksa koga treba pretražiti se polovi, pa se pretraga završava u broju koraka jednakom logaritmu broja dužina prefiksa. Uvek se pretražuje heš tabela na polovini preostalog intervala, i ako se

pronađe poklapanje pretražuju se veće dužine, a ako se ne pronađe poklapanje pretražuju se kraće dužine. Da bi bilo moguće na osnovu poklapanja utvrditi da li postoji duži prefiks koji odgovara odrišnog IP adresi, u heš tabele se pored prefiksa ruta smeštaju i markeri. Markeri su specijalni prefiksi koji označavaju da postoje duži prefiksi koji sadrže prefikse markera.

Pored vrednosti prefiksa, svaki marker sadrži i prethodno proračunatu vrednost najduže rute koja je i prefiks markera. Ovo omogućava bržu pretragu u slučaju kada postoje duži prefiksi koji sadrže prefiks markera i marker usmeri pretragu ka njima, ali duži prefiksi ne predstavljaju prefikse odredišne adrese. Tada se za rezultat pretrage uzima ruta zapamćena pri prolasku kroz poslednji marker. Ako se binarna pretraga završi u heš tabeli koja sadrži prefiks koji odgovara IP adresi, onda taj ruta koja sadrži taj prefiks predstavlja rezultat pretrage.



1*	a
010*	b
0010010*	c
10110*	d
1011*	e
00101*	f
001*	g

Slika 3-14 Pretraga po dužinama prefiksa

Binarno pretraživanje omogućava pronalaženje rute u logaritamskom broju pristupa heš tabelema. Korišćenjem procesora koji u jednom trenutku može izvršavati pristup jednom heš bloku nije moguće dodatno povećati ovu brzinu. Međutim korišćenjem više procesora, ili specijalizovanih logičkih blokova za pristup heš tabelama moguće je dodatno ubrzati pretragu. Ako bi se svakoj heš tabeli pridružio po jedan logički blok koji u njoj vrši pretragu tada bi se mogle paralelno izvršiti pretrage svih heš tabela i korišćenjem dodatnog logičkog bloka među pronađenim rutama sa poklapanjem pronaći ona sa najdužim prefiksom. U slučaju paralelne pretrage ne bi bilo potrebno koristiti markere što bi dovelo do uštede memorije. Mana ovog rešenja je cena logičkih blokova koji vrše pretrage heš tabela, pa je ekonomičnije koristiti opisano rešenje jednim logičkim blokom koji vrši binarnu pretragu.

3.1.3.1 Ažuriranje modula za pretragu po dužinama prefiksa

Ažuriranje modula za pretragu po dužinama prefiksa sastoji se od upisivanja podataka u heš tabele. Za svaku vrednost dužine prefiksa postoji jedna heš tabela i ona sadrži podatke o rutama sa prefiksima te dužine, kao i markere koji ukazuju na postojanje prefiksa veće dužine.

Pri dodavanju nove rute, informacije o prefiksu i izlaznom portu upisuju se u heš tabelu koja čuva prefikse iste dužine kao što je dužina novog prefiksa. Pored ove operacije, potrebno je po potrebi ažurirati markere u heš tabelama koje drže kraće prefikse od novog prefiksa. To je potrebno da bi se u heš tabelama sa kraćim prefiksima binarna pretraga ispravno usmerila ka novom prefiksu. Pri dodavanju novog prefiksa potrebno je ažurirati samo one heš tabele sa novim prefiksima koje se mogu naći na putu binarne pretrage ka heš tabeli u koju se dodaje prefiks. Taj spisak tabela se može prethodno preračunati sa svaku dužinu novog prefiksa. U tabelama sa tog spiska je potrebno proveriti da li sadrže prefiks koji je istovremeno i prefiks nove rute, bilo u formi rute ili u formi markera. Ako te tabele ne sadrže takav prefiks, tada je potrebno dodati marker sa tim prefiksom.

Uz svaki marker potrebno je čuvati i najdužu rutu koja predstavlja prefiks tog markera. Jedan način za pronalaženje te rute je pretraga heš tabela po dužinama prediksa. Time se koriste postojeće heš tabele što pojednostavljuje modul za ažuriranje i omogućava manju potrošnju memorije.

U slučaju brisanja rute, potrebno je obrisati rutu iz heš tabele koja čuva prefikse iste dužine kao što je prefiks koji se briše. Markere koji ogovaraju brisanoj ruti nije potrebno brisati zato što oni u sebi sadrže prethodno proračunatu rutu sa najdužim prefiksom koji je i prefiks tog markera. Brisanje markera se ne radi pošto nije jednostavno utvrditi da li je brisani prefiks jedini prefiks na koga taj marker vrši usmeravanje. Jedno rešenje za ovaj problem bilo bi uvođenje dodatne informacije u marker koja predstavlja broj ruta na koje marker usmerava. Drugo rešenje, koje je predloženo u radu [47], bi bilo periodično preračunavanje sadržaja svih heš tabela na osnovu tabela rutiranja.

U slučaju čuvanja dodatne brojača u markeru o broju ruta na koje marker usmerava, pri brisanju prefiksa bilo bi potrebno jednom proći kroz heš tabele do

prefiksa i dekrementirati ove brojače. Čuvanje brojača bi povećalo i utrošak memorije, ali bi se izbegao ponovni proračun svih tabela rutiranja. Ipak, ponovni proraču je potrebno samo ukoliko ponestane memorije, i ima efekta samo ako se oblik jednobitskog stabla koga prefiski čine značajno promenio, što je retka pojava. Iz tog razloga, ponovni proračun je moguće koristiti umesto čuvanja dodatnih brojača u markerima.

3.1.3.2 Ubrzanje pristupa heš tabelama korišćenjem Blumovih filtara

Korišćenje heš tabela za lukap često je praćeno korišćenjem Blumovih filtara (eng. *Bloom filters*) [49]. Blumovi filtri zauzimaju malo mesta u memoriji i omogućavaju utvrđivanje da li se određena vrednost nalazi u heš tabeli. Blumovi filtri sa sigurnošću daju informaciju da se vrednost ne nalazi u heš tabeli, i sa određenom verovatnoćom daju informaciju da se vrednost nalazi u heš tabeli.

U osnovnoj varijanti, Blumov filtar je niz bita. Biti filtra se inicijalno postavljaju na nulu. Za svaku vrednost koja se dodaje u Blumov filtar proračunava se nekoliko različitih heš funkcija, čiji je opseg vrednosti jednak broju bita u Blumovom filtru. Kada se vrednost dodaje u Blumov filtar, biti na pozicijama koji odgovaraju rezultatima heš funkcija se postavljaju na jedan.

Kada se radi upit da li je vrednost prethodno dodavana u Blumov filtar, proračunavaju se heš funkcije filtra za tu vrednost, i proverava se da li su svi biti koji odgovaraju rezultatima funkcija postavljeni na jedan. Ako svi biti nisu postavljeni na jedan, to znači da vrednost prethodno nije dodavana u Blumov filter. Ako svi biti jesu postavljeni na jedan, to znači da je vrednost ranije možda dodavana. Ako su svi biti postavljeni na jedan, nije sigurno da je vrednost ranije dodavana, pošto je moguće da su ti biti postavljeni na jedan kao rezultat proračuna heš funkcija za neku drugu vrednost koja je dodavana u filtar. Događaj kada su na svim mestima određenim rezultatima Blumovog filtra biti postavljeni na jedinice, a vrednost nije bila dodavana u filtar, naziva se lažno pozitivno poklapanje. Povećavanjem broja vrednosti dodatih u filtar povećava se verovatnoća lažnog pozitivnog poklapanja.

U slučaju realizacije Blumovog filtra sa nizom bita, ne postoji mogućnost brisanja vrednosti iz filtra, pošto nije moguće utvrditi da li biti pridruženi toj vrednosti nisu pridruženi i nekoj drugoj vrednosti dodatoj u filter. Ovo je moguće rešiti

korišćenjem Blumovih filtara sa brojačima, u kojima se umesto bita koriste brojači koji se inkrementiraju kada se vrednost dodaje u filtar i dekrementiraju kada se vrednost briše iz filtra. Korišćenje brojača je nepovoljno jer se uvećava količina memorije potrebna za smeštanje Blumovog filtra. Brojači korišćeni u Blumovom filtru su ograničene veličine, i u slučaju da dođu do maksimalne vrednosti, sledeća dodavanja vrednosti ne mogu da rezultuju inkrementiranjem brojača. Zbog toga, vrednosti se ne mogu brisati iz filtra ako brojači koji odgovaraju tim vrednostima imaju maksimalnu vrednost.

Korišćenje Blumovih filtara može značajno poboljšati performanse traženja izlaznog porta za odredišnu IP adresu, jer mogu smanjiti broj pristupa heš tabelama. Ušteda vremena se ostvaruje zato što je kompaktne Blumove filtre moguće čuvati u memoriji na čipu na kojoj se nalazi i logički blok koji izvršava pretragu heš tabela. Pristup memoriji na čipu je bar jedan red veličine brži od pristupa heš tabelama u memorijama na drugim čipovima.

Pri korišćenju Blumovih filtara, svaka heš tabela ima pridružen Blumov filtar i vrednosti koje se dodaju u heš tabelu dodaju se i u pridruženi Blumov filtar. Pre svakog pristupa heš tabeli, prvo se proverava da li pridruženi Blumov filtar sadrži tu vrednost. Ako je ne sadrži tada nije potrebno ni pristupati heš tabeli, jer heš tabela ne sadrži tu vrednost. Na ovaj način se smanjuje broj pristupa heš tabelama u drugim memorijskim čipovima i ubrzava pretraga.

3.1.3.2.1 Ažuriranje Blumovih filtara

Ažuriranje Blumovih filtara je dodatna operacija koju treba obaviti u slučaju kada se oni koriste za ubrzanje rada sa heš tabelama. Pri dodavanju nove rute, u Blumovom filtru je potrebno ažurirati polja koja odgovaraju vrednostima heš funkcija Blumovog filtra. U slučaju Blumovog filtra sa nizom bita, potrebno je postaviti vrednosti bita na jedan, a u slučaju Blumovog filtra sa brojačima potrebno je uvećati vrednosti brojača, pod uslovom da nisu dostigli maksimalnu vrednost.

Pri brisanju rute potrebno je proračunati vrednosti haš funkcija Blumovog filtra za vrednost koja se briše. Brisanje je moguće samo kod Blumovog filtra sa brojačima, i to samo ako ni jedan od brojača nije dostigao maksimalnu vrednost. U tom slučaju

potrebno je dekrementirati vrednosti svih brojača čime je brisanje vrednosti iz Blumovog filtra završeno.

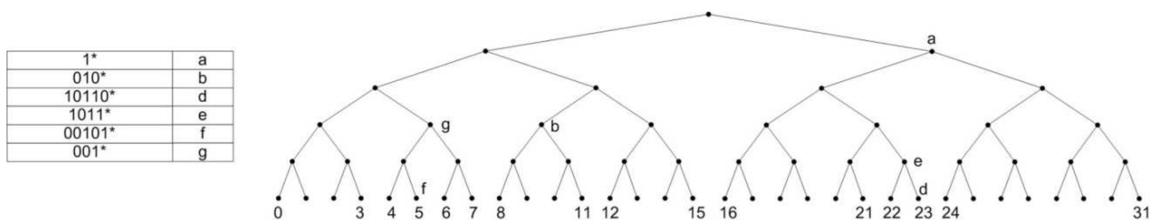
3.1.4 Pretraga po granicama opsega prefiksa

Jedan od uzroka složenosti pretrage tabele rutirana je potreba za pronalaženjem najdužeg poklapanja u skupu prefiksa različitih dužina. Pretraga po granicama opsega [37] prefiksa eliminiše potrebu za poređenjem odredišne IP adrese sa nizovima bita različitih dužina.

U pretrazi po granicama opsega prefiksa, svi prefiksi se prvo produžavaju na maksimalnu dužinu koja je jednaka dužini IP adrese. Ovaj proces rezultuje velikim brojem prefiksa, i poređenje odredišne IP adrese sa takvim skupom prefiksa ne bi bilo efikasno. U pretrazi po granicama opsega koristi se činjenica, da postoje opsezi u kojima prefiksi maksimalne dužine imaju iste vrednosti. Na Slici 3-15 prikazano je stablo u kome su prefiksi produženi do dužine 5. Tabela rutiranja kojoj prefiksi pripadaju takođe je prikazana na Slici 3-15. Na slici je korišćena mala maksimalna dužina prefiksa za koju je moguće prikazati kompletno stablo prefiksa produženih na maksimalnu dužinu.

Na Slici 3-15 u poslednjem redu stabla su prikazane vrednosti prefiksa koje predstavljaju granice opsega sa jednakim vrednostima izlaznih portova. Na primer, prefiksi sa vrednostima između 24 i 31 u desnom delu stabla svi potiču od rute sa prefiksom 1* i svi imaju istu vredost izlaznog porta.

Pretraga po granicama opsega se vrši tako što se utvrđuje opseg kome pripada vrednost odredišne adrese. Na primer, ako je vrednost adrese u primeru sa slike jednaka 25 potrebno je utvrditi da ona pripada opsegu između vrednosti 24 i 31, čime se utvrđuje i vrednost izlaznog porta za tu adresu, *a*.



1*	a
010*	b
10110*	d
1011*	e
00101*	f
001*	g

Slika 3-15 Formiranje granica opsega prefiksa

Pretragu granica opsega je moguće vršiti binarnim pretraživanjem, čime se rezultat pretrage dobija u broju koraka manjem ili jednakom logaritmu broja granica opsega. U tom slučaju sve granice zapisuju se u jedan niz na kome se vrši binarno pretraživanje.

3.1.4.1 Ažuriranje granica opsega prefiksa

Donju granicu opsega za svaku pojedinačnu rutu moguće je proračunati dopisivanjem niza nula na prefiks tog opsega do maksimalne dužine prefiksa. Ako se prefiksu dopišu sve jedinice, tada se dobija gornja granica opsega.

U slučaju da prefiksi tabele rutiranja predstavljaju skup nepreklapajućih prefiksa, dovoljno je generisati donju i gornju granicu opsega za svaki prefiks čime bi bile generisane sve vrednosti granica opsega. Međutim, u tabeli rutiranja često postoje prefiksi od kojih jedan predstavlja prefiks drugog. Na primer, na Slici 3-15 prefiks koji odgovara ruti *a* je prefiks rutama *e* i *d*. Iz tog razloga, granice opsega za prefikse *e* i *d*, su sadržani u okviru granica opsega za prefiks *a*. Da bi skup granica opsega bio uspešno generisan potrebno je dodati dodatne vrednosti u granice opsega, koje označavaju novonastale podopsege. Na primer, na Slici 3-15, vrednosti granica 21 i 24 dodate su da bi ograničile deo opsega prefiksa *a* posle generisanja opsega prefiksa *e* i *d*. Na taj način formirana su dva opsega prefiksa *a*, od kojih prvi ima granice 16 i 21, dok drugi opseg ima granice 24 i 31. Prefiksu *e* odgovara opseg čija se gornja i donja granica poklapaju i iznose 22, dok donja i gornja granice prefiksa *d* iznose 23.

Da bi algoritam ažuriranja uspešno ažurirao granice opsega posle dodavanja nove rute, potrebno je da proveri sve postojeće prefikse koji su istovremeno i prefiksi novog prefiksa, i da proveri sve prefikse koji sadrže u sebi novi prefiks. Uzimajući u obzir ove podatke, algoritam ažuriranja može odrediti eventualnu pojavu novih granica opsega. Nove granice opsega neće se pojaviti u slučaju kada na svakoj putanji od novog prefiksa do listova stabla postoji bar jedan duži prefiks. Ako postoje nove granice opsega njih je potrebno dodati u niz granica opsega na kome se vrši binarna pretraga. Ovo dodavanje može dovesti do pomerenja velikog broja elemenata niza da bi se napravilo mesto za nove elemente, što nije povoljno jer zahteva veliki broj upisa u memoriju koja se koristi za pretragu izlaznih portova, i što može prouzrokovati pauze u procesu traženja izlaznih portova za pakete koji prolaze kroz ruter. Broj elemenata niza

koji se pomera (a sa njima i vrednosti izlaznih portova upisanih u memoriju u kojoj se čuvaju izlazni portovi) zavisi od mesta u nizu na koje se dodaju nove granice.

Slično kao i kod dodavanja rute, i pri brisanju rute potrebno je uzeti u obzir sve prefikse koji su kraći od prefiksa koji se briše i predstavljaju njegov prefiks, kao i prefikse koji sadrže prefiks koji se briše. Na osnovu ovih podataka moguće je odrediti da li postoje elementi niza granica opsega koje je potrebno obrisati. Ako postoje, pri brisanju je potrebno popuniti upražnjena mesta. Da bi niz granica opsega ostao sortiran potrebno pri brisanju elementa niza potrebno je sve elemente niza sa jedne strane za jedan, što može predstavljati veliki broj pomeranja.

U nekim slučajevima pri dodavanju ili brisanju rute nije potrebno menjati broj elemenata niza već samo promeniti vrednost nekog od elemenata. Na primeru na Slici 3-15, pri dodavanju i brisanju prefiksa *d*, potrebno je samo promeniti informaciju o ruti uz četrnaesti element niza granica prefiksa čija je vrednost jednaka 23.

3.2 Analiza performansi

U ovom odeljku biće predstavljena analiza performansi POLP i BPFL lukap algoritama. Specifičnost POLP i BPFL lukap algoritama u odnosu na ostale analizirane algoritme je korišćenje većeg broja memorijskih blokova koji omogućavaju paralelno izvršavanje više operacija pronalaženja izlaznog porta. Pošto POLP i BPFL zahtevaju paralelno izvršavanje više operacija, to se uobičajeno realizuje implementacijom više specijalizovanih logičkih blokova koji vrše ove operacije. Visoki stepen paralelizacije omogućava i veoma visoke protoke paketa, pa POLP i BPFL algoritme čine dobrim kandidantima za korišćenje u ruterima visokih performansi.

U ovom poglavlju biće razmotrene performanse POLP i BPFL algoritma sa stanovišta utroška memorije, broja upisa u lukap modul zbog promena tabele rutiranja i vreme izvršavanja algoritama ažuriranja.

3.2.1 Analiza najgoreg slučaja memorijskih zahteva za IPv6 tabele rutiranja podrazumevajući raspodelu prefiksa na Internetu

U cilju poređenja različitih lukap algoritama poželjno je imati kriterijum za određivanje memorijskih zahteva. Najčešće su u literaturi memorijski zahtevi ilustrovani na primerima realnih ili generisanih tabela rutiranja [48]. Međutim, takve

tabele sa svojom veličinom i raspodelom prefiksa mogu pogodovati nekoj lukap šemi, dok za drugu mogu biti nepovoljne. Zbog toga je potrebno definisati generalni kriterijum za poređenje dva lukap algoritma u pogledu memorijskih zahteva.

U ovom radu biće predstavljen analitički postupak za određivanje memorijskih zahteva za najgori slučaj prefiksa sa određenom raspodelom koja je u skladu sa raspodelom dužina prefiksa na Internetu [37]. U okviru predstavljene analize broj prefiksa date dužine je parametar, a vrednosti prefiksa mogu biti slobodno izabrane. Mi ćemo izračunati potrebnu memoriju za najgori skup prefiksa zadatog kardinalnog broja za svaku dužinu prefiksa.

3.2.2 Analiza najgoreg slučaja za memorijske zahteve za slučaj POLP lukap algoritma

Memorija u POLP-u se koristi za čuvanje čvorova jednobitskog stabla. Iz tog razloga, skup prefiksa koji generišu maksimalan broj čvorova jednobitskog stabla će zauzeti maksimalnu memoriju.

Ovaj skup se može formirati tako što se formira jedan po jedan prefiks, pri čemu svaki novi prefiks generiše maksimalni broj novih čvorova jednobitskog stabla. Skup generisanih prefiksa formira jednobitsko stablo. Kada se generiše novi prefiks zadate dužine, potrebno je da njegov maksimalni zajednički početni deo sa bilo kojim od već generisanih prefiksa bude minimalan. Ako je to ispunjeno, novi prefiks će generisati prvi novi čvor već na prvom nivou jednobitskog stabla koji nije potpuno popunjen, pa će nastaviti generisanje novih čvorova dok ne dostigne zadatu dužinu.

U cilju opisa formiranja prefiksa na predstavljeni način, koristićemo jednobitsko stablo koje sadrži sve čvorove do dubine jednake maksimalnoj dužini prefiksa, odnosno potpuno popunjeno jednobitsko stablo dubine jednake dužini prefiksa. U cilju pronalaženja slobodne pozicije na najmanjoj dubini jednobitskog stabla, numerisaćemo sve čvorove jednobitskog stabla od korena na dole i sa leva na desno na svakoj dubini stabla u skladu sa tehnikom "obilazak stabla po širini" (eng. *breadth-first traversal*) [31]. Način numeracije čvorova stabla prikazan je na Slici 3-16.

Teorema 3.1: Pretpostavimo da su moguće pozicije u stablu numerisane odozgo na dole i sa leva na desno, tako da manji broj čvora bude na manjoj dubini stabla.

Pretpostavimo takođe da su na početku generisanja prefiksa svi čvorovi neoznačeni i da se pri generisanju prefiksa čvorovi pridruženi prefiksu označavaju. Pod navedenim pretpostavkama, skup prefiksa zadatih dužina će zauzimati maksimalnu količinu memorije ako se prefiksi generišu redom tako da svaki prefiks sadrži neoznačeni čvor numerisan najmanjim brojem.

Dokaz:

Pod datim pretpostavkama čvorovi na jednoj dubini su ili svi označeni ili, ako nisu svi označeni, onda svaki od označenih čvorova posmatrane dubine pripada jednom prefiksu.

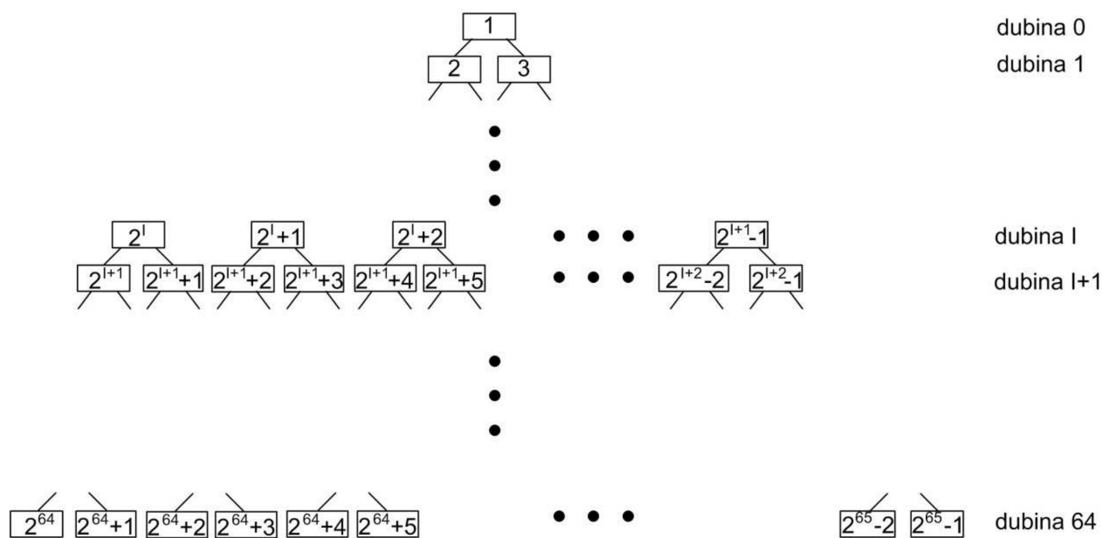
Tvrdnju da svaki od označenih čvorova posmatrane dubine pripada jednom prefiksu možemo dokazati polazeći od suprotne pretpostavke, odnosno da jedan označeni čvor nepopunjenog nivoa pripada bar dvoma prefiksima. Tada drugi od ta bar dva prefiksa ne bi prilikom njegovog generisanja sadržao neoznačeni čvor numerisan najmanjim brojem, što je u suprotnosti sa polaznom pretpostavkom. Dakle, kontradikcijom dolazimo do zaključka da svaki od označenih čvorova posmatrane dubine pripada jednom prefiksu.

Zaključujemo da su čvorovi na jednoj dubini ili svi označeni (maksimalno moguće zauzeti) ili je njihov broj jednak broju prefiksa čija je dužina veća ili jednaka tom nivou, pa prefiksi na tom nivou ne mogu generisati veći broj novih čvorova.

Kako je na svakoj dubini broj zauzetih čvorova maksimalno mogući, to predstavljeni algoritam rezultuje maksimalnom zauzetom memorijom za skup prefiksa zadate dužine. ■

Kako bi pojednostavili opis postupka generisanja prefiksa, smatraćemo da prvo generišemo najduže prefikse, pa potom kraće, u opadajućem redosledu dužina. Svaki prolaz kroz neoznačeni čvor i njegovo označavanje odgovara zauzimanju memorije u POLP stepenu potrebne za čuvanje zapisa čvora. U procesu kretanja kroz neoznačene nodove, izabrano je da pomeranje bude u smeru čvora-deteta sa manjim brojem, odnosno u smeru levog čvora-detata. U procesu označavanja čvorova, uvek se označava i čvor-parnjak (čvor koji ima istog roditelja) čvora koji se označava, što je u saglasnosti sa POLP algoritmom koji uvek alocira memoriju za oba čvora-deteta. Biti koji

odgovaraju putanji do dubine koja odgovara prefiksnoj dužini čine prefiks tabele rutiranja koja generiše maksimalno zauzeće memorije sa datim brojem ruta.



Slika 3-16 Numerisanje čvorova stabla pri proračunu maksimalnog zauzeća memorije

Opisani pohlepni (eng. *greedy*) algoritam zasnovan na tehnici numeracije po širini, na osnovu Teoreme 3.1, generiše skup prefiksa koji zauzima maksimalnu količinu POLP memorije.

Broj čvorova u POLP memorijama generisan predstavljenom procedurom može se sračunati prema jednačini (2.2). Ukoliko je u procesu generisanja prefiksa pozicija neoznačenog čvora ispod nivoa deljenja stabla *I* (pododeljak 3.1.2.9) svaki prefiks kreira dva nova čvora na nivou, $d\big(p(l,x)\big)$, na kome će u toku prolaza kroz stablo biti napravljen prvi novi čvor, i ispod tog nivoa. Broj generisanih prefiksa označen je simbolom *p*, i on je funkcija tekuće dužine prefiksa koji se generišu, *l*, i rednog broja generisanog prefiksa dužine *l*, označenog simbolom *x*, pri navedenom pravilu da se prvo generišu svi prefiksi najveće dužine, pa zatim svi prefiksi prve manje dužine, i tako redom. Nivo na kome će biti napravljen prvi novi čvor proračunava se pomoću jednačine (2.3). Prvih 2^I prefiksa će generisati prvi čvor na nivou deljenja *I*, i nastaviti da generiše po dva nova čvora, sve do dubine koja odgovara dužini prefiksa. Čvorovi iznad nivoa *I* se ne čuvaju u memorijama pajplajnova. Kada se generisanjem 2^I prefiksa nivo deljenja *I* popuni, biće popunjen i prvi sledeći nivo zato što se za nivoe ispod nivoa deljenja pri generisanju svakog čvora generiše i njegov parnjak. Pošto se generiše prefiks na nivou deljenja *I,* na nivou *I*+2 će biti generisan čvor koji odgovara bitu

prefiksa, i njegov čvor parnjak. Potom će na nivou *I*+2 biti generisan čvor koji odgovara bitu prefiksa i njegov čvor parnjak i tako redom do dubine koja odgovara dužini prefiksa. Posle generisanja čvora na nivou *I*, oba njegova čvora-deteta na nivou *I*+1 će biti generisana, i dva od četiri njegova čvora-potomka na nivou *I*+2 će biti generisana. Iz tog razloga, prvi sledeći neoznačen čvor posle popunjavanja nivoa *I* će biti na nivou *I*+2, i na tom nivou će biti 2^{I+1} neoznačenih čvorova. Od ovih 2^{I+1} neoznačenih čvorova, po dva čvora su parnjaci, pa je označavanjem čvorova na nivou *I*+2 moguće generisati 2^{I} novih prefiksa. Kada svi čvorovi na nivou *I*+2 budu označeni, na nivou *I*+3 će biti 2^{I+2} neoznačenih čvorova, od kojih su po dva čvora parnjaci. Sledi da je na nivou *I*+3 moguće generisati 2^{I+1} novih prefiksa. Na svakom sledećem nivou *n* ispod nivoa deljenja *I* moguće je generisati 2^{n-2} novih prefiksa, i postupak generisanja prefiksa se nastavlja dok u datoj raspodeli prefiksa postoje negenerisani prefiksi čija je dužina veća od dubine na kojoj se trenutno nalazi prvi neoznačeni čvor.

$$n = \sum_{l=64}^{d(p(l,x))} \sum_{x=1}^{P(l)} \left(\left(1 + h(p(l,x) - 2^{I})\right) + 2 \cdot \left(l - d\big(p(l,x)\big)\right) \right) \tag{2.2}$$

$$d\big(p(l,x)\big) = \begin{cases} I, & \text{za } p(l,x) \leq 2^{I} \\ \lfloor \log_2(p(l,x) - 1) \rfloor + 2, & \text{za } p(l,x) > 2^{I} \end{cases} \tag{2.3}$$

$P(l)$ predstavlja broj prefiksa dužine l, $p(l,x)$ je redni broj tekućeg prefiksa, $h(x)$ je Hevisajdova jedinična funkcija. U jednačini (2.2) naznačeno je da redni broj prefiksa zavisi od toga do koje dužine prefiksa l se stiglo u toku generisanja prefiksa i od tekućeg rednog broja prefiksa dužine l, označenog sa x. Maksimalno zauzeće memorije se dobija množenjem maksimalnog broja čvorova *n* i memorije potrebne na čuvanje jednog čvora. U okviru formula, Hevisajdova funkcija uzima u obzir da se na nivou deljenja formira jedan čvor a na svim ostalim nivoima dva. Sabirak $1 + h(p(l,x) - 2^{I})$ se odnosi na prvi nivo na kome se prave novi čvorovi, dok sabirak $2 \cdot \left(l - d\big(p(l,x)\big)\right)$ uzima u obzir ostale nivoe. Hevisajdova funkcija u sabirku $1 + h(p(l,x) - 2^{I})$ pri generisanju prvog čvora na nivou deljenja *I* ima vrednost 0, čime se uračunava memorija potrebna za pravljenje jednog čvora. Pri pravljenju prvog čvora na ostalim nivoima Hevisajdova funkcija ima vrednost 1, čime se uračunava

pravljenje dva čvora na tim nivoima. Na nivoima ispod nivoa na kome se pravi prvi čvor generiše se po dva nova čvora, što je predstavljeno sabirkom $2 \cdot \Big(l - d\big(p(l,x)\big)\Big)$.

Jednačina (2.3) predstavlja proračun nivoa na kome će u toku prolaza kroz stablo biti napravljen prvi novi čvor. Kao što je navedeno, pri označavanju prefiksa na nivou deljenja *I* moguće je generisati 2^I prefiksa, što je obuhvaćeno prvim slučajem jednačine (2.3). Drugi slučaj jednačine (2.3) odnosi se na nivo *I*+2 i niže nivoe. Na ovim nivoima moguće je generisati po 2^{n-2} prefiksa, gde je *n* broj nivoa. Označavanjem čvorova na nivou *I*+2 generišu se prefiksi od rednog broja 2^I +1 do rednog broja 2^{I+1}. Ceo deo logaritma za osnovu dva ovih brojeva umanjenih za 1 jednak je *I*, što je deo drugog slučaja jednačine (2.3). U opštem slučaju, označavanjem čvorova na nivou *n* koji je ispod nivoa *I*+1 generišu se prefiksi od rednog broja 2^{n-2}+1 do rednog broja 2^{n-1}. Drugi slučaj jednačine (2.3) omogućava proračun nivoa na kome se vrši označavanje na osnovu rednog broja prefiksa.

Ukoliko je broj prefiksa manji od 2^I, prefiksi kraći od *I* bita mogu rezultovati novim čvorovima na nivou deljenja stabla (I) kao rezultat primene tehnike potiskivanja listova. U najgorem slučaju, takvi prefiksi su pozicionirani tako da generišu maksimalan broj prefiksa na nivou deljenja stabla. Prefiks dužine i, pri čemu je $i < I$ može da generiše do 2^{I-i} prefiksa na nivou deljenja stabla.

3.2.3 Analiza najgoreg slučaja za memorijske zahteve za slučaj BPFL lukap algoritma

BPFL algoritam će imati najveće zahteve za memorijom za onaj skup prefiksa koji će koristiti najveći mogući broj čvorova balansiranih stabala i odgovarajućih podstabala. Ukoliko je broj prefiksa manji od maksimalnog broj čvorova u balansiranim stablima na nivou i, N_{B_i}, najveća memorija će biti korišćena ukoliko svaki prefiks kreira čvor balansiranog stabla i odgovarajuće retko podstablo, kao što je prikazano u jednačini (2.4). U ovom slučaju prefiksi se biraju tako da svaki zauzme novi čvor balansiranog stabla. Memorija koju prefiks zauzima je memorija potrebna za smeštanje čvora balansiranog stabla m_{b_i} i memorija potrebna za smeštanje retko popunjenog podstabla m_{sst}. Ukoliko su svi čvorovi balansiranih stabala na nivou i zauzeti, najgori slučaj raspodele prefiksa će biti onaj koji ima najveći broj gustih podstabala. Zahtevana

memorija u tom slučaju predstavljena je jednačinom (2.5). U ovom slučaju, prefiksi se biraju tako da se smeštaju u isto podstablo dok to podstablo ne dostigne broj prefiksa kome odgovara gusti mod. Potom se prefiksi dodaju u sledeće podstablo dok ono ne pređe u gusti mod i tako redom dok sva podstabla ne budu u gustom modu. Prvi sabirak jednačine (2.5) predstavlja memoriju potrebnu za smeštanje čvorova balansiranih stabala. Drugi sabirak predstavlja memoriju za smeštanje s gusto popunjenih podstabala, dok treći sabirak predstavlja memoriju potrebnu za smeštanje preostalih $N_{B_i} - s$ retko popunjenih podstabala. Ukoliko je broj prefiksa na nekom nivou i dovoljno veliki tako da su sva podstabla u gustom modu, zauzeta memorija će dostići maksimum predstavljen jednačinom (2.6) i on ostaje konstantan tokom dodavanja novih prefiksa.

$$m_{BPFL_i} = \begin{cases} n_i \cdot \left(m_{b_i} + m_{sst}\right), \quad n_i \leq N_{B_i} & (2.4) \\ N_{B_i} \cdot m_{b_i} + s \cdot m_{dst} + m_{sst} \cdot \left(N_{B_i} - s\right), \quad s \leq N_{B_i} & (2.5) \\ N_{B_i} \cdot \left(m_{b_i} + m_{dst}\right), \quad s = N_{B_i} & (2.6) \end{cases}$$

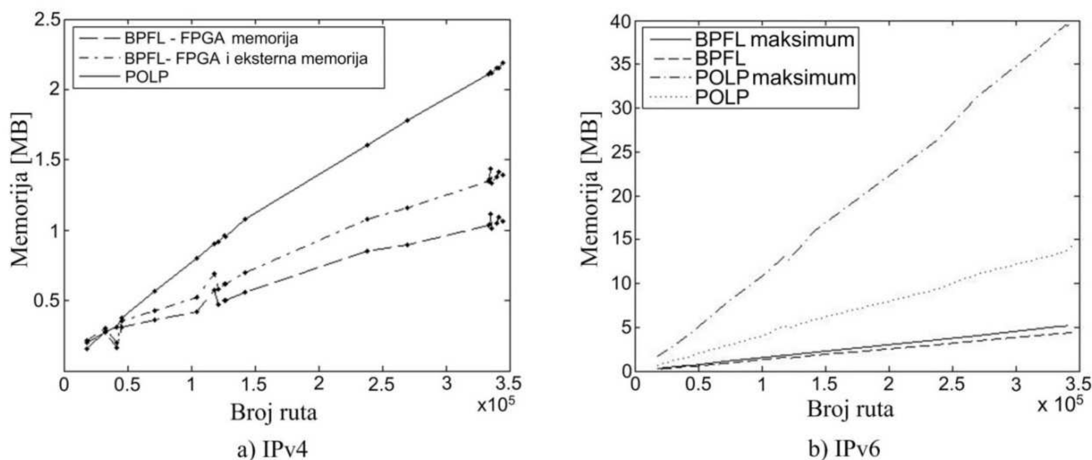
U predstavljenim jednačinama n_i označava broj prefiksa na nivou i, m_{b_i} je memorija potrebna za čvor balansiranog drveta na nivou i, m_{sst} i m_{dst} su memorije potrebne za retko i gusto podstablo, s je broj podstabala u gustom modu.

3.2.4 Memorijski zahtevi

Tri predstavljena algoritma ažuriranja biće upoređena po performansama, odnosno po memorijskim zahtevima i po brzini izvršavanja. Poređenje će biti izvršeno kako za IPv4 verzije algoritama ažuriranja tako i za IPv6 verzije. Za procenu performansi za slučaj IPv4 algoritama ažuriranja korišćene su realne tabele rutiranja koje su preuzete sa [50]-[51]. Kako je IPv6 još uvek u ranoj fazi razvoja, to su i postojeće tabele rutiranja relativno male. U cilju procene performansi za veće tabele rutiranja za koje se predviđa da će biti prisutne u budućnosti, generisane su tabele rutiranja koristeći statistike postojećih. Na osnovu algoritma predstavljenog u radu [48], implementiran je generator koji pravi IPv6 tabele rutiranja na osnovu postojećih IPv4 tabela. Generisanje vrednosti IPv6 prefiksa na osnovu IPv4 vrši se tako što prva dva bajta IPv6 prefiksa sadrže broj autonomnog sistema, zatim sledi vrednost IPv4 prefiksa,

i posle njega slede dva bajta koji imaju slučajni sadržaj. Dužina IPv6 prefiksa dobija se na osnovu dužine IPv4 prefiksa tako što se IPv4 prefiksi dužine 8 preslikavaju u IPv6 prefikse dužine 23. U ostalim slučajevima, dužina IPv4 prefiksa se duplira i, da bi se izbeglo da sve dužine IPv4 prefiksa budu parne, na svaki četvrti od ovih prefiksa se slučajnim izborom dodaje ili oduzima jedan.

U slučaju IPv4 tabela, za svaku od 21 realne tabele rutiranja izmeren je broj čvorova u stablu sa potisnutim listovima, za slučaj POLP algoritma ažuriranja, odnosno broj čvorova u svakom od balansiranih stabala za slučaj BPFL algoritma. Zauzeta memorija je sračunata i predstavljena na Slici 3-17a u funkciji od broja ruta tabele rutiranja. Za slučaj BPFL algoritma na grafiku je predstavljena memorija zauzeta na FPGA čipu, kao i ukupna memorija (FPGA i eksterna memorija koja se koristi za čuvanje informacija o izlaznim portovima). Za slučaj POLP algoritma, pretpostavljeno je da se 18 bita koristi za adresiranje čvorova dece i da su informacije o izlaznim portovima sačuvane u čvorovima listovima. Grafici potvrđuju da BPFL ima manje memorijske zahteve u odnosu na POLP.



Slika 3-17 Memorijski zahtevi BPFL I POLP lukap modula

U slučaju IPv6 tabela rutiranja, izvršeno je poređenje kako za realne i generisane tabele rutiranja, tako i za slučaj onih tabela rutiranja koje bi za zadati broj prefiksa i uz pretpostavljenu distribuciju prefiksa dovele do maksimalnog zauzeća memorije. U cilju određivanja maksimalno zauzete memorije, implementirana je aplikacija koja na osnovu generisanih IPv6 tabele rutiranja formira raspodelu dužina prefiksa, i za formiranu raspodelu formira IPv6 tabele rutiranja koje će biti najnepovoljnije po pitanju

memorijskog zauzeća. Na Slici 3-17b predstavljeni su rezultati za 22 IPv6 tabele rutiranja. Za slučaj POLP algoritma, stablo je podeljeno na dubini *I*=24, dok je dubina 8 izabrana za slučaj BPFL algoritma. S obzirom da su raspodele dužina prefiksa na osnovu kojih su generisane IPv6 tabele dobijene iz postojećih IPv4 tabela rutiranja koje imaju različitu raspodelu dužina prefiksa, to rezultuje u nelinearnosti na grafiku zavisnosti zauzete memorije u odnosu na broj prefiksa (Slika 3-17b). Ove nelinearnosti nisu velike, s obzirom da se raspodele dužina prefiksa ne razlikuju značajno za različit broj prefiksa u tabelama rutiranja. Kada bi se krive za maksimalno zauzeće memorije pravile za tabele rutiranja sa istom raspodelom dužina prefiksa, krive na Slici 3-17b bile bi linearne.

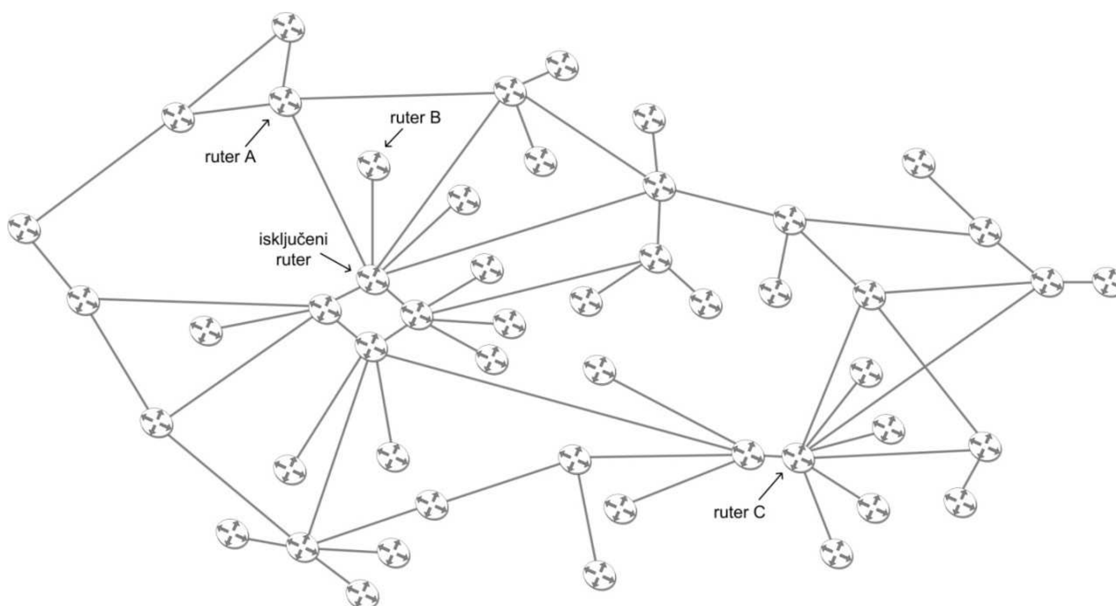
Za slučaj POLP algoritma, najgore zauzeće memorije je značajno veće u odnosu na zauzeće memorije kod pretpostavljenih realnih tabela rutiranja. Ova razlika je posledica zajedničkog prefiksa deljenog među ulazima tabela rutiranja. U slučaju najgoreg zauzeća memorije, zajednički prefiksi su minimizovani. Pošto u najgorem slučaju zauzeća memorije prefiksi dele minimalan broj zajedničkih čvorova, broj generisanih čvorova u tom slučaju je najveći.

U slučaju BPFL algoritma, efekat zajedničkih prefiksa je smanjen, jer se stablo na svakom nivou nezavisno obilazi. Sa Slike 3-17b, bliskost linija koje predstavljaju najgori slučaj memorijskog zauzeća i zauzeća za očekivane tabele rutiranja ukazuje da mali broj prefiksa deli podstabla, odnosno da su tabele rutiranja retke. Kada podstablo pređe u gusti mod, dalje dodavanje prefiksa ne dovodi do zauzumanja memorije, pa bi za veće brojeve prefiksa u podstablima zauzeće memorije bilo manje, i linija zauzeća memorije za očekivane tabele rutiranja ne bi bila bliska liniji najgoreg slučaja zauzeća memorije. Analiza je pokazala da je 99.5% podstabala u očekivanim tabelama rutiranja u „retkom“ modu što se poklapa sa najgorim slučajem. Međutim, pošto je zauzeće u najgorem slučaju i dalje malo, BPFL zahteva malu količinu memorije.

Možemo zaključiti da je POLP osetljiviji na statistiku IP adresa u odnosu na BPFL. Memorijski zahtevi POLP modula su veći od BPFL za IPv4 tabele rutiranja, i ova razlika se dodatno uvećava za IPv6 tabele rutiranja koje imaju znatno duže prefikse. S obzirom das u memorijski zahtevi za POLP su znatno veći u odnosu na memorijske zahteve za BPFL, POLP bi podržao manje tabele rutiranja u odnosu na BPFL.

3.2.5 Broj upisa

Značajna metrika perfomansi algoritama ažuriranja lukap tabela je broj pristupa memoriji u toku procesa ažuriranja. U toku promene memorijskog sadržaja, lukap za dolazne pakete može biti ometen. Iz tog razloga, algoritam ažuriranja je bolji ukoliko zahteva manji broj pristupa memoriji. Kako bi se izvršila procena broja pristupa memoriji koje zahtevaju POLP i BPFL algoritmi ažuriranja, implementacije BPFL i POLP algoritama u slučaju IPv4 verzije, su integrisane u OSPF softver Džona Moja (*John Moy*).

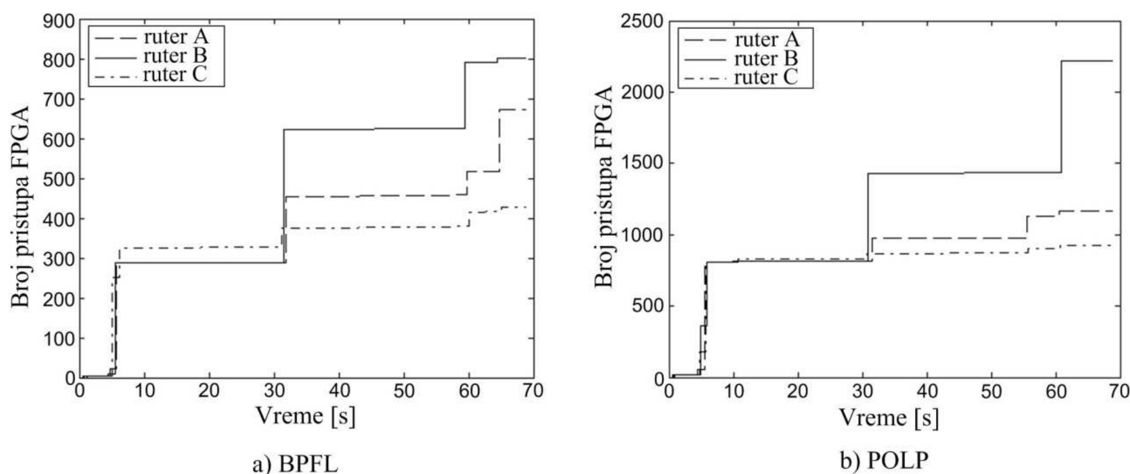


Slika 3-18 Simulaciona mreža

U cilju simulacije OSPF mreže, korišćen je *ospfd_sim* program koji je deo OSPF imlementacije. Simulaciona mreža je prikazana na Slici 3-18. Topologija simulacione mreže je inspirisana Češkom edukacionom i naučnom mrežom CESNET [52]. Simulaciona mreža se sastoji iz 50 rutera sa lokalnom mrežom prikačenom na svaki od rutera. Ruteri su startovani na početku eksperimenta, i u 30. sekundi merenja označeni ruter je isključen. U 50. sekundi ovaj ruter je ponovo startovan. Slika 3-19 prikazuje broj pristupa lukap memorijama, za tri rutera označena na Slici 3-18 sa A, B i C. Ruter A je izabran zato što se nalazi blizu kvara ali ima alternativne putanje do ostalih delova mreže. Ruter B je izabran jer ostaje izolovan u slučaju kvara, dok je ruter C izabran zato što se nalazi daleko od kvara. Može se primetiti da se pristupi memoriji dešavaju na

početku eksperimenta, kada su ruteri startovani, i nakon toga posle 30. i 50. sekunde merenja, kada je označeni ruter bio isključen i ponovo uključen.

Slika 3-19a prikazuje broj pristupa memoriji za slučaj BPFL algoritma. Broj pristupa memoriji je prikazan za tri rutera u simulacionoj mreži. Ruter B je imao veći broj pristupa memoriji posle isključenja rutera u 30. sekundi jer je bio izolovan od ostatka mreže i ponovnog uključenja u 50. sekundi kada se ponovo uspostavila njegova tabela rutiranja. Nakon što je označeni ruter bio isključen i ponovo uključen, ruteri A i C su imali manji broj promena u lukap memoriji zato što su njihove tabele rutiranja imale manji broj izmena. Pri tome, više upisa imao je ruter A koji je imao više izmena u tabeli rutiranja jer se nalazi bliže kvaru.



Slika 3-19 Broj pristupa lukap memoriji za BPFL i POLP

Slika 3-19b prikazuje broj pristupa memoriji za POLP algoritam. Kao i u slučaju BPFL algoritma, ruter B ima najveći broj pristupa memoriji. Može se primetiti da POLP algoritam ažuriranja zahteva veći broj memorijskih pristupa u odnosu na BPFL algoritam. Ovo se može objasniti činjenicom da za male tabele rutiranja nove rute kreiraju veći broj čvorova u strukturi stabla koju koristi POLP. Balansirana stabla korišćena u BPFL algoritmu imaju mali broj čvorova, i njihova obrada zahteva manji broj pristupa memoriji.

Za realne IPv4 i IPv6 tabele rutiranja, izmeren je broj pristupa memoriji i za POLP i za BPFL algoritam, kada su slučajno izabrane rute bile izbrisane. Srednja vrednost, maksimalna vrednost i standardna devijacija broja FPGA pristupa su prikazani u Tabeli 3-1 i Tabeli 3-2, za slučaj IPv4 i IPv6 respektivno.

Tabela 3-1 Broj pristupa lukap memoriji za IPv4

Broj prefiksa		32505	41328	45184	118190	121613	127157	339585	341138	344858
Srednja vrednost	BPFL	2.37	3.26	2.17	2.59	1.50	1.57	3.60	2.81	4.58
	POLP	2.43	2.09	1.72	2.09	1.27	1.37	1.37	1.68	1.68
Maksimum	BPFL	262	451	266	452	114	192	390	513	579
	POLP	32	87	11	67	22	46	18	61	61
Standardna devijacija	BPFL	14.93	25.37	13.98	19.41	6.59	8.89	29.25	27.42	37.76
	POLP	2.76	5.38	1.31	4.76	1.26	2.88	1.95	4.75	4.42

Za slučaj IPv4 tabela rutiranja, možemo primetiti da su srednje vrednosti slične za oba algoritma. Maksimalni broj pristupa je veći za BPFL algoritam, zato što on zahteva veliki broj pristupa memoriji kada se briše poslednja ruta iz podstabla. U tom slučaju, odgovarajući čvor balansiranog stabla treba da bude izbrisan, i balansirano stablo treba da bude preuređeno tako da se održi uslov kompletnosti balansiranog stabla. Takođe, postoji mogućnost da čvor mora biti pomeren u susedno stablo ukoliko ono ima bar dva čvora manje u odnosu na stablo čiji je čvor obrisan. U slučaju IPv6 tabela rutiranja, ažuriranje kod BPFL algoritma zahteva znatno veći broj pristupa memoriji, iz razloga što su stabla „retko” popunjena, pa brisanje ili dodavanje rute dovodi do brisanja ili formiranja novog podstabla.

Tabela 3-2 Broj pristupa lukap memoriji za IPv6

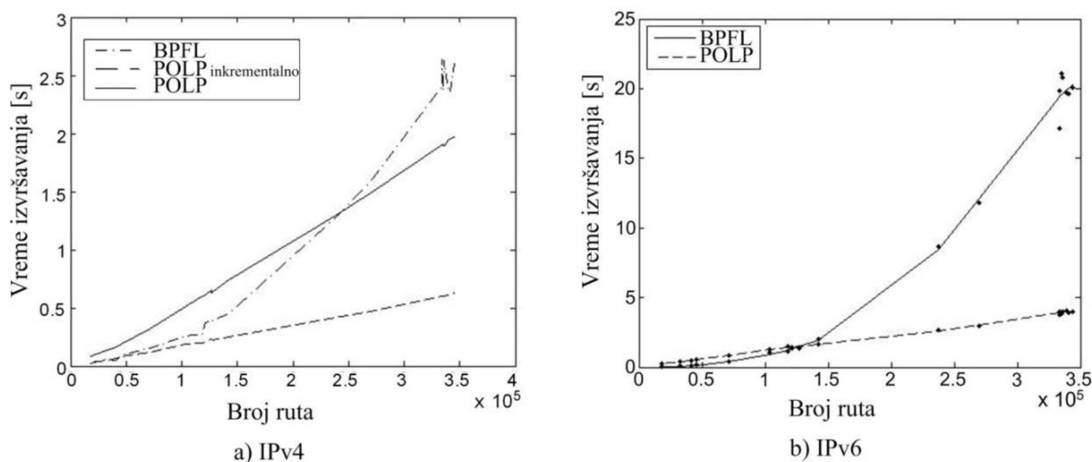
Broj prefiksa		32505	41328	45184	118190	121613	127157	339585	341138	344858
Srednja vrednost	BPFL	72.96	105.32	99.73	250.16	281.74	293.41	537.15	459.41	520.34
	POLP	6.14	13.84	7.29	14.58	13.98	14.11	13.25	13.10	13.42
Maksimum	BPFL	343	429	523	1032	1259	1291	2148	2081	2475
	POLP	30	46	34	46	46	34	56	46	48
Standardna devijacija	BPFL	72.96	96.04	97.60	220.58	257.45	241.29	490.20	457.76	491.14
	POLP	7.59	7.27	8.56	7.89	6.62	6.99	6.46	6.63	6.83

Da bi se precizno procenilo ometanje prosleđivanja paketa usled ažuriranja lukap tabela, potrebno je analizirati rad samog lukap modula. Ukoliko lukap modul ima

blokove koji se paralelno izvršavaju i ukoliko ovi blokovi imaju razdvojene memorije, ažuriranje lukap tabele smeštene u jednoj od ovih memorija ne utiče direktno na lukap u drugoj memoriji. Na primer, pristup memoriji jednog nivoa u BPFL algoritmu, ne ometa lukap process na drugom nivou. Međutim, IP lukap ne može da se završi dok se ne dobije rezultat sa nivoa koji je ometen. Slično, pristup memoriji jednog stepena u POLP lukap modulu će uticati samo na lukap procedure u pajplajnu kome pripada taj stepen. Sa druge strane, pajplajnovi su potpuno nezavisni jedni od drugih.

3.2.6 Vreme izvršavanja

BPFL i POLP agoritami ažuriranja u ovoj sekciji biće upoređeni po vremenu potrebnom za konstrukciju cele lukap tabele kada se rute dodaju iterativno jedna za drugom. Poređenje je izvršeno i za IPv4 i za IPv6 verzije algoritama. Slika 3-20a prikazuje vreme izvršavanja za IPv4, dok Slika 3-20b prikazuje vreme izvršavanja za IPv6. Prikazano vreme uključuje i memorijske pristupe i vreme obrade. Merenja su izvršena za 22 realne tabele rutiranja.



Slika 3-20 Vreme formiranja lukap tabele za IPv4 I IPv6 protokole

Sa Slike 3-20 možemo primetiti da je POLP algoritmu ažuriranja potrebno manje vremena da doda sve rute iz tabele rutiranja. Na Slici 3-20a prikazane su kriva vremena izvršavanja za POLP ažuriranje sa kompletnim proračunom pajplajna pri svakoj promeni tabele rutiranja i kriva vremena izvršavanja sa inkrementalnim proračunom. Pošto se pri inkrementalnom proračunu pri većini izmena tabele rutiranja proračunava samo promena jednobitskog stabla i odgovarajuće promene u pajplajnu, ono se izvršava brže. Kako dodavanje prefiksa u stablo ne zavisi od broja prefiksa već

prisutnih u stablu, kompleksnost dodavanja prefiksa u POLP je $O(1)$, dok je kompleksnost dodavanja n prefiksa $O(n)$. Kompleksnost dodavanja n prefiksa u BPFL je $O(n^2)$, zato što srednji broj čvorova u balansiranim stablima raste linearno sa n, i broj pomeranja čvorova za vreme dodavanja prefiksa linearno raste sa brojem čvorova balansiranog stabla. Iz tog razloga je rekonstrukcija velikih tabela rutiranja znatno sporija u BPFL-u nego u POLP-u.

IPv6 tabele rutiranja sa 300K prefiksa mogu biti konstruisane za 20s (na procesoru skromnih performansi) za BPFL, što je vreme uporedivo sa tipičnim vremenom reakcije protokola rutiranja na promene u mreži. Ometanje u prosleđivanju paketa će trajati znatno kraće, pošto ruteri sa najvećim brojem promenjenih ruta tipično treba da modifikuju samo deo tabele rutiranja u slučaju pada nekog rutera ili linka u mreži.

4 Implementacija za prototip internet rutera

4.1 Implementacija protokola za rutiranje

Protokoli za rutiranje sa balansiranjem opisani u poglavlju 2 obuhvataju sledeće implementacione celine:

- komunikaciju između rutera u cilju razmene informacija potrebnih za proračun koeficijenata balansiranja
- proračun koeficijenata balansiranja u okviru rutera
- usmeravanje paketa u skladu sa proračunatim koeficijentima balansiranja

Prva implementaciona celina podrazumeva razmenu podataka o zahtevanom generisanom i terminisanom saobraćaju rutera. Pošto se implementacija rutiranja sa balansiranjem oslanja na implementaciju OSPF protokola rutiranja, dostupne su informacije o topologiji distribuirane korišćenjem OSPF LSA paketa. Pored informacija o topologiji, za proračun koeficijenata balansiranja potrebne su i informacije o zahtevanim generisanim i terminiranim protocima rutera u mreži, kao i informacije o kapacitetima linkova u mreži. Ove informacije se prenose korišćenjem netransparentnih LSA paketa (eng. *opaque* LSA), koji su OSPF standardom predviđeni za prenos dodatnih informacija. Na osnovu izloženog, implementacija komunikacije između rutera obuhvata proširenje OSPF implementacije uvođenjem netransparentnih LSA paketa koji prenose informacije o zahtevanom generisanom i terminisanom saobraćaju rutera i informacije o brzinama linkova. Ove informacije se prenose u okviru netransparentnog informacionog polja LSA paketa. U okviru prve implementacione celine uključeno je i skladište informacija o ovim protocima koje u okviru softvera rutera sadrži sve prikupljene zahtevane protoke i kapacitete linkova.

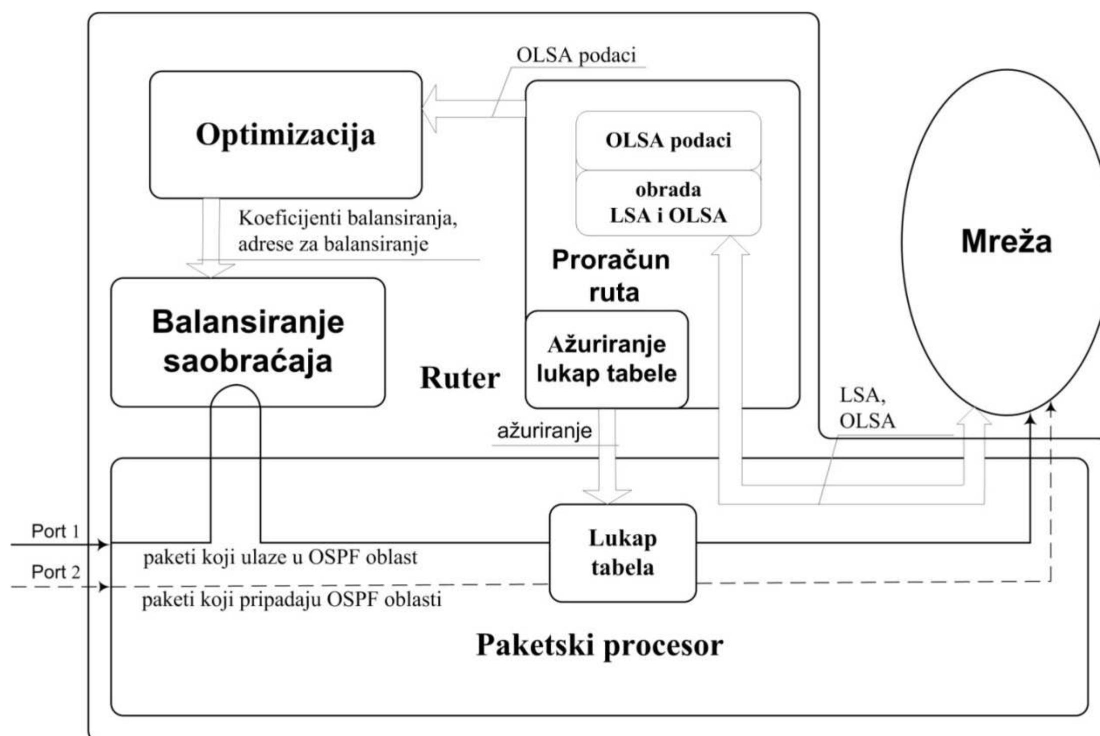
Druga implementaciona celina obuhvata proračun koeficijenata balansiranja na osnovu prikupljenih informacija o topologiji i zahtevanim generisanim i terminiranim saobraćajima rutera. U okviru ove celine potrebno je formirati koeficijente formula jednog od linearnih modela opisanih u odeljcima 2.3.3 i 2.3.4, i rešiti linearni model u cilju dobijanja koeficijenata balansiranja. Proračun koeficijenata formula vrši se na osnovu opisa u pododeljku 4.1.4.3. Pri tome se koriste skladišta informacija o topologiji

i saobraćajnim zahtevima rutera prikupljenih korišćenjem LSA paketa. Samo rešavanje jednačina vrši se metodom linearnog programiranja i u tu svrhu se koristi biblioteka *lpsolve* [53] koja sadrži implementaciju ovog metoda.

Treća implementaciona celina obuhvata implementaciju rutiranja paketa u skladu sa proračunatim koeficijentima balansiranja. U okviru ove implementacione celine nalazi se softverski blok koji na osnovu koeficijenata balansiranja svakom paketu generisanom na ruteru određuje balansirajući ruter. Takođe, ova implementaciona celina sadrži i samo rutiranje paketa preko balansirajućih rutera. Rutiranje sa balansiranjem se oslanja na osnovni protokol rutiranja (OSPF, ECMP i sl.) za usmeravanje paketa između izvorišnog i balansirajućeg, kao i između balansirajućeg i odredišnog rutera, a u okviru ove implementacije, usmeravanje paketa preko balansirajućih rutera implementirano je korišćenjem izvornog rutiranja (eng. *source routing*).

4.1.1 Organizacija implementacije rutiranja sa balansiranjem

Implementacija rutiranja sa balansiranjem urađena je u C++ programskom jeziku i zasnovana je na proširenju OSPF implementacije Džona Moja (eng. *John Moy*) [54]-[56]. Blok šema organizacije softvera prikazana je na Slici 4-1.



Slika 4-1 Blok šema implementacije rutiranja sa balansiranjem

Prva implementaciona celina predstavljena u prethodnom potpoglavlju obuhvata blokove „OLSA podaci” i „obrada LSA i OLSA”. Druga implementaciona celina predstavljena je blokom „Optimizacija”. Treća implementaciona celina obuhvata blok „Balansiranje saobraćaja” i elemente bloka „Paketski procesor” koji se odnose na filtriranje paketa koji ulaze u OSPF oblast i njihovo prosleđivanje u blok za balansiranje saobraćaja. Ukoliko je paket ušao u OSPF oblast u kojoj se nalazi posmatrani ruter kroz neki drugi ruter on se prosleđuje korišćenjem standardnog OSPF protokola.

4.1.2 Organizacija programskog koda *ospfd* implementacije

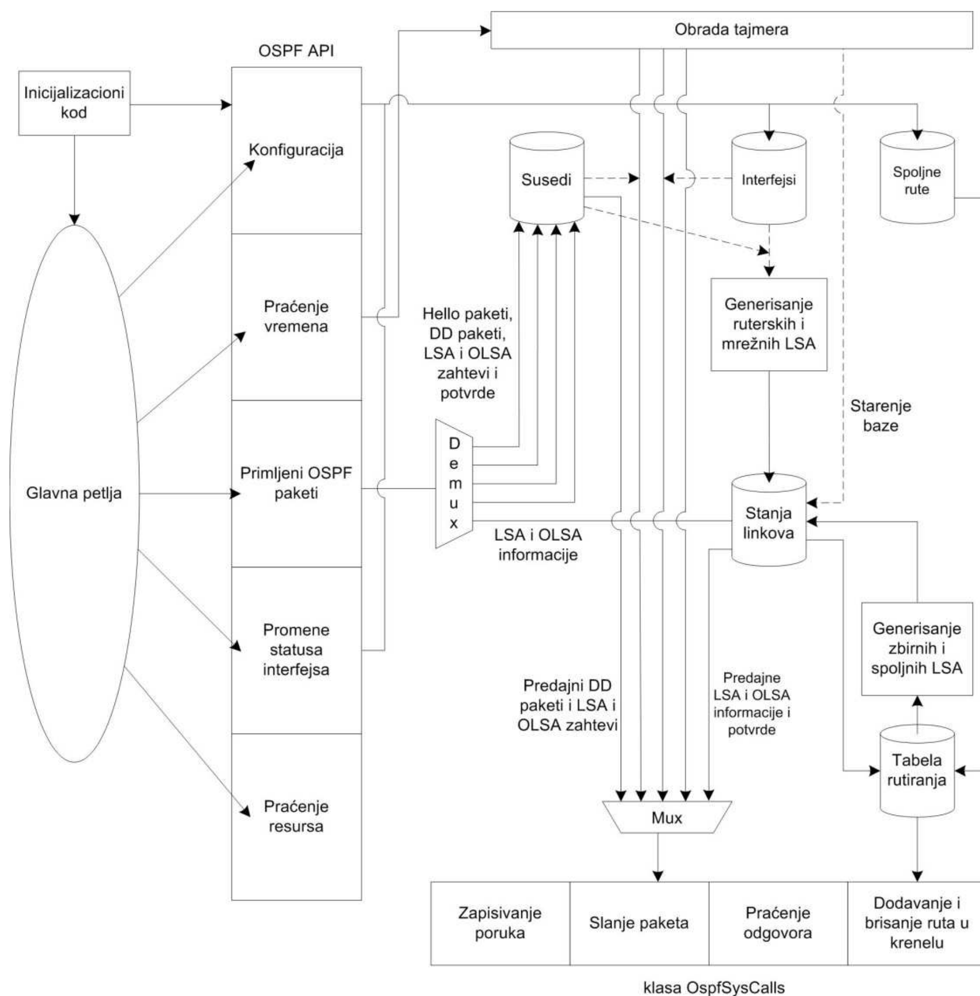
Softverski paket *ospfd* autora Džona Moja obuhvata implementaciju OSPF protokola rutiranja i simulatora mreže OSPF rutera. Implementacija OSPF protokola rutiranja sadržana je u folderima *src* i *linux*. Folder *src* sadrži implementaciju OSPF funkcionalnosti, nezavisnu od korišćenog operativnog sistema, dok folder *linux* sadrži klase koje omogućavaju izvršenje implementacije u okviru *Linux* operativnog sistema.

Knjiga *OSPF Complete Implementation* [54] autora Džona Moja sadrži detaljan opis strukture *ospfd* implementacije. Slika 4-2, preuzeta iz knjige *OSPF Complete Implementation* [54] prikazuje tok podataka u okviru *ospfd* programa.

Osnovna klasa *ospfd* implementacije je *OSPF* klasa, definisana u fajlu ospf.C. Ova klasa upravlja OSPF funkcionalnošću koja obuhvata komunikaciju sa susedima, održavanje LSA baze, i proračun tabele rutiranja. Klasa *OSPF* očitava konfiguraciju portova rutera i OSPF oblasti iz konfiguracionog fajla ospfd.conf. Blokovi inicijalizacije i glavne petlje, prikazani na Slici 4-2 pozivaju funkcije *OSPF* klase.

OSPF oblasti kojima ruter pripada predstavljene su klasom *SpfArea* (fajl SpfArea.C). Portovi konfigurisani za rad sa OSPF protokolom predstavljeni su klasom *SpfIfc* (fajl spfifc.C), u okviru koje su implementirani razmena paketa preko interfejsa i određivanje izabranog rutera u mrežama sa deljenim medijumom (eng. *Designated Router*). Klasa *SpfIfc* omogućava i prijem OSPF *Hello* poruka u mrežama koje omogućavaju brodkast. Za mreže koje ne podržavaju brodkast, *Hello* poruke se šalju korišćenjem klase *SpfNbr* (fajl spfnpr.C), koja služi i za predstavljanje samog suseda i sadrži adresu suseda, stanje povezanosti sa susedom i redni broj suseda. Redni broj suseda nije deo OSPF standarda, i *ospfd* implementacija ga interno koristi za raspoređivanje slanja *Hello* paketa u toku vremena. Slanje *Hello* paketa je primer

periodične operacije u okviru *ospfd* implementacije i za izvršavanje takvih operacija zadužena je klasa *timer* (fajl timer.C). Navedene klase omogućavaju funkcionisanje blokova za prijem i slanje OSPF paketa i bloka tajmera, prikazanih na Slici 4-2. Klasa *OspfSysCalls* prikazana u donjem delu Slike 4-2 predviđena je kao interfejs koji omogućava pozivanje funkcija operativnog sistema kao što je slanje paketa, nezavisno od tipa operativnog sistema. U *ospfd* implementaciji, u klasama *Linux* i *LinuxOspfd*, koje su nasleđene iz klase *OspfSysCalls,* implementirani su pozivi funkcija *Linux* operativnog sistema.



Slika 4-2 Tok podataka u okviru *ospfd* programa

OSPF ruteri međusobno razmenjuju informacije korišćenjem LSA paketa. U okviru *ospfd* implementacije, za predstavljanje svakog tipa LSA paketa koriste se klase *rtrLSA* (fajl rtrlsa.C), *netLSA* (fajl netlsa.C), *summLSA* (fajl summlsa.C), *asbrLSA* (fajl asbrlsa.C), *ASextLSA* (fajl asexlsa.C) i *grpLSA* (fajl grpLSA.C). Navedene klase nasleđene su iz *LSA* klase (fajl lsa.h). U okviru implementacije, LSA svakog tipa se

čuva u okviru svog AVL stabla. AVL stablo [31] je balansirano binarno stablo koje je dobilo ime po njegovim autorima Adisonu, Veskiju i Ladisu. AVL stablo se balansira u okviru operacija dodavanja i brisanja čvorova i ima osobinu da se dubine listova stabla razlikuju najviše za jedan. Svaki objekat klase *SpfArea* koji predstavlja OSPF oblast sadrži AVL stabla sa *LSA* objektima koji predstavljaju LSA pakete generisane u okviru te oblasti. Ova stabla predstavljaju LSA bazu rutera predstavljenu na Slici 4-2 blokom „Stanja linkova". Dodavanje LSA poruka u bazu i njihovo brisanje iz baze implementirano je u fajlu lsdb.C. LSA paketi se prenose kroz mrežu plavljenjem. Slanje i prijem LSA paketa implementirani su u fajlu spfflood.C, dok su slanje i prijem potvrda za LSA paketa implementirani u fajlu spfack.C.

Za proračun tabele rutiranja OSPF koristi informacije iz LSA, i proračun je potrebno izvršiti po promeni ovih informacija, odnosno po pristizanju promenjenog LSA paketa. Promenu LSA paketa detektuje metod *cmp_contents* klase *LSA*, i na osnovu promene koja se desila metod *rtsched* klase *OSPF* određuje koji proračun treba da bude izvršen. U slučaju promene LSA ruterskog ili mrežnog tipa pokreće se Dijkstra algoritam za proračun najkraćih putanja implementiran u fajlu spfcalc.C, koji ažurira tabelu rutiranja prikazanu na Slici 4-2.

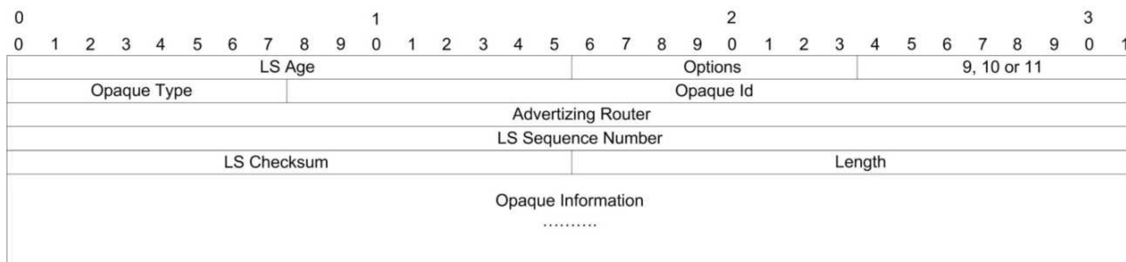
Implementacija simulatora sadržana je u folderu *ospf_sim*. Simulator se oslanja na implementaciju OSPF protokola i implementira funkcionalnost povezivanja rutera u mrežu na osnovu konfiguracionog fajla i grafičkog interfejsa koji omogućava formiranje mreže, izvršavanja simulacije i prikaza stanja rutera u toku simulacije.

4.1.3 Razmena informacija između rutera

Proračun koeficijenata balansiranja, pored informacija o topologiji koje se standardno prikupljaju u okviru OSPF protokola, zahteva i informacije o željenim terminiranim i generisanim saobraćajima rutera u mreži i informacije o kapacitetima linkova. Ove informacije se u okviru implementacije prenose korišćenjem netransparentnih LSA (eng. *opaque* LSA) paketa, koji su OSPF standardom predviđeni za prenos dodatnih informacija pri nadogradnji OSPF protokola.

Na Slici 4-3 prikazan je format netransparentnih LSA paketa [57]. Polja specifična za netransparetne LSA su tip netranspoarentnog LSA paketa (*Opaque Type*), identifikator netransparentnog LSA paketa (*Opaque Id*) i informacije prenošene u

netransparentnom LSA paketu (*Opaque Information*). Ostala polja su ista kao u ostalim tipovima LSA paketa [58].



Slika 4-3 Format netransparentnih LSA

Polje za tip netransparentog LSA paketa zauzima 8 bita i može imati vrednosti 0-255. Vrednosti tipova netransparentnih LSA 1-127 su predviđene za dodeljivanje u toku standardizacije, dok su vrednosti 128-255 predviđene za eksperimentalnu i privatnu upotrebu. U okviru implementacije je korišćena vrednost tipa 200 za LSA koji nose informacije o zahtevanim generisanim i terminiranim saobraćajima rutera, dok je za prenos informacija o brzinama linkova korišćena vrednost tipa 201. U okviru netransparentnih LSA korišćenih u implementaciji, polje identifikatora netransparetnog paketa ima vrednost 0.

Informacije prenošene u netransparentnim LSA u okviru implementacije organizovane su u dve strukture programskog jezika C++. Informacije o zahtevanom terminiranom i generisanom saobraćaju rutera sadržane su u strukturi *NodeTraffic*.

```
struct NodeTraffic {
        guint64 intraffic;
        guint64 outtraffic;
        struct in_addr advrouter;
        NodeSpeed *next;
};
```

Polje *intraffic* nosi informaciju o zahtevanom terminiranom saobraćaju rutera. Ovo polje se sastoji od 64 bita. Polje *outtraffic* nosi informaciju o zahtevanom generisanom saobraćaju rutera, i takođe se sastoji od 64 bita. Polje *advrouter* sadrži IP adresu rutera koji je generisao strukturu. Pokazivač *next* je predviđen za formiranje ulančane liste struktura *NodeTraffic* i pri prenosu se njegova vrednost postavlja na nulu. Navedena struktura se prenosi u okviru netransparentnih LSA sa tipom 200.

Struktura koja se koristi za prenos informacija o kapacitetima linkova nazvana je *LinkSpeed*.

```
struct LinkSpeed {
        guint64 speed;
        struct in_addr advrouterA;
        struct in_addr advrouterB;
        LinkSpeed *next;
};
```

Polje *speed* veličine 64 bita sadrži brzinu linka, dok adrese *advrouterA* i *advrouterB* predstavljaju adrese početnog i odredišnog rutera linka, definišući na taj način smer linka od rutera A ka ruteru B. Odvojeno definisanje brzine linka u svakom smeru omogućava rad i sa asimetričnim linkovima. Pokazivač *next* se koristi pri grupisanju struktura tipa *LinkSpeed* u ulančanu listu i pri prenosu u okviru LSA njegova vrednost se postavlja na nulu. Strukture *NodeTraffic* i *LinkSpeed* definisane su u fajlu lsaopqsend.h.

U implementaciji se koriste postojeće funkcije *ospfd* softvera, koje omogućavaju slanje i prijem netransparentnih LSA paketa. Za slanje se koristi *adv_area_opq* funkcija implementirana u okviru *OSPF* klase (fajl opqlsa.C). Funkciji *adv_area_opq* predaje se identifikator oblasti u koju treba poslati netransparentni LSA. Drugi parametar sadrži vrednosti tipa i identifikatora netransparentnog LSA u jednoj celobrojnoj vrednosti, čineći tako drugi red formata prikazanog na Slici 4-3. Treći parametar je pokazivač na memorijsku lokaciju od koje počinje sadržaj koga treba poslati, četvrti parametar je dužina sadržaja koga treba poslati, i peti parametar je promeljiva kojom se kaže da li treba obaviti slanje. Funkcija *adv_area_opq* je deo originalne *ospfd* implementacije.

```
bool OSPF::adv_area_opq(aid_t a_id, lsid_t lsid, byte *body, int blen,bool adv);
```

U našoj implementaciji se za slanje netransparentih LSA koriste sledeće dve funkcije:

```
bool sendlsa_traffic(OSPF *ospfc, NodeTraffic *ns) {
        return (ospfc->adv_area_opq(0, 200<<24, (byte*)ns, sizeof(NodeTraffic), true));
}
```

```
bool sendlsa_link(OSPF *ospfc,LinkSpeed *capacity,int brojl){
        return (ospfc->adv_area_opq(0, 201<<24, (byte*)capacity,
        brojl*sizeof(LinkSpeed), true));
}
```

Navedene funkcije omogućavaju slanje netransparentnih LSA samo na osnovu pokazivača na objekat *ospf* koji je deo *ospfd* softvera i pokazivača na strukturu koju treba preneti u okviru netransparentnog LSA. U slučaju slanja brzina linkova, dodatni parametar je broj linkova, na osnovu koga se određuje veličina memorije koju zauzimaju strukture sa brzinama svih linkova rutera.

Skladište informacija o netransparentnim LSA implementirano je klasama *TrafficList* (fajl lsaopbase.C) i *LinksList* (fajl lsaspeeds.C). Ove dve klase, u skladu sa Slikom 4-1, sa jedne strane omogućavaju prihvat informacija iz pristiglih netransparentnih LSA, dok sa druge strane omogućavaju očitavanje ovih informacija od strane bloka koji proračunava optimalne koeficijente balansiranja.

```
class TrafficList {
        NodeTraffic* head;
        Nodes* NodeL;
public:
        void additem(NodeTraffic * nd);
        void removeitem(NodeTraffic * nd);
        void printlist();
        void setlist(Nodovi* nodelist);
        guint64 getin(int i);
        guint64 getout(int j);
        TrafficList ();
        ~ TrafficList ();
};
```

U okviru klase *TrafficList*, pokazivač *head* pokazuje na prvi element ulančane liste struktura tipa *NodeTraffic*. Ova ulančana lista sadrži sve primljene informacije o zahtevanim generisanim i terminiranim saobraćajima rutera. Pokazivač *NodeL* pokazuje na ulančanu listu struktura tipa *Nodes* (fajl sprtraffic.C) koja se koristi pri izvršavanju funkcija *getin* i *getout* čiji su parametri identifikatori elemenata te liste. Za dodavanje i

brisanje elemenata u listi *head* koriste se funkcije *additem* i *removeitem*. Blok za optimizaciju sa druge strane koristi funkcije *getin* i *getout* da za zadati identifikator rutera u listi *Nodes* dobije njegov zahtevani generisani i terminirani saobraćaj.

```
class LinksList
{
        LinkSpeed* head;
        Nodes* NodeL;
public:
        void additem(LinkSpeed* nd);
        void removeitem(LinkSpeed* nd);
        void printlist();
        void setlist(Nodovi* nodelist);
        guint64 get(int i, int j);
        LinksList();
        ~LinksList();
};
```

Klasa *LinksList* sadrži pokazivač *head* koji pokazuje na ulančanu listu struktura tipa *LinkSpeed*. Ove strukture čuvaju sve informacije o brzinama linkova u mreži primljene posredstvom netransparentnih LSA. Nove strukture se dodaju u listu funkcijom *additem*, dok se brišu iz liste funkcijom *removeitem*. Funkcija *get* omogućava očitavanje brzine linka između čvorova sa indentifikatorima i i j. Lista *NodeL* sadrži spisak rutera sa njihovim identifikatorima, i omogućava pronalaženje strukture tipa *LinkSpeed* u listi *head* na osnovu indentifikatora koji su parametri funkcije *get*.

4.1.4 Optimizacioni modul

Optimizacioni modul određuje optimalne koeficijente balansiranja u cilju omogućavanja maksimalnog protoka kroz mrežu. Ovaj modul dobija od proširenog OSPF modula informacije neophodne za proračun koeficijenata balansiranja u skladu sa linearnim modelima opisanim u odeljcima 2.3.3 i 2.3.4.

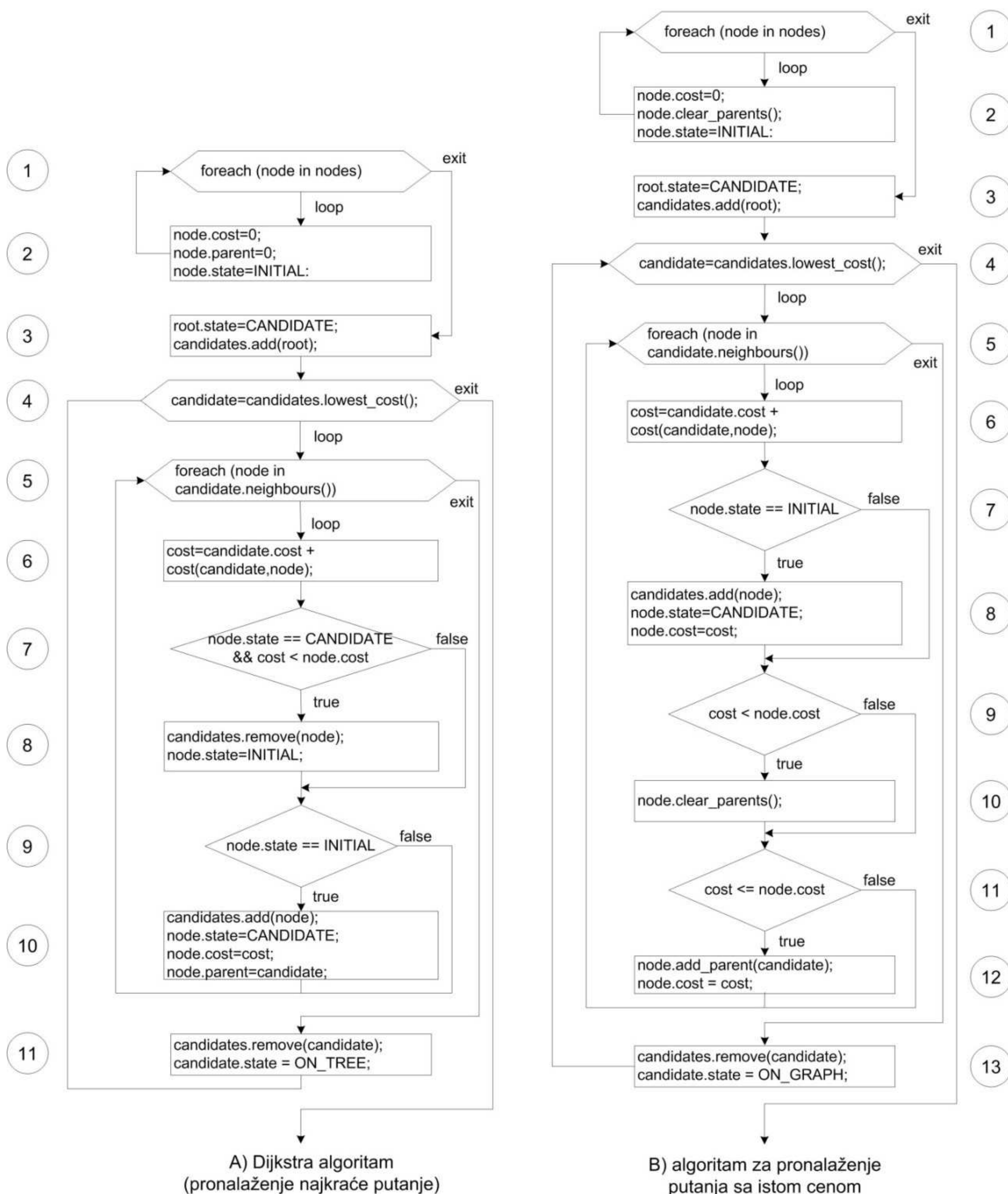
Prvi korak pri generisanju formula linearnog modela je dobijanje koeficijenata opterećenosti linkova F_{ij}^{k}. Koeficijent F_{ij}^{k} označava zauzetost linka k, pri komunikaciji između čvorova i i j, pri čemu se posmatraju jednosmerni linkovi. Za svaki link se formira matrica koja sadrži koeficijente za taj link. Da bi bilo moguće dobiti matrice

koeficijenata, potrebno je proračunati putanje saobraćaja pri komunikaciji bilo koja dva čvora u mreži. Za rutiranje u mreži je u implementaciji moguće koristiti rutiranje najkraćim putem (eng. SPR – *Shortest Path Routing*), ili rutiranje putanjama sa istom cenom (eng. ECMP – *Equal-Cost MultiPath*). Dobijanje putanja za rutiranje najkraćim putem moguće je korišćenjem Dijkstra algoritma.

4.1.4.1 Proračun putanja paketa kroz mrežu

Slika 4-4 A prikazuje Dijkstra algoritam, dok Slika 4-4 B prikazuje algoritam za pronalaženje putanja sa istom cenom. Prikazani Dijkstra algoritam deo je *ospfd* programa [56], dok je algoritam za pronalaženje putanja sa istom cenom realizovan u okviru ove implementacije. Rezultat Dijkstra algoritma je struktura stabla u kome je koren početni ruter, i čije grane označavaju najkraću putanju od bilo kog čvora stabla ka korenu stabla. Čvorovi stabla predstavljaju rutere ili LAN mreže na koje može biti povezano više rutera. Rezultat algoritma za pronalaženje putanja sa istom cenom je u opštem slučaju graf, u kome putanje od proizvoljno izabranog čvora grafa do početnog čvora predstavljaju putanje sa jednakom cenom. Koreni stabala, odnosno početni čvorovi grafova, su ruteri od kojih počinje građenje stabla, odnosno grafa. U daljem tekstu će se u objašnjenjima koja se odnose na oba algoritma koristiti termin graf i za stablo prvog algoritma i za graf drugog algoritma.

Uloga Dijkstra algoritma u *ospfd* programu je proračun izlaznih portova za tabelu rutiranja, i ta funkcionalnost je u okviru originalnog *ospfd* programa implementirana u fajlu spfcalc.C. Funkcionalnost proračuna najkraćih putanja u cilju proračuna opterećenosti linkova izdvojena je u fajl sprtraffic.C, u okviru koga je implementiran Dijkstra algoritam prikazan na Slici 4-4 A. Dijkstra algoritam je implementiran u klasi *Nodes*. Algoritam za pronalaženje putanja sa istom cenom implementiran je u fajlu ecmptraffic.C, u okviru klase *NodesMultipath*. U oba algoritma, koraci 1-3 predstavljaju inicijalizaciju, u kojoj se u svim objektima čvorova cene putanje ka korenu stabla postavljaju na nulu, postavlja se da čvorovi nemaju roditelje i da imaju inicijalno početno stanje. Pored inicijalnog početnog stanja, moguća su i stanje pripadanja skupu kandidata za pridruživanje grafu, i stanje pripadanja grafu. U trećem koraku inicijalizacije, čvor koji predstavlja koren grafa, pridružuje se skupu kandidata za pridruživanje grafu.



Slika 4-4 Algoritmi za proračun stabla rutiranja najkraćim putanjama i putanjama sa jednakim cenama

Opisana inicijalizacija se u oba slučaja vrši u funkcijama koje se zovu *dijk_init*, jedna u klasi *Nodes*, a druga u klasi *NodesMultipath*. Preostali deo oba algoritma je implementiran u funkcijama pod nazivom *dijkstra*, takođe pridruženim klasama *Nodes* i *NodesMultipath*. Ove funkcije imaju ista imena zato što se algoritam za proračun putanja sa istom cenom zasniva na Dijkstra algoritmu.

Po završenoj inicijalizaciji, svaki od algoritama ulazi u dvostruku petlju. Uslov spoljne petlje označen je brojem 4, i tim uslovom se iz skupa kandidata uzima kandidat sa najmanjom cenom i pridružuje se identifikatoru *candidate*. Petlja se izvršava dok skup kandidata ima bar jedan element. Unutrašnja petlja opisana je blokom 5 i ona prolazi kroz sve susede izabranog kandidata. Prvi korak unutrašnje petlje u oba algoritma, označen brojem 6, računa cenu od suseda u tekućoj iteraciji petlje, preko izabranog kandidata, do korena grafa.

U slučaju Dijkstra algoritma, u unutrašnjoj petlji vrše se operacije sa čvorovima-kandidatima i čvorovima u inicijalnom stanju. U slučaju da je čvor kandidat i da je njegova cena do korena stabla veća od cene preko tekućeg kandidata izabranog u spoljnoj petlji, čvor se briše iz skupa kandidata, i potom se ponovo dodaje skupu kandidata. Pri ponovnom dodavanju u skup kandidata, roditelj čvora postaje tekući kandidat izabran u spoljnoj petlji, a cena do korena postaje cena preko kandidata izabranog u spoljnoj petlji. U drugom slučaju, kada je čvor u inicijalnom stanju on se dodaje u skup kandidata, i povezuje se na kandidata izabranog u spoljnoj petlji. Pošto su svi susedi kandidata izabranog u spoljnoj petlji obrađeni, ovaj kandidat briše se iz skupa kandidata i stavlja se na stablo. Pošto svi čvorovi pređu iz skupa kandidata na stablo, algoritam formiranja stabla je završen.

U slučaju algoritma za pronalaženje putanja sa istom cenom, posle proračuna cene tekućeg čvora do korena grafa preko izabranog kandidata, prvo se proverava da li je tekući čvor u inicijalnom stanju. Ako je tekući čvor u inicijalnom stanju, prvo se vrši njegovo dodavanje u skup kandidata. Potom se vrši obrada koja ne zavisi od stanja čvora. Prvi korak ove obrade je provera da li je nova cena manja od cene čvora i ako jeste, briše se skup roditelja čvora. U sledećem koraku se proverava da li je nova cena manja ili jednaka od cene čvora i ako jeste, tekući kandidat se dodaje u skup roditelja čvora, i čvoru se dodeljuje cena preko tog kandidata. Navedenim operacijama u unutrašnjoj petlji se postiže da se u skupu roditelja čvora nađu svi čvorovi preko kojih je do korena grafa moguće doći po minimalnoj ceni. Po završenoj obradi svih suseda kandidata izabranog u spoljnoj petlji, ovaj kandidat se premešta iz skupa kandidata u graf, i počinje se sa obradom sledećeg kandidata sa najmanjom cenom iz skupa kandidata. Kada se svi čvorovi pridruže grafu algoritam formiranja grafa je završen.

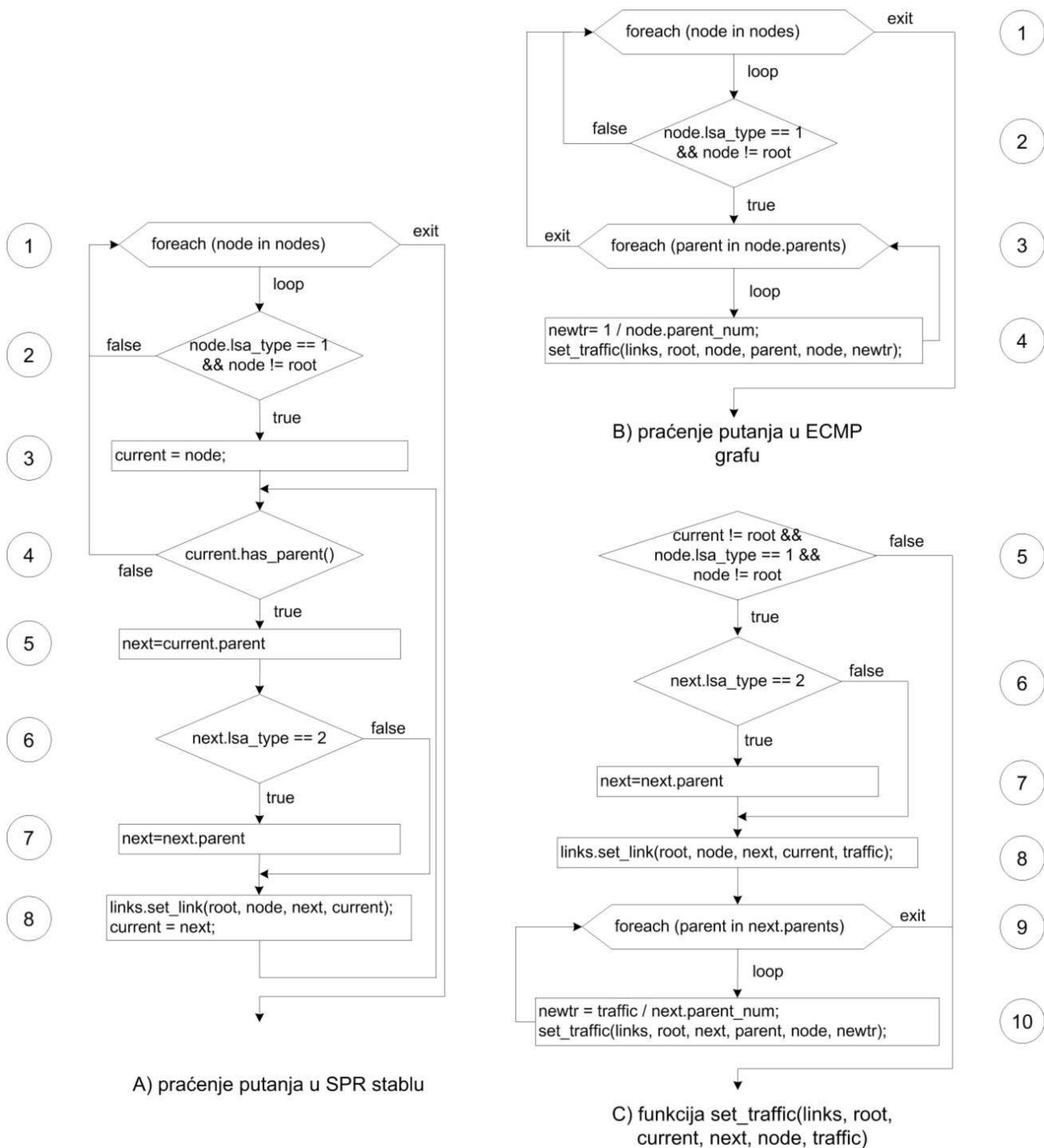
SPRm algoritam predstavlja modifikaciju Dijkstra algoritma za pronalaženje najkraćih putanja u kojoj se pri dodavanju čvora na stablo od svih kandidata sa jednakom cenom bira onaj čiji roditelj ima najmanje čvorova-dece. Izmenom algoritma prikazanog na Slici 4-4 A u koraku 4, kojom bi se iz objekta *candidates* uzimao kandidat čiji roditelj ima najmanje čvorova-dece omogućio bi se proračun putanja koje odgovaraju SPRm algoritmu.

4.1.4.2 Proračun opterećenosti linkova

U cilju dobijanja koeficijenata opterećenosti linkova F_{ij}^k potrebnih za formiranje optimizacionih formula, potrebno je za svaki link *k* proračunati saobraćaj koga na njemu generiše komunikacija između čvorova i i j. U prethodnom pododeljku je opisan proračun grafa putanja od svakog čvora j do čvora i koji predstavlja početni čvor za graf. Prolaskom kroz graf od čvora j do čvora i, za svaki link preko koga se pređe moguće je izračunati procenat saobraćaja između čvorova i i j koji prolazi kroz taj link. U slučaju rutiranja po najkraćoj putanji, kroz link preko koga se pređe na putu od i ka j prolazi sav saobraćaj, dok u slučaju rutiranja po putanjama sa jednakim cenama zavisi od broja grananja i spajanja putanja sa jednakim cenama pre tog linka. Kada do linka dođemo posle g grananja, tada saobraćaju tog linka treba dodati $1/\prod_{l=1}^{g} p_l$ saobraćaja koji se prenosi od i ka *j*, pri čemu je p_l broj putanja sa jednakom cenom pri l-tom grananju.

Algoritam na Slici 4-5 A implementiran je u fajlu sprtraffic.C, u funkciji *setcoefficients* koja pripada klasi *Nodes*. Algoritam prikazan na Slici 4-5 B implementiran je u fajlu ecmptraffic.C, u funkciji koja se takođe naziva *setcoefficients* i pripada klasi *NodesMultipath*. Funkcija *set_traffic* takođe je implementirana u okviru klase *NodesMultipath* u fajlu ecmptraffic.C.

Algoritam prikazan na Slici 4-5 A omogućava proračun opterećenosti linkova u slučaju stabla najkraćih putanja, dok algoritam predstavljen na Slikama 4-5 B i C služi za proračun opterećenosti linkova pri rutiranju po putanjama sa jednakom cenom. Ovi algoritmi se baziraju na prolasku kroz graf putanja, od svakog čvora ka početnom čvoru grafa. Koeficijent F_{ij}^k ima vrednost između 0 i 1, i predstavlja deo saobraćaja koji pri komunikaciji od čvora *i* ka čvoru *j* prolazi kroz link *k*.



Slika 4-5 Algoritmi za proračun opterećenosti linkova i nalaženje adresa balansiranja

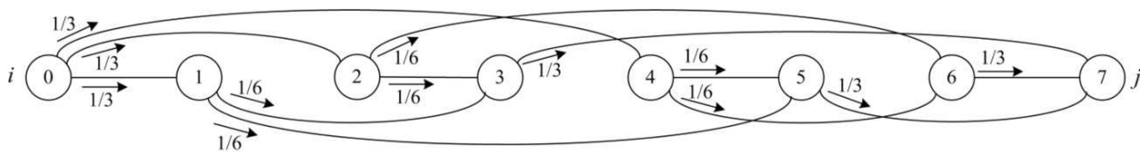
Pri prolazu kroz stablo najkraćih putanja, saobraćaj se ni u jednom čvoru ne deli, pa pri prelasku kroz granu koja odgovara linku k, koeficijentu F_{ij}^{k} se dodeljuje vrednost 1. Pri prolasku kroz stablo, sa svakog čvora u stablu se prelazi ka njegovom čvoru-roditelju, dok se ne stigne do korena stabla.

Na Slici 4-5 A, algoritam se sastoji iz dvostruke petlje. Spoljna petlja, čiji je uslov definisan u koraku 1 na Slici 4-5 A prolazi kroz sve čvorove stabla najkraćih putanja. Ova petlja prolazi kroz polazne rutere za prolaze kroz stablo. U koraku 2 se za

svaki čvor stabla proverava da li čvor predstavlja ruter i da li čvor nije koren stabla. Ako su ova dva uslova ispunjena, prelazi se na unutrašnju petlju koja predstavlja prolaz kroz stablo od čvora izabranog u spoljnoj petlji do korena stabla. U koraku 3 se vrednost promenljive *current* inicijalizuje vrednošću čvora od koga prolaz kroz stablo počinje. Potom se ulazi u unutrašnju petlju čiji je uslov predstavljen korakom 4. U okviru unutrašnje petlje, promenljiva *current* predstavlja čvor do koga se stiglo pri prolazu kroz stablo. U koraku 4 vrši se provera da li tekući čvor ima roditelja. Pošto je koren stabla jedini čvor u stablu koji nema roditelja, ova provera ekvivalentna je proveri da li je tekući čvor koren stabla. Ako tekući čvor nema roditelja, odnosno predstavlja koren stabla, prolazak kroz stablo je završen. Ako tekući čvor ima roditelja, potrebno je utvrditi koji je sledeći ruter na putu ka korenu stabla. U slučaju da je tip LSA koji odgovara sledećem čvoru stabla jednak 1, odnosno da je u pitanju ruterski LSA, tada je sledeći ruter roditelj tekućeg rutera. Promenljiva *next*, koja predstavlja sledeći čvor, u koreku 5 postavlja se na vrednost roditelja tekućeg čvora. U koraku 6 se proverava da li čvor sadrži LSA tipa 2, odnosno da li čvor odgovara mrežnom LSA. U slučaju da čvor odgovara mrežnom LSA, tada se u koraku 7 prelazi na roditelja mrežnog LSA u stablu, i taj roditelj predstavlja sledeći ruter na putu ka korenu stabla. Tipovi LSA čvorova mogu biti samo ruterski i mrežni, pa dalje provere tipa LSA nisu potrebne. Kada su poznati tekući i sledeći ruter, u koraku 8 se poziva funkcija *set_link* koja koeficijentu F_{ij}^{k} za link između tekućeg i sledećeg rutera dodeljuje vrednost 1 i promeljiva koja predstavlja tekući ruter dobija vrednost sledećeg rutera čime se završava iteracija unutrašnje petlje.

Funkcija *set_link* u okviru algoritma na Slici 4-5 A implementirana je kao deo klase *Links* u fajlu Links.C. Uloga klase *Links* je da čuva koeficijente opterećenosti linkova F_{ij}^{k}. Slično algoritmu za proračun opterećenosti linkova pri rutiranju najkraćom putanjom, algoritam za proračun opterećenosti linkova pri rutiranju putanjama sa jednakom cenom ima svoju verziju funkcije *set_link*, implementiranu u fajlu LinksMultipath.C, u okviru klase *LinksMultipath*. Klasa *LinksMultipath* čuva koeficijente opterećenosti linkova za rutiranje sa jednakim cenama, i pri prolasku kroz graf putanja sa jednakim cenama opterećenje je moguće dodati korišćenjem funkcije *set_link.*

Pri rutiranju po putanjama sa jednakim cenama, algoritam za prolazak kroz graf prikazan na Slikama 4-5 B i C, uzima u obzir da saobraćaj može biti podeljen između dva ili više linkova preko kojih se stiže od izvorišnog čvora do destinacije po putanjama jednakih. Primer raspodele saobraćaja između čvora 0 i čvora 7 u *flattened butterfly* topologiji prikazan je na Slici 4-6. Od čvora 0 saobraćaj polazi preko tri linka, ka čvorovima 1, 2 i 4. Ovaj saobraćaj se dalje deli po linkovima, pa kasnije koristi zajedničke linkove ka čvoru 7. Na primer, saobraćaj čvora 1 se deli na linkove prema čvorovima 3 i 5, dok u čvor 3 pristiže saobraćaj iz čvorova 1 i 2, i koristi zajednički link ka čvoru 7. Vrednosti opterećenosti prikazane uz linkove predstavljaju vrednosti koeficijenata $F_{0,6}^{k}$. Vrednosti koeficijenata $F_{0,6}^{k}$ koji nisu prikazani na slici jednake su nuli.



Slika 4-6 Opterećenosti linkova pri rutiranju sa jednakim cenama (ECMP)

Prilikom određivanja vrednosti koeficijenata opterećenosti linka F_{ij}^{k} čiji su algoritmi prikazani na Slikama 4-5 B i C, krećemo se po grafu od odredišta do izvorišta. Pošto je moguć slučaj velikog broja uzastopnih deljenja saobraćaja, algoritam prolaska kroz graf putanja sa jednakim cenama realizovan je korišćenjem rekurzivne funkcije *set_traffic*, koja se poziva za svaki link na koga se naiđe pri prolasku kroz graf. U slučaju nailaska na čvor od koga postoji više linkova sa istom cenom ka početnom čvoru, funkcija *set_traffic* poziva se za svaki od linkova, pri čemu se u poziv prosleđuje i veličina dela saobraćaja koji je pristigao do čvora, realni broj vrednosti od 0 do 1. Pošto se poziva za svaki link po kome prolazi saobraćaj pri komunikaciji između čvorova i i j pri rutiranju po putanjama sa jednakim cenama, funkcija *set_traffic* za svaki od tih linkova poziva funkciju *set_link* koja proračunava vrednost koeficijenta opterećenosti linka F_{ij}^{k} na osnovu vrednosti koja odgovara delu saobraćaja koji prolazi kroz link. U slučaju rutiranja po najkraćim putanjama nije bilo potrebno koristiti funkciju *set_traffic* zato što je u svakom koraku putanja prelazila na jedan sledeći link, dok je u slučaju rutiranja po putanjama sa jednakim cenama u svakom koraku moguće

da putanja pređe na više sledećih linkova. Pri rutiranju po putanjama sa jednakim cenama, funkcija *set_traffic* omogućava prelazak na više sledećih linkova proizvoljan broj puta, zato što se ta funkcija poziva za sve linkove na koje se saobraćaj deli, i u sledećem koraku se opet saobraćaj svakog od tih linkova može podeliti, i funkcija *set_traffic* se opet na isti način poziva za sve te linkove.

Algoritam prikazan na Slici 4-5 B petljom čiji je uslov predstavljen korakom 1 prolazi kroz sve čvorove grafa putanja sa jednakim cenama. Ukoliko čvor predstavlja ruter i različit je od korena stabla (korak 2), taj čvor postaje polazni čvor za prolaz kroz graf pa se ulazi u unutrašnju petlju koja prolazi kroz sve roditelje tekućeg rutera u grafu putanja sa jednakim cenama (korak 3). Pošto se u slučaju postojanja više roditelja saobraćaj od polaznog čvora grafa ravnomerno šalje preko svakog od roditelja, u koraku 4 se proračunava vrednost saobraćaja koja se šalje od roditelja ka posmatranom čvoru i za svakog roditelja se poziva rekurzivna funkcija *set_traffic* koja izračunava koeficijent za link od roditelja i poziva se rekurzivno za linkove ka tom roditelju od njegovih roditelja. Funkcija *set_traffic* prikazana je na Slici 4-5 C. Na početku algoritma, u koraku 5, se proverava da li je prolaz stigao do početnog čvora grafa. Ako nije, u koraku 6 se proverava da li čvor-roditelj sadrži mrežni LSA i, ako sadrži, u koraku 7 se vrednost sledećeg rutera dobija kao roditelj tog čvora u grafu. Potom se u koraku 8 poziva funkcija *set_link* koja koeficijentu F_{ij}^{k} dodeljuje vrednost proračunatog dela saobraćaja koji se kroz link *k* prenosi između tekućeg i sledećeg čvora, predstavljenih promenljivim *current* i *next*. Deo saobraćaja koji prolazi kroz link *k* sadržan je u promenljivoj *traffic*, koja se dobija deljenjem saobraćaja koji ulazi u tekući čvor na onoliko jednakih delova koliko tekući čvor ima roditelja. Promenjiva *traffic* predstavlja koeficijent opterećenosti linka F_{ij}^{k}. Potom se petljom čiju je uslov predstavljen u koraku 9 prolazi kroz sve roditelje čvora, i u koraku 10 se proračunava deo saobraćaja koji se šalje ka roditeljima, predstavljen promenljivom *newtr*. Za svakog roditelja se rekurzivno poziva funkcija *set_traffic*, pri čemu parametar *traffic* ima vrednost *newtr*.

U slučaju kada prolaz kroz graf od čvora *i* ka čvoru *j* ne prolazi preko linka *k* vrednost koeficijenta F_{ij}^{k} jednaka je nuli.

Funkcija *set_link* održava objekat koji sadrži vrednosti koeficijenata F_{ij}^{k}. Ova funkcija se u toku izvršavanja algoritama prikazanih na Slici 4-5 poziva pri svakom

prolasku kroz link i ažurira vrednost koeficijenta F_{ij}^{k} koji odgovara krajnjim čvorovima veze i linku za koji je pozvana. Pre izvršenja algoritama prikazanih na Slici 4-5 vrednosti svih koeficijenata postavljaju se na nulu. U slučaju rutiranja najkraćom putanjom poziv funkcije *set_link* postavlja vrednost koeficijenta na jedan. Pri rutiranju sa jednakim cenama moguće je da putanje koje su se u nekom čvoru podelile prelaze preko istih linkova i posle deobe, pa se za te linkove opterećenje dobija tako što se sabiraju delovi saobraćaja za svaki od prolaza. Iz tog razloga je bitno da se čuvaju tekuće vrednosti koeficijenata F_{ij}^{k}.

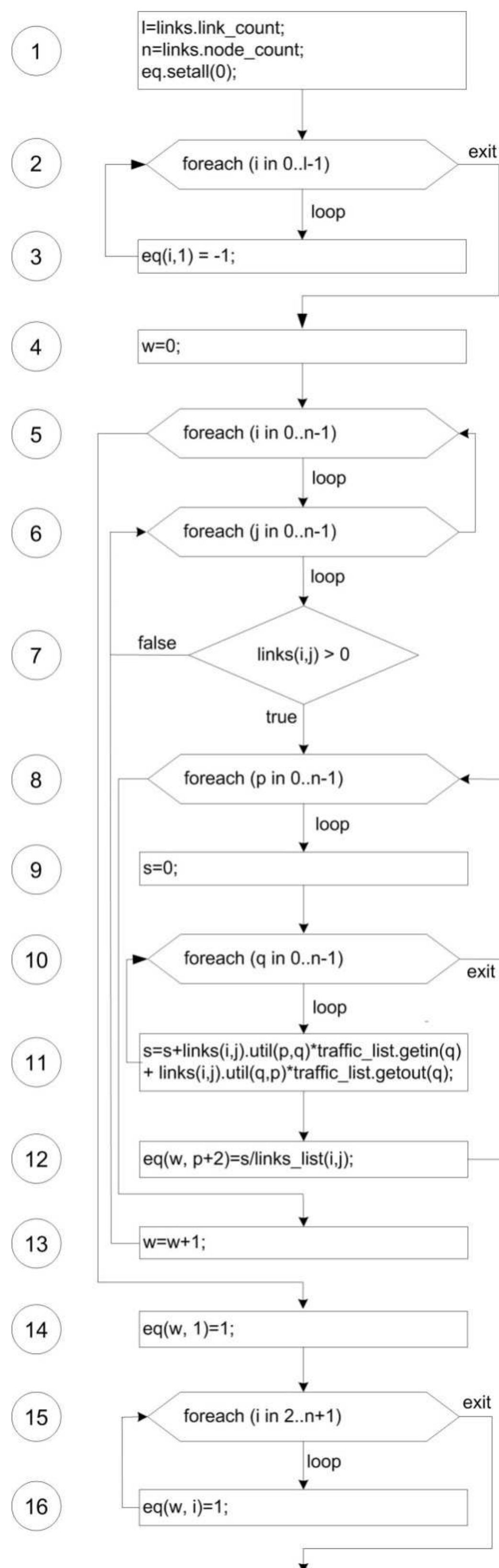
Funkcija *set_link* ima i ulogu generisanja adresa za balansiranje korišćenih u sviču koji odgovara početnom čvoru grafa. Ove adrese se dobijaju kao adrese izlaznih portova svičeva koji pripadaju najkraćim putanjama od svakog sviča mreže do sviča predstavljenog početnim čvorom grafa. Korišćenje ovih adresa omogućava da paket pri slanju iz izvorišnog sviča ka balansirajućem sviču koristi put koji je predviđen algoritmom i koji se koristi i pri prolasku kroz graf u cilju formiranja koeficijenata F_{ij}^{k} potrebnih za generisanje formula optimizacionog modela. Korišćenje neke druge adrese balansirajućeg sviča moglo bi rezultovati izmenjenom putanjom paketa, koja nije u skladu sa postavkom optimizacionog modela, pa ni raspodela saobraćaja u mreži ne bi odgovarala rezultatima optimizacionog proračuna.

4.1.4.3 Formiranje i rešavanje optimizacionog modela

Pošto su izračunati koeficijenti F_{ij}^{k} (pododeljak 4.1.4.2), i prikupljeni podaci o zahtevanim generisanim i terminiranim saobraćajima svičeva u strukturi *TrafficList* (odeljak 4.1.3) i podaci o brzinama linkova u strukturi *LinksList* (odeljak 4.1.3), moguće je formirati formule optimizacionih modela opisanih u odeljcima 2.3.3 i 2.3.4.

Algoritam na Slici 4-7 proračunava formule optimizacionog modela, koje obuhvataju po jednu nejednačinu za svaki link u mreži i jednačinu koja definiše da je skup koeficijenata balansiranja jednak jedan.

Algoritam prikazan na Slici 4-7 implementiran je u okviru funkcije *resultslp* za rutiranje najkraćim putanjama i funkcije *resultslpmultipath* za rutiranje putanjama sa jednakim cenama. Ove dve funkcije implementirane su u okviru klase *LPmodel* u fajlu equationslp.c.



Slika 4-7 Algoritam za proračun formula optimizacionog modela

U funkcijama *resultslp* i *resultslpmultipath* implementiran je isti algoritam, i one se razlikuju po korišćenim tipovima promenljivih. U ovom slučaju nije korišćena bazna klasa koja bi omogućila formiranje optimizacionog modela za dva tipa rutiranja u jednoj funkciji zato što ovi tipovi rutiranja nisu predviđeni za istovremenu upotrebu u programu. Tip rutiranja korišćen u programu određuje se unapred izborom modula za rutiranje najkraćim putanjama ili rutiranje putanjama sa jednakim cenama.

Koraci algoritma 1-13 formiraju nejednačine linkova, predstavljenih fomulama (3.1) i (3.2), i opisanih u poglavlju 2.3.4. Prvo se u koraku 1 algoritma formiraju promenljiva *l* koja sadrži broj linkova u mreži, *n* koja sadrži broj čvorova, i u objektu *eq* koji sadrži formule, svi koeficijenti formula inicijalno se postavljaju na vrednost 0. Pošto svaka od *l* nejednačina ima faktor -1 uz maksimalnu opterećenost linka Q koji je na levoj strani nejednakosti, u koracima 2 i 3 se vrednost prvog koeficijenta u nejednačinama postavlja na -1.

$$\min Q$$

$$\sum_{i=1}^{N} k_i = 1 \tag{3.1}$$

$$\forall l \in E: \frac{\sum_{(i,m)} F_{im}^{l}(k_i r_m + k_m s_i)}{C^l} \leq Q \tag{3.2}$$

Potom se izračunavaju vrednosti koeficijenata nejednačina koji se nalaze uz koeficijente balansiranja. Promenjiva *w* označava broj tekuće nejednačine i njena vrednost se u koraku 4 inicijalno postavlja na 0. Petlja koja obuhvata korake 5-13 proračunava nejednačine za svaki link. Petlje označene brojevima 5 i 6 prolaze kroz sve parove čvorova u mreži. Za svaki par čvorova se u uslovu 7 proverava da li postoji link koji povezuje te čvorove. Ako postoji, tada se u koracima 8 do 12 izračunavaju koeficijenti nejednačine tog linka.

Petlja označena brojem 8 prolazi kroz svaki čvor u mreži i u okviru nje se proračunava faktor uz koeficijent balansiranja tog čvora, i u koraku 12 smešta u objekat *eq* koji sadrži formule. Vrednost koeficijenta se akumulira u promenjivoj *s* čija se vrednost inicijalno postavlja na nulu. U petlji označenoj brojem 10, prolazi se kroz sve čvorove u mreži. Akcija petlje 10 izvršava se u koraku 11. U koraku 11 se u promenljivoj *s* akumulira saobraćaj između balansirajućeg rutera *p* i bilo kog drugog rutera u mreži. U taj saobraćaj ulazi vrednost saobraćaja od izvorišnog rutera *q* ka

balansirajućem ruteru *p*, i od balansirajućeg rutera *p* ka odredišnom ruteru *q*, preko linka od čvora *i* ka čvoru *j*. Vrednost saobraćaja po linku između čvorova *i* i *j* se dobija sabiranjem saobraćaja ka balansirajućem ruteru *p* koji prolazi kroz link između čvorova *i* i *j* i saobraćaja od balansirajućeg rutera *p* koji prolazi kroz taj link. Saobraćaj kroz link se dobija množenjem opterećenosti tog linka pri komunikaciji rutera *q* sa balansirajućim ruterom *p* i saobraćaja rutera *q*.

Opterećenost linka između čvorova *i* i *j* se dobija kao vrednost funkcije *util* objekta *links*(*i,j*), pri čemu su argumenti funkcije *util* krajnji čvorovi koji komuniciraju, odnosno ruter *q* i balansirajući ruter *p*. Ulazni i izlazni saobraćaji rutera se dobijaju korišćenjem funkcija *getin* i *getout* objekta *traffic_list*. Parametar ovih funkcija je identifikator rutera čiji se saobraćaj traži.

Pošto se u petlji 10 prođe kroz sve rutere *q*, u koraku 12 se postavlja izračunata vrednost koeficijenta u objekat *eq* koji sadrži formule. Po proračunu svih koeficijenata nejednačine za link od čvora *i* ka čvoru *j* u koraku 13 se uvećava brojač *w* i prelazi se na sledeću nejednačinu.

Poslednja formula optimizacionog modela je jednačina koja definiše da je suma svih koeficijenata balansiranja jednaka 1. Koraci algoritma 14 i 15 postavljaju koeficijente uz sve koeficijente balansiranja na 1, dok koeficijent uz maksimalnu opterećenost linka Q ostaje jednak inicijalnoj vrednosti 0.

Po završetku algoritma na Slici 4-7 objekat *eq* sadrži vrednost svih formula optimizacionog modela, koje su napisane u obliku u kome na desnoj strani formula stoji vrednost 0. Proračunati optimizacioni model se rešava korišćenjem funkcija biblioteke lpsolve, koja je objavljena pod GNU LGPL licencom otvorenog koda. Svaka od formula modela unosi se korišćenjem funkcije *add_constraint*, dok se sam model rešava korišćenjem funkcije *solve*. Parametri funkcije *add_constraint*(*lprec *lp, REAL *row, int constr_type, REAL rh*) su pokazivač *lp* na strukturu tipa *lprec* u koju se popunjavaju svi parametri modela, pointer *row* na niz u kome su smešteni koeficijenti formule, identifikator *constr_type* da li se radi o jednakosti ili nejednakosti i vrednost na desnoj strani formule. Ciljna funkcija linearnog modela upisuje se u strukturu tipa *lprec* korišćenjem funkcije *set_obj_fn*(*lprec *lp, REAL *row*) čiji su parametri pokazivač *lp* na strukturu tipa *lprec* i pointer na niz u kome su smešteni koeficijenti ciljne funkcije.

Kada se sve formule upišu u strukturu tipa *lprec*, poziva se funkcija *solve*(*lprec *lp*), koja rešava linearni model i čiji je jedini parametar struktura tipa *lprec*. Funkcija *solve* rezultat smešta u strukturu tipa *lprec* koja joj je predata kao argument. Po završenom rešavanju modela, vrednosti rezultata za koeficijente balansiranja i maksimalnu opterećenost linka dobijaju se korišćenjem funkcije *get_variables*(*lprec *lp, REAL *var*). Argumenti funkcije *get_variables* su struktura tipa *lprec* koja sadrži rešenje linearnog modela, i pokazivač na niz u koga funkcija *get_variables* treba da upiše rešenje modela.

4.1.5 Usmeravanje paketa u mreži sa balansiranjem

Pošto je završen proračun koeficijenata balansiranja i adresa balansirajućih rutera na koje treba slati pakete, što je opisano u odeljku 4.1.4, dobijeni rezultati se koristite za usmeravanje paketa u mreži. Na Slici 4-1 je prikazano da se koeficijenati balansiranja i adrese za balansiranje prenose iz bloka za optimizaciju u blok za balansiranje saobraćaja. U okviru implementacije, blok za balansiranje saobraćaja implementiran je kao samostalan program. Pošto se blok za optimizaciju opisan u odeljku 4.1.4 nalazi u okviru proširenog *ospfd* programa, koeficijenti balansiranja i adrese za balansiranje se prenose do programa za balansiranje saobraćaja korišćenjem TCP protokola. Pri komunikaciji, prošireni *ospfd* program ima ulogu servera, i program za balansiranje saobraćaja se kao klijent povezuje na prošireni *ospfd* program. Funkcije balansiranja saobraćaja i TCP klijenta sadržane su u fajlu changedest.C.

Usmeravanje paketa u mreži sa balansiranjem ostvaruje se korišćenjem izvornog rutiranja sa delimičnim spiskom rutera (eng. *loose source routing*). Pri izvornom rutiranju, u polje odredišne adrese paketa se upisuje sledeći ruter kroz koga paket treba da prođe, dok se u opciono polje paketa upisuje redom spisak ostalih rutera kroz koje paket treba da prođe zaključno sa odredišnom adresom paketa. Svaki put kada paket stigne do destinacije, proverava se spisak preostalih adresa kroz koje treba proći, i ako postoje preostale adrese, prva od njih postaje sledeća destinacija paketa. Izvorno rutiranje sa delimičnim spiskom rutera se razlikuje od izvornog rutiranja sa potpunim spiskom rutera (eng. *strict source routing*) po tome što pri kretanju ka tekućoj adresi iz spiska paket može prolaziti i kroz druge rutere koji su na putu ka toj adresi. Pri izvornom rutiranju sa potpunim spiskom rutera potrebno je u spisku navesti sve rutere

kroz koje paket redom treba da prođe. U programu za balansiranje saobraćaja u polje odredišne adrese paketa upisuje se adresa za balansiranje, dok se odredišna adresa paketa upisuje u opciono polje za izvorno rutiranje sa delimičnim spiskom adresa. Na taj način, paket će prvo biti prenet do balansirajućeg rutera, gde će u njegovo polje odredišne adrese biti upisana adresa odredišta paketa. Dodavanje adrese za balansiranje paketu vrši se u funkciji *addlssr*, prikazanoj na Slici 4-8, čiji su parametri pokazivač na paket *packet*, dužina paketa *payload_len*, pokazivač na pokazivač na memoriju sa rezultujućim paketom *lssrpacket*, pokazivač na memoriju u koju je smešten sadržaj opcije izvornog rutiranja koju treba ubaciti u paket *lssropt* i dužina opcije izvornog rutiranja *opt_len*. Adresa rezultujućeg paketa se upisuje u pokazivač na memoriju sa rezultujućim paketom, a funkcija vraća broj bajtova rezultujućeg paketa.

```
int  addlssr(unsigned char *packet,int payload_len,unsigned char **lssrpacket,unsigned char
*lssropt,int opt_len) {
  struct iphdr *iphdr,*iphdr1;
  unsigned char *pkt;
  *lssrpacket= ipbuffer; //pokazivac lssrpacket pokazuje na ipbuffer
  pkt=*lssrpacket;
  memset(pkt,0,payload_len+opt_len); // inicijalizujemo lssrpacket na nultu vrednost
  iphdr = (struct iphdr *)packet;// formiramo pointer tipa iphdr na bafer koji sadrzi paket radi
                                    //lakseg pristupa
  memcpy(pkt,packet,4*iphdr->ihl); // kopiramo header packeta u pkt
  memcpy(pkt+4*iphdr->ihl,lssropt,opt_len); // kopiramo lsrr opciju u pkt
  memcpy(pkt+4*iphdr->ihl+opt_len,packet+4*iphdr->ihl,payload_len-4*iphdr->ihl);
                                                                  // kopiramo payload u pkt
  iphdr1=(struct iphdr*) pkt;
  iphdr1->ihl=iphdr->ihl+(opt_len/4); // u novi paket(pkt) upisujemo duzinu njegovog headera u
                                          //ihl polje
  iphdr1->tot_len=htons(ntohs(iphdr->tot_len)+opt_len);// upisujemo duzinu paketa
  return ( payload_len+opt_len);
}
```

Slika 4-8 Funkcija *addlssr*

Kao što je predstavljeno na Slici 4-1, opcija izvornog rutiranja dodaje se samo paketima koji ulaze u oblast balansiranja. U ove pakete spadaju paketi koji u ruter stižu sa portova koji nisu u oblasti balansiranja, i koji ulaze u oblast balansiranja. Paketi koji ulaze u oblast balansiranja prosleđuju se programu za balansiranje saobraćaja korišćenjem *Linux* komande *iptables* [59]. Obrada IP paketa koji prolaze kroz *Linux* kernel implementirana je u okviru *netfilter* bloka kernela. Konfigurisanje *netfilter* bloka omogućeno je korišćenjem komande *iptables*. Slanje svih paketa koji prolaze kroz ruter programu za balansiranje omogućeno je komandom:

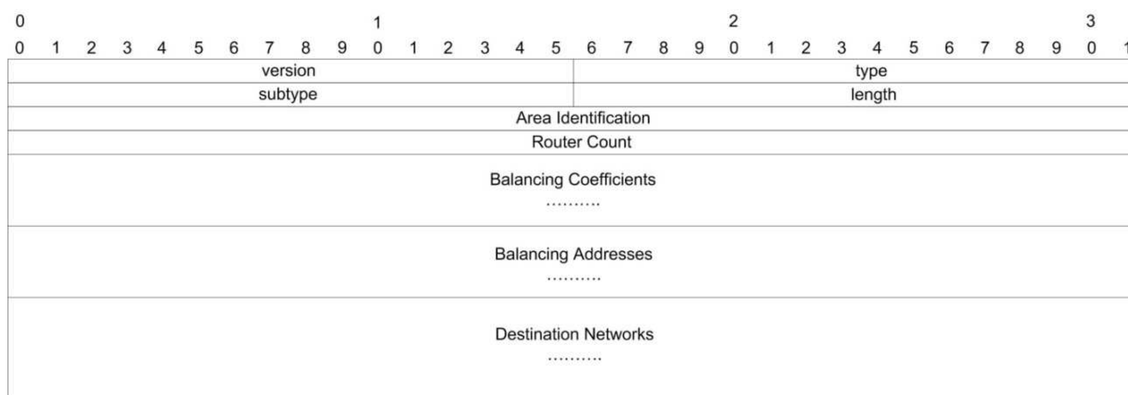
iptables –I FORWARD –p all –j NFQUEUE

Navedena komanda definiše pravilo za obradu paketa koji prolaze kroz *Linux* kernel. Prvi parametar komande (*–I FORWARD*) specificira da se komanda odnosi na pakete koji prolaze sa jednog porta rutera na drugi port. Drugi parametar (*–p all*) specificira da se komanda odnosi na sve protokole, dok poslednji parametar (*–j NFQUEUE*) specificira da se paketi koji zadovoljavaju navedene kriterijume stave u red za čekanje koga može čitati korisnički program koji se izvršava na *Linux* operativnom sistemu. Korisnički programi mogu čitati pakete iz tog reda za čekanje korišćenjem funkcija iz *libnetfilter_queue* biblioteke [60]. Program za balansiranje koristi ove funkcije da bi iz kernela očitavao pakete filtrirane na osnovu *iptables* komande. Posle očitavanja paketa, promene odredišne adrese i dodavanja opcije za izvorno rutiranje, program za balansiranje saobraćaja vraća ove pakete u *netfilter* blok kernela. Po prijemu u *netfilter* blok, paket će nastaviti put kroz kernel sa istog mesta odakle je bio prosleđen programu za balansiranje. Kada dođe do bloka za rutiranje, paket će u bloku za rutiranje biti prosleđen ka izlaznom portu na osnovu odredišne adrese zadate u programu za balansiranje.

Program za balansiranje saobraćaja, pored koeficijenata balansiranja i adresa za balansiranje, dobija i spisak koji sadrži dostupne odredišne mreže, kao i rutere preko kojih su te mreže dostupne. Ovi podaci se dobijaju posredstvom TCP protokola od proširenog *ospfd* programa. Zaglavlje paketa koji se prenosi TCP protokolom sadrži verziju, tip, podtip i dužinu paketa, posle zaglavlja slede identifikator OSPF oblasti na koju se podaci odnose i broj rutera u toj oblasti. U paketu zatim slede koeficijenti balansiranja za sve rutere, pa posle njih adrese za balansiranje na koje treba slati pakete pri balansiranju za svaki ruter. Polje sa odredišnim mrežama za svaki ruter sadrži spisak mreža dostupnih preko tog rutera. Format paketa za prenos podataka o balansiranju prikazan je na Slici 4-9.

Spisak dostupnih odredišnih mreža i pridruženih rutera se radi brzog pretraživanja čuva u *Patricia* stablu. Pri obradi paketa, za odredišnu adresu paketa pronalazi se mreža u *Patricia* stablu, i u tom čvoru stabla se nalazi pokazivač na ruter preko koga se dolazi do te mreže. U strukturi odredišnog rutera je zabeležen identifikator tekućeg balansirajućeg rutera i broj preostalih paketa koje treba balansirati

preko tog rutera. Sa ovim informacijama, adresa balansirajućeg rutera se postavlja za odredišnu adresu paketa, odredišna adresa se premešta u zaglavlje izvornog rutiranja, i u strukturi rutera se broj preostalih paketa za balansiranje preko tekućeg balansirajućeg rutera umanjuje za jedan. U slučaju da broj preostalih paketa za balansiranje dostigne nulu, prelazi se na sledeći ruter za balansiranje i broj paketa za balansiranje preko tog rutera se postavlja na vrednost proporcionalnu koeficijentu balansiranja tog rutera.



Slika 4-9 Format paketa za prenos podataka o balansiranju

U implementaciji je *Patricia* stablo predstavljeno klasama *PatEntry* (fajl pat.h) i *PatTree* (fajl pat.C) koje su deo originalne implementacije *ospfd* programa [56], i opisane su u [54]. U okviru programa za balansiranje saobraćaja, čvor *Patricia* stabla predstavljen je klasom *PatNode* (fajl pat.h). Ova klasa predstavlja odredišne mreže u oblasti balansiranja i njima pridružene rutere, i omogućava brzo pronalaženje rutera pridruženog odredišnoj mreži za koju je paket namenjen. Klasa *PatNode* sadrži pokazivač na strukturu rutera tipa *Router* (fajl changedest.h) koja predstavlja sam ruter pridružen odredišnoj mreži, ključ koga čini adresa mreže, i pokazivač na objekat tipa *PatNode*. Pokazivač na objekat tipa *PatNode* koristi se u strukturi *Router* za formiranje liste čvorova u *Patricia* stablu pridruženih ruteru (fajl changedest.C), što omogućava da se pri brisanju rutera obrišu i sve mreže koje su pridružene ruteru. Pored liste pridruženih čvorova u *Patricia* strukturi, struktura *Router* sadrži identifikator rutera za balansiranje, broj paketa koji je preostao za slanje preko tog rutera, i pokazivač na strukturu tipa *areabal* koja sadrži informacije o oblasti balansiranja kojoj ruter pripada i pokazivač na sledeći ruter. Poslednja bitna struktura u programu za balansiranje je *areabal* (fajl changedest.h), koja sadrži identifikator oblasti balansiranja, broj rutera, veličinu tekućeg bafera za koeficijente balansiranja rutera u oblasti, pokazivač na bafer

sa koeficijentima balansiranja, pokazivač na bafer sa adresama balansiranja, pokazivač na listu struktura *Router* koja predstavlja rutere u oblasti i pokazivač na sledeću strukturu tipa *areabal*.

Pri balansiranju, na osnovu odredišne adrese paketa u *Patricia* stablu se pronalazi čvor koji predstavlja odredišnu mrežu paketa. Zatim se prelazi na strukturu tipa *Router* na koju pokazuje čvor *Patricia* stabla i koja predstavlja odredišni ruter. U toj strukturi je zapisan identifikator tekućeg balansirajućeg rutera za taj odredišni ruter i broj preostalih paketa za balansiranje preko tog rutera. Na osnovu identifikatora balansirajućeg rutera iz strukture *areabal* koja predstavlja oblast balansiranja i sadrži polja opisana u prethodnom pasusu dobija se adresa za balansiranje na koju treba poslati paket.

4.1.6 Organizacija programskog koda

U uvodu poglavlja 4 navedene su tri implementacione celine rutiranja sa balansiranjem. Funkcionalnost prve dve implementacione celine, komunikacija među ruterima i proračun parametara balansiranja, implementirana je kao proširenje postojeće implementacije OSPF protokola, *ospfd* programa čiji je autor Džon Moj i koji je objavljen pod GNU GPL licencom otvorenog koda.

4.1.6.1 Implementacija rutiranja sa balansiranjem u okviru *ospfd* programa

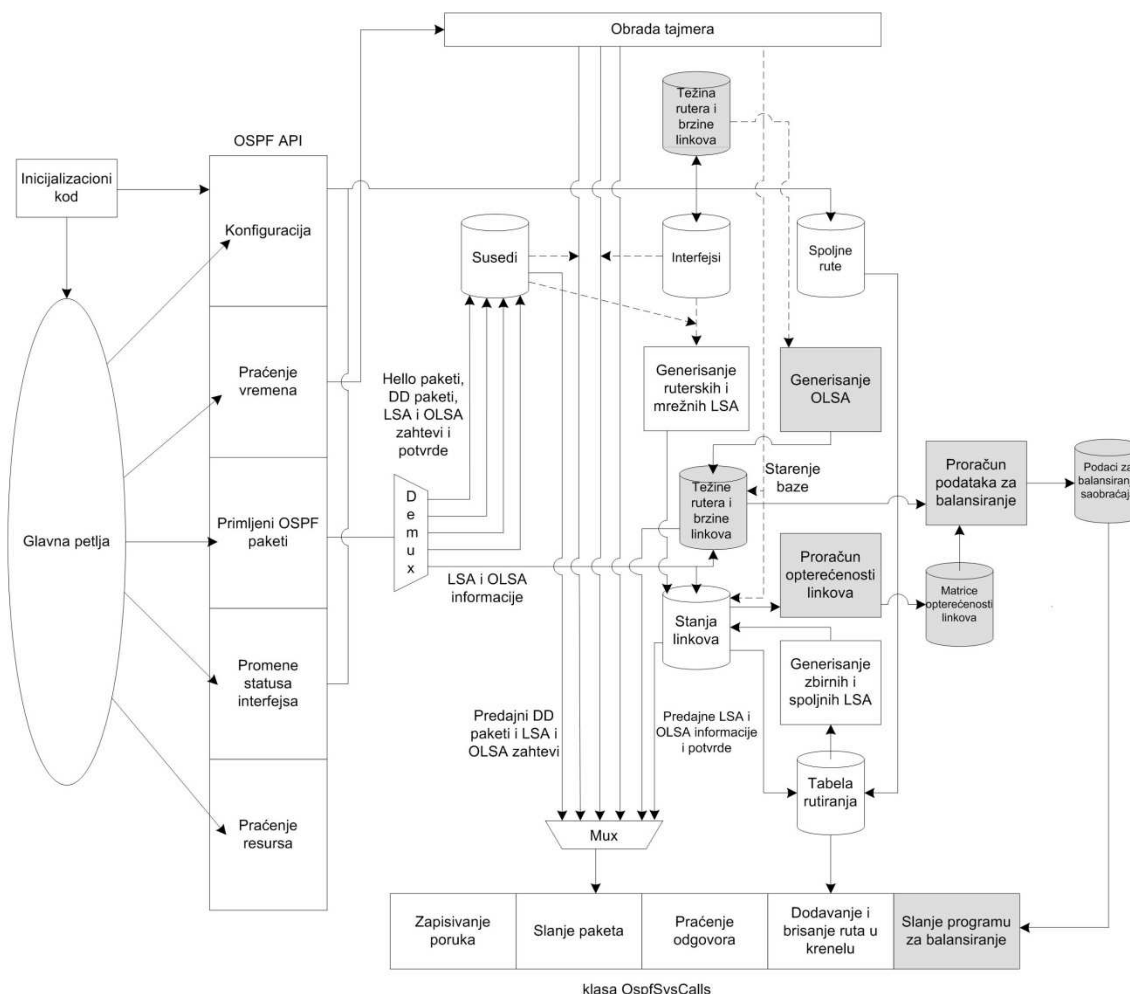
Slika 4-11 ilustruje tok podataka u okviru proširene *ospfd* implementacije. Ova slika predstavlja proširenje Slike 3.1 iz knjige *OSPF Complete Implementation* [54], na kojoj su sivom bojom označeni elementi dodati u okviru implementacije rutiranja sa balansiranjem.

U gornjem delu Slike 4-11 prikazan je element „Težina rutera i brzine linkova”. Ovaj element označava podatke o generisanom i terminiranom saobraćaju rutera i o brzinama njegovih linkova, koji se učitavaju iz konfiguracionog fajla ospfd.conf. U cilju učitavanja ovih podataka iz konfiguracionog fajla, u ospfd_linux.C fajl *ospfd* softvera dodata je mogućnost dekodovanja komandi konfiguracionog fajla *externport* i *ifcspeed*, tako što su implementirane funkcije *ExternPort* i *IfcSpeed*, i prijavljene *tcl* interpeteru ugrađenom u *ospfd* softver korišćenjem funkcije *Tcl_CreateCommand*. Primer konfiguracionog fajla sa komandama *externport* i *ifcspeed* prikazan je na Slici 4-10. Fajl

se odnosi na ruter sa adresom 100.3.7.2, koji ima linkove ka susednim ruterima 100.3.9.1 i 100.3.10.1 i lokalnu mrežu čiji prijemni i predajni saobraćaj su definisani poslednjom komandom u fajlu.

```
Routerid 100.3.7.2
Area 0.0.0.0
ifcspeed 100.3.7.2 100.3.9.1 1000
ifcspeed 100.3.7.2 100.3.10.1 1000
externport 100.3.7.2 1500 2500
```

Slika 4-10 Konfiguracioni fajl rutera u mreži sa balansiranjem



Slika 4-11 Tok podataka u okviru modifikovanog *ospfd* programa

Konfiguraciona komanda *externport* služi za definisanje portova rutera koji nisu u oblasti balansiranja i njeni parametri su IP adresa porta i zahtevani ulazni i izlazni protok po tom portu. Na osnovu protoka svih portova navedenih u *externport* komandama računaju se zahtevani generisani i terminirani saobraćaj rutera. Komanda *ifcspeed* služi za definisanje brzine portova u oblasti balansiranja i njeni parametri su IP

adresa porta rutera i porta na udaljenom kraju linka, kao i sama brzina linka. Podaci definisani konfiguracionim komandama *externport* i *ifcpspeed* čuvaju se u ulančanim listama i, kao što je prikazano na Slici 4-11, koriste se za generisanje netransparentnih LSA paketa koji ove podatke prenose do drugih rutera. Generisanje, prijem, i obrada netransparentnih LSA implementirani su u fajlu lsaopqsend.C, pri čemu su funkcije *sendlsa_traffic* i *sendlsa_link*, opisane u poglavlju 4.1.3, implementirane u fajlu lsaop1.C.

Strukture podataka za generisane i terminirane saobraćaje svih rutera u mreži i brzine svih linkova u mreži su na Slici 4-11 predstavljene elementom „Težine rutera i brzine linkova", i opisane su u odeljku 4.1.3. U strukture podataka u elementu „Težine rutera i brzine linkova" ulaze i podaci primljeni od drugih rutera posredstvom netransparentnih LSA paketa. Klasa *TrafficList* implementirana je u fajlu lsaopbase.C, dok je klasa *LinksList* implementirana u fajlu lsaspeeds.C. Klasa *TrafficList* sadrži podatke o zahtevanim terminiranim i generisanim saobraćajima rutera, dok klasa *LinksList* sadrži podatke o brzinama linkova u mreži. Ove klase su detaljno opisane u poglavlju 4.1.3.

Na osnovu podataka u elementu „Težine rutera i brzine linkova" proračunavaju se podaci za balansiranje saobraćaja, predstavljeni istoimenim elementom na Slici 4-11. Za proračun podataka o balansiranju potrebne su i matrice opterećenosti linkova, koje su prikazane na Slici 4-11. Proračun matrica opterećenosti linkova predstavljen je na Slici 4-11 blokom „Proračun opterećenosti linkova" i vrši se korišćenjem algoritama opisanih u pododeljcima 4.1.4.1 i 4.1.4.2. Ovaj proračun za rutiranje najkraćim putanjama sadržan je u fajlu sprtraffic.C, a rezultati proračuna sadržani su u fajlu Links.C. Za rutiranje po putanjama sa istom cenom proračuni su sadržani u fajlovima ecmptraffic.C i LinksMultipath.C.

Proračun podataka za balansiranje predstavljen je na Slici 4-11 blokom „Proračun podataka za balansiranje", opisan je u odeljku 4.1.4.3 i implementiran je u fajlu equations.C. Ovi podaci se sastoje od koeficijenata balansiranja i adresa balansirajućih rutera.

Podaci za balansiranje šalju se programu za balansiranje, što je prikazano na Slici 4-11, blokom „Slanje programu za balansiranje". Slanje podataka implementirano je u

fajlu sendbalance.C, koji koristi funkciju *tcp_sendp* implementiranu u fajlu tcpserver.C. Da bi slanje podataka bilo moguće, u fajlu ospf.c se, u toku inicijalizacije *ospfd* softvera prikazane na Slici 4-11, otvara TCP soket za komunikaciju sa programom za balansiranje. Soket se otvara funkcijom *socket*(*int namespace, int style, int protocol*) koja je deo *libc* biblioteke. Na serveru se IP adresa i port na kome će server primati konekcije definišu pozivom funkcije *bind*(*int socket, const struct sockaddr *address, socklen_t address_len*). Na serveru se pozivom funkcije *listen*(*int socket, int backlog*) prelazi u stanje praćenja zahteva za konekcije, i konekcije se prihvataju pozivom funkcije *accept*(*int socket, struct sockaddr *restrict address, socklen_t *restrict address_len*). Na strani klijenta, konekcija se vrši pozivom funkcije *connect*(*int socket, const struct sockaddr *address, socklen_t address_len*). Pošto je konekcija uspostavljena, komunikacija se vrši pozivima funkcija *send*(*int socket, const void *buffer, size_t length, int flags*) i *recv*(*int socket, void *buffer, size_t length, int flags*). Po završenoj komunikaciji soket se zatvara pozivom funkcije *close*(*int socket*).

Program za balansiranje, opisan u odeljku 4.1.5, implementiran je u okviru fajla changedest.C, i koristi implementaciju *Patricia* stabla iz *ospfd* implementacije, sadržane u fajlu pat.C.

4.2 Implementacija algoritama ažuriranja

Na blok šemi rutera prikazanoj na Slici 4-1 izdvajaju se dve celine. Jedna obuhvata blokove koji proračunavaju putanje paketa kroz mrežu. Ovi blokovi za proračun putanja sa balansiranjem paketa su opisani u poglavlju 4.1. Blokovi opisani u poglavlju 4.1 proračunavaju koeficijente balansiranja i adrese za balansiranje, i omogućavaju usmeravanje paketa preko balansirajućeg rutera korišćenjem izvornog rutiranja.

Druga celina rutera predstavlja paketski procesor, blok koji na osnovu proračunatih putanja paketa, vrši prosleđivanje paketa sa jednog porta rutera na drugi. Dve osnovne komponente paketskog procesora su lukap blok i krosbar. Lukap blok za svaki IP paket koji se usmerava, na osnovu odredišne adrese tog paketa i tabele rutiranja određuje izlazni port rutera. Krosbar, ili neki drugi komutator, na osnovu informacije o izlaznim portu prosleđuje paket sa porta sa koga je stigao na proračunati izlazni port.

Lukap blok ima zadatak da na osnovu odredišne adrese IP paketa u tabeli rutiranja pronađe rutu sa najdužim prefiksom koji je istovremeno i prefiks odredišne IP adrese. Tabele rutiranja mogu imati i stotine hiljada ruta i, da bi se postigle potrebne velike brzine prosleđivanja paketa, u lukap bloku se informacije iz tabele rutiranja čuvaju u formi pogodnoj za brzo pronalaženje rute sa najdužim poklapanjem prefiksa. Paralelno izvršavanje više pretraga ruta omogućava dodatno povećanje protoka paketa kroz lukap blok.

Zapise podataka iz tabele rutiranja u lukap blok vrši algoritam ažuriranja lukap bloka. Položaj bloka za ažuriranje u okviru rutera prikazan je na Slici 4-1. Algoritam ažuriranja predstavlja vezu između bloka za proračun putanja i bloka za prosleđivanje paketa. U okviru implementacije, veza između programa za proračun tabela rutiranja i modula za ažuriranje lukap bloka je poziv funkcije *obrada* čiji su parametri mrežna adresa odredišne mreže, dužina mrežne adrese, izlazni port rutera na koga treba prosleđivati pakete za tu mrežu i parametar koji označava da li se ruta dodaje ili briše. Na osnovu ovih parametara modul za ažuriranje podešava podatke u lukap bloku. Algoritam dodavanja rute obuhvata i slučaj zamene izlaznog porta rute. U ovom slučaju ruta se ponovo dodaje sa novom vrednošću izlaznog porta i tada će algoritam dodavanja rute rezultovati zamenom izlaznog porta postojeće rute.

U ovome odeljku biće opisane implementacije algoritama ažuriranja za BPFL i POLP, dve lukap procedure koje omogućavaju pronalaženje izlaznih portova pri velikim protocima paketa. BPFL i POLP opisani su u odeljcima 3.1.2.10 i 3.1.2.9 respektivno. Pored BPFL-a i POLP-a, biće opisana implementacija algoritma ažuriranja u slučaju lukap algoritma baziranog na pretrazi pomoću prioritetnog stabla, koje predstavlja novije rešenje sa potencijalom da uz uvođenje paralelizacije upita takođe bude korišćeno u ruterima visokih protoka.

4.2.1 Algoritam ažuriranja za BPFL lukap tabele

Pri izmeni, brisanju ili dodavanju rute u BPFL lukap tabeli, prvi korak je pronalaženje BPFL nivoa koga treba ažurirati, i ovaj korak se izvršava na osnovu dužine maske pridružene ruti.

Sledeći korak je pronalaženje rute u okviru BPFL nivoa, i ovaj korak se izvršava pronalaženjem opsega u kome se prefiks nalazi, i pretragom balansiranog stabla

pridruženog tom opsegu. U slučaju pronalaženja traženog prefiksa u balansioranom stablu, poslednji korak pri pronalaženju rute je pretraga podstabla pridruženog pronađenom čvoru balansiranog stabla.

Po pronalaženju rute, izvršavanje operacije izmene izlaznog porta se svodi na upis nove vrednosti izlaznog porta na memorijsku lokaciju određenu podstablom. U slučaju da je tražena operacija ažuriranja brisanje rute ili dodavanje rute potrebno je izvršiti dodatne operacije, opisane u sledećim poglavljima.

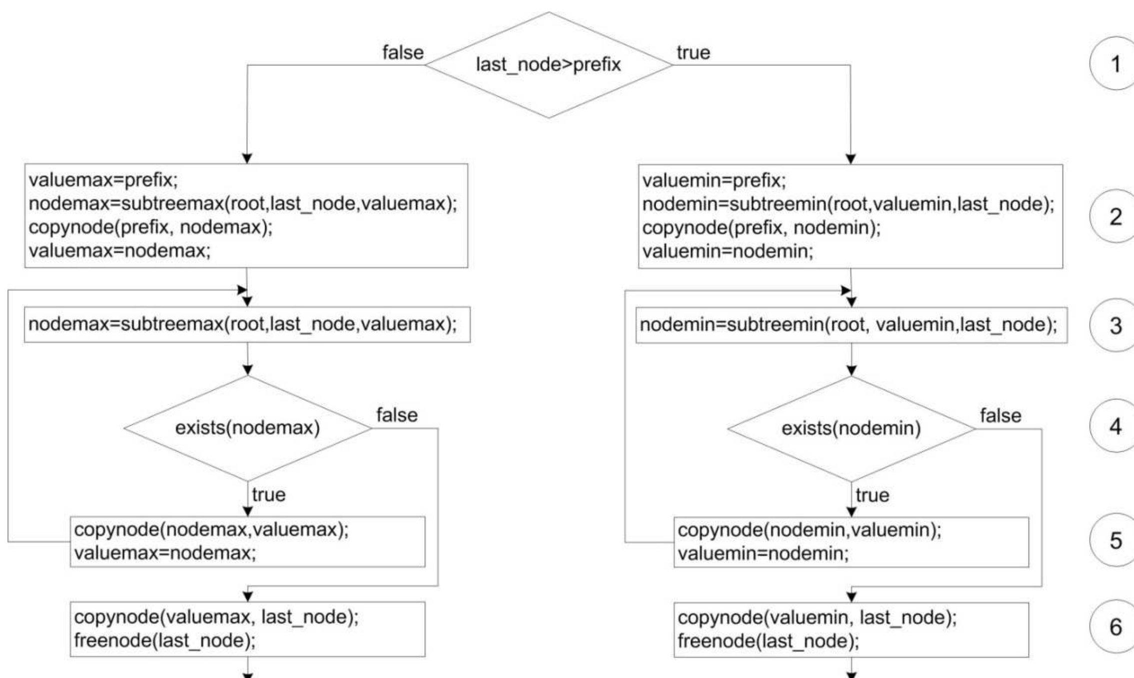
4.2.1.1 Brisanje rute

U slučaju brisanja rute, zavisno od vrednosti ostalih ruta, potrebno je izvršiti različite operacije. U najjednostavnijem slučaju, kada u podstablu u kome je ruta zapisana postoje i druge rute, dovoljno je samo u podstablu obrisati podatak o ruti. BPFL koristi dve vrste zapisa podstabla, opisane u poglavlju 3.1.2.10. Brisanje podataka o ruti se u slučaju zapisa gusto popunjenog podstabla svodi na postavljanje bita koji odgovara vrednosti prefiksa na vrednost 0. Veličina memorije koju zauzima zapis gusto popunjenog podstabla zavisi od dubine podstabla. U slučaju korišćenja podstabala dubine osam, zapis gusto popunjenog podstabla zauzima s_d = 510 bita, od kojih svaki označava da li njemu pridružen čvor podstabla sadrži prefiks. Pronalaženje bita koji odgovara prefiksu se vrši određivanjem koji čvor podstabla odgovara prefiksu i koji je njegov redni broj u podstablu. Zapis retko popunjenog podstabla omogućava predstavljanje podstabala koja sadrže do osam prefiksa, i sadrži redne brojeve pozicija tih prefiksa u podstablu. Uklanjanje rute iz podstabla vrši se brisanjem rednog broja koji odgovara obrisanoj ruti.

U komplikovanijem slučaju briše se jedina ruta pridružena podstablu. Tada se iz balansiranog stabla briše čvor koji odgovara tom podstablu. Brisanje čvora vrši se korišćenjem algoritma brisanja čvora balansiranog stabla, prikazanog na Slici 4-12. Algoritam prikazan na Slici 4-12 implementiran je u okviru klase stabla u fajlu balancedtrees.C. BPFL stablo je kompletno balansirano stablo, odnosno balansirano stablo koje ima sve popunjene nivoe sem poslednjeg i u poslednjem zauzete čvorove počevši od levog kraja nivoa, i iz tog razloga posle brisanja čvora potrebno je da se oslobodi samo poslednji čvor u poslednjem nivou. Iz tog razloga se pri brisanju čvora vrši premeštanje čvorova tako da se na kraju oslobodi poslednji čvor u poslednjem

nivou. Premeštanje se vrši tako što se čvorovi čije su vrednosti prefiksa između vrednosti prefiksa čvora koji se briše i poslednjeg čvora u poslednjem nivou redom premeštaju ka upražnjenom mestu čvora koji se briše.

U algoritmu prikazanom na Slici 4-12, u prvom koraku se određuje da li je vrednost prefiksa poslednjeg čvora u poslednjem nivou veća od vrednosti prefiksa čvora koji se briše. U slučaju kada je vrednost prefiksa pridruženog poslednjem čvoru u poslenjem nivou veća od vrednosti prefiksa pridruženog čvoru koji se briše, tada se premeštanje vrši tako što se na mesto prefiksa koji se briše kopira čvor koji ima prvu veću vrednost prefiksa od čvora koji se briše, zatim se na poziciju tog čvora kopira čvor sa sledećom većom vrednošću prefiksa. Kopiranje čvora sa prvom sledećom većom vrednošću prefiksa na upražnjeno mesto se nastavlja dok se ne prekopira poslednji čvor u poslednjem nivou, čime se proces brisanja završava. U algoritmu na Slici 4-12 u drugom koraku vrši se kopiranje čvora sa prvim sledećim većim prefiksom na mesto obrisanog čvora. Funkcija *subtreemin* se koristi za pronalaženje minimalnog prefiksa u datom opsegu u određenom podstablu. Argumenti funkcije *subtreemin*(*snode *node, unsigned int lowerbound, unsigned int upperbound*) su koren podstabla u kome se vrši pretraga, sa oznakom *node*, i gornje i donje granice u okviru kojih se traži najmanji čvor, sa oznakama *lowerbound* i *upperbound*.



Slika 4-12 Brisanje čvora iz balansiranog stabla (BPFL)

Posle kopiranja prvog sledećeg manjeg čvora na vrednost obrisanog čvora, ulazi se u petlju koju čine koraci 3-5, u kojoj se redom pomera prvi sledeći veći čvor na upražnjeno mesto. Petlja se završava kada se premesti čvor koji ima prvu manju vrednost od vrednosti prefiksa poslednjeg čvora u poslednem nivou. Potom se u koraku 6 algoritma na upražnjeno mesto premešta poslednji čvor u poslednjem nivou čime se algoritam završava.

U slučaju kada je vrednost provere u prvom koraku algoritma prikazanog na Slici 4-12 negativna, takođe se vrši premeštanje sa ciljem oslobađanja poslednjeg čvora u poslenjem nivou, sa razlikom da se ovoga puta na upražnjeno mesto premešta čvor sa prvim sledećim manjim prefiksom. Koraci premeštanja u slučaju kada je vrednost prefiksa poslednjeg čvora u poslednjem redu manja od vrednosti prefiksa čvora koji se briše prikazani su na levoj strani Slike 4-12 i odgovaraju oznakama koraka 2-6.

Po izvršenom brisanju čvora, sledeći korak je izvršavanje algoritma za održavanje ravnomerne raspodele čvorova po balansiranim stablima. Ovaj algoritam podrazumeva premeštanje čvora u susedno stablo ukoliko u njemu ima manje čvorova. U implementaciji je usvojeno da se premeštanje vrši ukoliko je razlika u broju čvorova veća od tri. Da bi granice između opsega vrednosti prefiksa sadržanih u balansiranim stablima mogle biti validne, uvek se u balansirano stablo sa većim vrednostima čvorova premešta čvor sa najvećom vrednošću, dok se u balansirano stablo sa manjim vrednostima čvorova premešta čvor sa najmanjom vrednošću. Sama operacija premeštanja sastoji se iz brisanja prefiksa u jednom balansiranom stablu, dodavanja prefiksa u drugo balansirano stablo i ažuriranja granica opsega. Algoritam premeštanja čvorova između balansiranih stabala opisan je u poglavlju 4.2.1.3.

U slučaju brisanja najmanjeg ili najvećeg čvora balansiranog stabla, po završenom brisanju čvora stabla vrši se ažuriranje vrednosti granica opsega stabla.

4.2.1.2 Dodavanje rute

Kao i u slučaju brisanja rute, operacije koje se izvršavaju u procesu dodavanja rute zavise od postojećih ruta u BPFL lukap tabeli.

U početku popunjavanja, dok nisu zauzeta sva balansirana stabla, nova ruta se dodaje u prazno stablo. U tom stablu se u korenu stabla dodaje čvor koji odgovara

prefiksu, i ažuriraju se podstablo i vrednost izlaznog porta koji pripadaju tom čvoru. Zatim se za balansirano stablo kome je dodat prefiks određuju vrednosti granica opsega. Granice opsega su po dodavanju prvog čvora jednake i odgovaraju prefiksu pridruženom tom čvoru.

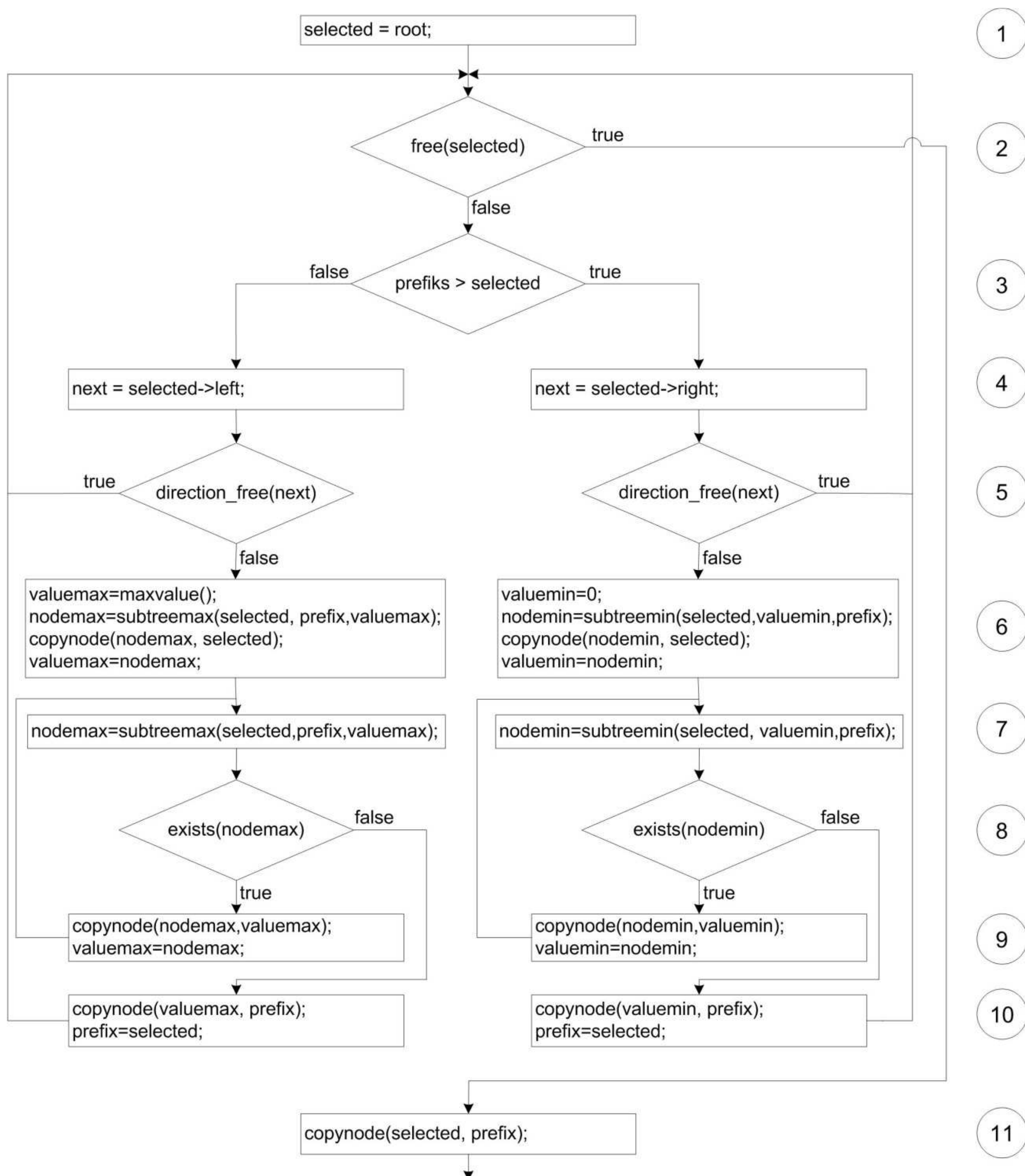
Ako su sva balansirana stabla kreirana, traži se stablo čiji čvorovi sadrže prefikse najbliže prefiksu rute koja se dodaje, tj. stablo čiji opseg obuhvata novi prefiks, ili je najbliži novom prefiksu. Po pronalasku tog balansiranog stabla, proverava se da li je čvor koji odgovara prefiksu već napravljen, što će biti slučaj ukoliko su prethodno dodavane rute sa prefiksima koje odgovaraju istom čvoru balansiranog stabla.

Ukoliko balansirano stablo već postoji, dodavanje nove rute se svodi na dodavanje informacija o ruti u podstablo i na upisivanje vrednosti izlaznog porta u memorijsku lokaciju određenu podstablom [43]-[44]. Postupak dodavanja zapisa prefiksa u podstablo suprotan je postupku brisanja prefiksa opisanom u poglavlju 4.2.1.1. U slučaju zapisa gusto popunjenog podstabla, dodavanje zapisa prefiksa svodi se na postavljanja vrednosti bita koji odgovara poziciji prefiksa u podstablu na 1. U slučaju zapisa retko popunjenog podstabla, u memoriju podstabla se upisuje identifikator pozicije prefiksa u podstablu.

U slučaju da u balansiranom stablu ne postoji čvor koji odgovara prefiksu nove rute, dodavanje novog čvora u balansirano stablo vrši se korišćenjem algoritma prikazanog na Slici 4-13. Algoritam prikazan na Slici 4-13 implementiran je u okviru klase stabla u fajlu balancedtrees.C. BPFL balansirano stablo je kompletno balansirano stablo, što znači da su svi nivoi stabla sem poslednjeg potpuno popunjeni i da je poslednji nivo delimično popunjen počev od leve strane. Iz tog razloga je novi čvor potrebno dodati na prvu sledeću slobodnu poziciju u poslednjem nivou ili, ako je poslednji nivo balansiranog stabla popunjen, u prvu poziciju u sledećem nivou stabla. Da bi ovaj uslov bio ispunjen, algoritam prikazan na Slici 4-13 polazi od korena balansiranog stabla (korak 1) i prolazi kroz stablo binarnom pretragom za vrednost prefiksa nove rute.

Na početku pretrage se proverava da li se stiglo do prazne pozicije u balansiranom stablu (korak 2), i ako jeste prazna pozicija se popunava i algoritam se završava. Da bi se došlo do željene, tj. prve sledeće slobodne pozicije, algoritam u

svakom koraku binarne pretrage treba da izabere granu koja vodi ka prvoj sledećoj slobodnoj praznoj poziciji.



Slika 4-13 Dodavanje čvora u balansirano stablo (BPFL)

U svakom koraku binarne pretrage, proverava se da li je vrednost prefiksa veća ili manja od vrednosti tekućeg čvora, i ako je manja bira se leva grana, a ako je veća bira se desna grana. Ako se ovaj izbor ne podudara sa smerom kretanja ka prvoj slobodnoj praznoj poziciji, tada je potrebno uzeti vrednost tekućeg čvora za dalju pretragu, tekući prefiks smestiti u podstablo koje nije u smeru kretanja, i izvršiti

premeštanje čvorova u podstablu koje nije u smeru kretanja tako da se zauzme i mesto do koga se stiglo u toku pretrage. Navedene operacije obavljaju se u koracima 3-10 algoritma prikazanog na Slici 4-13.

Premeštanje čvorova pri dodavanju rute je slično premeštanju čvorova pri brisanju rute opisanom u poglavlju 4.2.1.1 i prikazanom na Slici 4-13. U oba slučaja koristi se deo algoritama koji uklanja jedan čvor i ubacuje drugi čvor, i ovaj deo algoritma je na Slici 4-12prikazan koracima 2-6, dok je na Slici 4-13 prikazan koracima 6-10. U ovim koracima se u balansiranom stablu jedan čvor briše i ubacuje se drugi čvor, i između brisanja čvora i ubacivanja novog čvora vrši se premeštanje čvorova na način koji zavisi od toga koji od ova dva čvora je veći. Ako je čvor koji se briše veći tada se na njegovo mesto stavlja prvi manji čvor, pa na mesto tog čvora prvi manji od njega i tako dalje dok se ne dođe do vrednosti čvora koji se ubacuje kada on popunjava upražnjeno mesto i postupak se završava. . Provera da su svi veći čvorovi od čvora koji se ubacuje pomereni vrši se u koraku 4 algoritma na Slici 4-12, i u koraku 8 algoritma na Slici 4-13. Ako je čvor koji se briše manji, na njegovo mesto se premešta prvi veći čvor, pa na mesto tog čvora prvi sledeći veći i tako dalje dok se ne pomere svi manji čvorovi od čvora koji se ubacuje, kada se čvor koji se ubacuje upisuje na mesto na kome je bio poslednji pomereni čvor.

U algoritmima na Slikama 4-12 i 4-13 razlikuje se slučaj kada se premeštanje čvorova izvršava. U slučaju brisanja rute, opisanom u poglavlju 4.2.1.1, premeštanje poziva kada se čvor koji odgovara brisanoj ruti uklanja, dok se u stablo ubacuje poslednji čvor iz poslednjeg nivoa. U slučaju dodavanja rute, premeštanje se vrši kada kretanje kroz stablo nije moguće nastaviti ka prvom slobodnom mestu u poslednjem redu balansiranog stabla. Ako je potrebno nastaviti pretragu u levom podstablu, koje sadrži čvorove sa manjim vrednostima prefiksa, tada se čvor sa kojim se vrši pretraga dodaje u desno podstablo, a iz desnog podstabla se uzima najmanji čvor. Ovaj najmanji čvor se smešta na mesto do koga je stigla pretraga, a pretraga se nastavlja sa čvorom koji je bio na tom mestu. Ako se prvo slobodno mesto nalazi u desnom podstablu pretragu je potrebno nastaviti u njemu. Ako je vrednost čvora sa kojim se vrši pretraga takva da je potrebno nastaviti pretragu u levom podstablu, tada se ovaj čvor dodaje u levo podstablo, a iz levog podstabla se briše najveći čvor. Ovaj najveći čvor se smešta

na mesto do koga je stigla pretraga, pa se pretraga nastavlja u desnom podstablu sa čvorom koji je bio na mestu do koga je stigla pretraga.

Po izvršenoj zameni, moguće je nastaviti binarnu pretragu ka prvom slobodnom mestu u poslednjem redu. Kada se dodje do prvogn slobognog mesta u poslednjem redu balansiranog stabla i u njega se doda čvor sa kojim se u tom trenututku vrši binarna pretraga, proces dodavanja čvora se završava.

Po završenom dodavanju čvora balansiranog stabla, ukoliko taj čvor nije u okviru opsega stabla, vrši se ponovni proračun granica opsega za to stablo. Granice opsega stabla čine vrednosti najmanjeg i najvećeg čvora stabla.

4.2.1.3 Premeštanje čvorova između balansiranih stabala

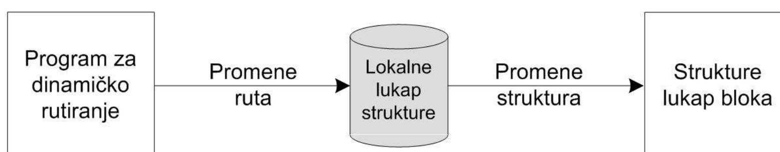
Da bi bilo moguće iskoristiti sve pozicije u balansiranim stablima u okviru BPFL nivoa, predviđeno je premeštanje čvorova između stabala sa susednim opsezima. Predviđeno je premeštanje čvorova u slučaju da razlika broja čvorova u balansiranim stablima bude veća od tri. Ovakvo premeštanje čvorova između stabala omogućava i ravnomernu raspodelu broja čvorova u balansiranom stablima, a time i uravnotežena vremena binarnih pretraga stabala.

Provera da li razlika između brojeva čvorova u susednim balansiranim stablima postala veća od tri vrši se posle svakog dodavanja čvora u stablo i brisanja čvora iz stabla. U slučaju kada se premeštanje vrši iz balansiranog stabla sa manjim vrednostima prefiksa u stablo sa većim vrednostima prefiksa, tada se iz stabla sa manjim prefiksima briše čvor sa najvećom vrednošću prefiksa, i taj čvor, zajedno sa pripadajućim podstablom i vrednostima izlaznih portova, se dodaje u stablo sa većim vrednostima prefiksa. Operacija brisanja čvora iz balansiranog stabla opisana je u odeljku 4.2.1.1, dok je operacija dodavanja čvora u stablo opisana u odeljku 4.2.1.2. Moguće je i premeštanje iz balansiranog stabla sa većim vrednostima prefiksa u stablo sa manjim vrednostima prefiksa, u kom slučaju se iz stabla sa većim vrednostima prefiksa premešta čvor sa najmanjom vrednošću prefiksa.

Po izvršenom premeštanju čvorova vrši se ažuriranje vrednosti granica opsega stabala koje su učestvovala u operaciji premeštanja čvora.

4.2.1.4 Organizacija programskog koda

Na Slici 4-14 prikazan je tok podataka pri izvršavanju algoritma ažuriranja lukap bloka. Rezultat programa za dinamički proračun tabela rutiranja, kao što je *ospfd* predstavljen u poglavlju 4.1.6, je tabela rutiranja i promene u tabeli rutiranja koje čine dodavanja, brisanja i izmene ruta. Algoritam za ažuriranje prihvata promene ruta i na osnovu njih i pristupa lokalnoj kopiji struktura podataka lukap bloka, vrši ažuriranje lokalne kopije i same strukture lukap bloka. Pri ažuriranju, tekuća vrednosti lukap struktura očitavaju se iz lokalne kopije, a lukap bloku se pristupa samo u prilikom upisa promenjenih vrednosti.



Slika 4-14 Tok podataka u toku ažuriranja lukap bloka

Funkcionalnost koja obuhvata dodavanje, brisanje i izmene izlaznog porta u okviru balansiranog stabla, podstabala i memorije koja sadrži vrednosti izlaznih portova obuhvaćena je u okviru klase *BalancedTrees*, čija se implementacija nalazi u fajlu balancedtrees.C. Funkcionalnost BPFL nivoa koja obuhvata pridruživanje prefiksa balansiranim stablima i premeštanje čvorova između stabala obuhvaćena je u klasi *Ranges*, čija se implementacija nalazi u fajlu ranges.C. Specijalni slučaj BPFL nivoa za prvih osam bita prefiksa, koji ne sadrži balansirana stabla i sadrži samo jedno podstablo i vrednosti izlaznih portova za rute u tom podstablu, implementiran je u fajlu levelzero.C.

Funkcija preko koje se pozivaju operacije brisanja, dodavanja i izmena izlaznih portova postojećih ruta naziva se *update_lookup*, i implementirana je u fajlu lookup.C. Argumenti ove funkcije su vrednost i dužina prefiksa, vrednost izlaznog porta, kao i tip operacija koju je potrebno izvršiti. Ova funkcija dalje koristi po jedan objekat klase *Ranges* za svaki BPFL nivo sem nultog, za koji koristi funkcije iz fajla nultinivo.C. Ova funkcija predstavlja interfejs koji program za formiranje ruta može koristiti za ažuriranje BPFL lukap tabele ako se kompajlira zajedno sa fajlovima koji sadrže implementaciju ažuriranja BPFL lukap tabela. U slučaju da zajedničko kompajliranje nije moguće (potpoglavlje 4.3), predviđeno je da se kompajlira samostalni program za

ažuriranje BPFL lukap tabela, i da zahteve za ažuriranje ruta prima posredstvom UDP poruka, što je opisano u potpoglavlju 4.3. Fajl koji sadrži fukciju *main* za samostalni program za ažuriranje BPFL lukap tabela naziva se tabela.C.

4.2.2 Algoritam ažuriranja za POLP lukap tabele

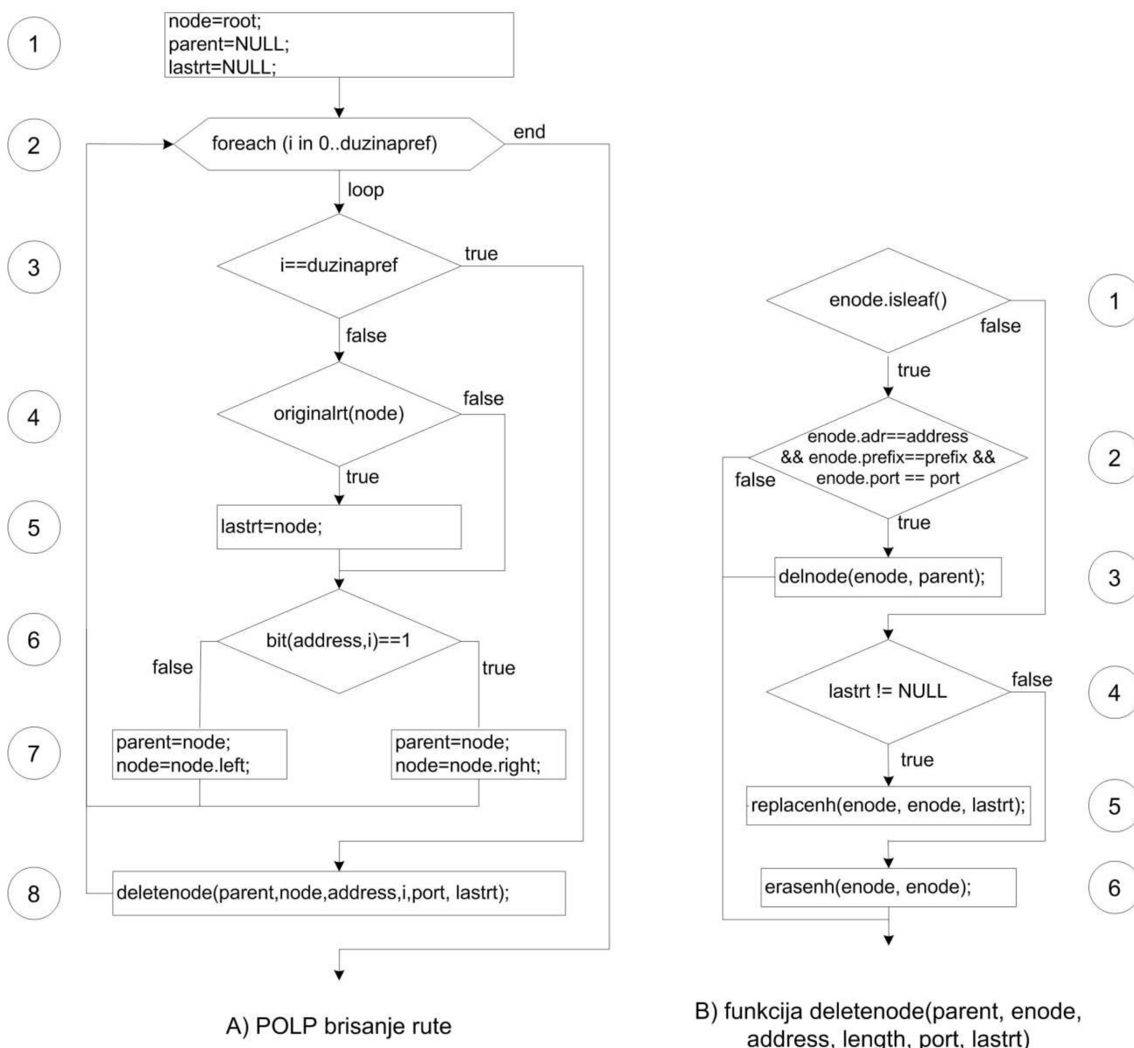
POLP lukap strukture podataka, opisane u poglavlju 3.1.2.9, zasnivaju se na jednobitskom stablu, u kome svaka grana stabla odgovara jednom bitu u okviru prefiksa. U cilju povećanja protoka paketa, POLP deli na određenoj dubini jednobitsko stablo na više podstabala, pridružuje svako podstablo jednom od pajplajna i raspoređuje čvorove stabla u memorijama pajplajna kojima je moguće pristupati istovremeno. U okviru implementacije algoritma ažuriranja za POLP lukap tabele, potrebno je postaviti vrednosti indeksa odredišnih podstabala, kao i memorijskih lokacija u okviru memorija pajplajna. Indeks odredišnih podstabala sadrži informacije o strukturi jednobitskog stabla do dubine na kojoj se vrši podela na podstabla, koja je konfigurabilna, pri čemu je podrazumevana vrednost za IPv4 osam bita, a za IPv6 tri bajta. Ovaj indeks sadrži sve vrednosti dela adrese iznad dubine deljenja koje se pojavljuju u prefiksima tabele rutiranja i pokazivače na pozicije prvog čvora podstabala u okviru pajplajna, i njegovo ažuriranje nije složeno. Čvorovi jednobitskog stabla ispod dubine deljenja raspoređuju se u okviru pajplajna, i moguće su razne varijante raspoređivanja, uz ograničenje da čvor ne može biti u istoj memoriji pajplajna kao njegov čvor-roditelj, već da mora biti u nekoj od memorija iza čvora-roditelja. Broj memorija pajplajna najmanje je jednak broju nivoa jednobitskog stabla ispod nivoa deljenja. Pajplajn omogućava istovremeno izvršavanje više zahteva za pronalaženje izlaznog porta za datu IP adresu tako što daje mogućnost istovremenog pristupa različitim memorijama, i sadrži pravilo da se čvor-dete uvek smešta u memoriju ispod čvora-roditelja. POLP lukap blok sadrži više pajplajna što uvećava broj lukap zahteva koji se istovremeno izvršavaju.

Implementacija POLP algoritma ažuriranja sadrži strukturu jednobitskog stabla na kojoj se vrše operacije dodavanja, brisanja i promena ruta. Svaki čvor u stablu sadrži i informacije o poziciji čvora u okviru memorije POLP pajplajna, i ovaj podatak se u toku proračuna ažurira u strukturi stabla. Pri ažuriranju strukture stabla, ažuriranju se indeksi odredišnih podstabala POLP bloka, i u memorijama pajplajna se ažuriraju adrese čvorova-dece i izlaznih portova. Implementacija sadrži i objekte koji

predstavljaju pajplajnove i njihove memorije, koji sadrže informacije kao što su broj slobodnih mesta u svakoj memoriji, stanja zauzetosti pozicija u memoriji i sledeća slobodna pozicija.

4.2.2.1 Brisanje rute

Osnovni deo algoritama za brisanje rute iz POLP lukap bloka prikazan je na Slici 4-15 A. Po dobijanju zahteva za brisanje rute, u koracima 1-7 algoritma na Slici 4-15 A, prolazi se kroz jednobitsko stablo, putanjom određenom vrednošću prefiksa koji se briše. Algoritam prikazan na Slici 4-15 implementiran je u okviru fajla leafpushing.C.



Slika 4-15 Brisanje rute (POLP)

Posle inicijalizacije promenljivih u prvom koraku algoritma, polazak kroz jednobitsko stablo se vrši u petlji prikazanoj u koraku 2 algoritma u okviru koje se

inkrementira promenljiva *i* koja predstavlja poziciju tekućeg bita u prefiksu. Svaka vrednost *i* manja od dužine prefiksa odgovara prolasku kroz jedan čvor stabla, i u svakom čvoru se u koraku 4 algoritma korišćenjem funkcije *originalrt* proverava da li čvor odgovara ruti iz tabele rutiranja. Funkcija *originalrt* jednostavno proverava da li je dužina rute zapisane u čvoru jednaka dubini čvora. Ako jeste, to znači da čvor odgovara ruti iz tabele rutiranja, za razliku od čvorova koji su kreirani u cilju formiranja putanje kroz stablo do čvora koji odgovara ruti, ili čvorovima koji su formirani u procesu potiskivanja ruta u grane stabla. Ako tekući čvor odgovara ruti iz tabele rutiranja, tada se pamti pokazivač na taj čvor u promenljivoj *lastrt*. Cilj je u petlji zapamtiti poslednji čvor na putnji koji odgovara ruti. Ruta zapamćenog čvora se koristi u procesu potiskivanja rute ka čvorovima ispod čvora koji odgovara brisanoj ruti, ako takvi čvorovi postoje.

U koracima 6 i 7 algoritma se, u skladu sa vrednošću bita brisane rute na poziciji *i*, vrednost pokazivača *node* postavlja na sledeći čvor na putu dok se pokazivač *parent* pridružuje tekućem čvoru, čime se iteracija petlje završava. Ovim premeštanjem pointera, prelazi se na jedno od čvorova-dece tekućeg čvora, zavisno od vrednosti bita na tekućoj poziciji prefiksa brisane rute.

U poslednjoj iteraciji petlje uslov u koraku 3 algoritma je ispunjen i u toj iteraciji se izvršava korak 8 algoritma u kome se poziva funkcija *deletenode*(*node *parent, node *enode, unsigned long address, unsigned long length, unsigned long port, unsigned long *lastrt*), čiji su argumenti čvor-roditelj čvora koji odgovara brisanoj ruti sa oznakom *parent*, zatim čvor koji odgovara brisanoj ruti sa oznakom *enode*, parametri brisane rute u koje spadaju vrednost prefiksa sa oznakom *address*, dužina rute sa oznakom *length*, izlazni port sa oznakom *port* i, kao poslednji parametar, pokazivač na poslednji čvor na putanji koji odgovara jednoj od ruta iz tabele rutiranja, sa oznakom *lastrt*.

Algoritam funkcije *deletenode* prikazan je na Slici 4-15 B. U prvom koraku se proverava da li je čvor koji odgovara brisanoj ruti list stabla. Ako je taj uslov ispunjen , i ako je ispunjen uslov sadržan u koraku 2 da je u čvoru zapisana ruta koja se briše izvršava se korak 3. U koraku 3 poziva se funkcija *delnode*(*node *current, node*

**parent*) koja briše čvor koji odgovara brisanoj ruti, i briše i sve čvorove iznad njega koji su u strukturi stabla suvišni posle njegovog brisanja.

Ukoliko čvor koji odgovara brisanoj ruti nije list stabla, tada se izvršavaju koraci 4-6 prikazani na Slici 4-15 B. Ovi koraci algoritma obezbeđuju da se u podstablu ispod čvora koji odgovara ruti uklone svi zapisi rute, i da se podstablo ažurira u skladu sa promenom u tabeli rutiranja. U koraku 4 se proverava da li na putanji do čvora koji odgovara brisanoj ruti postoji čvor koji odgovara nekoj drugoj ruti tabele rutiranja. Ako postoji, izvršava se korak 5 algoritma i funkcija *replacenh*(*node *current, node *enode, unsigned long *lastrt*), i vrednost rute tog čvora zamenjuje vrednost obrisane rute u svim čvorovima u kojima je do tada bila zapisana ruta koja se briše. Ukoliko na putanji do čvora koji odgovara brisanoj ruti nije postojala ni jedna ruta iz tabele rutiranja, tada se u koraku 6 poziva funkcija *erasenh*(*node *current, node *enode, unsigned long *lastrt*). Funkcija *erasenh* uklanja brisanu rutu iz čvora koji joj odgovara i svih čvorova ispod njega. Funkcije *replacenh* i *erasenh* rekurzivno prolaze kroz čvorove ispod čvora koji se briše i njihova prva dva argumenta su tekući čvor i čvor koji sadrži traženu rutu. U ove dve funkcije, svi čvorovi podstabla se posećuju korišćenjem algoritma pretrage stabla po dubini [31], i to tako što se u svakom čvoru funkcija prvo poziva za levi čvor-dete pa za desni čvor-dete. Pošto je u prvom pozivu ovih funkcija, prikazanom na Slici 4-15 B tekući čvor upravo čvor koji odgovara brisanoj ruti, prva dva argumenta u tom pozivu ovih funkcija imaju vrednost pokazivača na čvor koji odgovara brisanoj ruti. U rekurzivnim pozivima ovih funkcija, argument *current* uzima vrednost čvora do koga se stiglo pri pretrazi podstabla, dok argument *enode* sadrži parametre brisane rute, i omogućava utvrđuvanje da li čvor do koga se stiglo sadrži brisanu rutu. Ako čvor sadrži brisanu rutu, u slučaju funkcije *replacenh*, ova ruta se zamenjuje rutom sadržanom u parametru *lastrt*, dok se u slučaju funkcija *erasenh* čvor briše.

U toku ažuriranja jednobitskog stabla, sve potrebne izmene upisuju se u memorijske lokacije POLP bloka, na način opisan u poglavlju 4.3.2. U ovom poglavlju je opisano mapiranje adresa pridruženih lukap bloku u adresni prostor aplikacije koja vrši ažuriranje. Mapiranje omogućava upis podataka u lukap blok. U skladu sa Slikom 4-14, sve proračunate promene vrednosti memorijskih lokacija u POLP lukap bloku upisuju se u lukap blok.

4.2.2.2 Dodavanje rute

Algoritam dodavanja rute u POLP jednobitsko stablo prikazan je na Slici 4-16. Algoritam prikazan na Slici 4-16 implementiran je u okviru fajla leafpushing.C. Posle inicijalizacije promenljivih u prvom koraku, centralni deo algoritma čini petlja čiji je opseg vrednosti prikazan u koraku 2. Promenljiva petlje *i* prolazi kroz sve pozicije bita u prefiksu nove rute. U koraku 3 algoritma se proverava da li je vrednost promenljive *i* manja ili jednaka od dužine prefiksa. U slučaju da je manja izvršavaju se koraci algoritma na Slici 4-16, u kojima se prelazi jedan korak kroz stablo ka čvoru koji odgovara vrednosti prefiksa. U koraku 4 se proverava da li taj korak u stablu odgovara nuli ili jedinici u prefiksu nove rute, i u skladu sa tim, bira se leva ili desna grana od tekućeg čvora.

Koraci za levu i desnu granu su prikazani ispod uslova u koraku 4 i simetrični su, pa je dovoljno razmotriti slučaj prolaska levom granom, što odgovara slučaju bita 0 na poziciji *i* prefiksa. Prvo se, u koraku 5 proverava da li levi čvor postoji. Ukoliko ne postoji potrebno ga je napraviti. Pre pravljenja novog čvora stabla, u koraku 6 se proverava vrednost promenljive *newnode*, u kojoj se pamti da li se u toku kretanja kroz stablo do tog trenutka desilo da je bilo potrebno praviti novi čvor. Ako je vrednost promenljuve *newnode* jednaka *false*, radi se o prvoj situaciji u kojoj je potrebno napraviti novi čvor, i tada se u koraku 7 proverava da li je tekući čvor list, i ako jeste, pokazivač na njega se čuva u promenljivoj *tmpnode*, da bi se u koracima 16 i 17 izvršilo potiskivanje vrednosti rute iz tog čvora ka listovima. Ako tekući čvor, iza koga se prvi put pravi novi čvor nije list, tada nije potrebno vršiti potiskivanje rute ka listovima, zato što je to slučaj u kome čvor ima jedno dete-čvor i tada on u sebi nema sačuvanu rutu.

Prvi put kada je potrebno napraviti novi čvor, u koraku 9 se vrednost promenljive *newnode* postavlja na *true*, i ostaje sa tom vrednošću do kraja algoritma.

Ako sledeći čvor u prolasku ne postoji on se u koraku 10 kreira u funkciji *createnode*, čime se završava obrada slučaja u kome sledeći čvor ne postoji, koja je počela od koraka 6.

Slika 4-16 Dodavanje rute (POLP)

Kao što je opisano u poglavlju 3.1.2.9, u POLP algoritmu čvor-roditelj čuva jedan pokazivač na lokaciju na kojoj se jedan pored drugog nalaze oba njegova čvora-

deteta. Iz tog razloga se u koracima 11 i 12, po potrebi odmah pravi i drugi čvor-dete tekućeg čvora. Drugi čvor-dete se pravi samo u slučaju da je njegova dubina ispod dubine deljenja stabla na podstabla, jer se samo sadržaj podstabala čuva u pajplajnovima.

U poslednjoj iteraciji petlje prikazane u koraku 2, vrednost promenljive *i* jednaka je dužini prefiksa, i tada je uslov u koraku 3 ispunjen, pa se izvršavaju koraci 13-15. U koraku 13 se u čvor koji odgovara ruti postavljaju vrednost i dužina prefiksa, kao i vrednost izlaznog porta. U slučaju kada čvor koji odgovara ruti nije list stabla, u koraku 15 se poziva funkcija *pushnode*, koja izvršava guranje rute ka listovima stabla ispod čvora *node*.

Na kraju algorima se izvršavaju koraci 16 i 17 koji, u slučaju da je vrednost promenljive *tmpnode* postavljena u toku izvršavanja algoritma vrše potiskivanje rute sačuvane u čvoru na koga *tmpnode* pokazuje ka listovima stabla ispod tog čvora. Pokazivač *tmpnode* se postavlja pri prolazu kroz stablo, u trenutku kada se završava deo puta kroz postojeće čvorove i kada se počinje sa pravljenjem novih čvorova. Pokazivač *tmpnode* u tom slučaju pokazuje na poslednji čvor na putanji koji je postojao pre prolaska kroz stablo, i omogućava potiskivanje prefiksa od tog čvora ka čvorovima napravljenim u toku izvršavanja algoritma prikazanog na Slici 4-16.

Kao što smo ranije napomenuli, u toku izvršavanja algoritma dodavanja rute, novi čvorovi u jednobitskom stablu prave se pozivom funkcije *createnode*. U ovoj funkciji novom čvoru se podešava vrednost pokazivača na njegov roditeljski čvor, i ako se novi čvor nalazi na dubini deljenja na podstabla ili ispod nje, izvršava se dodavanje čvora u POLP pajplajn. Ako se novi čvor nalazi na dubini deljenja, tada on formira novo podstablo, koje se dodaje u POLP pajplajn u kome ima najviše slobodnog mesta. U slučaju da se čvor nalazi ispod dubine deljenja on se dodaje u pajplajn u kome se nalazi njegov čvor-roditelj, i to u prvu slobodnu memorijsku lokaciju ispod memorije u kojoj se nalazi čvor-roditelj. Prva sledeća slobodna lokacija pronalazi se linearnom pretragom. Prvo se proverava prvi stepen ispod stepena čvora-roditelja. Ako je on potpuno popunjen proverava se sledeći stepen i tako redom dok se ne pronađe stepen koji sadrži slobodnu lokaciju. U cilju bržeg izvršavanja ažuriranja, posle dodavanja čvora u stepen, u tom stepenu se pronalazi prva sledeća slobodna lokacija, i čuva se u

objektu klase *Pipeline* koji predstavlja pajplajn u kome se nalazi taj stepen. Time se pri sledećem dodavanju u taj stepen izbegava traženje slobodne lokacije počev od početka stepena. Za pronalaženje slobodne memorijske lokacije u stepenu pajplajna koriste se zapisi o zauzetosti memorijskih lokacija u okviru pajplajna, koji se takođe čuvaju u objektu klase *Pipeline*. Ovi zapisi se ažuriraju prilikom svakog dodavanja ili brisanja čvora u pajplajnu.

U slučaju kada ne postoji slobodna memorija pajplajna ispod memorije čvora-roditelja, funkcija *shiftnodes*(*node *cnode*) se koristi traženje mogućnosti da se oslobodi mesto u pajplajnu za novi čvor premeštanjem čvorova koji se u stablu nalaze iznad tog čvora u memorije koje odgovaraju višim stepenima pajplajna. Čvor *cnode* predstavlja čvor stabla za koga treba osloboditi mesto u pajplajnu. Mesto je moguće osloboditi ako između čvorova na putanji od korena stabla do čvora-roditelja od čvora *cnode* postoje preskočene memorije pajplajna, zato što su pri dodavanju prethodnih čvorova nije bilo slobodnih mesta u tim memorijama. Pošto je moguće da su se u međuvremenu pojavila slobodna mesta u nekim od preskočenih memorija, funkcija *shiftnodes* pronalazi takve memorije. Kada pronađe preskočenu memoriju u kojoj su se pojavila slobodna mesta, funkcija *shiftnodes* pomera prvi čvor ispod preskočene memorije u preskočenu memoriju, i sve čvorove ispod premešta na lokacije njihovih čvorova-roditelja, čime se oslobađa mesto za novi čvor.

U okvoru implementacije, sve navedene promene u pajplajnovima ažuriraju se u objektima klase *Pipeline*, i ažuriraju se u samoj memoriji POLP lukap bloka. Ovakvo ažuriranje memorija POLP lukap bloka je inkrementalno i vrši se pri svakom dodavanju i brisanju rute. Pored inkrementalnog ažuriranja implementacija sadrži i ponovni proračun celog pajplajna, kao i svih pajplajna istovremeno.

4.2.2.3 Proračun celog pajplajna

U toku iterativnog procesa dodavanja ruta u POLP lukap blok (pododeljak 3.1.2.9), algoritam predviđa mogućnost smeštanja čvorova-dece u jednu od memorija ispod čvora-roditelja. U slučaju da je prva memorija ispod čvora-roditelja zauzeta, traži se slobodno mesto u drugoj memoriji ispod čvora roditelja. Ako je i druga memorija ispod čvora-roditelja zauzeta traži se slobodno mesto u trećoj i tako dok se ne nađe slobodno mesto, ili dok se ne pretraže sve memorije ispod čvora roditelja.

Dodavanjem čvorova rute u memorije sa slobodnim mestima dovodi do toga da neke memorije mogu biti preskočene. Ako se pri dodavanju rute pri prolasku kroz stablo koje je opisano u poglavlju 4.2.2.2 preskaču memorije, moguće je da ne ostane dovoljno memorija za pravljenje novih čvorova potrebnih za predstavljanje nove rute, pri čemu je moguće da su se u međuvremenu oslobodila mesta u gornjim memorijama. Iterativni metod dodavanja ruta može korišćenjem funkcije *shiftnodes* opisane u poglavlju 4.2.2.2 da iskoristi slobodna mesta u gornjim memorijama ako su ta slobodna mesta nastala u preskočenim memorijama. Međutim, slobodna mesta koja postoje u memorijama u kojima se nalaze čvorovi-preci u stablu ne mogu biti iskorišćena, i tada iterativni metod ne može izvršiti dodavanje rute. Iz tog razloga vrši se dodatni proračun koji će drugačije rasporediti čvorove i na taj način potencijalno omogućiti dodavanje nove rute.

U slučaju kada iterativni metod ne može da izvrši dodavanje nove rute, program za ažuriranje POLP lukap tabele ima još dve operacije kojima pokušava dodavanje rute. U slučaju da ruta nije mogla biti dodata pozivom *shiftnodes* funkcije, prvo se pokušava proračun celog pajplajna. U ovome proračunu se brišu sve dodele čvorova jednobitskog stabla memorijama POLP lukap bloka, i dodavanje se vrši iz početka, uzimajući u obzir visinu ispod svakog čvora koji se dodaje i dajući proritet čvorovima koji imaju veću visinu ispod sebe, tj. veći broj nivoa stabla ispod sebe. U postupku ponovnog proračuna održavaju se liste čvorova *readylist* i *nextreadylist*, sortirane po visini čvorova od čvorova sa najvećom visinom ka čvorova sa najmanjom visinom. U prvom koraku se u *readylist* dodaju koreni svih podstabala dodeljenih pajplajnu. Zatim se ovi čvorovi pridružuju prvoj memoriji pajplajna, i pri svakom dodavanju čvora, u *nextready* listu se dodaju čvorovi-deca tog čvora. Kada se završi popunjavanje jedne memorije, bilo zato što je *readylist* ispražnjena ili zato što u memoriji više nema mesta, vrši se premeštanje čvorova iz *nextready* u *readylist* i prelazi se na popunjavanje sledeće memorije. Na ovaj način se memorije pajplajna popunjavaju uzimajući u obzir visine čvorova, i pronalazi se raspored u kome svi čvorovi mogu stati u pajplajn, ako takav raspored postoji.

Pri proračunu celog pajplajna, memorijama POLP bloka se dodeljuju čvorovi iz svih podstabala dodeljenih tom pajplajnu. Ako ovaj proračun ne uspe, i dalje je moguće da postoji raspored čvorova po pajplajnima koji omogućava smeštanje kompletnog stabla, pod uslovom da se izvrši drugačije dodeljivanje podstabala pajplajnima.

Kao što je opisano u poglavlju 4.2.2.2, u toku iterativnog dodavanja ruta, novo podstablo se dodeljuje pajpalajnu u kome u trenutku dodavanja podstabla ima najviše slobodnog mesta. U toku kasnijeg dodavanja ruta moguće je da podstabla dodeljena nekom pajplajnu porastu više nego podstabla dodeljena drugom pajplajnu, pa da u jedan pajplajn ne bude moguće dodati novu rutu iako u drugom pajplajnu ima slobodnog mesta. Iz tog razloga, ako proračun celog pajplajna ne uspe, vrši se ponovno dodeljivanje podstabala pajplajnovima, i posle toga ponovni proračun svih pajplajna. Pri dodeljivanju postabala, svako podstablo se dodaje u pajplajn koji ima najviše slobodnog mesta. Podstabla se dodaju redom, po opadajućem broju čvorova. Na ovaj način se postiže ravnomerna raspodela broja čvorova po pajplajnima, i omogućava dodeljivanje čvorova memorijama, koje nije bilo moguće pre ponovnog proračuna svih pajplana.

4.2.2.4 Organizacija programskog koda

Tok podataka u programu za ažuriranje POLP lukap bloka predstavljen je na Slici 4-14 predstavljenoj u pododeljku 4.2.1.4. Program za ažuriranje prihvata zahteve za dodavanje i brisanje ruta, na osnovu njih proračunava promene u POLP memorijama, i zatim te promene upisuje u POLP lukap blok. Proračun promena u POLP memorijama se vrši na osnovu lokalne kopije POLP lukap bloka, čime se izbegava potreba za pristup POLP memorijama u cilju očitavanja njegovog sadržaja.

Implementacija POLP algoritma ažuriranja sadržana je u klasama *Node*, *Subrtie*, *Pipeline*, *Pipelines* i *StageElement*. Sve ove klase sadržane su u fajlu leafpushing.C. U fajlu tabela.C sadržana je funkcija *update_lookup*, koja kao argumente prihvata parametre nove rute i identifikaciju da li se vrši dodavanje ili brisanje rute.

U okviru implementacije, čvorovi jednobitskog stabla modelirani su objektima klase *Node*. U ovim objektima sadržane su informacije o POLP memoriji kojoj je čvor pridružen i položaju čvora u toj memoriji, kao i informacije o pridruženoj ruti, dubini i visini čvora i broju čvorova-potomaka. Objekat klase *Node* sadrži i pokazivače na svoje čvorove-decu i na čvora-roditelja. Ove informacije omogućavaju efikasno izvršavanje operacija na stablu opisanih u poglavljima 4.2.2.1 i 4.2.2.2.

Objekat klase *Subtrie* predstavlja jedno podstablo i sadrži informacije o korenu i prefiksu podstabla kao i o broju čvorova u podstablima. Informacija o broju čvorova u podstablima se koristi pri sortiranju podstabala u opadajućem redosledu, potrebnom u

toku operacije proračunavanja celog pajplajna, opisane i poglavlju 4.2.2.3. Lista objekata klase *Subtrie* se koristi u toku proračuna svih pajplajnova, pri pridruživanju podstabala pajplajnovima, što je opisano u poglavlju 4.2.2.3. Svaki objekat tipa *Pipeline* sadrži listu objekata tipa *Subtrie* koji predstavlja njegova podstabla. Pored liste podstabala, objekat klase *pipeline* sadrži i broj sadržanih čvorova. Funkcije ove klase omogućavaju dodavanje čvorova pajplajnima i njihovo brisanje.

Objekat tipa *Pipelines* sadrži objekte tipa *Pipeline* i omogućava izvršavanje operacije dodavanja i brisanja čvorova na celom POLP bloku tako što na osnovu informacija u objektu tipa *Node* pronalazi odgovarajuće pajplajne kojima prosleđuje zahteve za izvršavanje operacija dodavanja i brisanja čvora. Ove operacije se pozivaju kada se doda ili briše čvor korišćenjem algoritama opisanih u poglavljima 4.2.2.1 i 4.2.2.2, i u toku tih operacije se ažuriraju podaci sadržani u objektu klase *Pipeline*. U ovu klasu spadaju lista podstabala, broj sadržanih čvorova u pajplajnu, veličine stepena, prva slobodna memorijska lokacija u svakom stepenu, i, za svaku memorijsku lokaciju u stepenima, indikacija da li je slobodna ili zauzeta. Objekti klase *StageElement* predstavljaju elemenate lista *readylist* i *nextreadylist* opisanih u poglavlju 4.2.2.3.

4.2.3 Algoritam ažuriranja lukap bloka sa prioritetnim stablom

Kao što je opisano u poglavlju 3.1.2.8, prioritetno stablo se zasniva na ideji da se prazni čvorovi jednobitskih stabala popune prefiksima iz nižih nivoa stabla. U slučaju kada čvor u koga je smešten prefiks iz nižeg nivoa stabla na putanji ispod sebe nema prefikse koji su duži od tog prefiksa, tada se čvor označava kao prioritetni. Kada se vrši pretraga prioritetnog stabla i kada adresa pretrage ima poklapanje sa prefiksom iz prioritetnog čvora, tada je nađen prefiks sa najdužim poklapanjem i pretraga se završava.

Na Slici 4-17 prikazani su algoritmi za dodavanje i brisanje rute u prioritetnom stablu. Algoritam brisanja rute je jednostavniji i prikazan je na Slici 4-17 A. Algoritmi prikazani na Slici 4-17 implementirani su u fajlu prioritytrie.C. U koraku 1 algoritma brisanja rute pronalazi se čvor koga treba obrisati, korišćenjem funkcije *search*. Argument funkcije *search* je objekat *prefix* koji sadrži informacije o ruti u koje spadaju prefiks, dužina maske i izlazni port. Funkcija *search* prolazi kroz stablo i pri svakom grananju odlučuje o izboru putanje na osnovu vrednosti bita prefiksa čija je pozicija

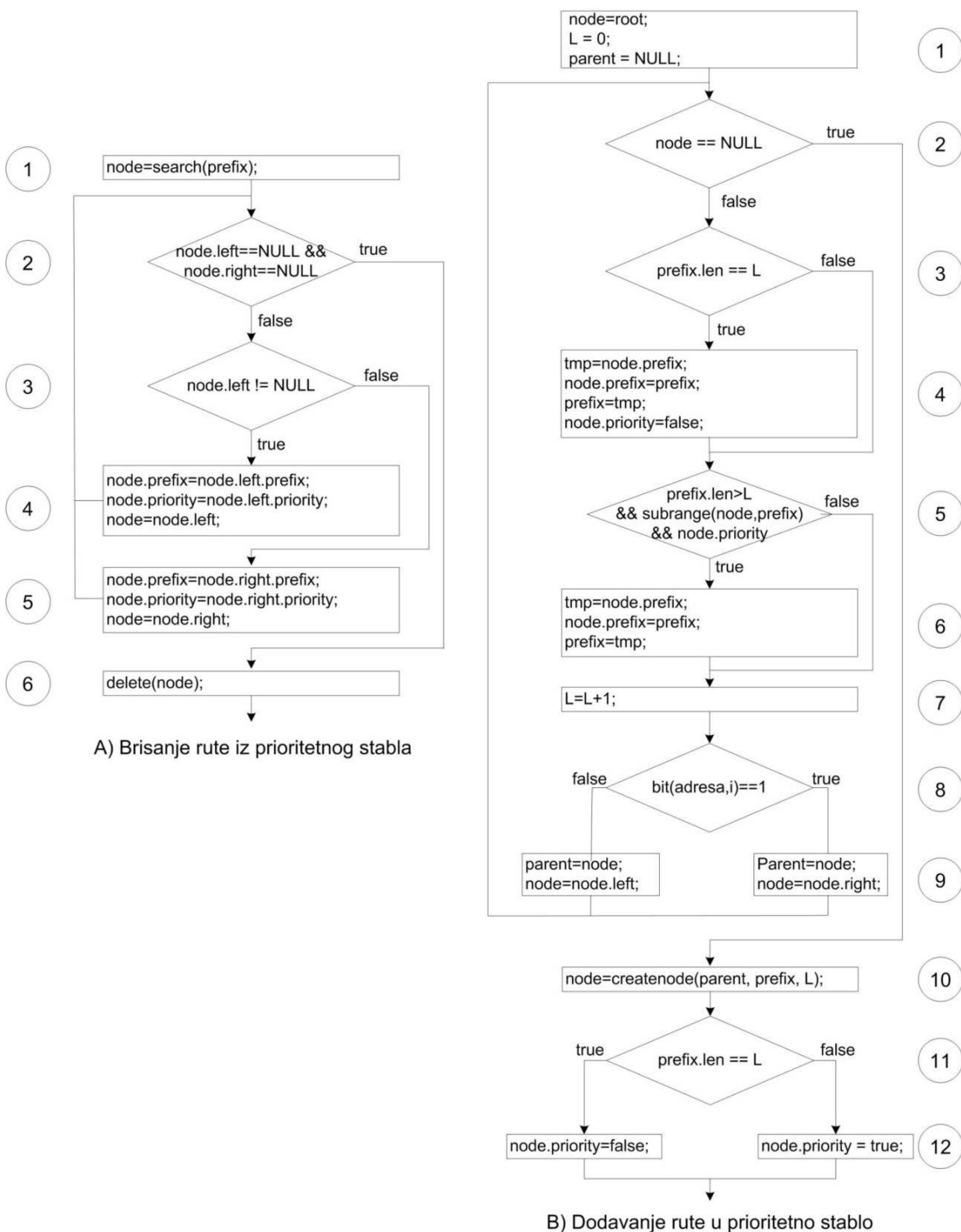
jednaka dubini do koje je pretraga stigla. Pri svakom grananju prefiks se poredi sa prefiksom sačuvanim u čvoru dok se ne pronađe poklapanje, čime se pretraga završava.

Pošto je čvor koga treba obrisati pronađen algoritam ulazi u petlju koju čine koraci 2-5. U koraku 2 se proverava da li tekući čvor (inicijalno čvor koji se briše) ima čvorove-decu. Ako ima, tada se na mesto tekućeg čvora premešta jedan od njegovih čvorova-dece. U koraku 3 se proverava da li levi čvor-dete postoji, i ako postoji, u koraku 4 se on premešta u tekući čvor. Ukoliko levi čvor-dete ne postoji, u obrisani čvor se premešta desni čvor-dete. Pri premeštanju čvora-deteta u roditeljski čvor prepisuje se vrednost prefiksa i informacija da li je čvor prioritetni. Pri premeštanju se zadržava prioritet, zato što u slučaju čvora koji nije prioritetan ne može biti garancije da ispod njega nema dužih prefiksa. Čvor ne može biti prioritetan ako ispod njega ima dužih prefiksa, zato što se prilikom poklapanja adrese sa prioritetnim prefiksom pretraga završava, koristeći informaciju da ispod prioritetnih prefiksa nema dužih prefiksa. U koracima 4 i 5 vrednost promenljive *node* dobija vrednost čvora-deteta koji je premešten, pa se u sledećoj iteraciji petlje vrši brisanje tog čvora-deteta. Petlja koja se sastoji iz koraka 2-5 redom premešta čvor-dete u čvora-roditelja dok se ne stigne do lista stabla kada će uslov u koraku 2 biti ispunjen, pa će list biti obrisan u koraku 6 čime se algoritam brisanja prefiksa završava.

Algoritam dodavanja rute u prioritetno stablo prikazan je na Slici 4-17 B. Pri dodavanju rute se prolazi kroz stablo u skladu sa bitima prefiksa nove rute. Pri prolasku kroz stablo, dužina prefiksa nove rute poredi se sa dužinom prefiksa sačuvanom u svakom prioritetnom čvoru. Kada se naiđe na odgovarajući prioritetni čvor sa kraćim prefiksom ili ako se dođe do dubine jednake dužini prefiksa nove rute, nova ruta se upisuje u tekući čvor, i prolazak kroz stablo se na isti način nastavlja sa rutom koja je do tada bila sadržana u tekućem čvoru. Kada se stigne do lista stabla i završi njegova provera, formira se novi čvor u skladu sa odgovarajućim bitom prefiksa rute sa kojom se u tom trenutku vrši pretraga, i ta ruta se smešta u novi čvor.

Pre prolaska stabla, u koraku 1 se promenljiva *node* koja predstavlja tekući čvor inicijalizuje da ukazuje na koren stabla, dok promenljiva *L* koja označava trenutnu dubinu u toku prolaska kroz stablo dobija vrednost 0. Vrednost promenljive *parent* u

toku pretrage pokazuje na čvora-roditelja tekućeg čvora i inicijalno se postavlja da ne ukazuje ni na jedan čvor.



Slika 4-17 Dodavanje i brisanje rute u prioritetnom stablu

U petlji koju čine koraci 2-9 prolazi se kroz stablo, i petlja se izvršava sve dok se ne dođe do lista stabla. Prolazak kroz stablo se vrši na osnovu bita prefiksa koji se

dodaje, i u svakom koraku se vrše dve provere prikazane u koracima 3 i 5. Ako neka od provera uspe, tada se prefiks snima u čvor do koga se stiglo, a vrednost prefiksa se menja i postaje vrednost koja je bila smeštena u čvoru. Dalja pretraga se vrši na osnovu nove vrednosti prefiksa, i završava se kada se stigne do lista stabla, što se proverava u uslovu 2.

Dve provere koje u slučaju pozitivnog ishoda rezultuju smeštanjem prefiksa u tekući čvor i promenom vrednosti prefiksa su prikazane u koracima 3 i 5. U koraku 3 se proverava da li tekuća dubina odgovara dužini prefiksa, i ako odgovara prefiks treba smestiti u tekući čvor jer je to poslednji čvor u koji je moguće smestiti taj prefiks. Pošto se prefiks smešta na dubinu stabla koja odgovara njegovoj dužini, u koraku 4 se specificira da čvor u koji se prefiks smešta nije prioritetni. U koraku 5 se proverava da li je dužina prefiksa veća od tekuće dubine, da li je prefiks smešten u tekućem čvoru uključen u prefiks koji se dodaje i da li je čvor prioritetni. Ako su ovi uslovi ispunjeni, prefiks sa kojim se vrši pretraga se snima u čvor i nastavlja se sa dodavanjem prefiksa koji je bio u čvoru. Pri tome čvor ostaje prioritetni. Navedeni uslovi garantuju da će struktura prioritetnog stabla i dalje biti takva da u podstablu ispod svakog prioritetnog prefiksa ne postoje duži prefiksi. Time je omogućeno da se pretraga završi kada dođe do poklapanja sa prioritetnim čvorom.

Prolazak kroz stablo se završava kada se u koraku 2 utvrdi da se došlo do prazne pozicije. Tada se izvršavaju koraci algoritma 10-12. U koraku 10 se kreira novi čvor korišćenjem funkcije *createnode*. Argument ove funkcije je promenljiva *parent* koja pokazuje na čvor do koga se došlo u prolasku kroz stablo. Ukoliko promenljiva *parent* ne ukazuje ni na jedan čvor, tada stablo nije formirano i kreira se čvor u korenu stabla. Ako *parent* pokazuje na neki čvor, kreira se čvor-dete i u njega se smešta vrednost prefiksa. U koracima 11 i 12 se utvrđuje da li je čvor prioritetni na osnovu provere da li je formiran na dubini stabla koja odgovara njegovoj dužini ili na manjoj dubini, i algoritam dodavanja čvora se završava.

4.2.3.1 Organizacija programskog koda

Implementacija algoritma ažuriranja prioritetnog stabla je relativno kratka i smeštena je u fajl prioritytrie.C. Pored funkcija korišćenih za operacije dodavanja i brisanja prefiksa opisanih u poglavlju 4.2.3, implementacija algoritma ažuriranja može

sadržati i *main* funkciju i implementaciju soketa preko koga može primati zahteve za uspostavljinje i raskidanje ruta od programa koji izvršava dinamički protokol rutiranja. Povezivanje programa koji izvršava dinamički protokol rutiranja sa softverom za ažuriranje detaljnije je opisano u odeljku 4.3.1. Način pristupa programa za ažuriranje lukap bloku opisan je u odeljku 4.3.2. Tok podataka prilikom ažuriranja strukture prioritetnog stabla isti je kao pri ažuriranju BPFL i POLP lukap blokova i prikazan je na Slici 4-14 u poglavlju 4.2.1.4. Po prijemu podataka o novoj ruti ili brisanju rute, program za ažuriranje prioritetnog stabla na osnovu lokalne strukture prioritetnog stabla izračunava promene, i te promene upisuje u lukap blok i u lokalnu strukturu prioritetnog stabla.

4.3 Integracija softverskih komponenti u okviru rutera

Softver za ažuriranje lukap bloka sa jedne strane dobija podatke o promenama tabela rutiranja od programa koji izvršava dinamički protokol rutiranja, i sa druge strane vrši upis u registre lukap bloka kako bi sadržaj lukap bloka odgovarao tabeli rutiranja.

4.3.1 Integracija softvera za rutiranje i softvera za ažuriranje

Implementacije softvera za ažuriranje BPFL i POLP lukap blokova i lukap bloka prioritetnog stabla, opisanih u poglavlju 4.2, urađene su u C++ programskom jeziku.

Komunikacija softvera za rutiranje i softvera za ažuriranje sastoji se od prenosa informacija o kreiranju ili brisanju rute, u smeru od softvera za rutiranje ka softveru za ažuriranje. U svakoj od tri opisane implementacije softvera za ažuriranje (prema BPFL, POLP algoritmu, i algoritmu prioritetnog stabla) u fajlu tabela.C nalazi se funkcija *update_lookup*(*unsigned long address, unsigned long prefixlength, unsigned long outputport,int routetype*), čiji su argumenti podaci koji opisuju rutu u koje spadaju vrednost prefiksa sa oznakom address, vrednost maske sa oznakom *prefixlength* i vrednost izlaznog porta sa oznakom *outputport*, kao i parametar koji sadrži informaciju da li se radi o operaciji kreiranja ili brisanja rute, sa oznakom *routetype*. U slučaju kada je i dinamički protokol rutiranja pisan u C++ programskom jeziku, integracija tog softvera sa softverom za ažuriranje vrši se povezivanjem na nivou programskog koda i zajedničkim kompajliranjem u jedan program. U slučaju kada je protokol za rutiranje pisan u C programskom jeziku, i dalje je moguće formirati jedan program, tako što se

C++ biblioteka kompajlira g++ kompajlerom, C kod se kompajlira gcc kompajlerom, pa se dobijeni objektni fajlovi povežu u jedan program korišćenjem g++ linkera.

Algoritmi ažuriranja za BPFL i POLP tabele rutiranja spojeni su sa *ospfd* implementacijom OSPF protokola [56], pozivom funkcije *update_lookup* iz funkcije *sys_install* koja pripada klasi *INrte*, i nalazi se u fajlu spfcalc.C. U *ospfd* softveru, klasa *INrte* sadrži podatke koji definišu jedan unos u tabeli rutiranja, i funkciju *sys_install* koristi za kreiranje i brisanje rute u lukap blok rutera.

U originalnom ospdf softveru funkcija *sys_install* kreira i briše rutu u kernelu *Linux* rutera, i za to, posredstvom funkcija *rtadd* i *rtdel* klase *LinuxOspfd*. Funkcije *rtadd* i *rtdel* vrše kreiranje i brisanje rute u *Linux* ruteru korišćenjem sistemskog poziva *ioctl*, čiji je prvi argument deskriptor soketa, drugi argument vrednošću SIOCADDRT označava da se radi o dodavanju rute, a vrednošću SIOCDELRT označava da se radi o brisanju rute. Treći argument funkcije *ioctl* je struktura tipa *rtentry* koja sadrži vrednosti prefiksa, maske i izlaznog porta. Softver za ažuriranje lukap tabele je moguće kompajlirati i kao odvojen program, kome je moguće slati poruke za kreiranje i brisanje ruta iz programa za dinamičko rutiranje. U okviru implementacija protokola ažuriranja predviđeno je da program za ažuriranje može primati UDP poruke koje sadrže prefiks, adresu i izlazni port rute, kao i komandu za brisanje i dodavanje rute. Za slučaj kada je dinamički protokol rutiranja napisan u C ili C++ programskom jeziku napisana je funkcija *requestupdate*(*unsigned long address, unsigned long prefixlength, unsigned long outputport,int routetype*) koja formira UDP paket i šalje ga programu za ažuriranje. Ova funkcija nalazi se u fajlu updatelookup.C koga je potrebno uključiti u program za dinamički protokol rutiranja, dok je funkciju *requestupdate* potrebno pozvati u tom programu svaki put kada se dodaje ili briše ruta. Druga funkcija koju je potrebno implementirati u dinamičkom protokolu rutiranja je *writetohardware*(*unsigned int adresa, unsigned int vrednost*). Ova funkcija se poziva sa adresom registra lukap bloka i vrednošću koju treba upisati u taj registar. Poziv se vrši iz fajla updatelookup.C kada od programa za ažuriranje UDP porukama stignu informacije o potrebnim upisima u lukap blok. Dodavanje fajla updatelookup.C, poziva funkcije *requestupdate* i implementacije funkcije *writetohardware* su jedine potrebne izmene u programu za dinamičko rutiranje. Pošto se komunikacija programa za dinamički protokol rutiranja i programa za ažuriranje odvija na istom računaru, UDP poruke se neće gubiti. U cilju uvođenja

sigurnog prenosa moguće je porukama dodati i redni broj, i mogućnost traženja ponovnog slanja u slučaju preskočenog rednog broja na prijemu.

4.3.2 Povezivanje ravni podataka i kontrolne ravni

Zadatak softvera za ažuriranje lukap bloka je da na osnovu tabele rutiranja izvrši upise u registre lukap bloka, koji će rezultovati konfiguracijom lukap bloka u skladu sa tabelom rutiranja. U toku rada dinamičkog protokola rutiranja, rute se kreiraju i brišu, i softver za ažuriranje treba da izračuna potrebne promene konfiguracije lukap bloka i da izvrši potrebne upise u njega. U ruterima se lukap blok nalazi u posebnom čipu, čiju registri su dostupni sa procesora koji izvršava program za ažuriranje lukap bloka. Registrima lukap bloka su dodeljene adrese u okviru adresnog prostora procesora. Ove adrese je moguće mapirati u adresni prostor operativnog sistema, ili adresni prostor programa za ažuriranje lukap bloka.

U slučaju mapiranja adresnog prostora lukap bloka u adresni prostor operativnog sistema, moguće je napisati drajver koji vrši upise u lukap blok. U tom slučaju program za ažuriranje lukap bloka može da koristi upise u drajver, koje bi drajver dalje prosleđivao u lukap blok.

U slučaju mapiranja adresnog prostora lukap bloka u adresni prostor programa za ažuriranje, program može direktno vršiti upise u lukap blok. U okviru *ospfd* implementacije je izvršeno mapiranje adresnog prostora lukap bloka u adresni prostor aplikacije, i implementirani su upisi u registre lukap bloka u cilju konfigurisanja lukap bloka u skladu sa vrednostima u tabeli rutiranja.

Mapiranje memorije se u *Linux* operativnom sistemu vrši korišćenjem fajla */dev/mem* koji se otvara korišćenjem funkcije *open*:

```
mem_fd = open("/dev/mem", O_RDWR);
```

Mapiranje memorije se vrši pozivom funkcije *mmap*:

```
ext_mem = (unsigned char *)mmap(
        NULL,
        EXTMEM_SIZE,
        PROT_READ|PROT_WRITE,
        MAP_SHARED,
        mem_fd,
        EXTMEM_BASE
);
```

Funkcija *mmap* vraća pokazivač na početak adresa na koje su mapirani registri lukap bloka. Prvi argument funkcije *mmap* je *NULL*, čime se naznačava da registri lukap bloka mogu biti mapirani na proizvoljnu adresu, tj. da vraćena adresa *ext_mem* može biti proizvoljna. U slučaju da prvi argument funkcije *mmap* nije jednak *NULL*, tada se vrednost prvog argumenta uzima kao željena adresa na koju treba izvršiti mapiranje. Drugi argument funkcije *mmap* je veličina memorije koja se mapira, treći argument definiše da je moguće čitanje i pisanje mapirane memorije, dok četvrti argument dozvoljava da i drugi programi mapiraju isti memorijsku opseg. Peti argument funkcije *mmap* je deskriptor otvorenog /*dev*/*mem* fajla, dok je poslednji argument početna adresa memorije lukap bloka u adresnom prostoru procesora. Definicija pokazivača *ext_mem* i njegova inicijalizacija korišćenjem funkcije *mmap* dodati su u fajl ospf.C *ospfd* programa.

Dobijeni pokazivač *ext_mem* pokazuje na početak memorijskog opsega u koga je mapiran adresni prostor lukap bloka, i upis u neki od sledećih *EXTMEM_SIZE* bajtova od tog pointera predstavlja upis u neki od registara lukap bloka, čime je izvršena sprega između programa za ažuriranje i lukap bloka.

5 Zaključak

Optimizovano dvofazno balansiranje bazirano na rutiranju putanjama istih cena omogućava unapređenje iskorišćenja data centra. S obzirom da savremeni data centri imaju ogroman broj mrežnih elemenata i da se taj broj stalno povećava, unapređenje iskorišćenja mreža data centara donosi značajne uštede. LB-ECR protokol za rutiranje vrši optimizaciju korišćenjem linearnog programiranja i kompleksnost njegovog modela omogućava korišćenje u velikim data centrima. U prvom delu ove disertacije je analitičkim i simulacionim poređenjem sa direktnim i dvofaznim protokolima rutiranja pokazano da je maksimalni garantovani protok bez gubitaka pri LB-ECR rutiranju veći ili, u nekim slučajevima, jednak protoku drugih ispitivanih rutiranja. I pored toga što se u nekim slučajevima protoci poklapaju, to poklapanje je posledica specifičnosti pojedinih topologija, dok LB-ECR predstavlja algoritam za optimizaciju protoka u svim posmatranim topologijama za date saobraćajne zahteve svičeva.

U disertaciji je opisan način proračuna najopterećenijeg linka u mreži i protoka pri kome dolazi do gubitaka paketa. Simulacija rutiranja u data centrima korišćenjem *ns*-3 mrežnog simulatora je prikazala ponašanje poređenih protokola rutiranja u širokom opsegu intenziteta saobraćaja. U skladu sa očekivanjima, pravilne topologije data centara sa alternativnim putanjama jednake cene pokazale su se pogodnim za primenu protokola rutiranja baziranih na dvofaznom balansiranju. U *dragonfly* topologiji LB-ECR je imao najviši garantovani protok bez gubitaka, koji je bio za 6% bolji od rezultata LB-SPRm protokola. U *flattened butterfly* topologiji LB-ECR je najviši garantovani protok bez gubitaka delio sa Valiant rutiranjem, dok je treći protokol bio LB-SPRm sa 30% nižim garantovanim protokom. U *fat-tree* topologiji svi dvofazni protokoli rutiranja, uključujući LB-ECR, imali su isti garantovani protok bez gubitaka koji je bio za približno 30% veći od protoka Valiant rutiranja.

U disertaciji je obrađena i tema ažuriranja lukap tabela rutera. U okviru razmatranja performansi BPFL i POLP algoritama ažuriranja opisan je proces generisanja tabele rutiranja koja rezultuje maksimalnim zauzećem memorije i izvedene su formule za proračun maksimalnog zauzeća memorije, za dva lukap algoritma visokih performansi. Pored maksimalnog zauzeća memorije, simulacijom je izmereno zauzeće

memorije za realne tabele rutiranja. Ovi rezultati se mogu iskoristiti kao osnova za projektovanje veličina memorija lukap bloka.

Merenja zauzeća memorije za realne tabele rutiranja pokazalo je da BPFL zahteva duplo manje memorije od POLP algoritma u slučaju IPv4 protokola, i približno sedam puta manje memorije u slučaju IPv6 protokola. Pri ispitivanju performansi algoritama ažuriranja meren je i broj upisa u memorije lukap bloka u toku ažuriranja. Ovaj podatak je bitan pošto upisi u memoriju lukap bloka mogu dovesti do pauza u čitanju podataka iz tih memorija, i time do pauza pri prosleđivanju paketa koje mogu biti kritične u ruterima velikih brzina. Merenje broja pristupa u simuliranoj mreži sa relativno malim tabelama rutiranja pokazalo je da BPFL ima približno duplo manje pristupa lukap bloku od POLP lukap procedure. Sa druge strane, merenja broja pristupa u slučaju većih tabela rutiranja pokazala su da POLP ima manji broj pristupa. Ovi rezultati merenja broja upisa objašnjeni su osobinama POLP i BPFL struktura podataka. Ispitivano je i vreme izvršavanja algoritma ažuriranja od koga zavisi izbor procesora rutera, i ono je je potvrdilo linernu kompleksnost algoritma ažuriranja POLP lukap bloka, kao i kvadratnu kompleksnost algoritma ažuriranja BPFL lukap bloka.

Disertacija sadrži i opise implementacija rutiranja sa balansiranjem i algoritama ažuriranja korišćenih za dobijanje simulacionih rezultata. Ove implementacije namenjene su za primenu u okviru prototipa Internet rutera koji je razvijen na Elektrotehničkom fakultetu.

Reference

[1] L. Valiant and G. Brebner, "Universal schemes for parallel communication," *Proceedings of the 13th annual symposium on theory of computing*, pages 263-277, May 1981.

[2] M. Kodialam, T. V. Lakshman and S. Sengupta, "Maximum Throughput Routing of Traffic in the Hose Model," *INFOCOM 2006*, Barcelona, Spain, April 23-29, 2006.

[3] M. Antić and A. Smiljanić, "Oblivious Routing Scheme Using Load Balancing Over Shortest Paths," *Proceedings of the ICCC '08,* Bejing, China, May 19-23, 2008.

[4] D. Abts and J. Kim, "High Performance Datacenter Networks: Architectures, Algorithms, and Opportunities," Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, March 2011.

[5] J. G. Koomey, "Growth in Data Center Electricity Use 2005-2010," Analytics Press, August 2011.

[6] B. Vamanan, J. Hasan and T. N. Vijaykumar, "Deadline-Aware Datacenter TCP (D2TCP) ," *Proceedings of the ACM SIGCOMM '12*, Helsinki, Finland, August 13-17, 2012.

[7] G. Shainer, "Networks: Topologies How to Design," HPC Advisory Council, 2011.

[8] M. Al-Fares, A. Loukissas and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," *SIGCOMM'08,* Seattle, Washington, USA, August 17–22, 2008.

[9] X. Yuan, W. Nienaber, Z. Duan and R. Melhem, "Oblivious Routing in Fat-Tree Based System Area Networks with Uncertain Traffic Demands," *ACM Sigmetrics,* pages 337-348*,* San Diego, CA*,* June 12-16, 2007.

[10] J. Kim, W. J. Dally and D. Abts, "Flattened Butterfly: A Cost-Efficient Topology for High-Radix Networks," *ISCA'07*, San Diego, California, USA, June 9–13, 2007.

[11] J. Kim, W. J. Dally, S. Scott and D. Abts, "Technology-Driven, Highly-Scalable Dragonfly Topology," *ISCA '08*, 35th International Symposium on Computer Architecture, Beijing, June 21–25, 2008.

[12] C. Guo et al., "BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers," *SIGCOMM'09*, Barcelona, Spain, August 17–21, 2009.

[13] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang and S. Lu, "DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers," *SIGCOMM'08*, Seattle, Washington, USA, August 17–22, 2008.

[14] C. Raiciu et al. "Improving datacenter performance and robustness with multipath TCP," *Proceedings of the ACM SIGCOMM*, Toronto, Canada, August 2011.

[15] L. Valiant and G. Brebner, "Universal schemes for parallel communication," *Proceedings of the 13th annual symposium on theory of computing*, pages 263-277, May 1981.

[16] H. Sullivan and T. R. Bashkow, "A large scale, homogeneous, fully distributed parallel machine," *Proceedings of the 4th annual symposium on Computer architecture*, pages 105–117, ACM Press, 1977.

[17] D. Seo, A. Ali, W. Lim, Nauman Rafique and Mithuna Thottethodi, "Nearoptimal worst-case throughput routing for two-dimensional mesh networks," *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 432–443, 2005.

[18] T. Nesson and S. L. Johnsson, "Romm routing on mesh and torus networks," *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 275–287, ACM Press, 1995.

[19] G. Sun, C. Chang and B. Lin, "A New Worst-Case Throughput Bound for Oblivious Routing in Odd Radix Mesh Network," *IEEE Computer Architecture Letters*, Vol. 12 no. 1, pp. 9-12, January-June 2013

[20] A. Singh, "Load-Balanced Routing in Interconnection Networks," PhD thesis, Stanford University, 2005.

[21] N. Jiang, J. Kim and W. J. Dally, “Indirect adaptive routing on large scale interconnection networks,” *Proceedings of the 36th annual international symposium on Computer architecture ISCA ’09*, pp. 220–231, 2009.

[22] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang and A. Vahdat, “Hedera: Dynamic flow scheduling for data center networks,” *Proceedings of Usenix NSDI 2010*.

[23] “OpenFlow Switch Specification,” Version 1.4.0, Open Networking Foundation, October 14, 2013.

[24] S. Kirkpatrick, C. D. Gelatt Jr, M. P. Vecchi, “Optimization by Simulated Annealing,” Science, Volume 220, Number 4598, May 1983.

[25] A. Borodin and J. E. Hopcroft, “Routing, Merginig, and Sorting on Parallel Models of Computation,” Journal of Computer and System Sciences 30, 130-145, 1985.

[26] C. Scheideler, Theory of Network Communication Lecture Notes, Fall Term 2004, *http://www.cs.jhu.edu/~scheideler/courses/600.348_F04/lecture_3.pdf*

[27] M. Kodialam, T. V. Lakshman and S. Sengupta, “Maximum Throughput Routing of Traffic in the Hose Model,” *INFOCOM 2006*, Barcelona, Spain, 23-29 April, 2006.

[28] M. Antić, N. Maksić, P. Knežević and A. Smiljanić, “Two Phase Load Balanced Routing using OSPF,” *IEEE Journal on Selected Areas of Communications*, January 2010.

[29] Xen Users’ Manual, Xen V2.0 for x86, [Online]. Available: http://wiki.xenproject.org/mediawiki/images/a/a4/Xen_2_user_manual.pdf

[30] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson, ”Inferring Link Weights Using End-to-End Measurements,” Proc. of the IMW 2002, Nov. 2002.

[31] T.H. Comen, C. E. Leiserson, R. L. Rivest, C. Stein, “*Introduction to Algorithms*,” pages 540-549, MIT Press, 2001.

[32] N. Maksić and A. Smiljanić, “Improving Utilization of Datacenter Networks,” *IEEE Communications Magazine*, November 2013.

[33] B. Towles and W. J. Dally, “Worst-case Taffic for Oblivious Routing Functions,” *Computer Architecture Letters*, 2002.

[34] H. W. Kuhn, “The Hungarian Method for the assigment problem,” Naval Reserach Logistics Quaterly, 2:83-97, 1955.

[35] The ns-3 network simulator [Online], Available: *http://www.nsnam.org/*

[36] The ns-3 network simulator documentation [Online], Available: *http://www.nsnam.org/ns-3-19/documentation/*

[37] M. Ruiz-Sanchez, E. Biersack and W. Dabbous, “Survey and taxonomy of IP address lookup algorithms," *IEEE Network Mag.*, March/April 2001.

[38] A. McAuley and P. Francis, “Fast routing table lookup using CAMs,” *Proceedings of the Conference on Computer* Communications, *INFOCOM 1993*, San Francisco, March/April 1993.

[39] A. Ramussen, A. Kragelund, M. Berger, H. Wessing and S. Ruepp, “TCAM-based High Speed Longest Prefix Matching with Fast Incremental Table Updates, ” *Proceedings of IEEE Conference on High Performance Switching and Routing 2013*, Taipei, Taiwan, July 8-11, 2013.

[40] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, “Small Forwarding Tables for Fast Routing Lookups,” *Proceedings of ACM SIGCOMM ’97*, Cannes, France, September 1997.

[41] W. Eatherton, Z. Dittia and G. Varghese, “Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates” *ACM SIGCOMM Computer Communication Review*, vol. 34(2), pp. 97-122, April 2004.

[42] W. Jiang, Q. Wang and V. K. Prasanna, “Beyond TCAMs: An SRAM-based parallel multi-pipeline architecture for terabit IP lookup”, *Proceedings of IEEE INFOCOM 2008*, Phoenix, USA, April 2008.

[43] Z. Čiča and A. Smiljanić, “Balanced Parallelised Frugal IPv6 Lookup Algorithm”, *IET Electronics Letters*, Vol. 47, No. 17, pp. 963-965, August 2011.

[44] Z. Čiča and A. Smiljanić, “Frugal IP lookup based on a parallel searh”, *Proc. of IEEE Workshop on High Performance Switching and Routing 2009*, Paris, France, June 2009.

[45] H. Lim, C. Yim and E. E. Swartzlander, Jr., “Priority Tries for IP Address Lookup”, *IEEE Transactions on Computers*, Vol. 59, No. 6, June 2010.

[46] W. Jiang and V. K. Prasanna, "Towards Practical Architectures for SRAM-based Pipelined Lookup Engines", *Proceedings of IEEE INFOCOM*, San Diego, California, 15-19 March, 2010.

[47] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High Speed IP Routing Lookups," *Proceedings of the ACM SIGCOMM 1997*, Cannes, France, 14-18 September, 1997.

[48] M. Wang, S. Deering, T. Hain and L. Dunn, "Non-random Generator for IPv6 Tables", *Proceedings of IEEE Symposium on High-Performance Interconnects 2004*, Stanford, USA, August 2004.

[49] S. Dharmapurikar, P. Krishnamurthy and D. E. Taylor, "Longest prefix matching using Bloom filters," *Proceedings of SIGCOMM 2003*, Karlsruhe, Germany, August 25-29, 2003.

[50] *BGP Reports [Online], Available:* http://*bgp.potaroo.net*

[51] Routing Information Service Raw Data [Online], Available: *http://data.ris.ripe.net*

[52] Czech Education and Scientific NETwork [Online], Available: *http://www.cesnet.cz*

[53] LpSolve, Reference Guide [Online], Available: *http://lpsolve.sourceforge.net*, 2005.

[54] John T. Moy, "*OSPF Complete Implementation*," Addison-Wesley Professional, 2000.

[55] John T. Moy, "*OSPF: Anatomy of an Internet Routing Protocol*," Addison-Wesley Professional, 1998.

[56] OSPFD Routing Software Resources [Online], Available: *www.ospf.org*

[57] RFC2370: The OSPF Opaque LSA Option [Online], Available: *http://www.ietf.org/rfc/rfc2370.txt*

[58] RFC2328: OSPF Version 2 [Online], Available: *https://www.ietf.org/rfc/rfc2328.txt*

[59] Gregor N. Purdy, "*Linux iptables Pocket Reference*," O'Reilly Media Inc., 2004.

[60] The netfilter.org "libnetfilter_queue" project [Online], Available: *http://www.netfilter.org/projects/libnetfilter_queue/*

Biografija kandidata

Nataša Maksić je rođena 6.2.1983. u Beogradu, gde je završila osnovnu školu i Matematičku gimnaziju kao nosilac Vukove diplome. Elektrotehnički fakultet u Beogradu upisala je 2002. i diplomirala je 2007. sa prosečnom ocenom 10. Dobila je nagradu Univerziteta u Beogradu kao student generacije Elektrotehničkog fakulteta za 2007/2008. U toku studiranja 2004. i 2005. dobila je nagrade kompanije YUBC SYSTEM A.D. kao najbolji student 3. i 4. godine Odseka za elektroniku, telekomunikacije i automatiku. 2006. kao student 5. godine bila je nagrađena od strane Elektrotehničkog fakulteta za najbolji uspeh na njenom odseku u školskoj 2005/2006. 2008. bila je nagrađena od strane Elektrotehničkog fakulteta za najbolji uspeh u studijama među studentima koji su diplomirali 2007/2008. na njenom odseku. Takođe, od Telenor fondacije dobila je nagradu prof. dr Ilija Stojanović za doprinos u oblasti telekomunikacija u kategoriji najboljeg diplomiranog studenta na Odseku za elektroniku, telekomunikacije i automatiku. Bila je stipendista Ministarstva prosvete i zadužbine Studenica. 2006. dobila je Eurobank EFG školarinu za studente završne godine državnih fakulteta za ostvarene izvanredne rezultate tokom studija.

Avgusta 2006. bila je na praksi u kompaniji ENSICO d.o.o iz Ljubljane. Od 2008. zaposlena je u Inovacionom centru Elektrotehničkog fakulteta kao istraživač. Radila je na projektu Ministarstva nauke „Sistemska integracija Internet rutera” od 2008. do 2010., a od 2011. radi na projektu „Razvoj servisa i bezbednosti Internet rutera visokog kapaciteta”.

Nataša Maksić je autor više radova na konferencijama i u časopisima. Autor je dva rada u žurnalima sa impakt faktorima preko 3. Dobitnik je nagrada za najbolji rad u oblasti Telekomunikacija u kategoriji mladih istraživača na konferencijama ETRAN 2009 i ETRAN 2012.

Nataša Maksić je recenzent za časopise *IEEE Communication Letters* i *IEEE Journal on Selected Areas in Communications*, kao i za konferenciju *IEEE Workshop on High Performance Switching and Routing*, u čijoj je organizaciji učestvovala 2012. godine. Za recenzije urađene u toku 2011. za časopis *IEEE Communication Letters* dobila je priznanje *Exemplary Reviewer*.

Прилог 1.

Изјава о ауторству

Потписани-а Наташа Максић

број индекса 5028/07

Изјављујем

да је докторска дисертација под насловом

Оптимизација и имплементација напредних протокола за рутирање

- резултат сопственог истраживачког рада,
- да предложена дисертација у целини ни у деловима није била предложена за добијање било које дипломе према студијским програмима других високошколских установа,
- да су резултати коректно наведени и
- да нисам кршио/ла ауторска права и користио интелектуалну својину других лица.

Потпис докторанда

У Београду, 10.07.2014.

Наташа Максић

Прилог 2.

Изјава о истоветности штампане и електронске верзије докторског рада

Име и презиме аутора НАТАША МАКСИЋ

Број индекса 5028/07

Студијски програм ТЕЛЕКОМУНИКАЦИЈЕ

Наслов рада ОПТИМИЗАЦИЈА И ИМПЛЕМЕНТАЦИЈА НАПРЕДНИХ ПРОТОКОЛА ЗА РУТИРАЊЕ

Ментор ДР АЛЕКСАНДРА СМИЉАНИЋ

Потписани/а НАТАША МАКСИЋ

Изјављујем да је штампана верзија мог докторског рада истоветна електронској верзији коју сам предао/ла за објављивање на порталу **Дигиталног репозиторијума Универзитета у Београду.**

Дозвољавам да се објаве моји лични подаци везани за добијање академског звања доктора наука, као што су име и презиме, година и место рођења и датум одбране рада.

Ови лични подаци могу се објавити на мрежним страницама дигиталне библиотеке, у електронском каталогу и у публикацијама Универзитета у Београду.

Потпис докторанда

У Београду, 10.07.2014.

Наташа Максић

Прилог 3.

Изјава о коришћењу

Овлашћујем Универзитетску библиотеку „Светозар Марковић" да у Дигитални репозиторијум Универзитета у Београду унесе моју докторску дисертацију под насловом:

ОПТИМИЗАЦИЈА И ИМПЛЕМЕНТАЦИЈА НАПРЕДНИХ ПРОТОКОЛА ЗА РУТИРАЊЕ

која је моје ауторско дело.

Дисертацију са свим прилозима предао/ла сам у електронском формату погодном за трајно архивирање.

Моју докторску дисертацију похрањену у Дигитални репозиторијум Универзитета у Београду могу да користе сви који поштују одредбе садржане у одабраном типу лиценце Креативне заједнице (Creative Commons) за коју сам се одлучио/ла.

1. Ауторство

2. Ауторство - некомерцијално

3. Ауторство – некомерцијално – без прераде

4. Ауторство – некомерцијално – делити под истим условима

5. Ауторство – без прераде

6. Ауторство – делити под истим условима

(Молимо да заокружите само једну од шест понуђених лиценци, кратак опис лиценци дат је на полеђини листа).

Потпис докторанда

У Београду, 10.07.2014.

Наташа Макеић

1. Ауторство - Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце, чак и у комерцијалне сврхе. Ово је најслободнија од свих лиценци.

2. Ауторство – некомерцијално. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца не дозвољава комерцијалну употребу дела.

3. Ауторство - некомерцијално – без прераде. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, без промена, преобликовања или употребе дела у свом делу, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца не дозвољава комерцијалну употребу дела. У односу на све остале лиценце, овом лиценцом се ограничава највећи обим права коришћења дела.

4. Ауторство - некомерцијално – делити под истим условима. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце и ако се прерада дистрибуира под истом или сличном лиценцом. Ова лиценца не дозвољава комерцијалну употребу дела и прерада.

5. Ауторство – без прераде. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, без промена, преобликовања или употребе дела у свом делу, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца дозвољава комерцијалну употребу дела.

6. Ауторство - делити под истим условима. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце и ако се прерада дистрибуира под истом или сличном лиценцом. Ова лиценца дозвољава комерцијалну употребу дела и прерада. Слична је софтверским лиценцама, односно лиценцама отвореног кода.