UNIVERZITET U BEOGRADU
ELEKTROTEHNIČKI FAKULTET

ALGORITMI ZA OSTVARIVANJE MAKSIMALNE PROPUSNOSTI TORUSNOG RUTERA

Master rad

Mentor:

Dr Zoran Čiča, docent

Kandidat:

Aleksandar Vasiljević
2015/3418

Beograd, Avgust 2016.

SADRŽAJ

1. UVOD

Sposobnost superkompjutera da rade posao sve brže i brže zadovoljava računarske potrebe, kako naučnih istraživanja tako i sve većeg broja industrijskih sektora. Snaga procesora je ključna za određivanje performanse sistema, ali nije jedini faktor. Jedan od ključnih aspekata paralelnih računara je komunikaciona mreža koja povezuje kompjuterske čvorove. Mreža je ta koja garantuje brzu interakciju između procesora dopuštajući im da sarađuju. Ovo je od suštinskog značaja za rešavanje kompleksnih računarskih problema na brz i efikasan način.

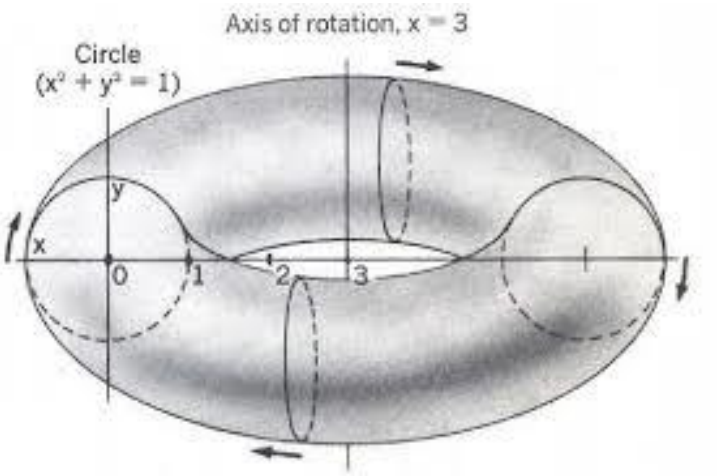
Na probijanje paketske tehnologije na telekomunikaciono tržište dominantno je uticao tehnološki razvoj jer je tek sa razvojem integrisanih čipova bilo moguće efikasno procesirati pakete i omogućiti razvoj paketske komutacije do današnjeg nivoa. Kako se celokupno procesiranje paketa vrši na čvorovima, javila se potreba da se razvijaju različite arhitekture paketskih komutatora kako bi se dobili što bolji rezultati u pogledu sledećih karakteristika: velika propusnost, malo kašnjenje, skalabilnost, smanjena kompleksnost, mala potrošnja energije za komunikaciju i na kraju ali ne i najmanje važno, cena.

U ovom radu biće obrađena jedna od tih arhitektura, torusna, koja se primenjuje u pojedinim mrežnim uređajima. Glavna ideja ove arhitekture je da se teret komutacije prebaci na same portove. Tačnije sada svaki port predstavlja mali mrežni čvor koji je povezan sa svojih šest suseda po *x*, *y* i *z* osi. Takođe, saobraćaj unutar same strukture može putovati po više različitih putanja od izvorišnog do odredišnog čvora. Sama struktura je zanimljiva jer je jednostavna, skalabilna i jeftina. U tu svrhu biće kreirana simulacija torusne arhitekture pomoću koje ćemo pratiti njeno ponašanje prilikom opterećenja saobraćajem, protoke po linkovima kao i uticaj na sam rad povećavanjem njenih dimenzija. Za potrebe rada, kod simulacije je pisan u programskom jeziku *C++* koristeći najnovije standarde *C++ 11* i *C++ 14*, koji se nametnuo kao prirodan izbor jer ispunjava sve zahteve za programiranje. Kod je pisan u *Microsoft Visual Studio 2013* programu jer pruža sve neophodne alate za ugodno pisanje i testiranje koda.

Rad je organizovan tako da ćemo se prvo opširnije pozabaviti samom torusnom arhitekturom, njenom strukturom, principom funkcionisanja, prednostima i manama i to će biti obuhvaćeno u drugom poglavlju. U trećem poglavlju ćemo prikazati kod simulacije, bavićemo se njegovom analizom i komentarisati primenjena rešenja i metode u kodu. Četvrto poglavlje je rezervisano za analizu dobijenih rezultata prilikom testiranja simulacije i njihovo komentarisanje da bi u petom poglavlju izveli zaključke i zapažanja vezana za torusnu arhitekturu.

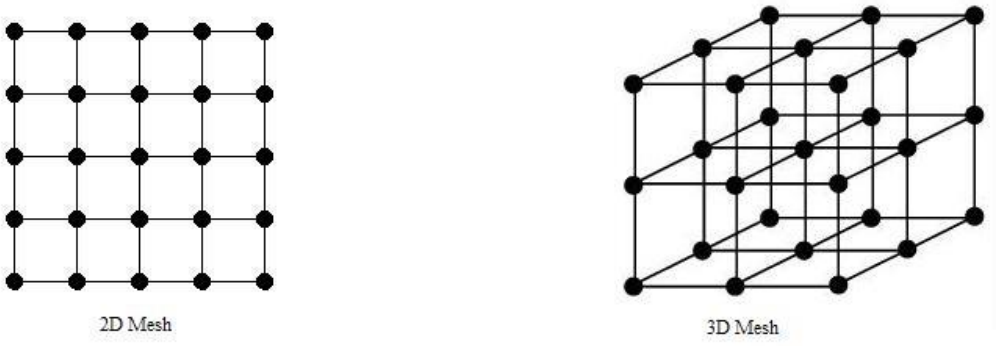
2. TORUSNA ARHITEKTURA

Torusna arhitektura je, kako joj samo ime kaže, ideju dobila od geometrijskog tela, torusa - obrtna površ koja se dobija kada se rotira kružnica u trodimenzionalnom prostoru oko ose koplanarne sa kružnicom, a koja ne dodiruje krug, poput unutrašnje gume automobila. Slika 2.1. prikazuje torusnu strukturu.



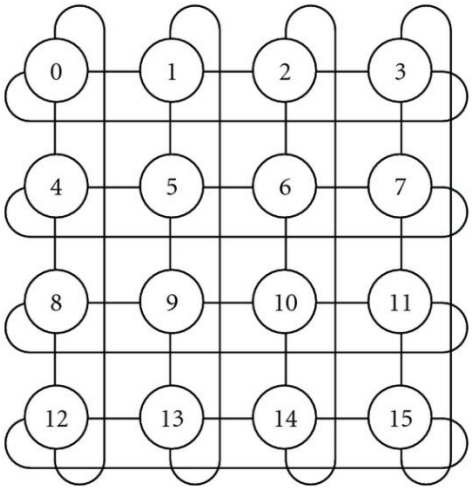
Slika 2.1. Torusna površ [11].

Kako bismo bolje razumeli povezanost arhitekture i geometrijskog tela krenimo od ravne ploče. Izdelićemo je na podjednake delove oblika kvadrata tako što ćemo povući jednak broj vertikalnih i horizontalnih linija i tako dobiti mrežu. Sada zamislimo da se na obodima ploče koji su ispresecani tim linijama kao i na presecima samih linija nalaze mali mrežni čvorovi. Na ovaj način smo dobili jednu meš mrežu mrežnih čvorova (Slika 2.2.).



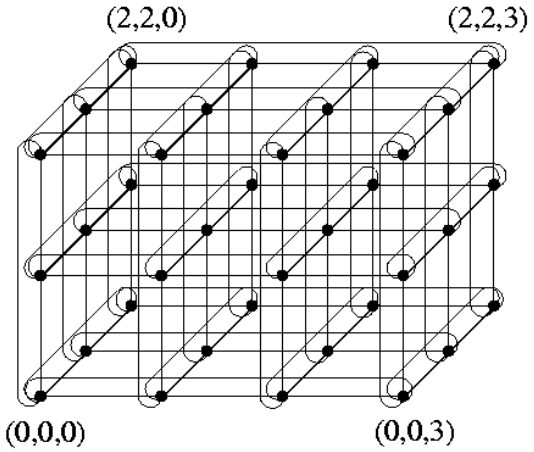
Slika 2.2. Čvorovi povezani u 2D i 3D meš mrežu [9].

Primećujemo da, ako bi čvor na jednom kraju mreže poslao paket čvoru na drugom kraju, taj paket bi morao, u zavisnosti od dimenzija mreže, da pređe dug put kroz celu mrežu. Ali ako dodamo jedan link između ta dva čvora, ceo put paketa se svodi na prelazak tog linka čime se značajno ubrzava proces prenosa. Kada isti princip primenimo na svaka dva krajnja čvora ove mreže, koja su postavljeni jedan naspram drugog, dobićemo 2D torusnu strukturu (Slika 2.3.).



Slika 2.3. Primer dvodimenzionalne torusne strukture [10].

Slaganjem ovih torusnih struktura, jedna na drugu, tj. dodavanjem treće dimenzije i povezivanjem čvorova po istom principu, dobijamo 3D torusnu strukturu (Slika 2.4.) koja povezuje *NxNyNz* čvorova od kojih svaki čvor ima po 6 suseda po svim osama dekartovog koordinatnog sistema *(-x, +x, -y, +y, -z, +z*) izraženih u paketskim skokovima (hop).



Slika 2.4. Trodimenzionalne torusna struktura - spoj više 2D struktura u jednu 3D strukturu [7].

Kako bi primetili prednost uvođenja treće dimenzije u torusnoj arhitekturi, poredićemo meš mrežu sa 32x32 čvora i 3D torus sa 10x10x10 čvorova. Ove dve arhitekture imaju približno isti broj čvorova koji iznosi oko 1000. Zadatak je da se pošalje paket od čvora koji se nalazi na obodu mreže do čvora koji se nalazi u središtu. Kako bi paket došao do čvora u središtu meš mreže on mora, ako je poslat sa čvora koji se nalazi u gornjem levom kraju mreže, da napravi 16 skokova po dužini, a zatim još toliko po širini, što je ukupno 32 skoka. Kod 3D torusne arhitekture, ako paket krene iz iste pozicije, napraviće pet skokova po dužini, pet skokova po širini i još pet po dubini, što je ukupno 15 skokova naspram 32 skoka kod meš mreže. Više skokova uzrokuje veća kašnjenja što loše utiče na rad cele mreže. Bitno je napomenuti da idealan torusni oblik ima jednak broj čvorova po svim dimenzijama, inače cela struktura postaje nebalansirana.

Torusna arhitektura funkcioniše tako što se celokupna komutacija paketa prebacuje na portove predstavljene mrežnim čvorovima. Ti mrežni čvorovi se ponašaju kao mali ruteri od kojih svaki ima po sedam portova, jedan eksterni i šest internih ka svojim susedima. Ruteri su spojeni kratkim linkovima. Postoji tri tipa rutiranja unutar torusne arhitekture:

- fiksno,
- adaptivno,
- balansirano.

Fiksno rutiranje podrazumeva da postoji obrazac, redosled, kojim se rutiranje obavlja i ono se unapred definiše. Npr. prvo se paket usmerava po osi *x*, zatim *y* osi i na kraju *z* osi dok ne stigne do odredišnog čvora.

Adaptivno rutiranje funkcioniše tako što se posmatra opterećenost linkova saobraćajem. Oni koji su manje opterećeni biće korišćeni za prenos paketa dok će linkovi sa većim opterećenjem biti zaobilaženi.

Balansirano rutiranje vrši predikciju putanja paketa na osnovu težinskih koeficijenata, cena, koji se dodeljuju linkovima.

Glavna prednost torusne arhitekture je njena mogućnost da se lako proširuje bez povećavanja kompleksnosti komutacije jer ona ne zavisi od broja ulaznih i izlaznih portova. Takođe, jednostavnost arhitekture utiče na cenu koja raste linearno sa brojem portova što je još jedna od prednosti.

Mana ove arhitekture je da je blokirajuća. Može se desiti da dođe do zagušenja linkova i nepotrebnog odbacivanja paketa iako izlazni portovi nisu preopterećeni. Još jedna bitna mana je i kašnjenje paketa koji prolaze kroz ovu strukturu. Ono zavisi od udaljenosti samih portova, a sa povećanjem dimenzija mreže postaje značajno veliko.

3. KODNA REALIZACIJA

3.1. Opis problema

Za zadatu matricu ponuđenog saobraćaja torusnom ruteru, treba pronaći puteve kroz torusni ruter za svaki komunikacioni par tako da što više saobraćaja prođe. Prilikom primene fer servisa smatrati da su svi tokovi iste težine. Svaki čvor po istom scenariju generiše saobraćaj tj. ima isti protok tokom čitave simulacije, gde je ukupan broj čvorova jednak *N*, a *N* je jednak proizvodu sve tri dimenzije komutatora. Pri tome treba varirati ponuđeni ulazni saobraćaj od 0.1 do 1 (0.1 znači da samo 10% kapaciteta ulaznog linka se koristi, a 1 znači da se koristi svih 100% kapaciteta). Ulazni podatak (pored dimenzije torusa) jeste matrica ponuđenog ulaznog saobraćaja i na osnovu nje, izabranog scenarija kako se raspoređuje saobraćaj kroz torus i načina izbora puteva izračunati propusnost rutera, tj. koliko saobraćaja prolazi a koliko je odbačeno. Propusnost je ono što je najbitniji izlazni podatak kao i ukupan saobraćaj koji je prošao i ukupan protok saobraćaja koji ne može da prođe. Nakon prvog izbora puta za svaki od tokova i nalaženja ostvarenih protoka, linkovi u torusnom ruteru koji su prepunjeni treba da budu umanjeni za proizvoljan korak sve dok ne postanu propusni. Nakon implementacije algoritma treba izvršiti testiranje za nekoliko saobraćajnih slučajeva i veličina torusnog rutera.

Scenariji za inicijalizaciju saobraćaja u torusu:

1. Čvor *i* (*i=1..N*) gađa portove *1..N* sa istom količinom saobraćaja, pri čemu svaki tok ima protok *u/N* gde je *u* ponuđeni ulazni saobraćaj (uniformni saobraćaj).
2. Čvor *i* gađa čvor *j* sa 50% ponuđenog ulaznog saobraćaja (*u/2*), a sve ostale sa podjednakom količinom saobraćaja koja iznosi *u/(2*(N-1))*.
   Podscenariji za *j*: 1. kada je *j* susedni čvor čvoru *i*, 2. kada je *j* što je više moguće udaljen od čvora *i*.

Napomena: logika odnosa između *i* i *j* treba da bude ista za svako *i* - ako je susedni čvor prvi sledeći po *x*-osi, onda se taj princip koristi za sve čvorove i (hot-spot scenario).

3. Čvor *i* gađa samo dva čvora *j* i *k*, svaki sa po *u/2* saobraćaja.
   Podscenariji: 1. *j* i *k* su susedni čvorovi, 2. *j* je susedni čvor, a *k* je udaljeni čvor, 3. *j* i *k* su udaljeni čvorovi.

Napomena: logika izbora čvora *j* i *k* treba da bude konzistentna kao i kod hot-spot scenarija (dijagonalni scenario).

Prilikom određivanja ruta u torusu koristiti:

1. Fiksno rutiranje *x-y-z*
2. Random izbor puta tj. redosleda *z-x-y*

3.2. Struktura programa

U ovom poglavlju ćemo se detaljno baviti samom simulacijom torusne arhitekture i analizom programskog koda. Pratićemo sve funkcije i komentarisati korake izvršavanja simulacije.

Kao što je pomenuto, simulacija torusne arhitekture je rađena u programskom jeziku *C++* dok je kod razvijan u *Microsoft Visual Studio* okruženju. Simulacija bi trebalo da oponaša rad torusnog komutatora koji se nalazi pod saobraćajnim opterećenjem. Ulazni podatak pored dimenzije torusa jeste matrica ponuđenog saobraćaja koja se generiše u čvorovima po uniformnoj raspodeli i ima vrednosti između 0.1 i 1. Destinacija saobraćaja, mrežni čvor, predstavljena je koordinatama tog odredišnog čvora i ona će biti definisana odabirom jednog od ponuđenih scenarija. Prilikom pisanja simulacije smo se opredelili za izlazne linkove koji će biti posmatrani kao sastavni deo čvora i svaki od njih će sadržati svoj vektor saobraćaja. U simulaciji će biti korišćena dva tipa rutiranja: Fiksno i Random. U zavisnosti od odabira načina rutiranja saobraćaj će se usmeravati po *x*, *y* ili *z* osi dok ne stigne do odredišnog čvora.

U simulaciji je omogućeno:

- Podešavanje dimenzija torusne arhitekture po svim koordinatama,
- Biranje tipa scenarija i podscenarija za hot-spot i dijagonalni scenarije,
- Podešavanje vrednosti za maksimalnu popunjenost linka kao i koraka za smanjivanje u slučaju kada je link pun,
- Izbor načina rutiranja.

Kako bi se izašlo u susret ovim zahtevima, programski kod simulacije organizovan je tako da se sastoji od tri celine: *Header.h*, *Source.cpp* i *DataStructures.cpp*. U *Header.h* fajlu nalazi se definicija i deklaracija samog torusa kao i njegove metode. Pored toga tu se još nalaze i deklaracije Čvora, Saobraćaja i Linka kao i deklaracije svih glavnih funkcija u programu. U *Source.cpp* fajlu nalazi se main funkcija koja poziva glavne funkcije i vrši ispis rezultata na ekran. *DataStructures.cpp* fajl sastoji se od definicija svih glavnih kao i pomoćnih funkcija.

Prilikom izrade koda simulacije bilo je potrebno rešiti sledeće zahteve:

- Generisanje matrice ulaznog saobraćaja,
- Generisanje torusa, mrežnih čvorova i linkova,
- Popunjavanje torusa saobraćajem u zavisnosti od odabira scenarija i podscenarija,
- Rutiranje paketa između čvorova u zavisnosti od odabranog tipa,
- Smanjivanje saobraćaja kako bi svi linkovi bili propusni,
- Prikupljanje podataka.

Za potrebe analize simulacije testiraćemo ponašanje torusne arhitekture tako što ćemo pokretati simulaciju sa različitim podacima koje smo prethodno naveli i porediti dobijene rezultate. U tu svrhu posmatraćemo sledeće parametre:

- Inicijalno opterećenje saobraćaja
- Finalno opterećenje saobraćaja (nakon smanjivanja)
- Prosečno opterećenje saobraćaja
- Najveće odstupanje saobraćaja
- Inicijalno opterećenje linkova
- Finalno opterećenje linkova (nakon smanjivanja)
- Prosečno opterećenje linkova
- Najveće odstupanje linka
- Procenat zagušenih linkova

Saobraćajni protok se posmatra na svim linkovima torusne arhitekture za vreme izvršavanja simulacije. Popunjenost linkova se odnosi na količinu saobraćaja koja prolazi kroz njega tokom i nakon izvršavanja simulacije.

3.3. Reprezentacija

Sada kada smo definisali zadatak, ulaz i izlaz iz programa pristupamo nalaženju najpogodnije strukture koja odgovara svim našim zahtevima. Opisani torus najbolje je predstaviti grafom čijim čvorovima želimo brzo da pristupimo na osnovu adrese čvora i čiji će nam susedi često biti potrebni. Naš izbor u ovom zadatku biće 3D matrica sa O(1) pristupom adresi i O(1) algoritmom za nalaženje suseda.

Matrica je zamišljena kao template klasa i nosiće informacije o dimenzijama torusa. Pored ovoga imaće i vektora podataka koji je zamišljen tako da predstavlja čvorove samog torusa i biće definisan malo kasnije. Svi neophodni podaci za računanje ulaza i izlaza, stanja linkova i bafera, kao i informacije o položaju čvorova i suseda biće smeštene u ovom vektoru.

Da bismo u potpunosti iskoristili mogućnosti ovako kreirane matrice moramo dodati javne metode koje će nam omogućiti pristup čvorovima matrice. Funkcije **pos()**, **coordinates()** nam vraćaju poziciju za zadate kordinate kao i pojedinačne kordinate za traženu poziciju respektivno, a funkcija **el()** nam omogućava pristup samom čvoru na osnovu kordinata ili pozicije.

Pored podrazumevanog konstruktora definisani su još: destruktor, konstruktor, konstruktor kopije, operator dodele, move konstruktor i move operator dodele. Biće nam potrebne još i funkcije koje vraćaju dimenzije matrice kao i njenu ukupnu veličinu.

```
template<class T>
class Matrix3D
{
private:
	int m_size_x = 0;
	int m_size_y = 0;
	int m_size_z = 0;
	std::vector<T>  m_data;
}
```

Za kretanje unutar matrice potrebno nam je da definišemo adrese čvorova, koje se najlakše mogu reprezentovati nizom brojeva od kojih će svaki predstavljati jednu koordinatu počevši od *x*. Strukture koja će nam ovo obezbediti naziva se *TripleInt*. Ovu strukturu ćemo takođe iskoristiti da kreiramo niz koji će nam olakšati određivanje susednih čvorova kao i smerova kretanja prilikom inicijalizacije scenarija i rutiranja. *NEIGHBOR_DIRECTIONS* predstavlja niz od šest *TripleInt* – ova od kojih svaki simbolizuje jednog suseda i smer u kome se on nalazi.

```
struct TripleInt {
	int x, y, z;
};

TripleInt const NEIGHBOR_DIRECTIONS[6] = {
	{ -1, 0, 0 },
	{ 1, 0, 0 },
	{ 0, -1, 0 },
	{ 0, 1, 0 },
	{ 0, 0, -1 },
	{ 0, 0, 1 }
};

using Address = TripleInt;
```

Osnovni parametar celog zadatka na osnovu koga kreiramo izlazne rezultate jeste saobraćaj. Prema zahtevima zadatka naš saobraćaj treba da sadrži izvorišnu i odredišnu adresu, kao i početno i krajnje opterećenje radi ispisa izlaznih rezultata. Za potrebe opisivanja saobraćaja uvodimo strukturu sa imenom *Traffic*. Pored opisanih atributa uvešćemo i jedan *bool* koji ćemo koristiti da markiramo saobraćaj ako prolazi kroz preopterećen link i na taj način pripremićemo ga za smanjivanje.

```
struct Traffic{
	Address source;
	Address destination;
	float load;
	float initial_load;
	bool  mark;
};
```

Sledeći u nizu podataka za koji treba da osmislimo strukturu jeste tok ili link kroz koji će biti prosleđivan saobraćaj. Pošto smo se opredelili za matričnu strukturu u kojoj ćemo skladištiti podatke o čvorovima, tok ćemo kreirati kao izlaznu strukturu iz svakog čvora ka i nazvaćemo ga *Link*. Svaki čvor imaće šest tokova ka svojim susedima kao i jedan dodatni ka samome sebi. Tok će biti jednoznačno određen izvorišnom i odredišnom adresom i sadržaće podatke o svom početnom i krajnjem opterećenju radi ispisa izlaznih rezultata. Bilo bi pogodno ako bismo znali spisak svih saobraćaja koji prolaze kroz jedan tok. U tu svrhu u svaki tok dodaćemo i vektor pokazivača saobraćaja tako da ćemo u svakom trenutku simulacije moći da kontrolišemo i po želji smanjimo opterećenje toka smanjujući svaki saobraćaj pojedinačno za određen korak.

```
struct Link{
	Address source;
	Address destination;
	float load;
	float initial_load;
	std::vector<Traffic *> traffic;
};
```

Nakon što smo definisali matricu, saobraćaj i tokove stekli su se uslovi i da definišemo čvorove.

U čvor je potrebno da smestimo sve podatke od interesa za zadatak i sam torus će nam simbolizovati vektor čvorova. Za ove potrebe kreiraćemo strukturu *Node* koja će biti jednoznačno određena sopstvenom adresom. Pored ove adrese svaki čvor sadržaće i tri vektora. Prvi će biti vektor saobraćaja *buffer* i predstavljaće saobraćaj koji je generisan u samom čvoru na osnovu ponuđene ulazne matrice saobraćaja i izabranog scenarija i po potrebi podscenarija. Drugi vektor će biti vektor pokazivača na čvorove koji će predstavljati susedne čvorove i zvaće se *neighbours*. Ovaj vektor će između ostalog biti iskorišćen prilikom kreiranja trećeg vektora izlaznih tokova. Kao što smo naglasili prilikom definisanja tokova svaki čvor sadržaće 7 izlaznih linkova, 6 ka svojim susedima i jedan ka samome sebi i oni će biti smešteni u vektor *links*.

```
struct Node{
	Address address;
	std::vector<Traffic> buffer;
	std::vector<Node *> neighbours;
	std::vector<Link > links;
};
```

I nakon svega stekli su se svi uslovi za torus:

```
using Torus = Matrix3D <Node>;
```

3.4. Implementacija grafovskih algoritama

3.4.1. Algoritam za nalaženje čvora sa datom adresom

Da bismo u potpunosti iskoristili mogućnosti ovako kreirane matrice moramo dodati javne metode koje će nam omogućiti pristup čvorovima matrice. Potrebne su nam funkcije koje za traženu poziciju vraćaju koordinate čvora i koje za predane koordinate vraćaju tačne pozicije. Poznajući sve tri dimenzije matrice kada nam se proslede trenutne koordinate, trenutnu poziciju čvora računamo na sledeći način:

```
int pos(int x, int y, int z) const
{
	return m_size_x*m_size_y*z + m_size_x*y + x;
}
```

U slučaju da imamo njegovu poziciju, zajedno sa dimenzijama matrice, možemo izračunati i njegove tačne koordinate sledećim formulama:

```
TripleInt coordinates(int position)  const
{
	int z_div = position / (m_size_x*m_size_y);
	int z_rem = position % (m_size_x*m_size_y);
	int y_div = z_rem / m_size_x;
	int y_rem = z_rem % m_size_x;
	return{ y_rem, y_div, z_div };
}
```

Nakon ovoga definišemo i metod **el()** koji nam vraća pristup traženom čvoru:

```
T& el(int x, int y, int z)
	{
		return m_data[pos(x, y, z)];
	}

T& el(TripleInt const & coord)
	{
		return m_data[pos(coord)];
	}
```

3.4.2. Algoritam za nalaženje suseda čvora

Potrebno nam je da napravimo algoritam za određivanje suseda. Ovo radimo tako što ćemo prvo napraviti jednu pomoćnu funkciju koja će vršiti cirkularno pomeranje. Ona će uzimati kao parameter početnu tačku, veličinu matrice i pravac u kome se pomeramo. Kako bismo ostali u okviru veličina matrice dodajemo *size* a zatim delimo po modulu sa *size*:

```
static int circular_move(int size, int direction, int start_point)
{
	return (start_point + direction + size) % size;
}
```

Uz pomoć ove funkcije sada možemo pronaći sve susede. Prolazimo kroz matricu pravaca suseda i za svaku koordinatu vršimo pomeraj u negativnom i pozitivnom smeru. Funkcija **circular_move()** će nam biti od koristi na više mesta u samom programu:

```
static std::vector<Node*> find_neighbours(Torus& torus, int x, int y, int z)
{
	std::vector<Node*> neighbours;
	for (auto const& direction : NEIGHBOR_DIRECTIONS)
		{
		TripleInt neighbour_position = {
			circular_move(torus.size_x(), direction.x, x)
		      , circular_move(torus.size_y(), direction.y, y)
		      , circular_move(torus.size_z(), direction.z, z)
		};
		neighbours.push_back(&torus.el(neighbour_position));
	}
	return neighbours;
}
```

3.4.3. Algoritam za nalaženje puteva

U ovom poglavlju osvrnućemo se i na algoritam za pronalazak putanje. U zavisnosti od tipa rutiranja računaćemo putanju za svaki saobraćaj posebno i to ćemo obaviti u dve različite funkcije **compute_path()** i **compute_random_path()**. Da bismo rešili ovaj problem uvodimo novi vektor adresa *path* koji će kasnije biti prosleđen funkciji **send_traffic_to_link()**. U ovaj vektor mi ćemo smestiti adresu svakog čvora kroz koji prolazi saobraćaj na svojoj putanji od izvorišta do odredišta računajući tu i prvi i poslednji čvor. Ovaj vektor će uvek imati minimum dve adrese u sebi, a u slučaju da je polazna tačka saobraćaja jednaka krajnjoj imaće dva zapisa polazne tačke. Funkcija za računanje putanje poredi krajnju i početnu tačku i ukoliko nisu iste stavlja adresu polazne u *path* i zove pomoćne funkcije. One zatim porede trenutnu startnu i ciljnu koordinatu i vrše pomeraj ka ciljnoj koordinati u najkraćem smeru upisujući pritom adresu svakog čvora kroz koji prođu u vektor *path*. Ovo se ponavlja za sve koordinate dok se ne izjednači polazna adresa sa odredišnom.

```
static std::vector<Address> compute_path(Torus const & torus, Address start,
Address goal)
{
	std::vector<Address> path;
	if (start == goal) {
		path.push_back(start);
		path.push_back(start);
	}
	else {
		Address start_address = start;
		path.push_back(start_address);
		start_address = push_x(torus, start_address, goal, path);
		start_address = push_y(torus, start_address, goal, path);
		start_address = push_z(torus, start_address, goal, path);
	}
	return path;
};
```

Da bismo ovo ostvarili potrebno je nekoliko pomoćnih funkcija **push_x()**, **push_y()**, **push_z()**, **push_xyz()** i **shortest_path_direction()**. Objasnićemo princip na pomeranju po *x* koordinati.

Poredimo trenutnu koordinatu *x* u kojoj se nalazimo sa odredišnom i ukoliko su iste vraćamo je u vektor *path*. Ukoliko nisu iste nalazimo najkraći pravac ka ciljnoj koordinati koristeći **shortest_path_direction()** i počinjemo da se krećemo ka odredištu u koracima za po jedan uz pomoć **circular_move()** funkcije upisujući u vektor *path* adresu svakog čvora kroz koji prođemo. Ovim smo završili obilazak po *x* koordinati i vraćamo tačku u kojoj se sada nalazimo. Ovo se zatim ponavlja za preostale koordinate.

```
static Address push_x(Torus const & torus, Address start, Address goal,
std::vector<Address> & path){
	if (start.x == goal.x)
	{
		return start;
	}
	int direction = shortest_path_direction(torus.size_x(), start.x, goal.x);
	start.x = circular_move(torus.size_x(), direction, start.x);
	while ( start.x != goal.x ){
		path.push_back(start);
		start.x = circular_move(torus.size_x(), direction, start.x);
	}
	path.push_back(start);
	return start;
}
```

Računanje najkraćeg pravca se radi tako što se uzima polazna tačka i računa rastojanje u smeru levo i desno ka krajnjoj tački. Kada se to izračuna ova dva rastojanja se porede i vraća se 1 ili -1 u zavisnosti od rezultata. Ovde smo opet iskoristili deljenje po modulu sa *size* kako bismo ostali u okviru čvorova matrice.

```
static int shortest_path_direction(int const size, int const a, int const b)
{
	int direction_left = calculate_distance(size, Direction::Left, a, b);
	int direction_right = calculate_distance(size, Direction::Right, a, b);
	return (direction_left <= direction_right ? -1 : 1);
}

static int calculate_distance(int const size, Direction const direction, int
const a, int const b)
{
	int distance = -1;
	switch(direction){
	case Direction::Left:
		distance = (size + a - b) % size;
		break;
	case Direction::Right:
		distance = (size + b - a) % size;
```

```
            break;
      }
      return distance;
}
```

Algoritam za određivanja Random ruta se pored navedenih funkcija oslanja i na **get_random_direction()** koja će biti opisana u sledećem poglavlju. Navedena funkcije vraća slučajnu osu i smer po kojem treba izvršiti rutiranje. Nakon ovoga poziva se funkcija za odabranu osu (*x*, *y* ili *z*) i radi određivanje putanje po principu koji je prethodno opisan. Proces se zatim ponavlja za preostale dve ose opet po slučajnom redosledu.

3.5. Inicijalizacija

U prethodnim poglavljima smo definisali strukture i algoritme koje ćemo koristiti prilikom izrade simulacije. U ovom poglavlju ćemo se baviti njihovom inicijalizacijom, tj. njihovim kreiranjem i postavljanjem početnih vrednosti. Krenućemo od generatora slučajnih brojeva kako bismo napravili našu matricu ulaznog saobraćaja. Za kreireanje svih slučajnih brojeva u projektnom zadatku koristićemo klasu *std::default_random_engine*. Od ove klase tražićemo da nam generiše *float* vrednost u rasponu od 0.1 do1. Ovako dobijena vrednost će se smestiti u čvor matrice saobraćaja i ona će reprezentovati sav saobraćaj koji će u toku simulacije izaći iz jednog čvora.

```
Matrix3D<float> generate_traffic_matrix(int size_x, int size_y, int size_z)
{
      std::random_device                  rd;
      std::default_random_engine          engine(rd());
      std::uniform_real_distribution<float> distribution(0.1f, 1.0f);
      Matrix3D<float> traffic_matrix(size_x, size_y, size_z);
      std::generate(
            traffic_matrix.begin(), traffic_matrix.end()
            , [&]{ return distribution(engine); }
      );
      return traffic_matrix;
}
```

Ostali smo dužni objašnjenje funkcije za **get_random_direction()**. Da bismo ovo rešili potreban nam je niz *NEIGHBOUR_DIRECTIONS*. Generisaćemo slučajan broj od 0 do 5 i iz funkcije ćemo vratiti niz *TripleInt* koji se nalazi na toj poziciji i koji će nam predstavljati i osu i pravac u kojem ćemo se kretati.

```
static TripleInt get_random_direction()
{
      static std::random_device                  rd;
      static std::default_random_engine          engine(rd());
      static std::uniform_int_distribution<int>  distribution(0, 5);
      return NEIGHBOR_DIRECTIONS[distribution(engine)];
}
```

Sledeći zadatak koji nas čeka jeste inicijalizacija torusa. Ona počinje i završava sa funkcijom **make_torus()** koja za zadate parametre *x*, *y* i *z* vraća kreiran torus sa svom potrebnom strukturom koju ćemo koristiti dalje u simulaciji. U ovoj funkciji prvo pozivamo konstruktor torusa, koji kreira čvorove a zatim tim čvorovima dodeljujemo adrese. Nakon što smo to uradili potrebno je da pronađemo susede svakog čvora. Ovo radimo opisanom funkcijom **find_neighbours()** kojoj kao parametar sada predajemo naš torus i adresu porta za kog tražimo susede. Na ovaj način kreiramo vektor pokazivača na susedne čvorove koji će biti smešten u svakom čvoru zasebno. U sledećem brojaču zovu se funkcije **find_neighbours()** i **make_links()** koje za svaki čvor kreiraju vektor susednih čvorova a zatim i vektor izlaznih linkova.

```
Torus make_torus(int size_x, int size_y, int size_z)
{
	auto torus = Torus(size_x, size_y, size_z);
	for (int pos = 0; pos < torus.size(); ++pos)
	{
		auto port_address = torus.coordinates(pos);
		auto & node = torus.el(port_address);
		node.address = port_address;
	}
	for (int pos = 0; pos < torus.size(); ++pos)
	{
		auto port_address = torus.coordinates(pos);
		auto & node = torus.el(port_address);
		node.neighbours = find_neighbours(torus, port_address);
		node.links = make_links(node);
	}
	return torus;
}
```

Ostao je još jedan korak kako bismo kompletirali naš torus a to je kreiranje linkova. Već smo naveli da ćemo za rešavanje zadatka koristiti izlazne linkove koje ćemo smestiti u svakom čvoru. Vektor linkova će se uvek sastojati od 7 linkova, 6 ka svojim susedim i 1 ka samome sebi. Za njihovo kreiranje iskoristićemo vektor suseda tako što ćemo proći kroz njega i za svakog suseda kreirati po jedan link koji će biti jednoznačno određen odredišnom adresom. Kada smo kreirali sve linkove ka susedima ostaje da dodamo još jedan link na kraj vektora sa sopstvenom početnom i odredišnom adresom. Opterećenje svih linkova ćemo u početku podesiti na 0.

```
static std::vector <Link> make_links(Node const & node )
{
	std::vector<Link> links(node.neighbours.size() + 1);
	for (unsigned int j = 0; j < node.neighbours.size(); ++j)
	{
		auto & link = links[j];
		link.source = node.address;
		link.destination = node.neighbours[j]->address;
		link.load = 0;
	}
	links.back().source = node.address;
	links.back().destination = node.address;
	links.back().load = 0;
```

```
	return links;
}
```

Kako bi kompletirali inicijalizaciju celog zadatka potrebno je još da izvršimo odabir scenarija.

Scenariji su opisani na početku ovog poglavlja a mi ćemo krenuti od prvog. Svakom scenariju biće predan sam torus i matrica ponuđenog ulaznog saobraćaja. Prvo određujemo opterećenje za svaki saobraćaj koje će u prvom slučaju iznositi *u/N* gde je *u* generisani početni saobraćaj svakog čvora pojedinačno a *N* broj svih čvorova u torusu. Nakon toga za svaki čvor kreiramo vektor saobraćaja *buffer* veličine *N* i u njega ubacujemo *N* saobraćaja sa definisanim opterećenjem. Ovom prilikom postavljamo i inicijalnu vrednost svakog saobraćaja na trenutno opterećenje i ovaj podatak ćemo koristiti kada budemo radili analizu rezultata simulacije.

```
void init_scenario1(Torus& torus, Matrix3D<float> const& initial_traffic)
{
	int const nodes_count = torus.size();
	for (int i = 0; i < nodes_count; ++i)
	{
		auto& node = torus.el(torus.coordinates(i));
		float const load_per_destination = initial_traffic.el(node.address)
/ nodes_count;
		node.buffer.reserve(nodes_count);
		for (int j = 0; j < nodes_count; ++j)
		{
			auto const & source_port = node;
			auto const & destination_port =
torus.el(torus.coordinates(j));
			auto const traffic = Traffic{ source_port.address,
destination_port.address, load_per_destination, load_per_destination, false };
			node.buffer.push_back(traffic);
		}
	}
}
```

U drugom scenariju zahtevi su malo drugačiji pa će nam biti potrebne pomoćne funkcije. Sada imamo zahtev da *u/2* saobraćaja bude prosleđeno jednom odredištu koje će biti definisano podscenarijem dok će se druga polovina saobraćaja ravnomerno rasporediti na preostale čvorove. *u/2* saobraćaja biće prosleđeno ili slučajnom najbližem čvoru ili najudaljenijem čvoru u matrici. Za ove potrebe koristićemo funkciju **get_random_direction()** koja će nam vrati slučajnu osu i smer kretanja. Ukoliko *u/2* saobraćaja ide ka slučajnom susedu, mi uz pomoć kreiranog slučajnog pravca i funkcije **circular_move()**, vršimo pomeraj za jedno mesto i selektujemo suseda. Ukoliko je u pitanju najudaljeniji čvor pozivamo pomoćnu funkciju **get_farthest()** koja će izračunati destinaciju tako što će za svaku osu računati najdalji čvor po sledećoj formuli:

```
static Address get_farthest(Torus const & torus, Address const & address)
{
	Address destination;
	destination.x = (address.x + torus.size_x() / 2) % torus.size_x();
```

```
	destination.y = (address.y + torus.size_y() / 2) % torus.size_y();
	destination.z = (address.z + torus.size_z() / 2) % torus.size_z();
	return destination;
}
```

Sada kao i u prethodnom scenarijumu kreiramo u svakom čvoru vektor saobraćaja *buffer* čija će veličina biti *N*. Na osnovu podscenarija postavljamo vrednosti za opterećenje i inicijalno opterećenje.

```
void init_scenario2(Torus& torus, Matrix3D<float> const& initial_traffic,
SubScenario2 const & sub_scenario)
{
	Address dir = get_random_direction();
	int const nodes_count = torus.size();
	for (int i = 0; i < nodes_count; ++i)
	{
		auto& node = torus.el(torus.coordinates(i));
		node.buffer.reserve(nodes_count);
		float const half_load = initial_traffic.el(node.address) / 2;
		float const rest_load = half_load / (nodes_count - 1);
		Address destination_half;
		switch (sub_scenario){
		case SubScenario2::Nearest:
			destination_half = circular_move(torus, dir, node.address);
			break;
		case SubScenario2::Farthest:
			destination_half = get_farthest(torus, node.address);
			break;
		default: assert(false && "invalid subscenario value");}
		for (int j = 0; j < nodes_count; ++j)
		{
			auto& dest_node = torus.el(torus.coordinates(j));
			auto traffic = Traffic{ node.address, dest_node.address, 0 };
			traffic.initial_load = destination_half == dest_node.address
				? half_load
				: rest_load;
			traffic.load = traffic.initial_load;
			traffic.mark = false;
			node.buffer.push_back(traffic);
		}
	}
}
```

U trećem scenariju mi ćemo iz svakog čvora slati saobraćaj samo ka dvema destinacijama. Važna napomena za scenario 3 jeste da moramo da vodimo računa da nam se ne poklope odredišni čvorovi za parove saobraćaja i to radimo tako što proveravamo da li se slučajan pravac prvog razlikuje od pravca drugog, tj. obezbeđujemo da se pomeranje ne vrši po istoj osi. Podscenarijima definišemo

da li saobraćaj šaljemo susedima ili najudaljenijem čvoru po osi. Kao što smo naveli prvo ćemo napraviti dva slučajna pravca uz pomoć **get_random_direction()** i obezbediti da se oni ne poklapaju.

Slučajnog suseda dobijamo uz pomoć **circular_move()** kao i u prethodnom scenariju dok će nam za najudaljenijeg po pravcu trebati nova pomoćna funkcija. Ona će preuzimati pravac u kojem želimo udaljenu destinaciju i u zavisnosti od njega definisati korak pomeraja, zatim zvati funkciju **circular_move()** i vratiti željenu destinaciju.

```
static Address get_far_away_in_direction(Torus const & torus, Address const&
origin, Address const & dir)
{
	Address destination = origin;
	int steps = 0;
	if (dir.x != 0){
		steps = torus.size_x() / 2;
	}
	if (dir.y != 0){
		steps = torus.size_y() / 2;
	}
	if (dir.z != 0){
		steps = torus.size_z() / 2;
	}
	for (int i = 0; i < steps; ++i)
	{
		destination = circular_move(torus, dir, destination);
	}
	return destination;
};
```

Kada su određene destinacije ostaje da kreiramo vektor saobraćaja *buffer* za svaki čvor koji će imati dva člana i da u njega smestimo saobraćaj za suseda ili najudaljenije čvorove. Takođe pored opterećenja postavljamo vrednost i za inicijalno opterećenje. Ovime je naša inicijalizacija kompletno završena.

```
void init_scenario3(Torus& torus, Matrix3D<float> const& initial_traffic,
SubScenario3 sub_scenario)
{
	Address dir1 = get_random_direction();
	Address dir2 = get_random_direction();
	while (dir1 == dir2 || dir1 == -dir2)
	{
		dir2 = get_random_direction();
	}
	int const nodes_count = torus.size();
	for (int i = 0; i < nodes_count; ++i)
	{
		auto& node = torus.el(torus.coordinates(i));
		node.buffer.reserve(2);
		float const half_load = initial_traffic.el(node.address) / 2;
		Address destination1;
		Address destination2;
```

```
            if (SubScenario3::Nearest == sub_scenario)
            {
                  destination1 = circular_move(torus, dir1, node.address);
                  destination2 = circular_move(torus, dir2, node.address);
            }
            if (SubScenario3::NearAndFar == sub_scenario)
            {
                  destination1 = circular_move(torus, dir1, node.address);
                  destination2 = get_far_away_in_direction(torus, node.address,
dir2);
            }
            if (SubScenario3::Farthest == sub_scenario)
            {
                  destination1 = get_far_away_in_direction(torus, node.address,
dir1);
                  destination2 = get_far_away_in_direction(torus, node.address,
dir2);
            }
            Traffic traffic1, traffic2;
            traffic1 = Traffic{ node.address, destination1, half_load,
half_load, false };
            node.buffer.push_back(traffic1);
            traffic2 = Traffic{ node.address, destination2, half_load,
half_load, false };
            node.buffer.push_back(traffic2);
      }
}
```

3.6. Simulacija

Do ovog momenta mi imamo kreiranu matricu sa svom infrastrukturom koja je napunjena saobraćajem u zavisnosti od scenarija. Nakon inicijalizacije odabranih scenarija ostalo je još samo jedno a to je da se inicijalni saobraćaj prosledi na svoja odredišta. Ova logika je smeštena u jednu veću funkciju **send_all_traffic()** koja računa putanju a zatim prosleđuje saobraćaj, vrši korekciju u slučaju zagušenja linkova i ispisuje željeni izveštaj u konzolu.

Da bismo počeli sa prosleđivanjem saobraćaja potrebno je da izaberemo tip rutiranja. Ovo ćemo uraditi putem konzole dozvoljavajući samom korisniku da definiše rutu. Algoritmi za rutiranje su obrađeni u prethodnom poglavlju te ćemo sada podrazumevati da su rute kreirane. Funkcija **send_traffic_to_link()** uzimaće torus, vektor adresa *path* koji je prethodno kreiran i sam saobraćaj koji želimo da prosledimo. Ideja je da u samome linku imamo vektor pokazivača svih saobraćaja koji prolaze kroz njega. Ovo će nam omogućiti da umanjujemo saobraćaj za korak koji je unapred definisan ukoliko dođe do preopterećenja linkova. Algoritam je realizovan tako da uzima par adresa iz vektora *path* gde prva predstavlja polaznu a druga odredišnu i na osnovu njih traži odgovarajući link u koji će da smesti saobraćaj. U ovoj funkciji iskoristićemo metod prozora, tj. kreiraćemo okvir koji će se sastojati od dve adrese i pokušaćemo da nađemo link na adresi početnog čvora. Kada budemo imali poklapanje smeštamo pokazivač na saobraćaj u vektor *traffic* na linku. Ovo ponavljamo

za svaki okvir dok ne prođemo kroz ceo vektor *path*. Prolaskom kroz svaki saobraćaj u torusu mi smo završili popunjavnje svih njegovih linkova.

```
static void send_traffic_to_link(Torus& torus, std::vector<Address> const path,
Traffic* traffic)
{
	assert(path.size() > 1);
	int first = 0;
	int second = 1;
	int const size = path.size();
	for (; second != size; ++first, ++second){
		auto& start_node = torus.el(path[first]);
		auto& goal_node = torus.el(path[second]);
		for (std::size_t i = 0; i < start_node.links.size(); ++i){
			if (start_node.links[i].destination == goal_node.address){
				start_node.links[i].traffic.push_back(traffic);
				break;
			}
		}
	}
}
```

Sada imamo sve podatke koji su nam potrebni i možemo da počnemo sa popunjavanjem linkova.

Prolazimo kroz sve čvorove, a zatim i kroz njihove linkove, i za svaki sumiramo opterećenja saobraćaja i smeštamo u ukupno opterećenje linka. Ovom prilikom postavljamo i parameter inicijalne popunjenosti linka koji će nam biti od koristi prilikom obrade rezultata.

```
static void refresh_all_links_loads(Torus& torus)
{
	for (auto& node : torus){
		for (auto& link : node.links){
			refresh_link_load(link);
		}
	}
}
static void refresh_link_load(Link& link)
{
	link.load = 0;
	for (auto t : link.traffic){
		link.load += t->load;
	}
}
void set_initial_link_load(Torus & torus)
{
	for (auto & node : torus)
	{
		for (auto & link : node.links)
```

```
            {
                  link.initial_load = link.load;
            }
      }
}
```

Nakon toga ulazimo u petlju koja se izvršava sve dok postoji makar jedan link koji je pun. Cilj ove petlje je da umanji sav saobraćaj koji prolazi kroz pune linkove za unapred definisan korak – *delta*.

To se postiže na sledeći način:

```
static bool check_is_there_full_link(Torus & torus, float max_load)
{
      for (auto& node : torus)
      {
            for (auto& link : node.links)
            {
                  if (link.load >= max_load)
                  {
                        return true;
                  }
            }
      }
      return false;
}


refresh_all_links_loads(torus);
do
{
      check_full_links(torus, max_load);
      update_marked_traffics(torus, delta);
      refresh_all_links_loads(torus);
} while (check_is_there_full_link(torus, max_load));
```

Samo smanjivanje saobraćaja radiće se tako što ćemo proći kroz sve linkove u torusu i ukoliko je link pun u njegovom vektoru pokazivača *traffic*, postavićemo vrednost *mark* polja na *true* za svaki saobraćaj koji se u njemu nalazi. *Mark* atribut predstavlja vrednost tipa *bool* koji smo kreirali u strukturi *Traffic*.

```
static void check_full_links(Torus & torus, float max_load)
{
      for (auto& node : torus)
      {
            for (auto& link : node.links)
            {
                  check_and_mark_if_full_link(link, max_load);
            }
      }
}
```

```
static void check_and_mark_if_full_link(Link & link, float max_load)
{
	if (link.load <= max_load)
	{
		return;
	}
	for (auto t : link.traffic)
	{
		t->mark = true;
	}
}
```

Kada je svaki saobraćaj markiran, moramo ponovo proći kroz torus i osvežiti za svaki čvor vektor *buffer* ukoliko je potrebno. Pritom moramo voditi računa da saobraćaj u tom vektoru ima opterećenje koje je veće od nule. Nakon ovoga zovemo funkciju **refresh_all_link_loads()** koja će osvežiti vrednosti opterećenja u linkovima koristeći promenjene podatke u saobraćaju. Ovo se izvršava sve dok ne preostane ni jedan link čije opterećenje nije veće od granične vrednosti. Izlaskom iz ove petlje naša simulacija je završena i možemo pristupiti prikupljanju željenih podataka.

```
static void update_marked_traffics(Torus & torus, float delta)
{
	for (auto& node : torus)
	{
		for (auto& traffic : node.buffer)
		{
			if (true == traffic.mark && traffic.load > 0)
			{
				traffic.load -= delta;
				if (traffic.load <= 0.0){
					traffic.load = 0.0;
				}
				traffic.mark = false;
				if (traffic.load <= 0)
				{
					traffic.load = 0;
				}
			}
		}
	}
}
```

3.7. Izlaz programa

Za potrebe izračunavanja izlaza programa formiraćemo strukturu *Report*. Parametri koji su za nas od interesa, a koji će biti upisani u ovu strukturu su: inicijalni saobraćaj u čvorovima, finalni saobraćaj u čvorovima (nakon smanjivanja), prosečan saobraćaj, maksimalno odstupanje saobraćaja, inicijalna popunjenost linkova, finalna popunjenost linkova (nakon smanjivanja), prosečna popunjenost linkova, maksimalno odstupanje saobraćaja u linkovima i broj linkova koji su preopterećeni.

```
struct Report
{
	float initial_link_load;
	float initial_traffic_load;
	float final_link_load;
	float final_traffic_load;
	float average_link_load;
	float average_traffic_load;
	float highest_link_deviation;
	float highest_traffic_deviation;
	int   over_loaded_links;
	};
```

Nakon odabira tipa rutiranja mi ćemo odrediti putanje saobraćaja i napravićemo vektore u linkovima. Kada je to završeno opterećenje iz čvorova biće prebačeno u odgovarajuće linkove i to će predstavljati inicijalnu popunjenost linkova. Sada možemo da prođemo kroz ceo torus i da izvršimo sume svih saobraćaja i svih linkova i da kreiramo izlazne podatke. Takođe u ovom trenutku proći ćemo kroz linkove i prebrojati sve one koji su popunjeni. Ovo radimo na sledeći način:

```
set_initial_link_load(torus);
float over_loaded_links = ( get_over_loaded_links(torus, max_load) /
(torus.size() * 7) ) * 100;
float initial_link_load = sum_link_loads(torus);
float initial_traffic_load = sum_all_trafics(torus);

float get_over_loaded_links(Torus const & torus, float max_load)
{
	float number_of_links = 0;
	for (auto const & node : torus)
	{
		for (auto & link : node.links)
		{
			if (link.load >= max_load)
			{
				number_of_links += 1;
			}
		}
```

```
	}
	return number_of_links;
}

static float sum_all_trafics(Torus & torus)
{
	float result = 0;
	for (auto const & node : torus)
	{
		for (auto const & traffic : node.buffer)
		{
			result += traffic.load;
		}
	}
	return result;
}



static float sum_link_loads(Torus & torus)
{
	float result = 0;
	for (auto const & node : torus)
	{
		for (auto const & link : node.links)
		{
			result += link.load;
		}
	}
	return result;
```

Zatim ulazimo u petlju koja proverava linkove i vrši smanjivanje saobraćaja ukoliko je to potrebno. Izlaskom iz ove možemo kreirati finalna opterećenja i preostale parametre. To radimo koristeći prethodno definisane funkcije, dok za računanje maksimalnog odstupanja prolazimo kroz svaki čvor i svaki link i računamo razliku inicijalne i krajnje vrednosti. Ovo predstavlja i poslednji korak u našoj simulaciji.

```
float final_link_load = sum_link_loads(torus);
float final_traffic_load = sum_all_trafics(torus);
float average_link_load = final_link_load / (torus.size() * 7);
float average_traffic_load = final_traffic_load /
number_of_all_traffics(torus);
float highest_link_deviation = get_highest_link_deviation(torus);
float highest_traffic_deviation = get_highest_traffic_deviation (torus);

static float get_highest_link_deviation(Torus const & torus)
{
	float max_deviation = 0;
	for (auto const & node : torus)
```

```
	{
		for (auto & link : node.links)
		{
			float deviation = link.initial_load - link.load;
			if (deviation > max_deviation)
			{
				max_deviation = deviation;
			}
		}
	}
return max_deviation;
}

static float get_highest_traffic_deviation(Torus const &torus)
{
	float max_deviation = 0;
	for (auto const & node : torus)
	{
		for (auto & traffic : node.buffer)
		{
			float deviation = traffic.initial_load - traffic.load;
			if (deviation > max_deviation)
			{
				max_deviation = deviation;
			}
		}
	}
	return max_deviation;
}
```

4. ANALIZA REZULTATA SIMULACIJE

U ovom poglavlju ćemo se baviti analizom rezultata dobijenih prilikom testiranja simulacije. Poredićemo torusne arhitekture različitih dimenzija i njihovo ponašanje u simulaciji prilikom različitih tipova scenarija, takođe upoređivaćemo rezultate za različite tipove rutiranja. Kodna realizacija metoda prikupljanja podataka na osnovu koje je izvršena analiza rada torusne arhitekture rutera opisana je u prethodnom poglavlju.

4.1. Analiza rezultata dobijenih u simulaciji

Kao što smo prethodno naveli za potrebe analize simulacije testiraćemo ponašanje torusne arhitekture tako što ćemo pokretati simulaciju sa različitim podacima koje smo prethodno naveli i porediti dobijene rezultate. U tu svrhu posmatraćemo sledeće parametre:

- Inicijalno opterećenje saobraćaja (*initial_traffic_load*) - ukupan slučajno generisan ulazni saobraćaj koji ne zavisi od scenarija.

- Finalno opterećenje saobraćaja (*final_traffic_load*) - računaće se na kraju simulacije nakon smanjivanja saobraćaja na zagušenim linkovima ukoliko postoji potreba.

- Prosečno opterećenje saobraćaja (*average_traffic_load*) - predstavlja količnik finalnog opterećenja saobraćaja i ukupnog broja saobraćaja koji zavisi od izbora scenarija i podscenarija.

- Najveće odstupanje saobraćaja (*highest_traffic_deviation*) - predstavlja najveće odstupanje pojedinačnog finalnog saobraćaja od njegovog inicijalnog u slučaju kada je link kroz koji prolaze zagušen.

- Inicijalno opterećenje linkova (*initial_link_load*) - predstavljaće sumu opterećenja svih linkova u torusu nakon raspodele inicijalnog saobraćaja, a pre provere da li je došlo do zagušenja linkova.

- Finalno opterećenje linkova (*final_link_load*) - kada smo izračunali inicijalno opterećenje linkova umanjujemo saobraćaj koji prolazi kroz zagušene linkove a samim tim i linkove dok nam svi linkovi ne postanu propusni, a zatim računamo sumu opterećenja svih tako dobijenih linkova.

- Prosečnu popunjenost linkova (*average_link_load*) - predstavlja količnik finalnog opterećenja linkova i broja linkova u torusu.

- Najveće odstupanje linka (*highest_link_deviation*) - predstavlja najveće odstupanje pojedinačnog finalnog opterećenja linka od njegovog inicijalnog u slučaju kada je došlo do zagušenja u linku.

- Procenat zagušenih linkova (*over_loaded_links*) - računaćemo tako što ćemo deliti broj linkova koji su zagušeni posle prosleđivanja inicijalnog saobraćaja sa ukpnim brojem linkova u torusu.

Za potrebe rada analiziraćemo toruse dimenzija 5x5x5, 6x6x6, 8x8x8 i 7x8x9. Torusi sa ovim dimenzijama predstavljaju strukture sa 125, 216, 512 i 504 čvorova, tj. poredimo strukture od kojih svaka sledeća ima približno dvostruko, odnosno četvorostruko više portova i uporedićemo strukturu koja nema pravilan oblik. Takođe, posmatraćemo kako se svaka od navedenih struktura ponaša u različitim scenarijima i tipovima rutiranja.

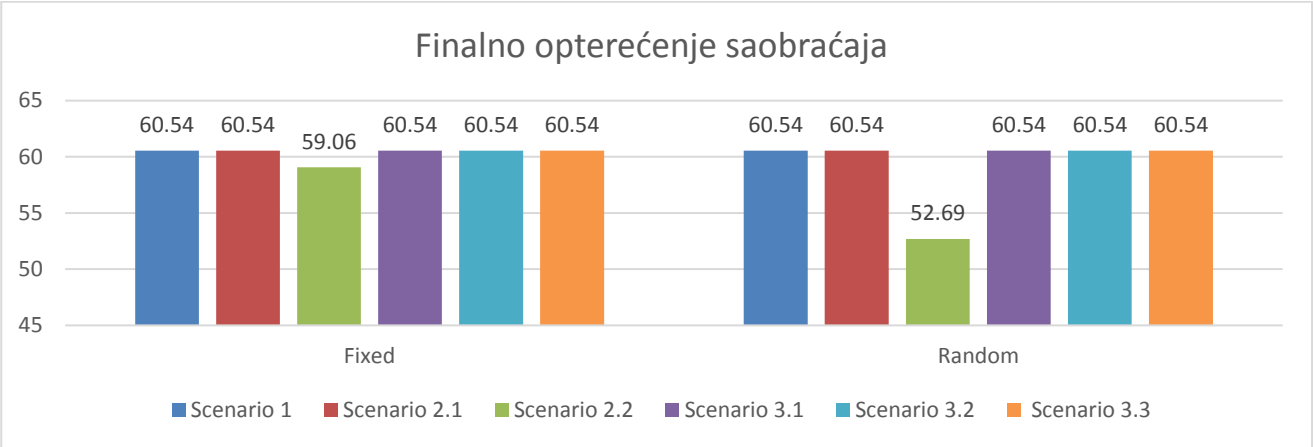
Prilikom testiranja simulacije analizirana su sva tri tipa scenarija sa svojim odgovarajućim podscenarijima. Opterećenja na čvorovima, tj. verovatnoće generisanja saobraćaja pri kojima je vršena simulacija variraju između 0.1 i 1 u zavisnosti od slučajne raspodele. Maksimalno opterećenje linka biće unapred definisano, neće se menjati tokom simulacije i iznosiće 1. Ukoliko dođemo u situaciju da moramo da smanjujemo saobraćaj na zagušenom linku smanjivaćemo ga u koracima od po 0.001 i ovo će takođe biti karakteristično za sve scenarije. Za određenu dimenziju torusa prikazaćemo svaki podatak na zasebnom grafiku. Vrednost podatka će biti prikazana za sve scenarije kroz koje prolazimo za jedan tip rutiranja a zatim na istom grafiku i za drugi. Ovo će nam obezbediti lakše poređenje dobijenih rezultata.

4.2. Torus 5x5x5

Na početku simulacije mi generišemo matricu ulaznog saobraćaja koje će biti raspoređeno po čvorovima torusa. Saobraćaj svakog čvora će imati slučajnu vrednost između 0.1 i 1. Zatim u svakom čvoru kreiramo vektor saobraćaja koji će rasporediti saobraćaj čvora prema odabranom scenariju. Kada je ovo završeno pozivamo funkciju **sum_all_trafics()** koja prolazi kroz vektore saobraćaja svih čvorova i sumira ih. Na ovaj način dobijamo inicijalno operećenje saobraćaja koje u stvari reprezentuje sav ulazni generisani saobraćaj. Kod zadatka je napisan tako da za unapred definisane dimenzije torusa mi generišemo ulaznu matricu saobraćaja koju sačuvamo i zatim je prosleđujemo scenarijima na raspodelu. Inicijalno opterećenja saobraćaja nam se zbog ovoga neće menjati kroz scenarije i biće uvek isto za jednu dimenziju torusa.

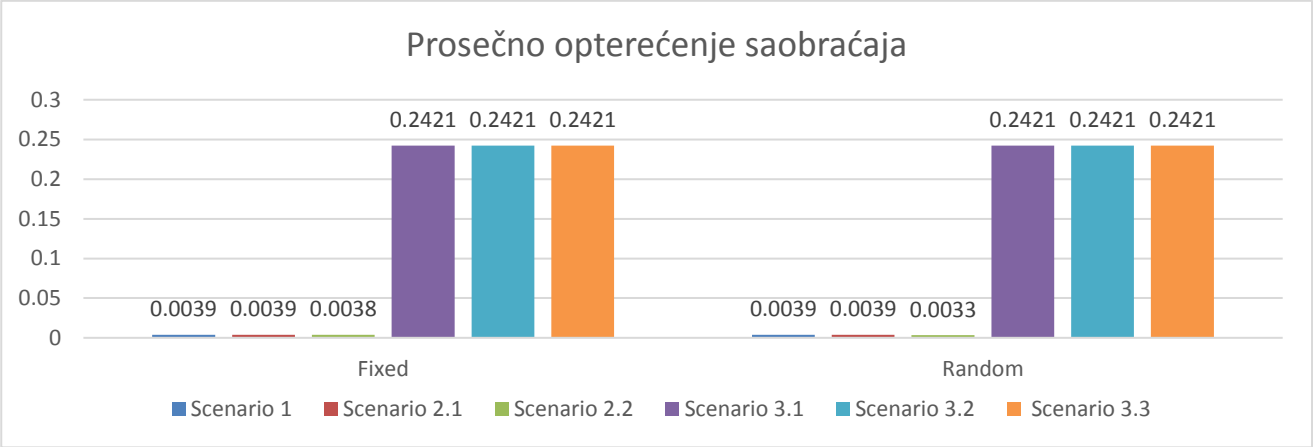
Inicijalno opterećenje saobraćaja = 60.54

Sledeći parametar je finalno opterećenje saobraćaja. Ovaj podatak će se razlikovati od inicijalnog ukoliko je došlo do prekoračenja maksimalne vrednosti na linkovima:



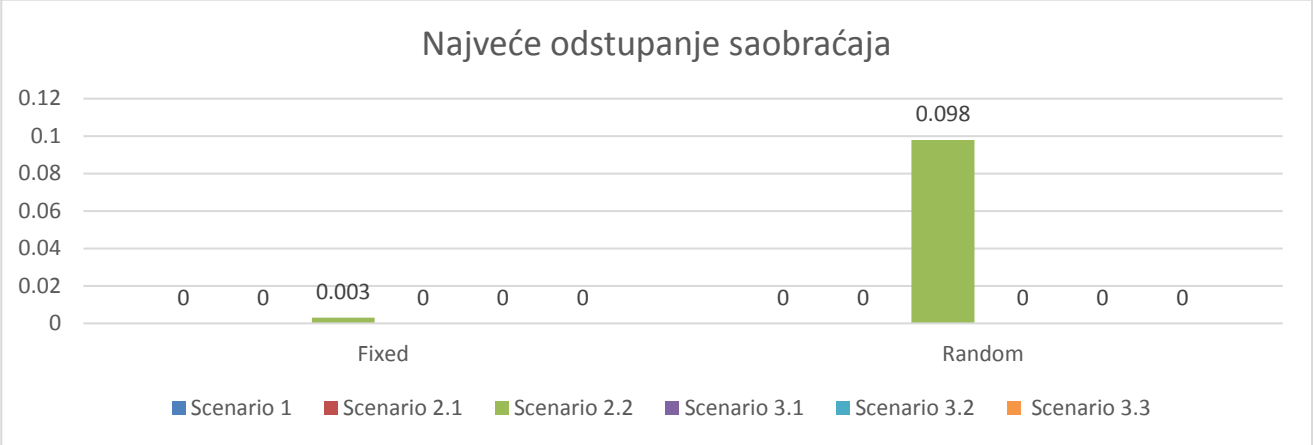
Slika 4.2.1. Finalno opterećenje saobraćaja.

Na Slici 4.2.1. vidimo da samo u scenariju 2.2 dolazi do zagušenja na linkovima. Zanimljivo je da je u ovoj situaciji korišćenje Random tipa rutiranja povećalo zagušenje linkova, tj. količina odbačenog saobraćaja se povećala. Zatim sledi prosečna vrednost saobraćaja koja će se razlikovati od scenarija do scenarija.



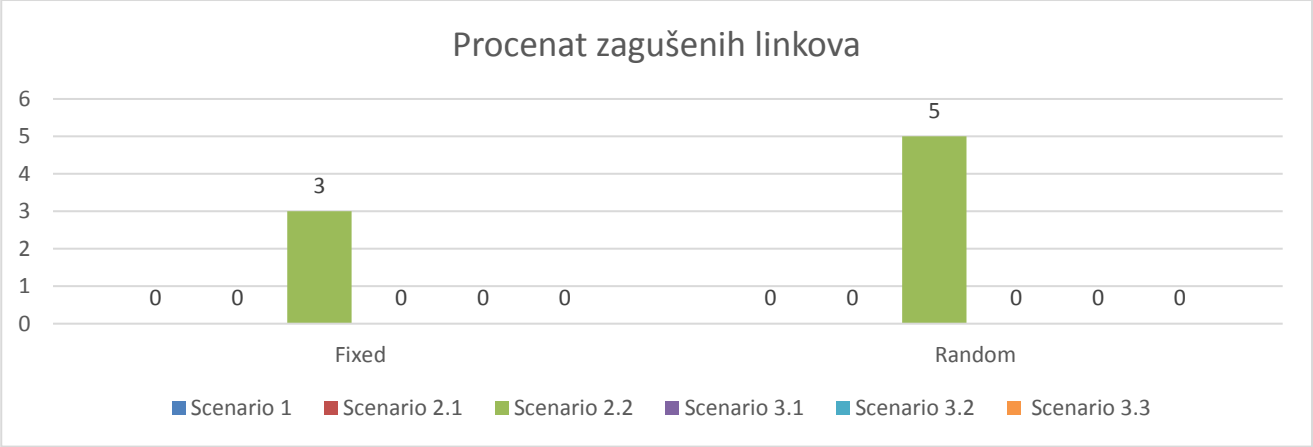
Slika 4.2.2. Prosečno opterećenje saobraćaja.

Do velike razlike prosečnog opterećenja saobraćaja dolazi zbog same definicije scenarija, kao i zbog propusnosti samog torusa jer će biti računato na samome kraju simulacije. Prvi i drugi scenario raspoređuju ulazni saobraćaj na *N* čvorova dok treći to čini za samo dva čvora iz torusa. Poslednji parametar za saobraćaj koji ćemo obraditi je najveće odstupanje od željenog saobraćaja. Ovaj parametar će nam se pojaviti u slučaju da dođe do zagušenja na linkovima.



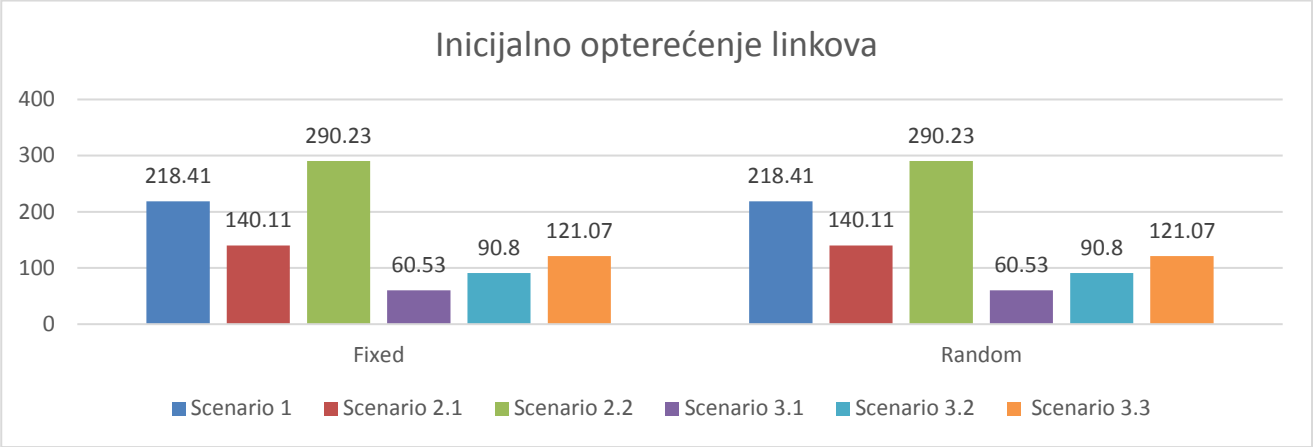
Slika 4.2.3. Najveće odstupanje saobraćaja.

Ovaj grafik potvrđuje prethodne konstatacije a to je da za Random tip rutiranja u scenariju 2.2 imamo mnogo veća odstupanja saobraćaja. Sada na red dolaze linkovi. Prvo ćemo prikazati procenat zagušenih linkova nakon prvog propuštanja saobraćaja, a pre smanjivanja ukoliko bude potrebe za njim.



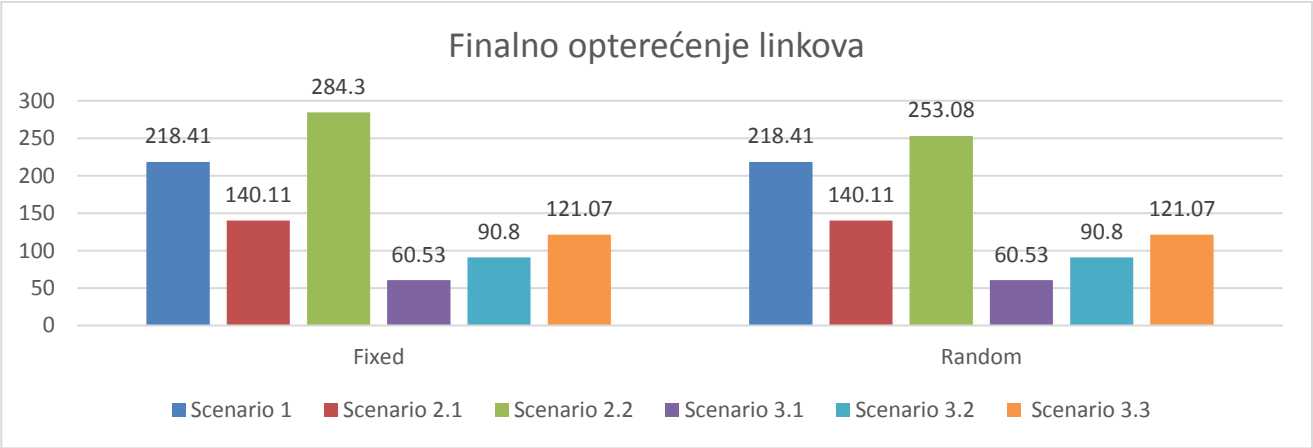
Slika 4.2.4. Procenat zagušenih linkova.

Na Slici 4.2.4. vidimo i razliku od 2% u korist Fiksnog tipa rutiranja. Analizu linkova nastavljamo sa inicijalnim optrerećenjima. Ovo ćemo prikazati na Slici 4.2.5.



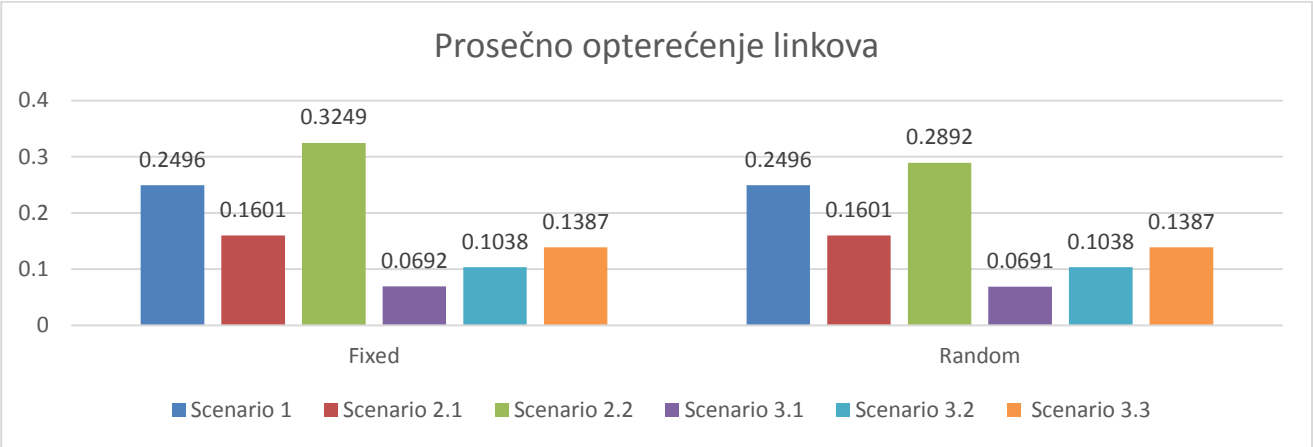
Slika 4.2.5. Inicijalno opterećenje linkova.

Inicijalno opterećenje ne zavisi od tipa rutiranja i imaće istu vrednost u oba slučaja. Zatim prikazujemo i finalno opterećenje linkova. Ovaj podatak će se razlikovati od prethodnog u slučaju da je došlo do zagušenja linkova.



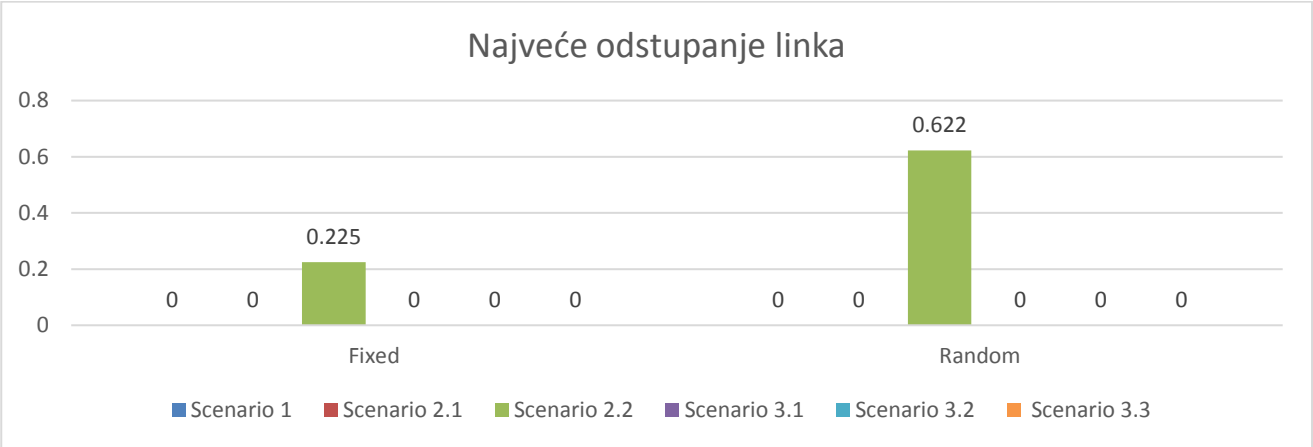
Slika 4.2.6. Finalno opterećenje linkova.

U scenariju 2.2 vidimo odstupanje od inicijalne vrednosti opterećenja linka za Fiksni tip rutiranja, kao i za Random. Nakon ovoga možemo prikazati i prosečno opterećenje linkova.



Slika 4.2.7. Prosečno opterećenje linkova.

Vrednosti za prosečno opterećenje linkova nam govore da je naš torus slabo opterećen. Maksimalna prosečna vrednost za sve scenarije i tipove rutiranja iznosi 0.3249 i predstavlja približno trećinu maksimalne popunjenosti linka. Ostaje nam još jedan rezultat za prikazivanje a to je najveće odstupanje popunjenosti linka. I ovaj podatak će nam zavisiti od toga da li je došlo do zagušenja u linkovima.



Slika 4.2.8. Najveće odstupanje linka.

Sa Slike 4.2.8. očitavamo da je u ovoj simulaciji ostvareno približno 2.5 puta manje maksimalno odstupanje od inicijalne vrednosti linka korišćenjem Fiksnog tipa rutiranja. Takođe iako imamo slabu prosečnu popunjenost linkova vidimo da u scenariju 2.2 beležimo ogromna prekoračenja maksimalne propusnosti linka.

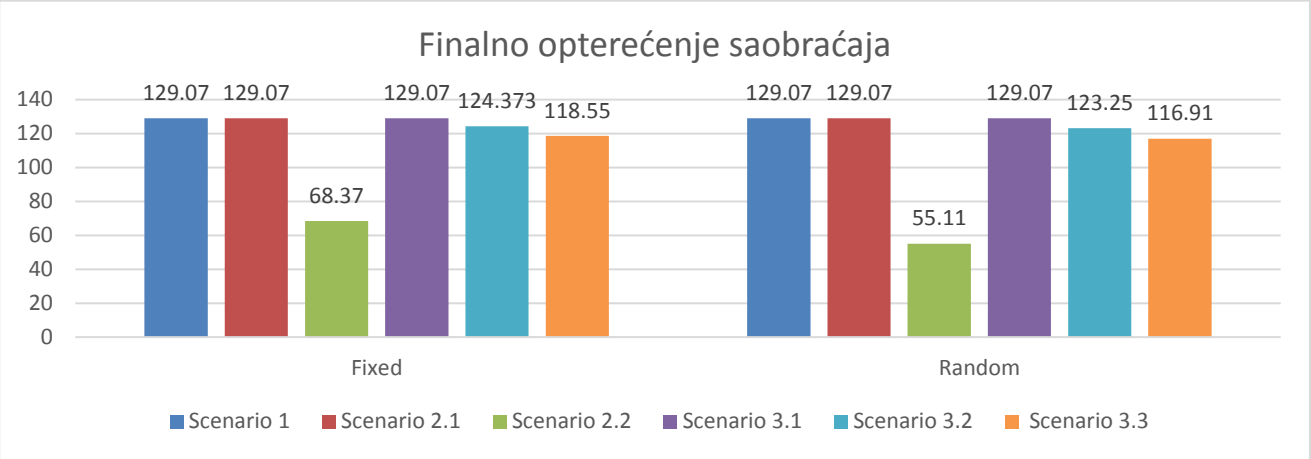
Ovim smo završili predstavljanje svih rezultata za matricu veličine 5x5x5.

4.3. Torus 6x6x6

Analizu počinjemo od inicijalnog opterećenja saobraćaja. Njegova vrednost zavisi isključivo od dimenzija torusa i biće jednaka kroz celu simulaciju za sve scenarije.

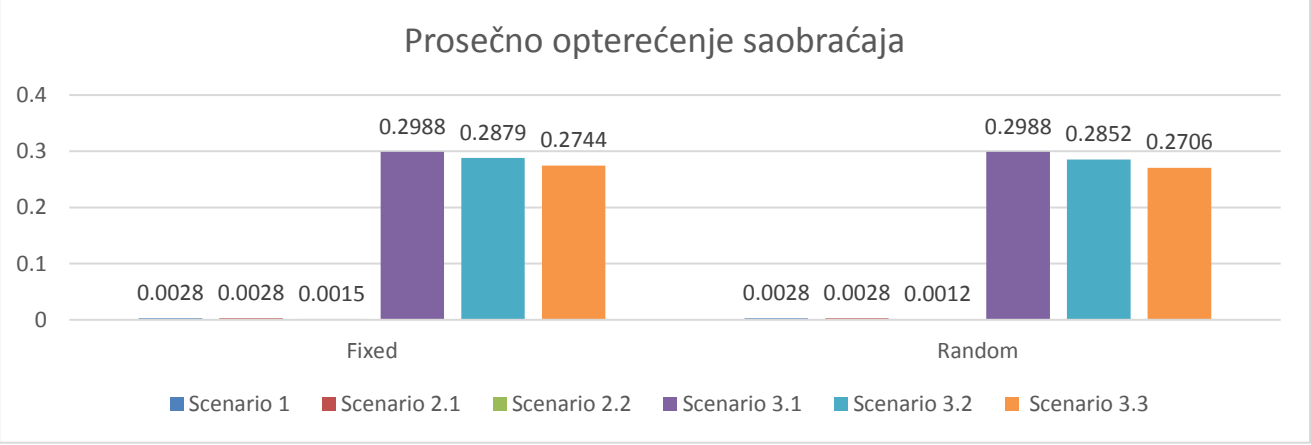
Inicijalno opterećenje saobraćaja = 129.07

Sledeći parametar je finalno opterećenje saobraćaja. Ovaj podatak će se razlikovati od inicijalnog ukoliko je došlo do zagušenja na linkovima:



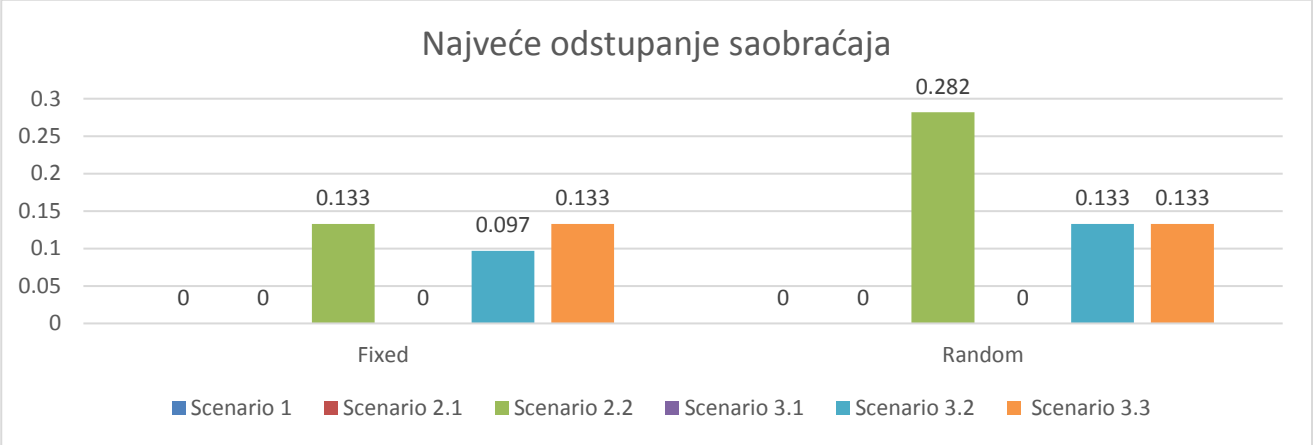
Slika 4.3.1. Finalno opterećenje saobraćaja.

Primećujemo sada da postoji znatno smanjenje inicijalnog saobraćaja u scenarijima 2.2, 3.2 i 3.3. Zatim sledi prosečna vrednost saobraćaja koja će se razlikovati od scenarija do scenarija.



Slika 4.3.2. Prosečno opterećenje saobraćaja.

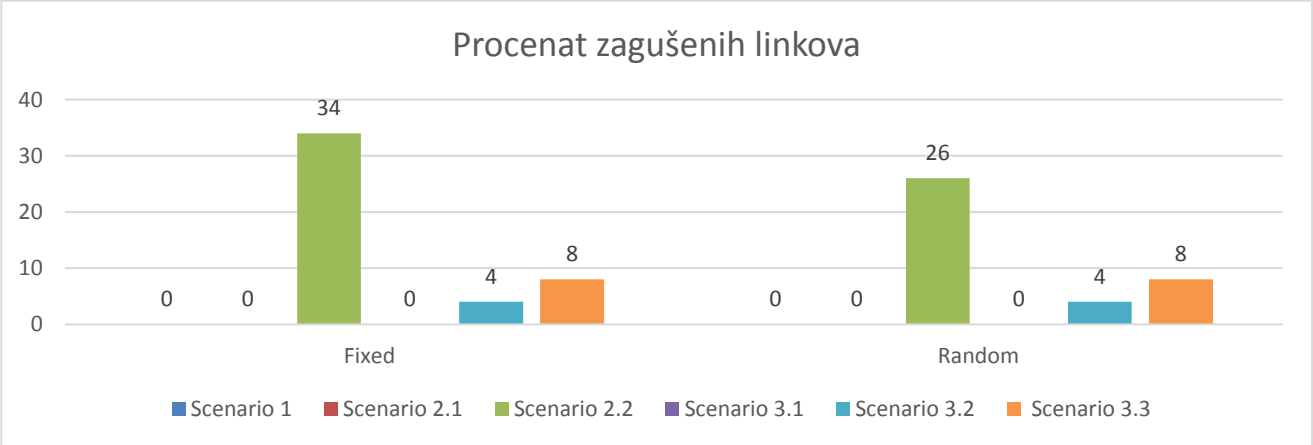
Kao i u prethodnom poglavlju opterećenje saobraćaja zavisi od samih definicija scenarija kao i od propusnosti torusa jer će biti računato na samome kraju simulacije. Poslednji parametar za saobraćaj koji ćemo obraditi je najveće odstupanje od željenog saobraćaja. Ovaj parametar će nam se pojaviti u slučaju da dođe do zagušenja na linkovima.



Slika 4.3.3. Najveće odstupanje saobraćaja.

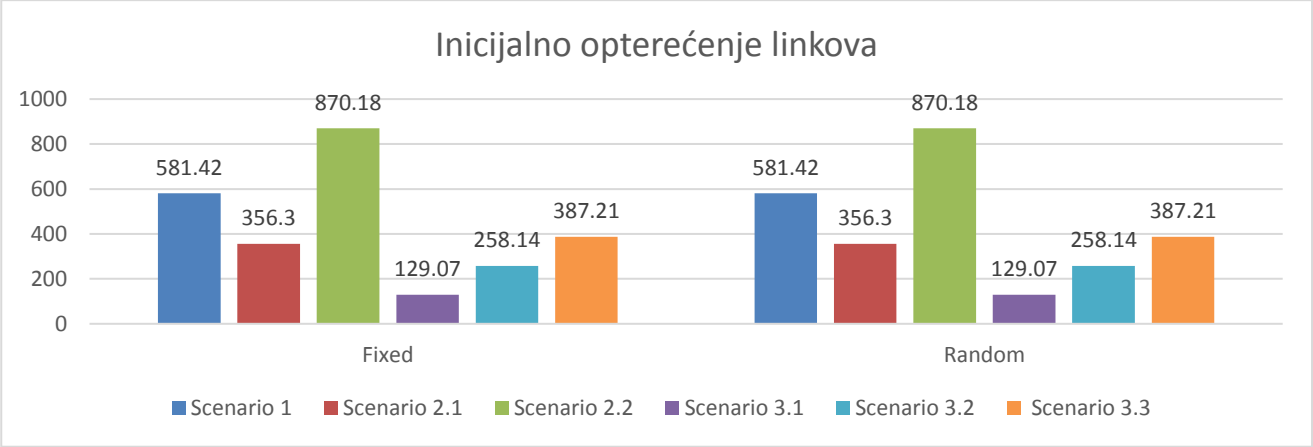
Sa Slike 4.3.3. očitavamo da Random tip rutiranja unosi veće smanjivanje inicijalnog saobraćaja za scenarije 2.2 i 3.2 dok je u scenariju 3.3 to zagušenje isto.

Prikazali smo sve parametre za saobraćaj i sada možemo da pređemo na linkove. Prvo ćemo prikazati procenat zagušenih linkova nakon prvog propuštanja saobraćaja, a pre smanjivanja ukoliko bude potrebe za njim.



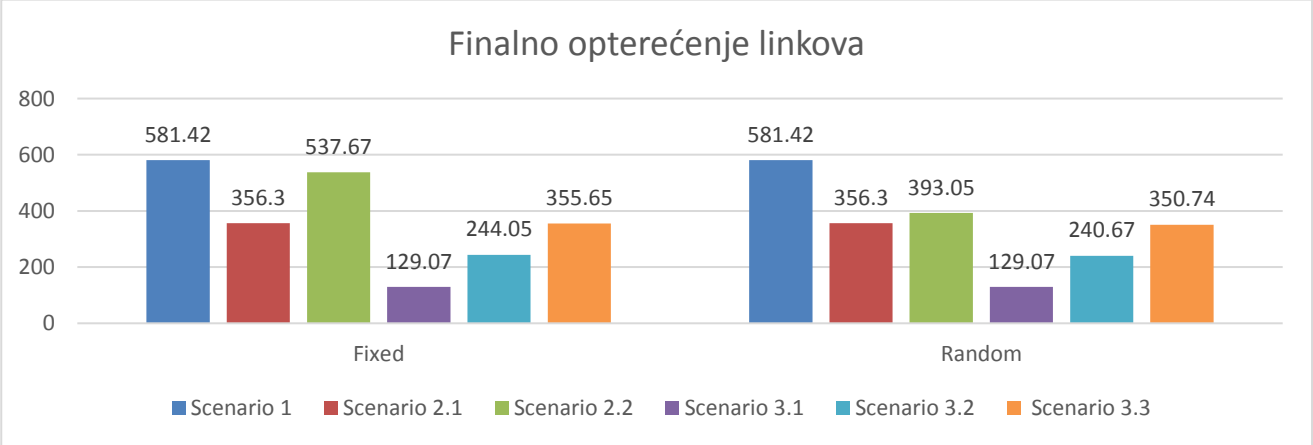
Slika 4.3.4. Procenat zagušenih linkova.

Slika 4.3.4. nam pokazuje da iako Random tip rutiranja unosi veće odstupanje od inicijalnog saobraćaja u scenariju 2.2 procenat zagušenih linkova u ovom slučaju biće manji. Analizu linkova nastavljamo sa inicijalnim optrerećenjima. Ovo ćemo prikazati na Slici 4.3.5.



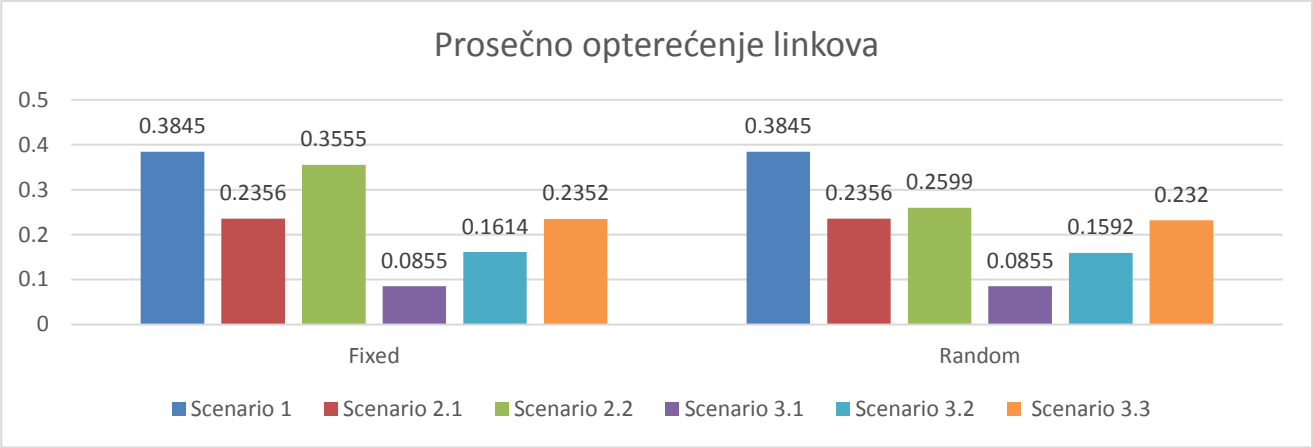
Slika 4.3.5. Inicijalno opterećenje linkova.

Zatim prikazujemo i finalno opterećenje linkova. Ovaj podatak će se razlikovati od prethodnog u slučaju da je došlo do zagušenja linkova.



Slika 4.3.6. Finalno opterećenje linkova.

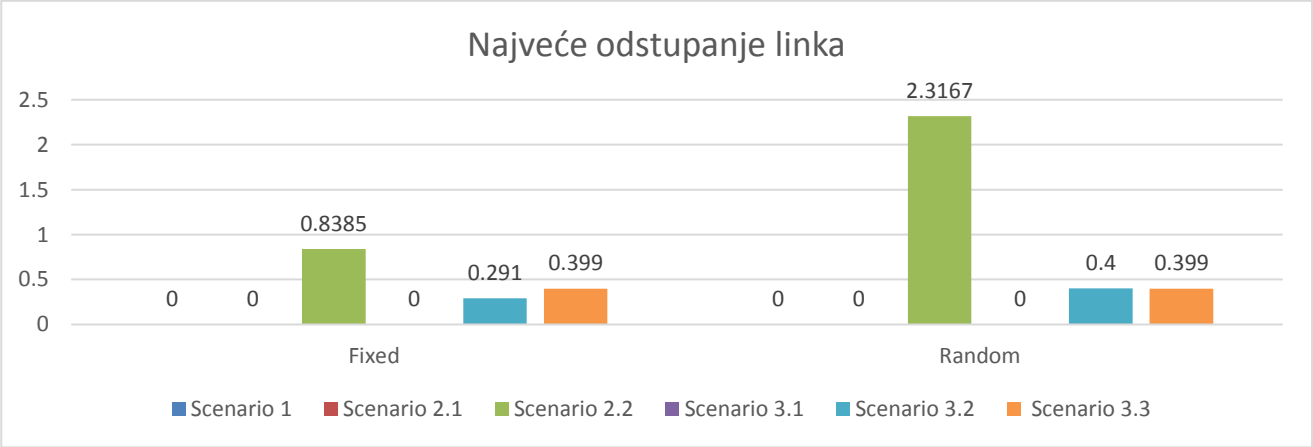
Za torus 6x6x6 u ovom slučaju se ispostavlja da Fiksno rutiranje unosi osetno manje zagušenja u linkove, tj. Fiksno rutiranje će nam obezbediti veću propusnost torusnog rutera. Nakon ovoga možemo prikazati i prosečno opterećenje linkova.



Slika 4.3.7. Prosečno opterećenje linkova.

Na osnovu Slike 4.3.7. zaključujemo da je torus za ponuđeni ulazni saobraćaj srednje opterećen. Najveće prosečno opterećenje linkova jeste u scenariju 1 i ono iznosi 0.3845.

Ostaje nam još jedan rezultat za prikazivanje a to je najveće odstupanje popunjenosti linka. I ovaj podatak će nam zavisiti od toga da li je došlo do zagušenja u linkovima.



Slika 4.3.8. Najveće odstupanje linka.

Sa Slike 4.3.8. očitavamo da je u ovoj simulaciji ostvareno približno 2.5 puta manje maksimalno odstupanje od inicijalne vrednosti linka korišćenjem Fiksnog tipa rutiranja. Takođe iako imamo slabu prosečnu popunjenost linkova vidimo da u scenariju 2.2 beležimo ogromna prekoračenja maksimalne propusnosti linka. Za pojedine linkove to prekoračenje iznosi približno 230%. Ova slika dodatno potvrđuje tvrdnju da nam je za ovu simulaciju veće odstupanje za Random tip rutiranja.

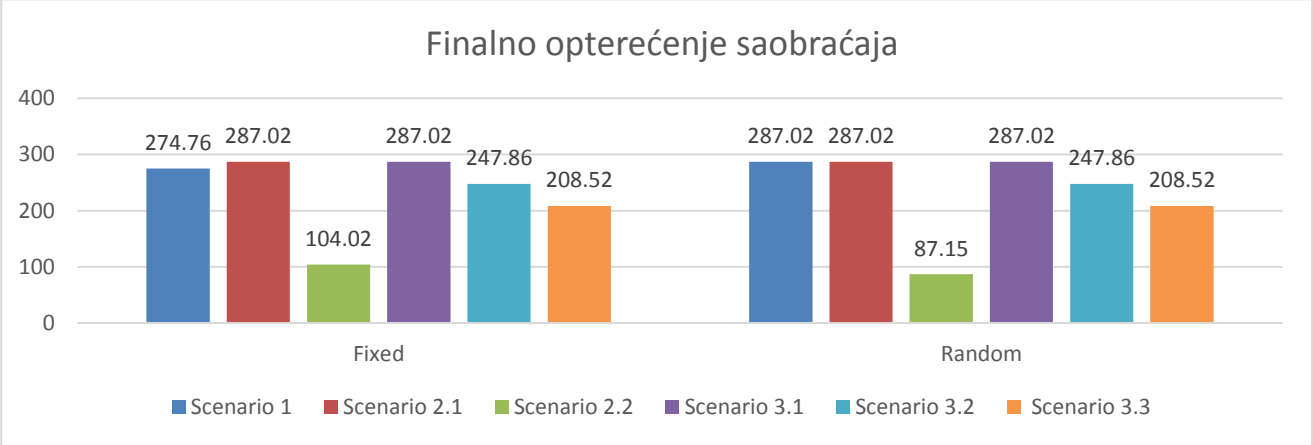
Ovim smo završili predstavljanje svih rezultata za matricu veličine 6x6x6.

4.4. Torus 8x8x8

Analizu počinjemo od inicijalnog opterećenja saobraćaja. Njegova vrednost zavisi isključivo od dimenzija torusa i biće jednaka kroz celu simulaciju za sve scenarije. Sa povećanjem dimenzija torusa proporcionalno raste i inicijalno opterećenje saobraćaja. Ono u ovom slučaju iznosi:

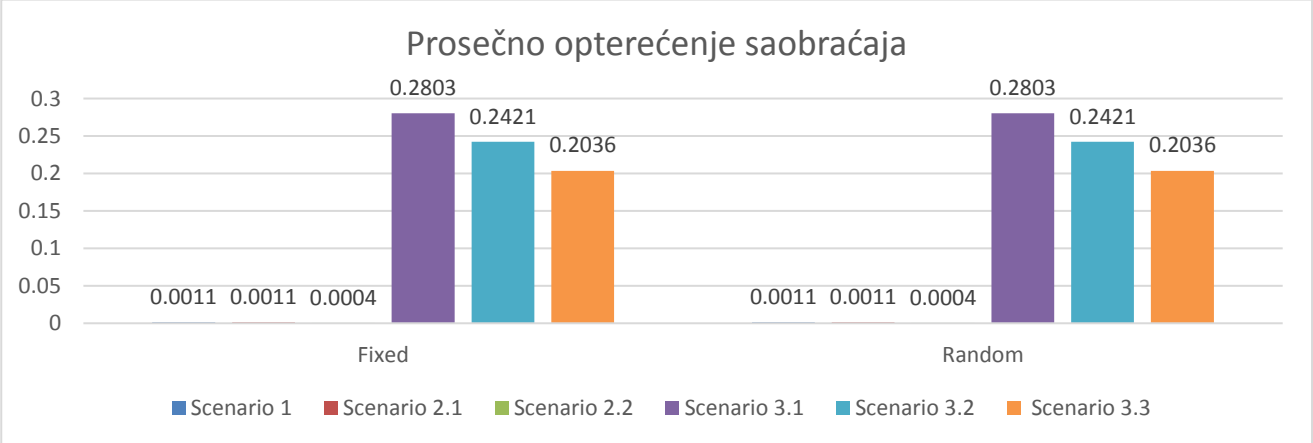
Inicijalno opterećenje saobraćaja = 287.02

Sledeći parametar je finalno opterećenje saobraćaja. Ovaj podatak će se razlikovati od inicijalnog ukoliko je došlo do zagušenja na linkovima:



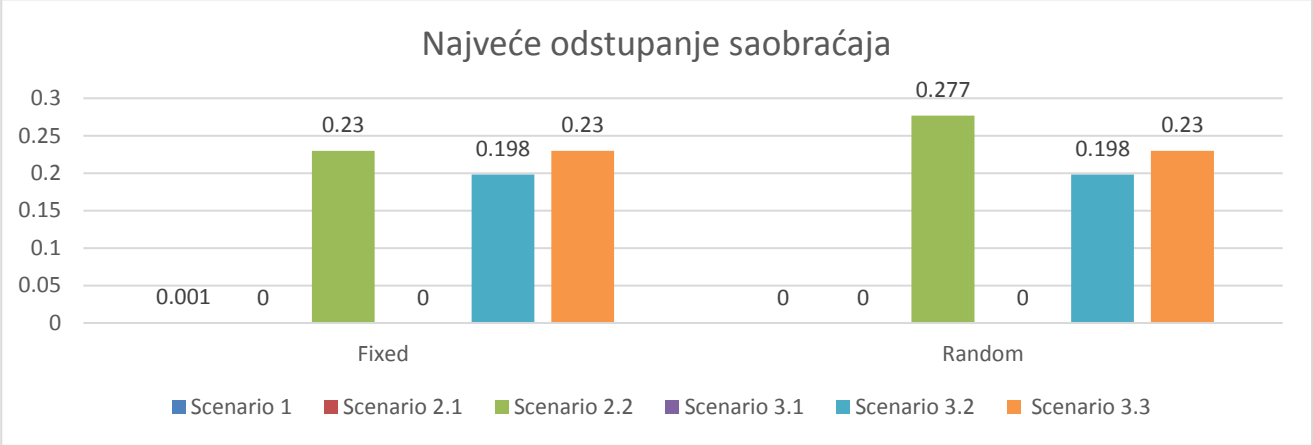
Slika 4.4.1. Finalno opterećenje saobraćaja.

Slika 4.4.1. pokazuje da u torusu 8x8x8 dolazi do odbacivanja inicijalnog saobraćaja za scenarije 1, 2.2, 3.2, 3.3. Očigledno je da sa povećanjem dimenzija matrice možemo očekivati i smanjenje propusnosti i u prvom scenariju. Zatim sledi prosečna vrednost saobraćaja koja će se razlikovati od scenarija do scenarija kao i od same propusnosti rorusnog rutera.



Slika 4.4.2. Prosečno opterećenje saobraćaja.

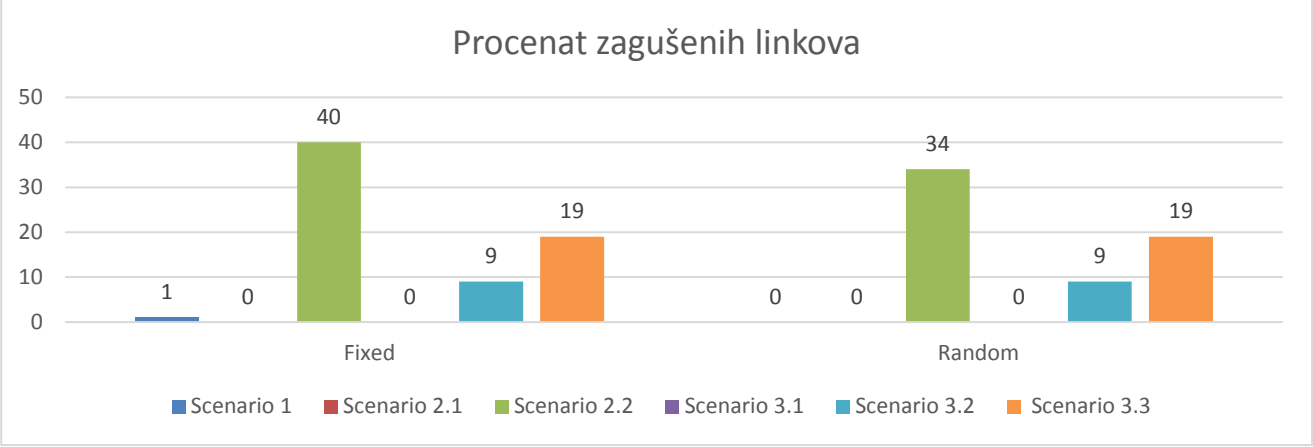
Poslednji parametar za saobraćaj koji ćemo obraditi je najveće odstupanje od željenog saobraćaja. Ovaj parametar će nam se pojaviti u slučaju da dođe do zagušenja na linkovima.



Slika 4.4.3. Najveće odstupanje saobraćaja.

Uzimajući u obzir da je korak smanjivanja saobraćaja definisan za sve simulacije i iznosi 0.001 sa Slike 4.4.3. vidimo da se za pojedine saobraćaje u scenarijima smanjivanje vršilo više od 200 puta.

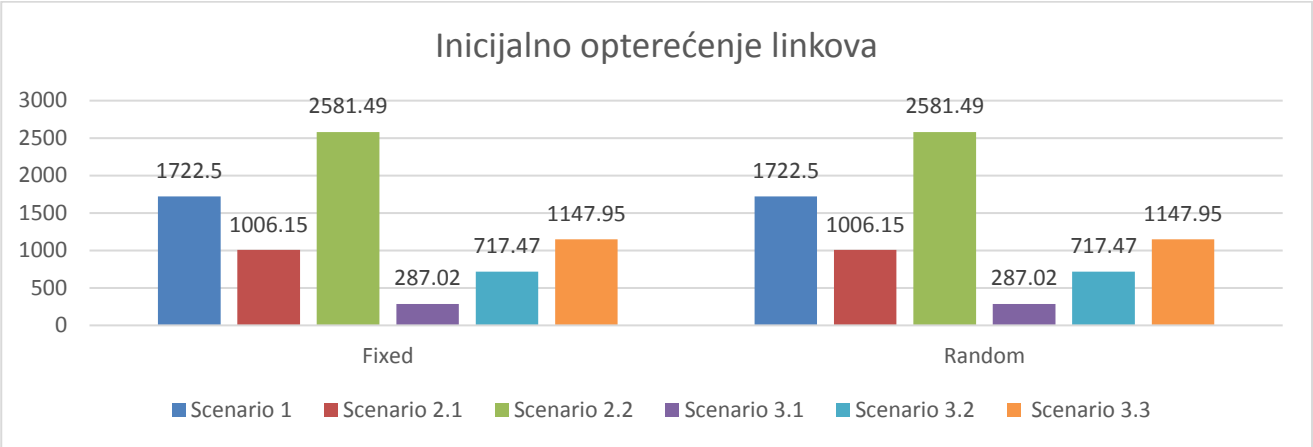
Zatim prelazimo na linkove. Prvo ćemo prikazati procenat zagušenih linkova nakon prvog propuštanja saobraćaja, a pre smanjivanja ukoliko bude potrebe za njim.



Slika 4.4.4. Procenat zagušenih linkova.

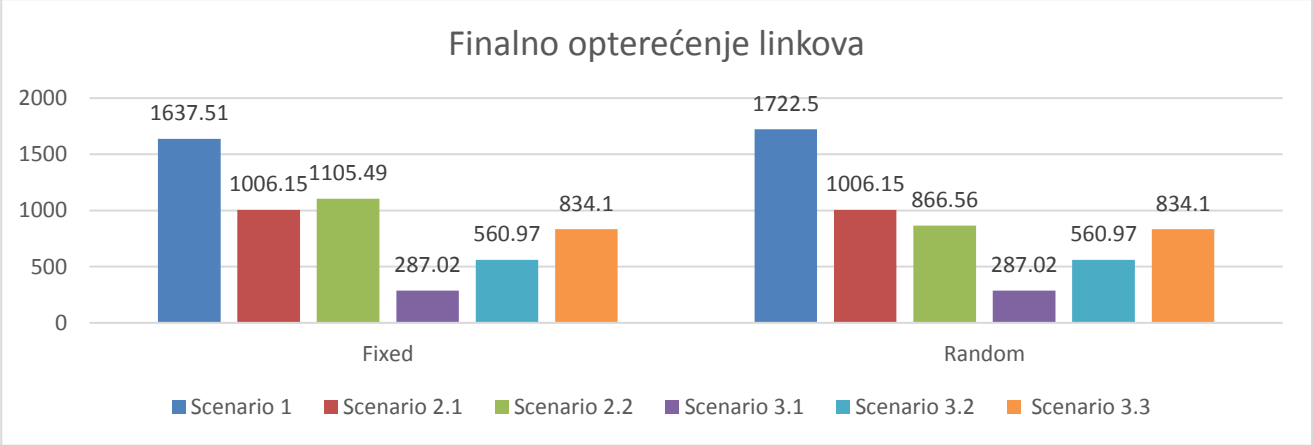
Sa Slike 4.4.4. očitavamo da je u torusu 8x8x8 za scenario 2.2 procenat zagušenih linkova skočio na 40%. Ovo će dosta uticati na količinu odbačenog saobraćaja, tj. samu propusnost rutera. Takođe ne treba zanemariti ni 19% zagušenja na linkovima u scenariju 3.3.

Analizu linkova nastavljamo sa inicijalnim optrerećenjima. Ovo ćemo prikazati na Slici 4.4.5.



Slika 4.4.5. Inicijalno opterećenje linkova.

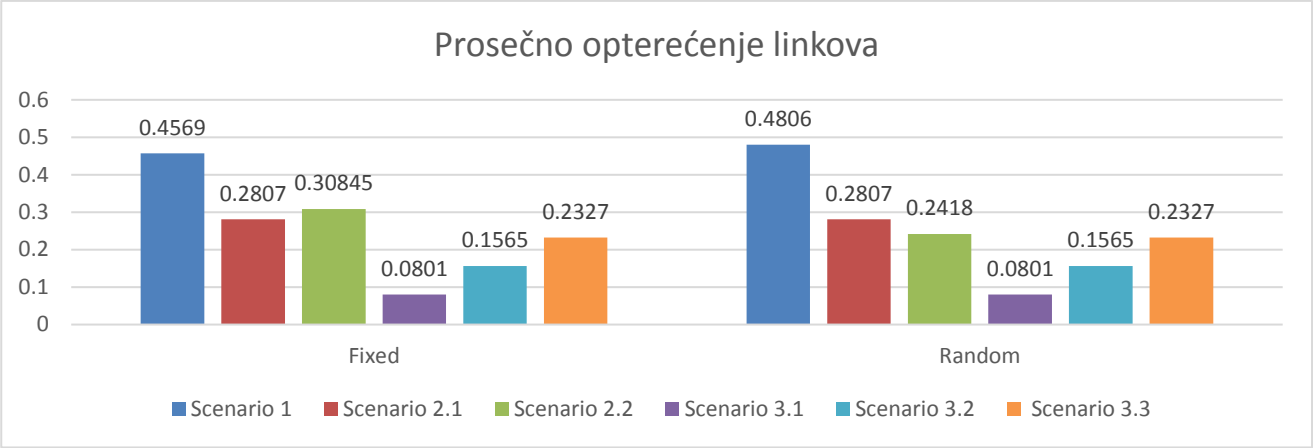
Zatim prikazujemo i finalno opterećenje linkova. Ovaj podatak će se razlikovati od prethodnog u slučaju da je došlo do zagušenja linkova.



Slika 4.4.6. Finalno opterećenje linkova.

Kao što smo i videli iz prethodnih rezultata najveće odstupanje opterećenja linkova jeste za scenarije 2.2 i 3.3 i ono je prikazano na Slici 4.4.6. Još jedan zanimljiv podatak je da se prilikom ove simulacije za scenarije 3.2 i 3.3 poklopilo da je za oba tipa rutiranja ostvarena jednaka propusnost linkova.

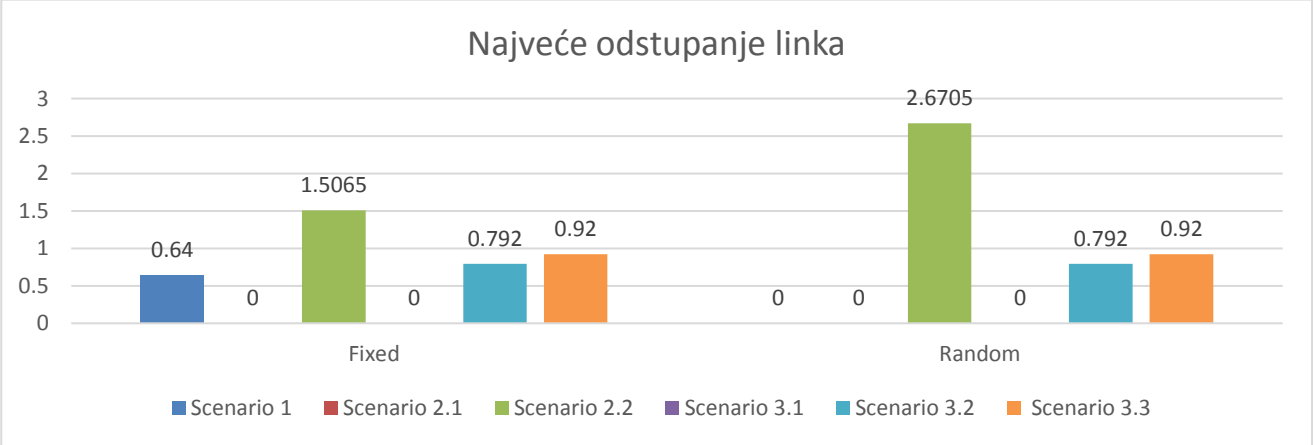
Nakon ovoga možemo prikazati i prosečno opterećenje linkova.



Slika 4.4.7. Prosečno opterećenje linkova.

Na osnovu Slike 4.4.7. zaključujemo da je naš torus 8x8x8 srednje popunjen i ta vrednost ni u jednom scenariju ne prelazi 50%.

Ostaje nam još jedan rezultat za prikazivanje a to je najveće odstupanje popunjenosti linka. I ovaj podatak će nam zavisiti od toga da li je došlo do zagušenja u linkovima.



Slika 4.4.8. Najveće odstupanje linka.

Nasuprot srednjem opterećenju linkova torusa 8x8x8 Slika 4.4.8. nam pokazuje da je to opterećenje neravnomerno raspoređeno i da u pojedinim scenarijima za određene linkove imamo premašaj od 267%.

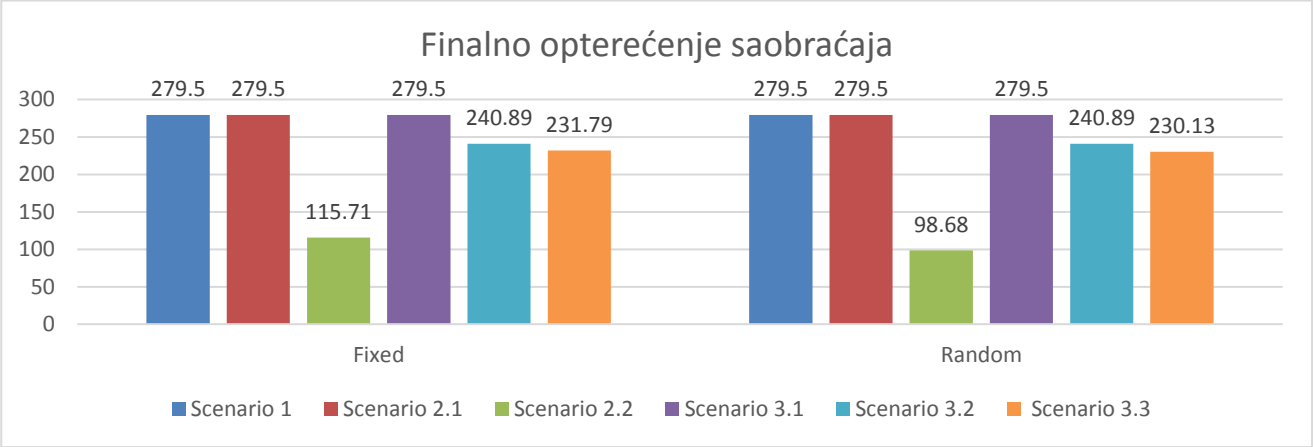
Ovim smo završili predstavljanje svih rezultata za matricu veličine 8x8x8.

4.5. Torus 7x8x9

Analizu počinjemo od inicijalnog opterećenja saobraćaja. Njegova vrednost zavisi isključivo od dimenzija torusa i biće jednaka kroz celu simulaciju za sve scenarije.

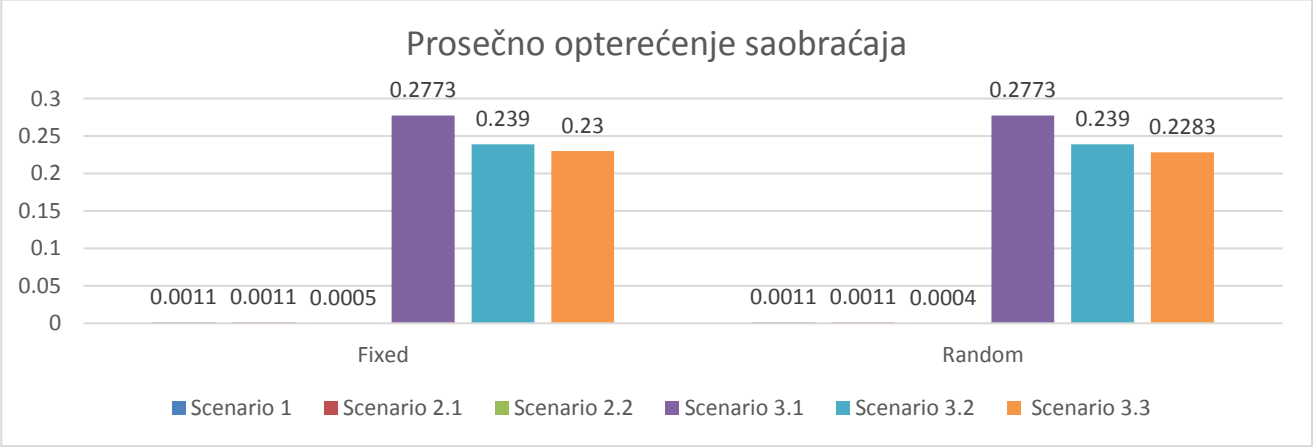
Inicijalno opterećenje saobraćaja = 279.5

Sledeći parametar je finalno opterećenje saobraćaja. Ovaj podatak će se razlikovati od inicijalnog ukoliko je došlo do zagušenja na linkovima:



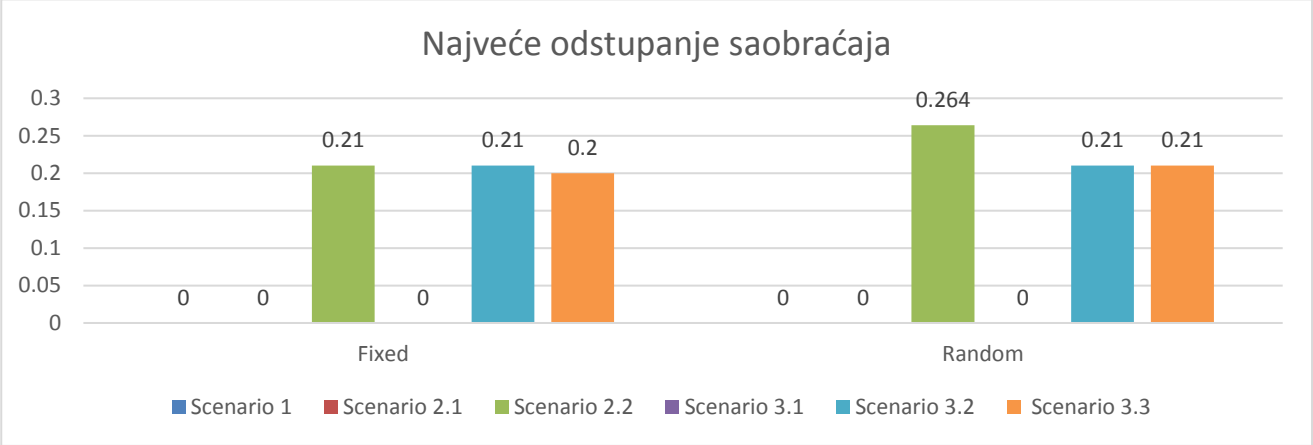
Slika 4.5.1. Finalno opterećenje saobraćaja.

Na Slici 4.5.1. zapažamo da ćemo za torus 7x8x9 u ovoj simulaciji imati smanjivanje propusnosti saobraćaja za scenarije 2.2, 3.2 i 3.3 i to za oba tipa rutiranja. Zatim sledi prosečna vrednost saobraćaja koja će se razlikovati od scenarija do scenarija kao i od same propusnosti torusnog rutera.



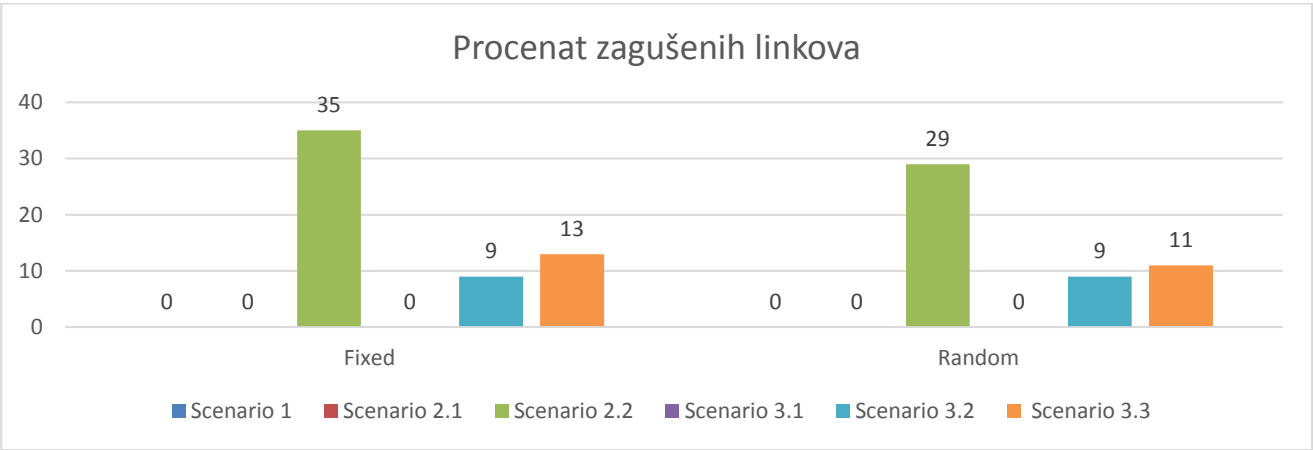
Slika 4.5.2. Prosečno opterećenje saobraćaja.

Poslednji parametar za saobraćaj koji ćemo obraditi je najveće odstupanje od željenog saobraćaja. Ovaj parametar će nam se pojaviti u slučaju da dođe do zagušenja na linkovima.



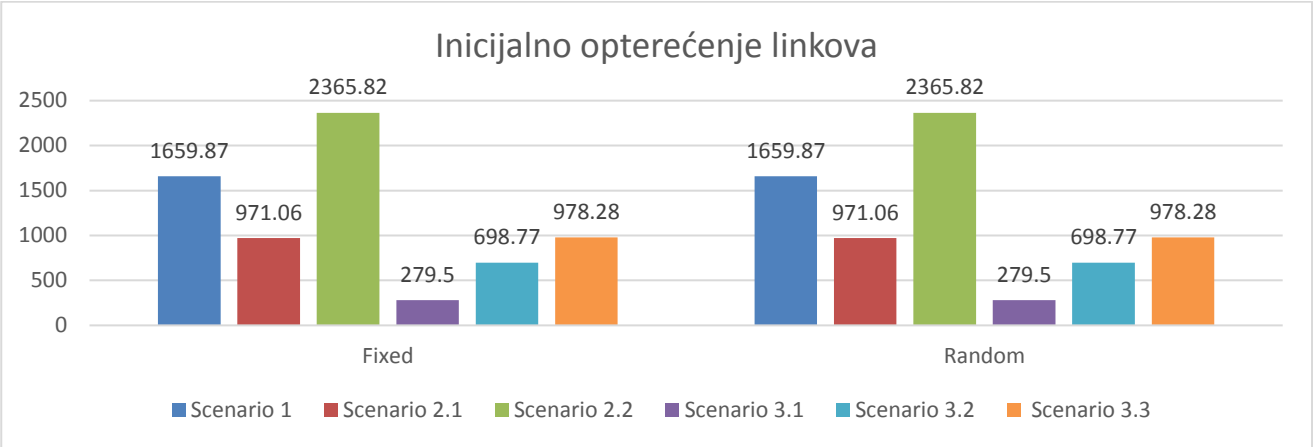
Slika 4.5.3. Najveće odstupanje saobraćaja.

Uzimajući u obzir korak smanjivanja od 0.001, na Slici 4.5.3. vidimo da je za pojedine scenarije vršeno redukovanje početnog saobraćaja više od 200 puta. Zatim prelazimo na linkove. Prvo ćemo prikazati procenat zagušenih linkova nakon prvog propuštanja saobraćaja, a pre smanjivanja ukoliko bude potrebe za njim.



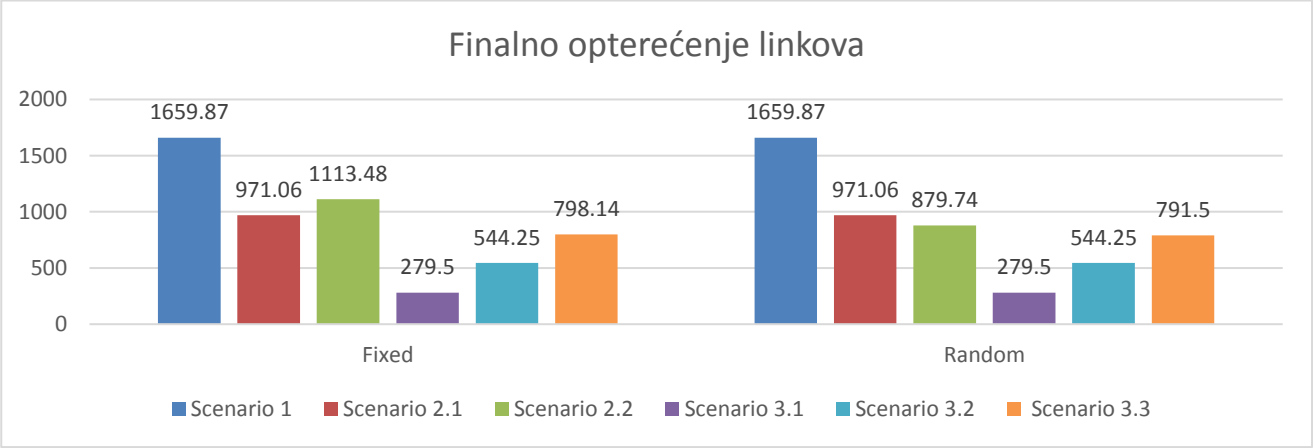
Slika 4.5.4. Procenat zagušenih linkova.

U ovoj simulaciji imamo slučaj da je sa Random tipom rutiranja ostvarena veća propusnost u scenarijima 2.2 i 3.3. Analizu linkova počinjemo od inicijalnog optrerećenja. Ovo ćemo prikazati na Slici 4.5.5.



Slika 4.5.5. Inicijalno opterećenje linkova.

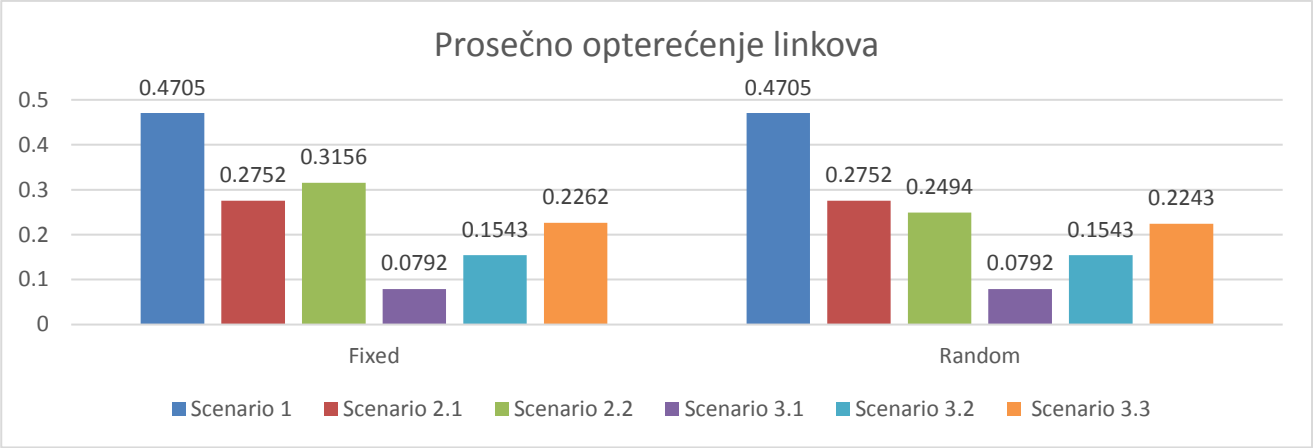
Zatim prikazujemo i finalno opterećenje linkova. Ovaj podatak će se razlikovati od prethodnog u slučaju da je došlo do zagušenja linkova.



Slika 4.5.6. Finalno opterećenje linkova.

Na Slici 4.5.6. vidimo da je veća propusnost torusa u scenariju 2.2 za Fiksni tip rutiranja iako je veći broj inicijalno zagušenih linkova. Ova razlika propusnosti nije zanemarljiva i iznosi više od 20%. Pored ovoga očitavamo da je takođe za Fiksni tip rutiranja u scenariju 3.3 ostvaren veći finalni protok uprkos većem broju zagušenih linkova.

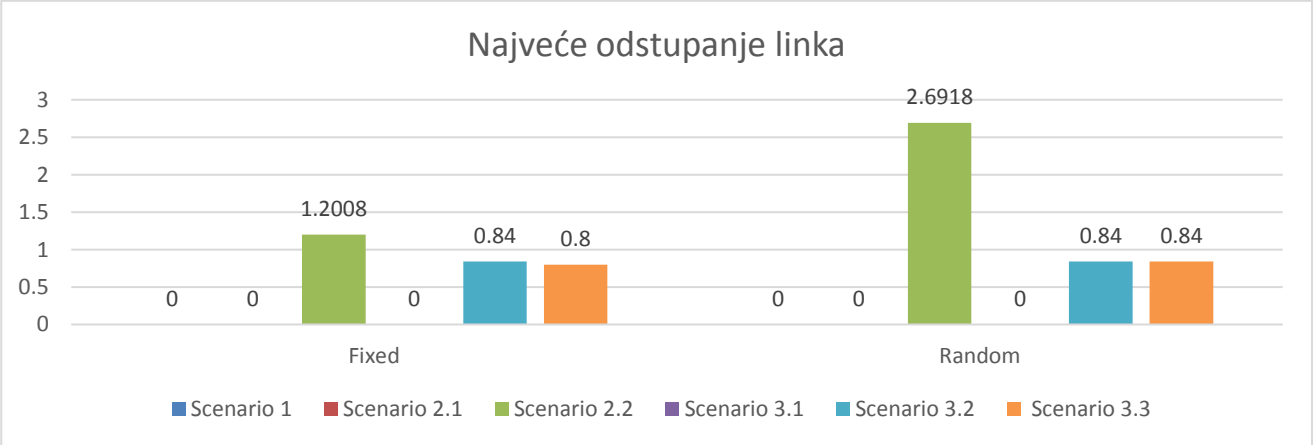
Nakon ovoga možemo prikazati i prosečno opterećenje linkova.



Slika 4.5.7. Prosečno opterećenje linkova.

Na Slici 4.5.7. vidimo da je torus 7x8x9 srednje opterećen saobraćajem kao i da prosečno opterećenje ne prelazi 50% maksimalne popunjenosti linka ni u jednom scenariju.

Ostaje nam još jedan rezultat za prikazivanje a to je najveće odstupanje popunjenosti linka. I ovaj podatak će nam zavisiti od toga da li je došlo do zagušenja u linkovima.



Slika 4.5.8. Najveće odstupanje linka.

Na Slici 4.5.8. očitavamo da će i za torus 7x8x9 biti potrebno više od 200 koraka simulacije kako bi se svi linkovi doveli u ne zagušeno stanje. Takođe, prikazana je velika razlika u korist Fiksnog tipa rutiranja gde je maksimalno odstupanje linka više nego duplo manje u poređenju sa Random tipom rutiranja.

Slikom 4.5.8. smo završili predstavljanje svih rezultata za matricu veličine 7x8x9.

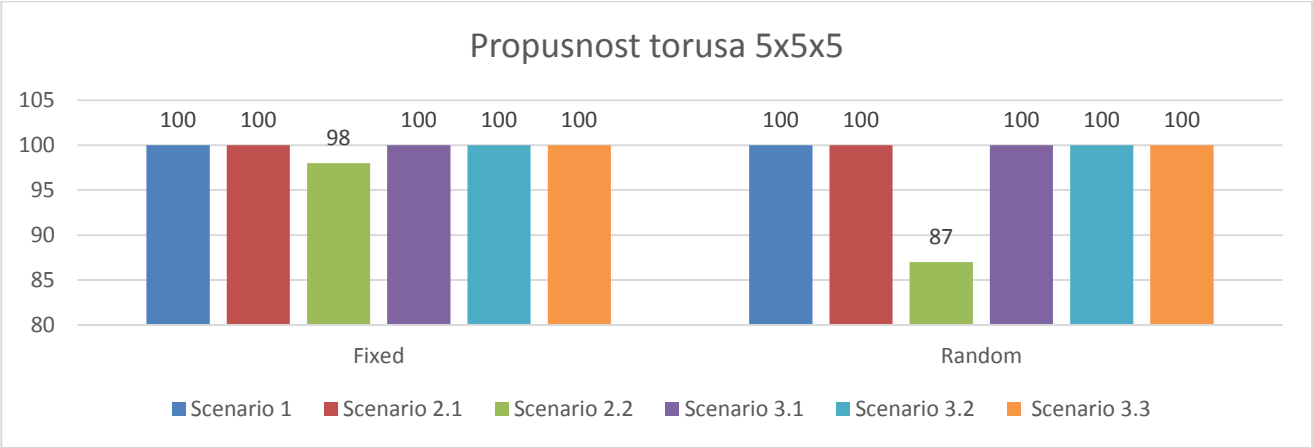
4.6. Analiza propusnosti saobraćaja torusa različitih dimenzija

Ovime je završeno predstavljanje rezultata očitanih iz pokrenutih situacija.

Sada ćemo se posvetiti još jednom podatku a to je propusnost celokupnog torusa. Ovaj podatak ćemo izračunati poredeći inicijalno ulazno opterećenje saobraćaja sa finalnim opterećenjem saobraćaja koje je dobijeno nakon umanjivanja opterećenja zagušenih linkova i biće izraženo u procentima. Za svaku veličinu torusa prikazaćemo po jedan grafik upoređujući propusnost za oba tipa rutiranja.

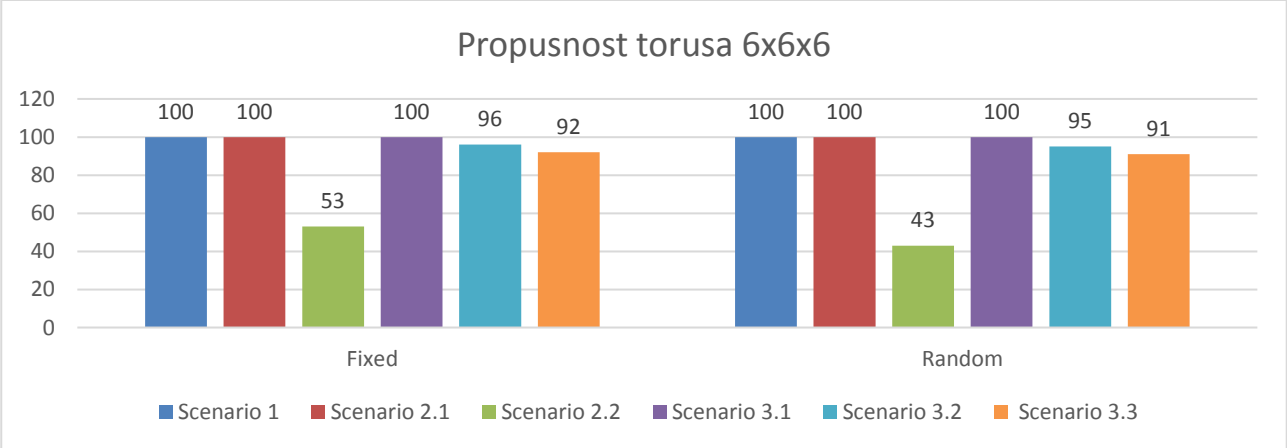
Počećemo od torusa 5x5x5.

Inicijalno opterećenje saobraćaja (5x5x5) = 60.54



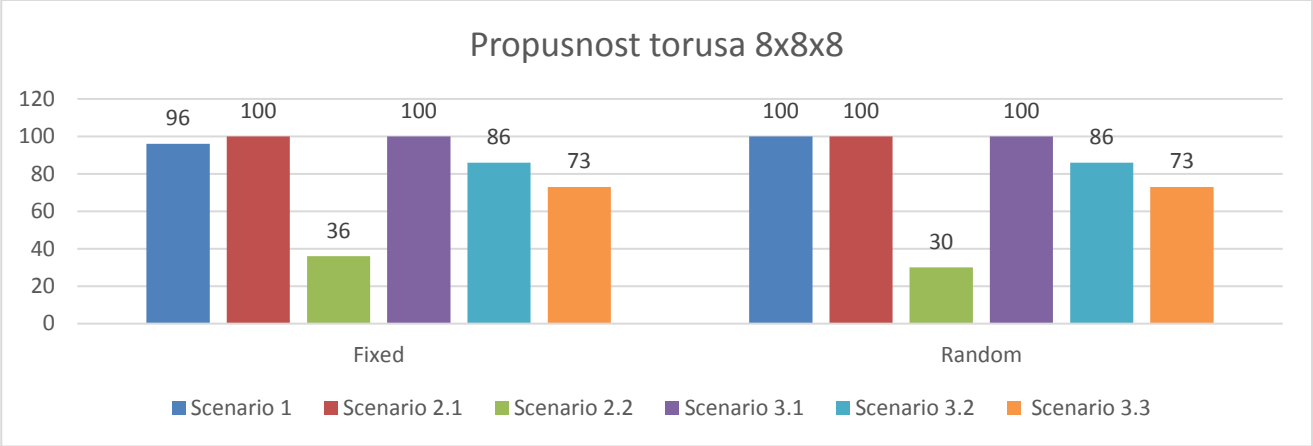
Slika 4.6.1. Propusnost torusa 5x5x5.

Inicijalno opterećenje saobraćaja (6x6x6) = 129.07



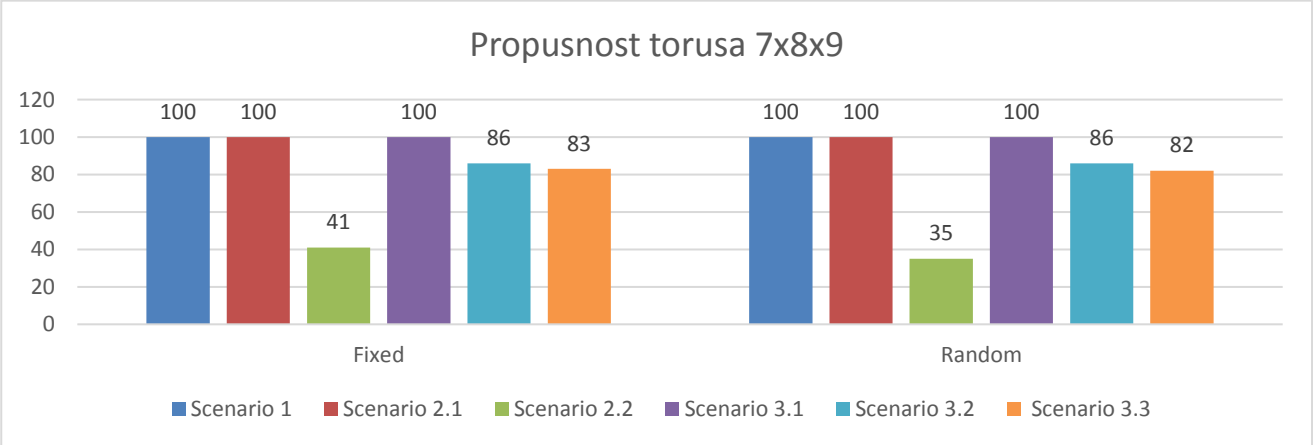
Slika 4.6.2. Propusnost torusa 6x6x6.

Inicijalno opterećenje saobraćaja (8x8x8) = 287.02



Slika 4.6.3. Propusnost torusa 8x8x8.

Inicijalno opterećenje saobraćaja (7x8x9) = 279.5



Slika 4.6.4. Propusnost torusa 7x8x9.

Prikazani grafici pokazuju da će nam propusnost samih rutera najviše zavisiti od izbora scenarija odnosno od načina raspodele generisanog ulaznog saobraćaja. Već sa prve Slike 4.6.1. vidimo da će najmanje ostvarenog protoka u torusu veličine 5x5x5 biti za scenario 2.2. Takođe možemo primetiti razliku u odnosu na tipove rutiranja. Prilikom ove simulacije Fiksni tip se pokazao kao bolji izbor jer je uspeo da propusti približno 12% više saobraćaja. Ovo se može objasniti time da Random tip rutiranja ima ne konzistentnu raspodelu saobraćaja ka odredištima dok je za Fiksni tip ta raspodela unapred definisana. Osim scenarija 2.2 svi ostali su imali maksimalnu propusnost od 100%.

Prelazimo na torus 6x6x6 koji ima skoro duplo više čvorova od torusa 5x5x5. Posmatrajući Sliku 4.6.2. vidimo da se za ovu dimenziju situacija sa propusnošću dosta izmenila. U ovom slučaju čak tri scenarija nisu ostvarila maksimalnu propusnost a to su 2.2, 3.2 i 3.3. Scenariji 3.2 i 3.3 su uspeli da održe visokom svoju propusnost - iznad 90 % dok je u scenariju 2.2 za Random tip rutiranja ta propusnost pala na 43%. Jasno je da je u ovakvim situacijama potrebna promena načina rutiranja ukoliko želimo da povećamo propusnost jer nam ova dva tipa koja smo koristili ne mogu to obezbediti. Sa Slike 4.6.2. takođe vidimo i da se prilikom ove simulacije Fiksni tip rutiranja pokazao boljim, tj. sa njime uspevamo da prosledimo veću količinu saobraćaja. Zaključujemo i da je porastom broja čvorova opala propusna moć našeg rutera za ovako definisane scenarije.

Sledeći na redu je torus 8x8x8 koji poređenja radi ima 4 puta više čvorova od torusa 5x5x5 i više od dva puta od torusa 6x6x6. Sa Slike 4.6.3. možemo da vidimo da za ovu simulacija čak 4 scenarija ne ostvaruju maksimalnu propusnost i to su 1, 2.2, 3.2 i 3.3. Poredeći Slike 4.6.3. i 4.6.2. možemo zaključiti da porastom broja čvorova protok u našim scenarijima znatno opada. Za torus 8x8x8 po prvi put se javilo ostupanje od maksimalne propusnosti u prvom scenariju i ovo nam nagoveštava da ćemo sa daljim povećanjem dimenzija torusa osetiti pad propusnosti i u ovom scenariju. Za scenarije 3.2 i 3.3 propusnost se spustila ispod 90% i iznosi 86% i 73%. Takođe zanimljivo je videti da je ostvarena ista propusnost u ovim scenarijima korišćenjem različitih tipova rutiranja. Za scenario 2.2 propusnost je pala na 36%, tj. na 30% za random tip rutiranja i kao što smo ranije naveli potrebno je razmisliti o uvođenju novog tipa rutiranja ili proširenju linkova kako bi naš torus uspešno opslužio ovaj scenario. U ovoj simulaciji svaki tip rutiranja se pokazao jedanput kao bolji i to Random tip u 1. scenariju, a Fiksni u scenariju 2.2.

Za kraj preostaje nam torus 7x8x9 čija je propusnost prikazana na Slici 4.6.4. Ovu dimenziju za torus smo izabrali kako bismo istestirali i jednu nepravilnu strukturu i uporedili je sa ostalim rezultatima i ovom smo prilikom vodili računa da broj čvorova bude približno jednaka broju čvorova torusa dimenzija 8x8x8. Sa Slike 4.6.4. vidimo da je propusnost umanjena kao i u prethodnim simulacijama za scenarije 2.2, 3.2 i 3.3. Za scenarije 3.2 i 3.3 ta propusnost iznosi približnih 85% dok je za scenario 2.2 ona znatno umanjena i za Random tip rutiranja iznosi 35%. Takođe upoređujući slike 4.6.3. i 4.6.4. zaključujemo da u slučaju nepravilne strukture 7x8x9 nije došlo do smanjenja propusnosti, već je naprotiv ostvarena propusnost veća i to u scenarijima 1, 2.2 i 3.3.

5. ZAKLJUČAK

Torusna arhitektura je jedno od jednostavnijih rešenja koje se može koristiti prilikom projektovanja komutacionog polja. U prilog torusnoj arhitekturi ide i mogućnost lakog proširivanja bez povećavanja kompleksnosti komutacije. Data karakteristika se mogla primetiti prilikom programiranja simulacije jer se nije javila potreba za dodavanjem novih funkcija niti je bilo potrebno izmeniti postojeće funkcije prilikom povećavanja dimenzija torusa.

S druge strane u prikazanim rezultatima uočena je i glavna mana dva opisana tipa rutiranja, a to je problem zagušenja na linkovima i samim tim smanjena celokupna propusnost torusnog rutera. Ovaj problem je značajan kod torusa sa većim dimenzijama i biva izraženiji u određenim scenarijima i podscenarijima sa daljim povećanjem dimenzija.

Analizom torusa nekoliko različitih dimenzija dolazimo do zaključka da će propusnost rutera primarno zavisiti od definicije samih scenarija i podscenarija za raspodelu slučajno generisanog saobraćaja kao i od samih dimenzija i tipa rutiranja. Na osnovu različitih simulacija zaključeno je da se sa Random tipom rutiranja ostvaruje manja propusnost u torusu, tj. sa Fiksnim načinom rutiranja uspeli smo da prosledimo veće količine ulaznog saobraćaja. Za scenario 2.2 primećeno je da propusnost dramatično opada sa povećanjem dimenzija i u ovom slučaju bi trebalo razmisliti o primeni drugačijeg tipa rutiranja. Takođe iz očitanih rezultata zaključujemo da strukture nepravilnih oblika 7x8x9 ne moraju podrazumevano imati manju propusnost od pravilnih struktura, šta više u našoj simulaciji imali smo upravo obrnut slučaj. Pokretanjem i testiranjem koda utvrđeno je da je vreme za izvršavanja simulacije za scenario 2.2 za oba tipa rutiranja znatno veće od ostalih i može iznositi i preko 300 sekundi za torus dimenzija 8x8x8 i u ovom scenariju postoji prostor da se pokuša sa optimizacijom koda radi povećanja preformansi.

Važno je napomenuti, da se postojeća tema može dodatno proširiti uključivanjem dodatnih opcija i ograničenja u simulaciju, kao što su adaptivno i balansirano rutiranje, menjanjem korišćenih vrednosti za maksimalnu propusnost linka i koraka smanjivanja saobraćaja, redefinisanjem ili dodavanjem novih scenarija, koji bi se mogli ispitivati u nekim od budućih radova.

LITERATURA

[1] Z. Čiča, *Komutacioni sistemi* – predavanja, [Online]. Available: http://telekomunikacije.etf.bg.ac.rs/predmeti/te4ks/ks.php

[2] L. Kraus, *Programski jezik C++ sa rešenim zadacima*, Akademska misao, Beograd, 2011.

[3] Cluster Design, *Torus Networks Design*, [Online]. Available: http://clusterdesign.org/torus/

[4] Wikipedia, *Torus*, [Online]. Available: http://en.wikipedia.org/wiki/Torus

[5] http://users.informatik.uni-halle.de/~jopsi/dpar03/chap2.shtml

[6] http://sankofa.loc.edu/savur/web/Parallel.html

[7] http://pages.cs.wisc.edu/~tvrdik/5/html/Section5.html

[8] http://www.eurotech.com/DLA/Products_Eurotech/Aurora/HPC_3D_Torus_Short_Paper.pdf

[9] http://wiki.expertiza.ncsu.edu/index.php/CSC/ECE_506_Spring_2010/ch_12_PP

[10] http://www.hindawi.com/journals/ijdsn/2012/124245/fig3/

[11] http://www.rugusavay.com/what-is-a-torus-shape/

SPISAK SKRAĆENICA

SPISAK SLIKA

Prilog:

Header.h

```
#pragma once

#include <cassert>
#include <vector>
#include <iostream>

struct TripleInt {
	int x, y, z;
};

inline bool operator==(TripleInt const & a, TripleInt const & b){
	return a.x == b.x && a.y == b.y && a.z == b.z;
}

inline bool operator!=(TripleInt const & a, TripleInt const & b){
	return !(a == b);
}

inline TripleInt operator + (TripleInt const & a, TripleInt const & b)
{
	return{a.x + b.x, a.y + b.y, a.z + b.z};
}

inline TripleInt operator - (TripleInt const & b){
	return {-b.x, -b.y, -b.z};
}

inline TripleInt operator - (TripleInt const & a, TripleInt const & b)
{
	return{ a.x - b.x, a.y - b.y, a.z - b.z };
}

TripleInt const NEIGHBOR_DIRECTIONS[6] = {
	{ -1, 0, 0 },
	{ 1, 0, 0 },
	{ 0, -1, 0 },
	{ 0, 1, 0 },
	{ 0, 0, -1 },
	{ 0, 0, 1 }
};

template<class T>
class Matrix3D
{
public: // typedefs
	using iterator = typename std::vector<T>::iterator;
	using const_iterator = typename std::vector<T>::const_iterator;

private: // fields
	int m_size_x = 0;
	int m_size_y = 0;
	int m_size_z = 0;
```

```
        std::vector<T>  m_data;

public: // ctors, assignments, dtors
        Matrix3D()
        {}

        Matrix3D(Matrix3D const& other)
                : m_size_x{ other.m_size_x }
                , m_size_y{ other.m_size_y }
                , m_size_z{ other.m_size_z }
                , m_data{ other.m_data }
        {}

        Matrix3D(Matrix3D&& other)
                : m_size_x{ other.m_size_x }
                , m_size_y{ other.m_size_y }
                , m_size_z{ other.m_size_z }
                , m_data(std::move(other.m_data))
        {}

        Matrix3D& operator= (Matrix3D const& other)
        {
                if (this != &other)
                {
                        this->m_size_x = other.m_size_x;
                        this->m_size_y = other.m_size_y;
                        this->m_size_z = other.m_size_z;
                        this->m_data = other.m_data;
                }
                return *this;
        }

        Matrix3D& operator= (Matrix3D&& other)
        {
                if (this != &other)
                {
                        this->m_size_x = other.m_size_x;
                        this->m_size_y = other.m_size_y;
                        this->m_size_z = other.m_size_z;
                        this->m_data = std::move(other.m_data);
                }
                return *this;
        }

        ~Matrix3D() {}

        Matrix3D(int size_x, int size_y, int size_z)
                : m_size_x{ size_x }
                , m_size_y{ size_y }
                , m_size_z{ size_z }
                , m_data(size_x*size_y*size_z)
        {
                assert(size_x > 0);
                assert(size_y > 0);
                assert(size_z > 0);
        }

public: // observers
        int size() const { return m_data.size(); }
        TripleInt sizes() const { return{ m_size_x, m_size_y, m_size_z }; }
```

```
        int size_x() const { return m_size_x; }
        int size_y() const { return m_size_y; }
        int size_z() const { return m_size_z; }

        iterator begin() { return m_data.begin(); }
        iterator end() { return m_data.end(); }

        const_iterator begin() const { return m_data.begin(); }
        const_iterator end() const { return m_data.end(); }

        const_iterator cbegin() const { return m_data.cbegin(); }
        const_iterator cend() const { return m_data.cend(); }

        T& el(int x, int y, int z)
        {
                return m_data[pos(x, y, z)];
        }

        T& el(TripleInt const & coord)
        {
                return m_data[pos(coord)];
        }

        T const& el(int x, int y, int z) const
        {
                return m_data[pos(x, y, z)];
        }

        T const& el(TripleInt const  & coord) const
        {
                return m_data[pos(coord)];
        }

public: // methods

        int pos(int x, int y, int z) const
        {
                assert(x < m_size_x);
                assert(y < m_size_y);
                assert(z < m_size_z);

                return m_size_x*m_size_y*z + m_size_x*y + x;
        }

        int pos(TripleInt const& coordinates)  const
        {
                return pos(coordinates.x, coordinates.y, coordinates.z);
        }

        TripleInt coordinates(int position)  const
        {
                assert(0 <= position && position < size());

                int z_div = position / (m_size_x*m_size_y);
                int z_rem = position % (m_size_x*m_size_y);

                int y_div = z_rem / m_size_x;
                int y_rem = z_rem % m_size_x;

                return{ y_rem, y_div, z_div };
```

```
        }
};

using Address = TripleInt;

struct Traffic{
        Address source;
        Address destination;
        float load;
        float initial_load;
        bool  mark;
};

struct Link{
        Address source;
        Address destination;
        float load;
        float initial_load;
        std::vector<Traffic *> traffic;
};

struct Node{
        Address address;
        std::vector<Traffic> buffer;
        std::vector<Node *> neighbours;
        std::vector<Link > links;
};

using Torus = Matrix3D <Node>;

Torus make_torus(int size_x, int size_y, int size_z);

Matrix3D<float> generate_traffic_matrix(int size_x, int size_y, int size_z);

void init_scenario1(Torus& torus, Matrix3D<float> const& initial_traffic);

enum class SubScenario2 { Nearest, Farthest };

void init_scenario2(Torus& torus, Matrix3D<float> const& initial_traffic, SubScenario2 const
& sub_scenario);

enum class SubScenario3 { Nearest, NearAndFar, Farthest };

void init_scenario3(Torus& torus, Matrix3D<float> const& initial_traffic, SubScenario3
sub_scenario);

enum class RoutingAlgorithm { Fixed, Random };

struct Report
{
        float initial_link_load;
        float initial_traffic_load;
        float final_link_load;
        float final_traffic_load;
        float average_link_load;
        float average_traffic_load;
        float highest_link_deviation;
        float highest_traffic_deviation;
        int   over_loaded_links;
};
```

```
Report send_all_traffic(Torus & torus, float max_load, float delta, RoutingAlgorithm
algorithm);
```

Source.cpp

```
#pragma once

#include "Header.h"
#include <iomanip>

int main()
{
	int size_x = 7;
	int size_y = 8;
	int size_z = 9;
	int scenario;
	int sub_scenario = -1;
	int routing_algorithm = -1;
	int repeat_scenario_simulation = -1;
	Report report;
	auto initial_traffic = generate_traffic_matrix(size_x, size_y, size_z);

	bool repeat = false;
	do
	{
		auto torus = make_torus(size_x, size_y, size_z);

		std::cout << "Izaberite jedan od ponudjenih scenarija (1-3): \n";
		std::cin >> scenario;

		switch (scenario)
		{
		case (1) :
			init_scenario1(torus, initial_traffic);
			break;
		case (2) :
			std::cout << "Izaberite jedan od ponudjenih podscenarija (1. Nearest 2.
Farthest): \n";
			std::cin >> sub_scenario;
			assert(sub_scenario == 1 || sub_scenario == 2);
			switch (sub_scenario)
			{
			case (1) :
				init_scenario2(torus, initial_traffic, SubScenario2::Nearest);
				break;
			case (2) :
				init_scenario2(torus, initial_traffic, SubScenario2::Farthest);
				break;
			}
			break;
		case (3) :
			std::cout << "Izaberite jedan od ponudjenih podscenarija (1. Nearest 2.
NearAndFar 3. Farthest): \n";
```

```
                        std::cin >> sub_scenario;
                        assert(sub_scenario == 1 || sub_scenario == 2 || sub_scenario == 3);
                        switch (sub_scenario)
                        {
                        case (1) :
                               init_scenario3(torus, initial_traffic, SubScenario3::Nearest);
                               break;
                        case (2) :
                               init_scenario3(torus, initial_traffic,
SubScenario3::NearAndFar);
                               break;
                        case (3) :
                               init_scenario3(torus, initial_traffic, SubScenario3::Farthest);
                               break;
                        }
                        break;
                default:
                        std::cout << "Neispravan broj scenarija!";
                        exit(1);
                }

                std::cout << "Izaberite jedan od ponudjenih algoritama za rutiranje (1 -
Fiksni, 2 - Random): \n";
                std::cin >> routing_algorithm;
                assert(routing_algorithm == 1 || routing_algorithm == 2);

                switch (routing_algorithm)
                {
                case (1) :
                        report = send_all_traffic(torus, 1.0f, 0.001f,
RoutingAlgorithm::Fixed);
                        break;
                case (2) :
                        report = send_all_traffic(torus, 1.0f, 0.001f,
RoutingAlgorithm::Random);
                        break;
                }

                std::cout
                        << "Initial_load                = " << std::setw(10) <<
report.initial_link_load << "\n"
                        << "Final_load                  = " << std::setw(10) <<
report.final_link_load << "\n"
                        << "Average_link_load           = " << std::setw(10) <<
report.average_link_load << "\n"
                        << "Highest_link_deviation      = " << std::setw(10) <<
report.highest_link_deviation << "\n"
                        << "Initial_traffic             = " << std::setw(10) <<
report.initial_traffic_load << "\n"
                        << "Final_traffic               = " << std::setw(10) <<
report.final_traffic_load << "\n"
                    << "Average_traffic_load        = " << std::setw(10) <<
report.average_traffic_load << "\n"
                        << "Highest_traffic_deviation   = " << std::setw(10) <<
report.highest_traffic_deviation << "\n"
                        << "Number of overloaded links  = " << std::setw(10) <<
report.over_loaded_links << " %" <<"\n";

                std::cout << "Ukoliko zelite da nastavite sa simulacijom unesite 1, u
suprotnom unesite 0:\n";
```

```
            std::cin >> repeat_scenario_simulation;
            assert(repeat_scenario_simulation == 1 || repeat_scenario_simulation == 0);

            switch (repeat_scenario_simulation)
            {
            case (1) :
                  repeat = true;
                  break;
            case (0) :
                  exit(1);
            }
      } while (repeat);

return 0;
}
```

DataStructures.cpp

```
#pragma once

#include "Header.h"
#include <algorithm>
#include <random>

//////////////////////////////////////////////////////////////////////
////////// Helper Functions //////////////////////////////////////////
//////////////////////////////////////////////////////////////////////

static int circular_move(int size, int direction, int start_point)
{
      assert(direction == -1 || direction == 0 || direction == 1);
      assert(0 < size);
      assert(0 <= start_point && start_point < size);

      return (start_point + direction + size) % size;
}

TripleInt circular_move(Torus const& torus, TripleInt const& dir, TripleInt const& origin)
{
      return{
            circular_move(torus.size_x(),dir.x ,origin.x ),
            circular_move(torus.size_y(), dir.y, origin.y),
            circular_move(torus.size_z(), dir.z, origin.z)

      };
}

Matrix3D<float> generate_traffic_matrix(int size_x, int size_y, int size_z)
{
      std::random_device                    rd;
      std::default_random_engine            engine(rd());
      std::uniform_real_distribution<float> distribution(0.1f, 1.0f);

      Matrix3D<float> traffic_matrix(size_x, size_y, size_z);
```

```
        std::generate(
                traffic_matrix.begin(), traffic_matrix.end()
                , [&]{ return distribution(engine); }
        );

        return traffic_matrix;
}

static TripleInt get_random_direction()
{
        static std::random_device                       rd;
        static std::default_random_engine               engine(rd());
        static std::uniform_int_distribution<int>       distribution(0, 5);
        return NEIGHBOR_DIRECTIONS[distribution(engine)];
}

static std::vector<Node*> find_neighbours(Torus& torus, int x, int y, int z)
{
        assert(0 <= x && x < torus.size_x());
        assert(0 <= y && y < torus.size_y());
        assert(0 <= z && z < torus.size_z());

        std::vector<Node*> neighbours;

        for (auto const& direction : NEIGHBOR_DIRECTIONS){
                TripleInt neighbour_position = {
                        circular_move(torus.size_x(), direction.x, x)
                        , circular_move(torus.size_y(), direction.y, y)
                        , circular_move(torus.size_z(), direction.z, z)
                };
                neighbours.push_back(&torus.el(neighbour_position));
        }

        return neighbours;
}

static std::vector<Node*> find_neighbours(Torus& torus, Address const& address)
{
        return find_neighbours(torus, address.x, address.y, address.z);
}

static std::vector <Link> make_links(Node const & node )
{
        std::vector<Link> links(node.neighbours.size() + 1);
        for (unsigned int j = 0; j < node.neighbours.size(); ++j)
        {
                auto & link = links[j];
                link.source = node.address;
                link.destination = node.neighbours[j]->address;
                link.load = 0;
        }
        links.back().source = node.address;
        links.back().destination = node.address;
        links.back().load = 0;

        return links;
}

enum class Direction { Left, Right };
```

```
static int calculate_distance(int const size, Direction const direction, int const a, int
const b)
{
	assert(size > 0);
	assert(0 <= a && a < size);
	assert(0 <= b && b < size);

	int distance = -1;

	switch(direction){
	case Direction::Left:
		distance = (size + a - b) % size;
		break;
	case Direction::Right:
		distance = (size + b - a) % size;
		break;
	}

	assert(distance != -1);

	return distance;
}

static int shortest_path_direction(int const size, int const a, int const b)
{
	int direction_left = calculate_distance(size, Direction::Left, a, b);
	int direction_right = calculate_distance(size, Direction::Right, a, b);
	return (direction_left <= direction_right ? -1 : 1);
}

static Address push_x(Torus const & torus, Address start, Address goal, std::vector<Address>
& path){
	if (start.x == goal.x)
	{
		return start;
	}
	int direction = shortest_path_direction(torus.size_x(), start.x, goal.x);
	start.x = circular_move(torus.size_x(), direction, start.x);
	while ( start.x != goal.x ){
		path.push_back(start);
		start.x = circular_move(torus.size_x(), direction, start.x);
	}
	path.push_back(start);

	return start;
}

static Address push_y(Torus const & torus, Address start, Address goal, std::vector<Address>
& path){
	if (start.y == goal.y)
	{
		return start;
	}
	int direction = shortest_path_direction(torus.size_y(), start.y, goal.y);
	start.y = circular_move(torus.size_y(), direction, start.y);
	while (start.y != goal.y){
		path.push_back(start);
		start.y = circular_move(torus.size_y(), direction, start.y);
	}
```

```
	path.push_back(start);

	return start;
}

static Address push_z(Torus const & torus, Address start, Address goal, std::vector<Address>
& path){
	if (start.z == goal.z)
	{
		return start;
	}
	int direction = shortest_path_direction(torus.size_z(), start.z, goal.z);
	start.z = circular_move(torus.size_z(), direction, start.z);
	while (start.z != goal.z){
		path.push_back(start);
		start.z = circular_move(torus.size_z(), direction, start.z);
	}
	path.push_back(start);

	return start;
}

static std::vector<Address> compute_path(Torus const & torus, Address start, Address goal)
{
	std::vector<Address> path;
	if (start == goal) {
		path.push_back(start);
		path.push_back(start);
	}
	else {
		Address start_address = start;
		path.push_back(start_address);
		start_address = push_x(torus, start_address, goal, path);
		start_address = push_y(torus, start_address, goal, path);
		start_address = push_z(torus, start_address, goal, path);
	}
	return path;
};

static Address push_xyz(Torus const & torus, Address start_address, Address const&
goal_address, std::vector<Address>& path, Address const& direction)
{
	if (direction.x != 0){
		start_address = push_x(torus, start_address, goal_address, path);
	}
	if (direction.y != 0){
		start_address = push_y(torus, start_address, goal_address, path);
	}
	if (direction.z != 0){
		start_address = push_z(torus, start_address, goal_address, path);
	}
	return start_address;
}

static std::vector<Address> compute_random_path(Torus const & torus, Address start, Address
goal)
{
	std::vector<Address> path;
	if (start == goal) {
		path.push_back(start);
```

```
            path.push_back(start);
        }
        else {
            Address start_address = start;
            path.push_back(start_address);

            Address dir1 = get_random_direction();
            start_address = push_xyz(torus, start_address, goal, path, dir1);

            Address dir2 = get_random_direction();
            while (dir1 == dir2 || dir1 == -dir2)
            {
                dir2 = get_random_direction();
            }
            start_address = push_xyz(torus, start_address, goal, path, dir2);

            Address dir3 = get_random_direction();
            while (dir1 == dir3 || dir1 == -dir3 || dir2 == dir3 || dir2 == -dir3)
            {
                dir3 = get_random_direction();
            }
            start_address = push_xyz(torus, start_address, goal, path, dir3);
        }

        return path;
};

static Address get_farthest(Torus const & torus, Address const & address)
{
        Address destination;
        destination.x = (address.x + torus.size_x() / 2) % torus.size_x();
        destination.y = (address.y + torus.size_y() / 2) % torus.size_y();
        destination.z = (address.z + torus.size_z() / 2) % torus.size_z();

        return destination;
}

static TripleInt get_random_nearest(Torus const & torus, Address const& origin)
{
        auto dir = get_random_direction();
        return circular_move(torus, dir, origin);
}

static Address get_far_away_in_direction(Torus const & torus, Address const& origin, Address
const & dir)
{
        Address destination = origin;
        int steps = 0;
        if (dir.x != 0){
            steps = torus.size_x() / 2;
        }
        if (dir.y != 0){
            steps = torus.size_y() / 2;
        }
        if (dir.z != 0){
            steps = torus.size_z() / 2;
        }

        for (int i = 0; i < steps; ++i)
        {
```

```
                destination = circular_move(torus, dir, destination);
        }

        return destination;
};

static bool is_buffer_empty(Node const & node)
{
        return node.buffer.empty();
}

static bool all_buffers_empty(Torus const & torus)
{
        return std::all_of(torus.begin(), torus.end(), is_buffer_empty);
}

static void send_traffic_to_link(Torus& torus, std::vector<Address> const path, Traffic*
traffic)
{
        assert(path.size() > 1);
        int first = 0;
        int second = 1;
        int const size = path.size();
        for (; second != size; ++first, ++second){
                auto& start_node = torus.el(path[first]);
                auto& goal_node = torus.el(path[second]);

                for (std::size_t i = 0; i < start_node.links.size(); ++i){
                        if (start_node.links[i].destination == goal_node.address){
                                start_node.links[i].traffic.push_back(traffic);
                                break;
                        }
                }
        }
}

static void check_and_mark_if_full_link(Link & link, float max_load)
{
        if (link.load <= max_load)
        {
                return;
        }
        for (auto t : link.traffic)
        {
                t->mark = true;
        }
}

static void check_full_links(Torus & torus, float max_load)
{
        for (auto& node : torus)
        {
                for (auto& link : node.links)
                {
                        check_and_mark_if_full_link(link, max_load);
                }
        }
}

static void update_marked_traffics(Torus & torus, float delta)
```

```
{
	for (auto& node : torus)
	{
		for (auto& traffic : node.buffer)
		{
			if (true == traffic.mark && traffic.load > 0)
			{
				traffic.load -= delta;
				if (traffic.load <= 0.0){
					traffic.load = 0.0;
				}
				traffic.mark = false;
				if (traffic.load <= 0)
				{
					traffic.load = 0;
				}
			}
		}
	}
}

static bool check_is_there_full_link(Torus & torus, float max_load)
{
	for (auto& node : torus)
	{
		for (auto& link : node.links)
		{
			if (link.load > max_load)
			{
				return true;
			}
		}
	}
	return false;
}

static void refresh_link_load(Link& link)
{
	link.load = 0;
	for (auto t : link.traffic){
		link.load += t->load;
	}
}

static void refresh_all_links_loads(Torus& torus)
{
	for (auto& node : torus){
		for (auto& link : node.links){
			refresh_link_load(link);
		}
	}
}

static float sum_link_loads(Torus & torus)
{
	float result = 0;
	for (auto const & node : torus)
	{
		for (auto const & link : node.links)
		{
```

```
                        result += link.load;
                }
        }
        return result;
}

static float sum_all_trafics(Torus & torus)
{
        float result = 0;
        for (auto const & node : torus)
        {
                for (auto const & traffic : node.buffer)
                {
                        result += traffic.load;
                }
        }
        return result;
}

void set_initial_link_load(Torus & torus)
{
        for (auto & node : torus)
        {
                for (auto & link : node.links)
                {
                        link.initial_load = link.load;
                }
        }
}

static int number_of_all_traffics(Torus const & torus)
{
        int result = 0;
        for (auto const & node : torus)
        {
                result += node.buffer.size();
        }
        return result;
}

static float get_highest_link_deviation(Torus const & torus)
{
        float max_deviation = 0;
        for (auto const & node : torus)
        {
                for (auto & link : node.links)
                {
                        float deviation = link.initial_load - link.load;
                        if (deviation > max_deviation)
                        {
                                max_deviation = deviation;
                        }
                }
        }
return max_deviation;
}

static float get_highest_traffic_deviation(Torus const &torus)
{
        float max_deviation = 0;
```

```
	for (auto const & node : torus)
	{
		for (auto & traffic : node.buffer)
		{
			float deviation = traffic.initial_load - traffic.load;
			if (deviation > max_deviation)
			{
				max_deviation = deviation;
			}
		}
	}
	return max_deviation;
}


float get_over_loaded_links(Torus const & torus, float max_load)
{
	float number_of_links = 0;
	for (auto const & node : torus)
	{
		for (auto & link : node.links)
		{
			if (link.load > max_load)
			{
				number_of_links += 1;
			}
		}
	}
	return number_of_links;
}

////////////////////////////////////////////////////////////////////////
////////// Implementations /////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////

Torus make_torus(int size_x, int size_y, int size_z)
{
	auto torus = Torus(size_x, size_y, size_z);

	for (int pos = 0; pos < torus.size(); ++pos)
	{
		auto port_address = torus.coordinates(pos);
		auto & node = torus.el(port_address);
		node.address = port_address;
	}

	for (int pos = 0; pos < torus.size(); ++pos)
	{
		auto port_address = torus.coordinates(pos);
		auto & node = torus.el(port_address);
		node.neighbours = find_neighbours(torus, port_address);
		node.links = make_links(node);
	}

	return torus;
}

void init_scenario1(Torus& torus, Matrix3D<float> const& initial_traffic)
{
	assert(torus.size_x() == initial_traffic.size_x());
```

```
	assert(torus.size_y() == initial_traffic.size_y());
	assert(torus.size_z() == initial_traffic.size_z());

	int const nodes_count = torus.size();
	for (int i = 0; i < nodes_count; ++i)
	{
		auto& node = torus.el(torus.coordinates(i));
		float const load_per_destination = initial_traffic.el(node.address) /
nodes_count;
		node.buffer.reserve(nodes_count);
		for (int j = 0; j < nodes_count; ++j)
		{
			auto const & source_port = node;
			auto const & destination_port = torus.el(torus.coordinates(j));
			auto const traffic = Traffic{ source_port.address,
destination_port.address, load_per_destination, load_per_destination, false };
			node.buffer.push_back(traffic);
		}
	}
}

void init_scenario2(Torus& torus, Matrix3D<float> const& initial_traffic, SubScenario2 const
& sub_scenario)
{
	assert(torus.size_x() == initial_traffic.size_x());
	assert(torus.size_y() == initial_traffic.size_y());
	assert(torus.size_z() == initial_traffic.size_z());

	Address dir = get_random_direction();
	int const nodes_count = torus.size();

	for (int i = 0; i < nodes_count; ++i)
	{
		auto& node = torus.el(torus.coordinates(i));
		node.buffer.reserve(nodes_count);
		float const half_load = initial_traffic.el(node.address) / 2;
		float const rest_load = half_load / (nodes_count - 1);
		Address destination_half;
		switch (sub_scenario){
		case SubScenario2::Nearest:
			destination_half = circular_move(torus, dir, node.address);
			break;
		case SubScenario2::Farthest:
			destination_half = get_farthest(torus, node.address);
			break;
		default:
			assert(false && "invalid subscenario value");
		}

		for (int j = 0; j < nodes_count; ++j)
		{
			auto& dest_node = torus.el(torus.coordinates(j));

			auto traffic = Traffic{ node.address, dest_node.address, 0 };
			traffic.initial_load = destination_half == dest_node.address
				? half_load
				: rest_load;
			traffic.load = traffic.initial_load;
			traffic.mark = false;
			node.buffer.push_back(traffic);
```

```
			}
		}
}

void init_scenario3(Torus& torus, Matrix3D<float> const& initial_traffic, SubScenario3
sub_scenario)
{
	assert(torus.size_x() == initial_traffic.size_x());
	assert(torus.size_y() == initial_traffic.size_y());
	assert(torus.size_z() == initial_traffic.size_z());

	Address dir1 = get_random_direction();
	Address dir2 = get_random_direction();
	while (dir1 == dir2 || dir1 == -dir2)
	{
		dir2 = get_random_direction();
	}
	int const nodes_count = torus.size();

	for (int i = 0; i < nodes_count; ++i)
	{
		auto& node = torus.el(torus.coordinates(i));
		node.buffer.reserve(2);
		float const half_load = initial_traffic.el(node.address) / 2;
		Address destination1;
		Address destination2;
		if (SubScenario3::Nearest == sub_scenario)
		{
			destination1 = circular_move(torus, dir1, node.address);
			destination2 = circular_move(torus, dir2, node.address);
		}
		if (SubScenario3::NearAndFar == sub_scenario)
		{
			destination1 = circular_move(torus, dir1, node.address);
			destination2 = get_far_away_in_direction(torus, node.address, dir2);
		}
		if (SubScenario3::Farthest == sub_scenario)
		{
			destination1 = get_far_away_in_direction(torus, node.address, dir1);
			destination2 = get_far_away_in_direction(torus, node.address, dir2);
		}

		Traffic traffic1, traffic2;
		traffic1 = Traffic{ node.address, destination1, half_load, half_load, false };
		node.buffer.push_back(traffic1);
		traffic2 = Traffic{ node.address, destination2, half_load, half_load, false };
		node.buffer.push_back(traffic2);
	}
}

Report send_all_traffic(Torus & torus, float max_load, float delta, RoutingAlgorithm
algorithm)
{
	bool repeat = false;

	switch (algorithm)
	{
	case(RoutingAlgorithm::Fixed) :
		for (int i = 0; i < torus.size(); ++i)
		{
```

```
                        auto& node = torus.el(torus.coordinates(i));
                        for (auto& traffic : node.buffer)
                        {
                                auto path = compute_path(torus, traffic.source,
traffic.destination);
                                send_traffic_to_link(torus, path, &traffic);
                        }
                }
                break;

        case(RoutingAlgorithm::Random) :
                for (int i = 0; i < torus.size(); ++i)
                {
                        auto& node = torus.el(torus.coordinates(i));
                        for (auto& traffic : node.buffer)
                        {
                                auto path = compute_random_path(torus, traffic.source,
traffic.destination);
                                send_traffic_to_link(torus, path, &traffic);
                        }
                }
                break;
        default:
                assert(false && "invalid subscenario value");
        }

        refresh_all_links_loads(torus);

        set_initial_link_load(torus);
        float over_loaded_links = ( get_over_loaded_links(torus, max_load) / (torus.size() *
7) ) * 100;
        float initial_link_load = sum_link_loads(torus);
        float initial_traffic_load = sum_all_trafics(torus);

        do
        {
                check_full_links(torus, max_load);
                update_marked_traffics(torus, delta);
                refresh_all_links_loads(torus);
        } while (check_is_there_full_link(torus, max_load));

        float final_link_load = sum_link_loads(torus);
        float final_traffic_load = sum_all_trafics(torus);
        float average_link_load = final_link_load / (torus.size() * 7);
        float average_traffic_load = final_traffic_load / number_of_all_traffics(torus);
        float highest_link_deviation = get_highest_link_deviation(torus);
    float highest_traffic_deviation = get_highest_traffic_deviation (torus);


        return{ initial_link_load, initial_traffic_load, final_link_load, final_traffic_load,
                average_link_load, average_traffic_load, highest_link_deviation,
highest_traffic_deviation, over_loaded_links };
}
```