SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1890

Modifikacija protokola TCP za podršku kvaliteti usluge

Ivan Varga

Zagreb, listopad 2011

Sadržaj

1. Uvod

Rastom prosječnih brzina prijenosa podataka na Internetu, usluge poput strujanja (streaming) multimedijskog sadržaja, audio i video poziva, dobivaju sve veći broj korisnika. Pojedini servisi za prijenos podataka koriste protokol UDP te u tom slučaju aplikacije moraju biti otporne na gubitak i krivi redoslijed pristiglih paketa. Pošto UDP nema nikakvu kontrolu zakrčenja, njegovo korištenje predstavlja mogući uzrok istog. Protokol TCP je prvotno definirao samo kontrolu toka podataka, odnosno razmjenu informacije o slobodnom prostoru u međuspremnicima. Kao posljedica nedostatka definirane kontrole zakrčenja, 80-ih godina prošlog stoljeća dolazi do kolapsa Interneta uslijed zakrčenja [1][3]. 1986. godine usvajaju se Van Jacobsonova načela koja su implementirana u Tahoe inačici protokola TCP. Iz navedenih razloga, ali i zbog činjenice da dio sigurnosnih stijena (firewall) blokira sve osim TCP prometa, određeni servisi koriste TCP za prijenos podataka. Za razliku od rješenja koji koriste UDP, ovakvi servisi ne predstavljaju rizik od zakrčenja te aplikacije ne moraju brinuti jesu li svi segmenti došli i jesu li u ispravnom poretku. No, zbog neučinkovitog iskorištenja širine propusnog pojasa i varijacije kašnjenja (engl. jitter), kvaliteta takvih usluga bit će niža nego kod servisa koji koriste UDP. Treća opcija su vlasnički (proprietary) aplikacijski protokoli nad TCP ili UDP prijenosnom protokolu, koje koriste aplikacije poput Skpyea [20], RealPlayera, Microsoft Windows Media Servicea . Takvi protokoli ne jamče usklađenost s ostalim tokovima podataka te predstavljaju mogućnost gušenja istih.

TCP nudi uslugu pouzdanog prijenosa, stoga prilikom prijenosa podataka u stvarnom vremenu zbog gubitka ili u posebnim slučajevima većeg kašnjenja segmenta troši se vrijeme na retransmisiju segmenata koji možda više nisu aktualni. U određenim slučajevima gubitak vremena na segment koji kasni može degradirati uslugu više nego da je segment odbačen. Sljedeći problem predstavljaju algoritmi kontrole zakrčenja koji, premda rješavaju problem konkurentnog prijenosa podataka, uzrokuju varijaciju kašnjenja prilikom detekcije zakrčenja. Ako pošiljateljska strana ne primi ništa unutar zadanog vremenskog roka ili primi za redom 3 potvrde istog segmenta dolazi do drastičnog smanjenja dozvoljenog broja nepotvrđenih segmenata (congestion window, skraćeno cwnd), što za posljedicu ima povećanu varijaciju kašnjenja. Situaciju dodatno komplicira činjenica da višestruke potvrde istog segmenta mogu biti posljedica gubitka jednog segmenta uslijed kratkotrajnih problema sa vezom ili usmjeravanja segmenata različitim putevima. Nadalje, različiti modeli TCP propusnosti (engl. bandwidth) pokazuju kako je propusnost obrnuto proporcionalna vremenu povratka (Round Trip Time, RTT) i korijenu vjerojatnosti gubitka paketa [5]:

$$B \propto \frac{1}{\sqrt{p} * RTT} \quad (1)$$

Postoje različiti radovi na temu poboljšanja TCP-a, kako kontrole zakrčenja tako i sustava potvrde segmenata. O njihovoj uspješnosti ne odlučujemo samo na temelju propusnosti i varijacije kašnjenja, već nas zanima kako svaka izmjena utječe na ostali TCP promet, ima li veću propusnost od prometa neizmijenjenog protokola TCP, i utječe li na TCP propusnost te u kojoj mjeri je izmjena kompatibilna sa standardnim protokolom. Svaka modifikacija bi trebala biti TCP prijateljska, pravedna i kompatibilna. Ukoliko predloženi protokol ne utječe na propusnost TCP prometa, odnosno ne izgladnjuje ga, protokol je TCP

prijateljski. Protokol je uz to i pravedan ako u sličnim uvjetima nikada nema veću propusnost od standardnog protokola TCP. Ako je protokol u mogućnosti komunicirati sa neizmijenjenim TCP-om, znači da je modificirana samo jedna strana, ili su modifikacije TCP kompatibilne. Prva dva svojstva, prijateljstvo i pravednost, procjenjuju se na osnovu mjerenja protoka podataka u različitim uvjetima i usporedno sa TCP prometom.

Cilj ovog rada je istražiti mogućnosti proširenja TCP protokola s podrškom za kvalitetu usluge. Osnovne ideje izmjena predloženih u ovome radu su:

1. Predložena kvaliteta usluge ne zahtjeva promjene u mrežnom sloju koji prenosi TCP segmente
2. Sve izmjene moraju biti samo na jednoj strani (pošiljateljskoj ili primateljskoj)
3. Sve izmjene moraju biti kompatibilne sa TCP protokolom
4. Izmjene ne smiju negativno utjecati na konkurentne tokove podataka, odnosno gušiti ih ili uzimati veći dio propusnosti veze.

Rad je strukturiran na sljedeći način; pojam kvalitete usluge i osnovni parametri kvalitete usluge objašnjeni su u drugom poglavlju rada. Elementi TCP protokola koji su potrebni za razumijevanje predloženih proširenja nalaze se u trećem poglavlju. U istome poglavlju nalazi se i pregled povezanih radova te ocjene njihovih uspješnosti. Četvrto poglavlje sadrži detaljnu specifikaciju predloženog rješenja kvalitete usluge za protokol TCP. Mjerenja i analiza rezultata nalaze se u petom poglavlju. Rad završava sa zaključkom i literaturom te dodatcima.

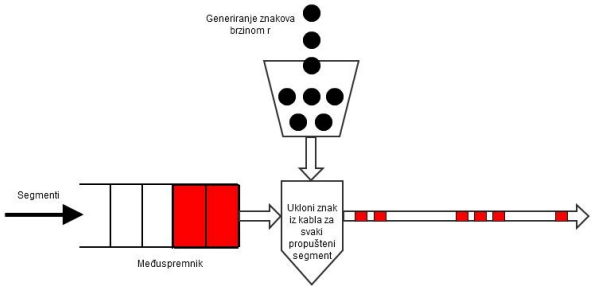
2. Kvaliteta usluge

Internet funkcionira kao mreža najbolje namjere (engl. best-effort), odnosno ne daje nikakve garancije da će poslani paket stići na odredište ili da će stići u zadanom vremenu. No postoji potreba za nekakvom garancijom od mreže, odnosno potreba za određenom kvalitetom usluge (engl. Quality of Service, QoS). Ciljevi kvalitete usluge su kontrola prioriteta propusnosti mreže, kašnjenja i varijacije kašnjenja, pouzdanosti, odnosno gubitka paketa, brzine prijenosa, itd. Pošto ne postoji dogovor između različitih pružatelja internetskih usluga o zajedničkom pružanju takvih usluga, na razini cijelog Interneta nije dostupna kontrola kvalitete usluga. Zbog navedenog razloga nije moguće ostvariti kvalitetu usluge koja bi zahtijevala specijaliziranu mrežnu opremu ili izmjenu postojeće.

Prijenos podataka u stvarnom vremenu posebno je osjetljiv na varijaciju u kašnjenju. Taj problem se uobičajeno rješava upotrebom međuspremnika (engl. buffer). Podaci koji pristižu čuvaju se u međuspremniku do trenutka kada se međuspremnik napuni s određenom količinom podataka. Nakon toga se podaci kontrolirano propuštaju dalje kako bi se smanjila varijacija kašnjenja. Premda ovakvo rješenje smanjuju varijaciju kašnjenja, istovremeno se povećava kašnjenje pošto je potrebno određeno vrijeme kako bi se međuspremnik napunio zadovoljavajućom količinom podataka. U slučaju audio ili video poziva, kašnjenje ne bi smjelo biti veće od 200-250ms kako bi kvaliteta poziva bila zadovoljavajuća, odnosno kako bi se očuvao privid komunikacije u stvarnom vremenu. Varijacija kašnjenja nije isključivo posljedica promjenjivih prilika u mreži, može biti uzrokovana i neprilagođenom kontrolom zakrčenja. Prilikom detekcije zakrčenja TCP drastično smanjuje dozvoljeni broj nepotvrđenih segmenata, što za posljedicu ima povećanje kašnjenja preostalih segmenata. Modificirani TCP protokol koji bi pružao kvalitetu usluge, mogao bi na temelju maksimalnog kašnjenja i maksimalne varijacije kašnjenja na koje se obavezao te stanja u mreži (RTT) odrediti veličinu međuspremnika i maksimalno smanjenje prozora zakrčenja.

Ponekad pouzdanost koju jamči TCP nije toliko bitna, pogotovo u slučaju kada okteti imaju vremenski rok do kada su aktualni. Ako za video poziv koristimo koder video sadržaja koji je otporan na gubitke paketa, nepoželjno je gubiti vrijeme čekajući na pakete koje više ne trebamo. Kvalitetu usluge stoga možemo definirati i za pouzdanost, odnosno stopu gubitaka paketa koja je podnošljiva. Tada segmente koji još nisu stigli, a blokiraju normalan nastavak toka, možemo potvrditi ukoliko je pouzdanost nakon preskakanja segmenata iznad minimalne zadovoljavajuće vrijednosti [1]. Potvrđivanje segmenata koji nisu stigli možemo obavljati svaki puta kada dobijemo segment izvan redoslijeda ili RTT vremena prije isteka maksimalnog kašnjenja sljedećeg očekivanog segmenta.

Audio i video koderima posebno je zanimljiva informacija o dostupnoj brzini prijenosa, to jest kvaliteti sadržaja koji mogu slati kanalom. Kako bi se prilagodili uvjetima na mreži, a ipak zadržali prosječnu brzinu konstantnom, možemo koristiti algoritam kabla sa znakovima (engl. token bucket) (slika 2.1) za određivanje broja segmenata koje šaljemo. Kabao se u konstantnim vremenskim intervalima puni znakovima brzinom r određenom željenom brzinom prijenosa. Prilikom slanja podataka uništava se broj znakova koji odgovara veličini pročitanih podataka. Ukoliko u trenutku čitanja nema znakova u spremniku čitanje nije moguće izvršiti. Dubina kabla određuje usnopljenost prometa, odnosno otpornost na oscilacije brzine.



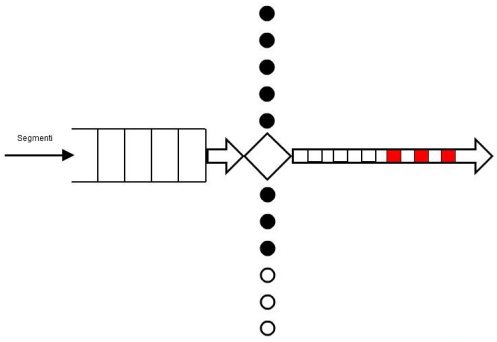
Slika 2.1 Kabao sa znakovima

Algoritam kabla sa znakovima kontrolira srednju brzinu i veličinu snopa. Maksimalna veličina snopa određena je izrazom:

$$C + rS = M \cdot S \quad (2)$$

Pri čemu je C kapacitet kabla, odnosno količina podataka koja stane u kabao, r brzina punjenja znakova, M brzina slanja podataka i S vrijeme koje prođe dok se snop isprazni.

No za provedbu kontrole prometnih tokova (engl. traffic policing) koristi se i algoritam kabla koji curi (engl. leaky bucket),prikazan na slici 2.2, koji kontrolira vršnu brzinu. Kabao koji curi u kablu čuva segmente koji se iz kabla šalju u konstantnim vremenskim intervalima. Ako se kabao napuni a pristižu novi segmenti oni se odbacuju. Moguće je kombinacija tih dvaju algoritama, odnosno dvostruki kabao, kako bi se kontrolirala vršna brzina, srednja brzina i usnopljenost.



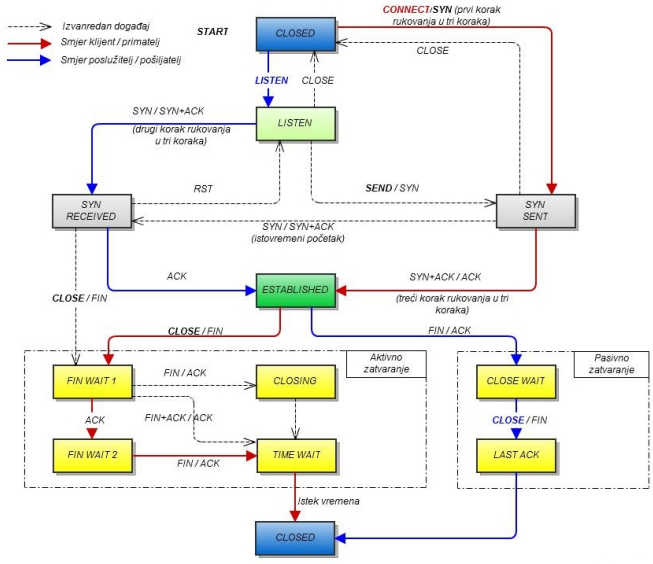
Slika 2.2 Kabao koji curi (Leaky bucket)

3. Protokol TCP

TCP (engl. Transmission Control Protocol) je spojni prijenosni protokol koji jamči pouzdanu isporuku podataka od izvorišta do odredišta sa očuvanim redoslijedom. U TCP su uključeni mehanizmi za detekciju i korekciju pogrešaka. Većina prometa na Internetu odvija se preko protokola TCP. Jedinica podataka protokola TCP naziva se segment , koji se pakira u IP paket i šalje preko mreže.

1969. godine istraživački odjel Ministarstva obrane SAD (DARPA - Defense Advanced Research Projects Agency) pokrenuo je projekt uspostave eksperimentalne mreže s prespajanjem paketa nazvane ARPANET, čija je namjena bila ispitati tehničke mogućnosti razmjene podataka pomoću računalne mreže. 1975. godine ARPANET je od eksperimentalne postala operativna mreža. TCP/IP skup protokola usvojen je kao vojni standard 1983. godine. Korištenje tog standarda bio je preduvjet za spajanje na ARPANET. DARPA je potakla ugradnju porodice protokola TCP/IP u operacijski sustav UNIX i time je stvorena prva veza između operacijskog sustava UNIX i TCP/IP skupa protokola. U vrijeme kad je TCP/IP postao standard počeo se pojavljivati termin Interneta, a mreža ARPANET podijeljena je na dva dijela: MILNET (dio podatkovne mreže Ministarstva obrane SAD) i manji ARPANET.

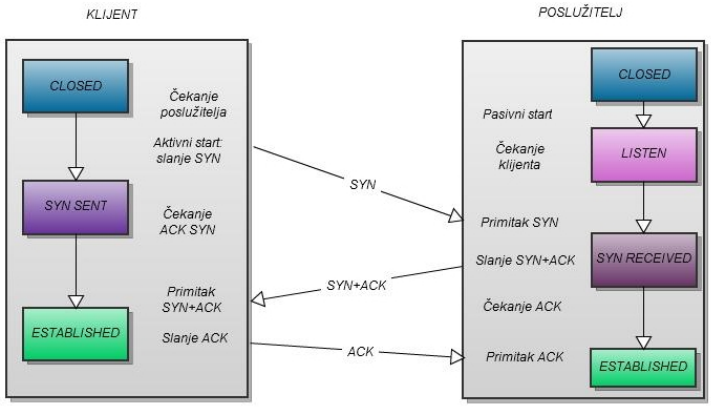
Osnovna svojstva TCP-a su pouzdanost, veza od točke do točke (engl. point-to-point) i dvosmjerni prijenos podataka. Svi se podaci tretiraju kao niz okteta, zato što aplikacije koje koriste TCP u međusobnoj komunikaciji nemaju informaciju o segmentima koji prenose podatke. Dijagram prijelaza stanja protokola TCP nalazi se na slici 3.1:



Slika 3.1 Dijagram prijelaza stanja protokola TCP

Prilikom korištenja ovog protokola entiteti prolaze kroz tri osnovne faze:

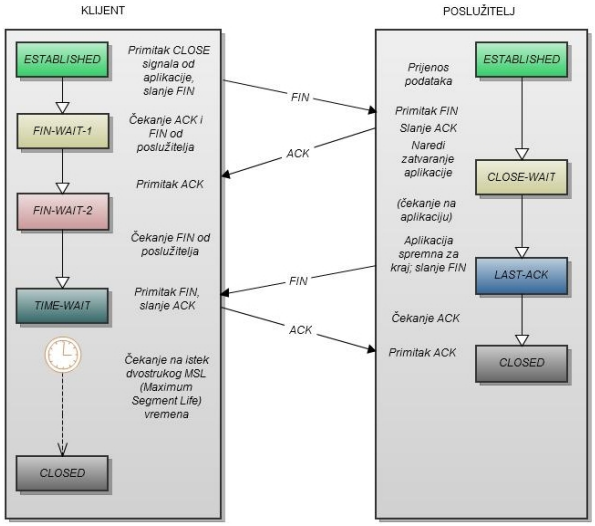
1. Uspostava veze: Proces koji se izvodi na klijentu želi uspostaviti vezu s procesom na poslužitelju. Klijentom, odnosno klijentskim TCP entitetom smatramo onaj TCP koji započinje vezu, dok se poslužiteljskim smatra onaj koji čeka zahtjev za uspostavom veze. Vezu uspostavljaju operacijski sustavi na kojima se nalaze procesi koji žele komunicirati. Povezivanje se vrši razmjenom tri segmenta. Klijentski proces informira klijentski TCP da želi uspostaviti vezu s poslužiteljem. Nakon toga se poslužitelju šalje prvi kontrolni segment. Taj segment ne sadrži podatke aplikacijske razine, već ima postavljen tzv. SYN bit zaglavlja na 1. Zbog toga se taj segment naziva SYN segment. Klijent odabire slučajni broj (client_isn) i stavlja ga u polje za redni broj inicijalnog TCP SYN segmenta. Poslužitelj odgovara segmentom kojim potvrđuje primitak i odobrava uspostavu veze klijentu. Taj segment odobravanja veze također ne sadrži podatke aplikacijske razine, ali sadrži tri važne informacije u zaglavlju segmenta. Prvo, SYN bit je postavljen na 1. Drugo, polje potvrde u zaglavlju TCP segmenta se namješta na client_isn+1. Na kraju, poslužitelj odabire svoj inicijalni redni broj (server_isn) i stavlja vrijednost u polje zaglavlja TCP segmenta. Klijent tada šalje poslužitelju još jedan segment koji potvrđuje da je dobio segment odobravanja veze. To radi tako da stavi vrijednost server_isn+1 u polje potvrde u zaglavlju. SYN bit postavlja se u 0 budući da je veza uspostavljena. Ova procedura se naziva sinkronizacija u tri koraka (engl. three-way handshake). Dijagram toka sinkronizacije nalazi se na slici 3.2. Kada se sva tri koraka obave, klijent i poslužitelj jedan drugome mogu slati segmente koji sadrže podatke. U svakom budućem segmentu SYN će biti postavljen na 0.



Slika 3.2 Sinkronizacija u tri koraka

2. Razmjena podataka: TCP entiteti razmjenjuju podatke u segmentima. Segment se sastoji od zaglavlja koje ima 20 okteta (uz opcionalni dio) za kojim slijedi nula ili više okteta podataka. Podaci koje segment prenosi ne odgovaraju nužno aplikacijskim podatkovnim jedinicama (engl. Application Protocol Data Unit – APDU), već može nositi samo dio APDU-a ili više njih zajedno. Veličina segmenta je varijabilna uz dva ograničenja. Prvo ograničenje je da svaki segment uključujući i TCP zaglavlje mora stati u 65 535 okteta IP paketa, a drugo da čvor fragmentira prevelike segmente (ako je isti veći od MTU mreže) u više manjih segmenata od kojih svaki dobiva svoje IP zaglavlje. Kontrola toka koju koriste TCP entiteti je protokol s kliznim prozorom.

3. Prekid veze: Kad klijentska aplikacija odluči prekinuti vezu s poslužiteljem šalje TCP segment sa FIN bitom postavljenim u 1 i uđe u FIN_WAIT_1 stanje. Dok je u FIN_WAIT_1 stanju, klijentski TCP čeka TCP segment potvrde od strane poslužitelja. Kada primi navedeni segment, klijentski TCP ulazi u FIN_WAIT_2 stanje. Dok je u FIN_WAIT_2 stanju, klijent čeka sljedeći segment od strane poslužitelja s FIN bitom postavljenim u 1. Nakon što primi taj segment klijentski TCP ulazi u TIME_WAIT stanje . Vrijeme provedeno u TIME_WAIT stanju ovisi o implementaciji, ali tipična vrijednost je 120 sekundi. Nakon čekanja veza se formalno zatvori. Tijek prekida veze može se pratiti na slici 3.3.

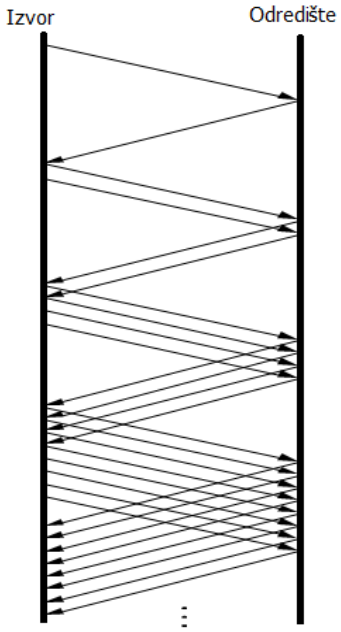


Slika 3.3 Prekid veze

TCP sam brine o zakrčenju mreže. Kako se prijenos segmenata odvija preko nepouzdane mreže, uvode se vremenska ograničenja koja kontroliraju stanje veze te se pomoću njih mogu otkriti segmenti koji nisu stigli na odredište. Vremensko ograničenje potrebno je prilagoditi mreži, odnosno, dinamički ga prilagođavati stanju veze. Ako je premalo, generira se više prometa no što je potrebno, a u slučaju da je vrijednost ograničenja prevelika, odziv na pogreške je prespor.

3.1. Izmjene protokola TCP

Glavni nedostatak korištenja protokola UDP za prijenosa podataka je nedostatak kontrole zakrčenja.. Potreba za mehanizmima kontrole zakrčenja postala je očigledna 80-ih godina prošlog stoljeća kada uslijed zakrčenja dolazi do kolapsa Interneta [11][3]. Kao što je ranije spomenuto, 1986. usvajaju se Van Jacobsonova načela, koja su implementirana u TCP Tahoe verziji. TCP Tahoe započinje s polaganim startom (engl. slow start), tokom koga cwnd raste eksponencijalno (slika 3.4), do unaprijed utvrđene vrijednosti (engl. slow start treshold, SSTRESH). Nakon polaganog starta, slijedi faza izbjegavanja zakrčenja (engl. congestion avoidance) tokom koje se prozor linearno povećava do detekcije zakrčenja, odnosno isteka vremenskog roka za dobivanje sljedeće potvrde. U tom trenutku TCP Tahoe postavlja SSTRESH na pola cwnd-a i ponovno započinje fazu polaganog starta. 1990. godine pojavljuje se poboljšana TCP verzija, TCP Reno, implementirana na operacijskom sustavu 4.3 BSD.



Slika 3.4 Polagani start

U slučaju primitka tri istovjetne potvrde TCP Tahoe i Reno pretpostavljaju gubitak segmenta. Za razliku od Tahoe verzije, TCP Reno se u trenutku detekcije gubitka segmenta ne vraća u fazu polaganog starta, već nakon brze retransmisije započinje fazu brzog oporavka. Faza brzog oporavka nova je faza u TCP Reno verziji, tokom koje se SSTRESH i cwnd postavljaju na pola prethodne cwnd vrijednosti. Cwnd se potom uvećava za svaku potvrdu istog segmenta. U trenutku primanja nove potvrde, Reno nastavlja sa fazom izbjegavanja zakrčenja od SSTRESH vrijednosti. Ako dođe do isteka predviđenog vremena za primitak potvrde, Reno kao i Tahoe odlazi u fazu polaganog starta.

TCP New-Reno manja je modifikacija protokola TCP Reno . Ta modifikacija omogućava

da TCP New-Reno bolje reagira na gubitak više segmenata. Za razliku od Renoa, New-Reno ne izlazi iz faze brzog oporavka dok ne primi potvrde za sve prethodno poslane segmente. Ukoliko dobije novu potvrdu koja ne potvrđuje zadnji poslani segment, New-Reno će pretpostaviti da je izgubljen i sljedeći nepotvrđeni segment te će obaviti retransmisiju tog segmenta. Nakon potvrde zadnjeg segmenta nastavlja sa fazom izbjegavanja zakrčenja. Za detekciju gubitka svakog segmenta potrebno je RTT vremena.

TCP Vegas najpotpunija je TCP verzija, temeljena na TCP Renou. Vegas čuva informaciju o vremenu slanja svakog nepotvrđenog segment. Zahvaljujući toj informaciji i poboljšanom algoritmu procjene RTT-a, već po primitku druge istovjetne potvrde Vegas donosi odluku treba li obaviti retransmisiju. Ukoliko je prošlo više od RTO vremena od slanja prvog nepotvrđenog segmenta obavlja se retransmisija navedenog segmenta. Primitkom prve nove potvrde nakon retransmisije Vegas provjerava koliko je vremena prošlo od sljedećeg nepotvrđenog segmenta. Faza izbjegavanja zakrčenja ne oslanja se na gubitak segmenta kao pokazatelja zakrčenja, već mjeri ostvarenu brzinu prijenosa i uspoređuje tu vrijednost s predviđenom. Ukoliko je izmjerena vrijednost daleko ispod predviđene, povećava cwnd. Naprotiv, ukoliko su te dvije vrijednosti preblizu, Vegas smanjuje cwnd kako ne bi iskoristio kapacitete veze do kraja. Tokom polaganog starta cwnd raste svaki drugi RTT, a u periodima neaktivnosti provjerava brzinu prijenosa te na osnovu te informacije detektira kada treba izaći iz faze polaganog starta.

3.2. Kontrola zakrčenja

Kontrola zakrčenja najosjetljiviji je mehanizam protokola TCP. Izmjene kontrole zakrčenja imaju utjecaj ne samo na vlastiti tok podataka već i na konkurentne tokove. TCP modifikacije spomenute u ovom poglavlju mijenjaju kontrolu zakrčenja. Za ocjenu uspješnosti mjeri se propusnost, pravednost i kompatibilnost modifikacije sa TCP protokolom. Premda neke od modifikacija ne zadovoljavaju uvjete izmjena koji su zadani u uvodu rada, njihovo proučavanje inspiriralo je idejno rješenje za novu modifikaciju.

3.2.1. Holywood

TCP Holywood u osnovi je fino podešen TCP Reno sa ciljem postizanja bolje propusnosti i manje varijacije kašnjenja [3]. Prednost Holywooda je činjenica da zahtjeva izmjenu samo pošiljateljske strane te jednostavnost njegove implementacije. Holywood ima sljedeće izmjene:

1. Tokom faze polaganog starta cwnd se množi sa 14/5 umjesto 2, stoga Holywood brže stiže do SSTRESH vrijednosti.
2. Tokom faze izbjegavanja zakrčenja, cwnd i dalje raste linearno ali 4 puta sporije, odnosno prozor se uvećava za 1 svakih 4RTT.Ova izmjena napravljena je sa ciljem postizanja gotovo konstantne vrijednosti prozora zakrčenja.
3. Nakon 3 istovjetne potvrde, SSTRESH i cwnd se smanjuju na 5/6 prijašnje cwnd vrijednosti umjesto na 1/2 (kao što je slučaj kod TCP Renoa koji je osnova Holywooda)
4. Nakon isteka maksimalnog vremena bez potvrda, odnosno detkcije gubitka segmenta Holywood postavlja SSTRESH na 13/20 cwnd-a, a cwnd na 3.

Jedan od glavnih ciljeva TCP Holywooda je smanjenje varijacije kašnjenja. Rezultati mjerenja obavljenih na mrežnom simulatoru ns-2 [21], pokazali su da za gubitak segmenata od 0-60% Reno pokazuje 38% veću varijancu od Holywooda, dok je ta vrijednost za Vegas čak 265%. Suprotno tome Westwood se pokazao jednako dobrim, ako ne i boljim od Holywooda. Po pitanju propusnosti Holywood je značajno učinkovitiji od Westwooda (30%), Vegasa (17%) i Renoa (53-77%). Iz navedenog vidljivo je da Holywood nije TCP pravedan. No, podatak koji bi bio zanimljiviji, a nije obrađen u mjerenjima, jest u kojoj je mjeri Holywood TCP friendly, odnosno utječe li na usporedni TCP promet.

3.2.2. Westwood

TCP Westwood (TCPW) je modifikacija pošiljateljske strane TCP Reno, koja mu omogućava da za razliku od TCP Reno razlikuje gubitak segmenata kao posljedice zakrčenja i od gubitka koji nije posljedica zakrčenja te primjerenije reagira na njih [12]. Westwood tokom prijenosa podataka prati potvrde segmenata kako bi procijenio trenutnu brzinu prijenosa podataka. U trenutku zakrčenja taj se podatak koristi kako bi odredio SSTRESH i cwnd tokom faze brzog oporavka. Pod pretpostavkom da je zakrčenje posljedica prekomjernog slanja podataka u dužem vremenskom periodu, TCP Westwood kao aproksimaciju koristi nisko propusni filtar:

$$\hat{b_k} = \frac{\frac{2T}{t_k - t_{k-1}} - 1}{\frac{2T}{t_k - t_{k-1}} + 1} \hat{b_{k-1}} + \frac{b_k - b_{k-1}}{\frac{2T}{t_k - t_{k-1}} + 1} \quad (3)$$

te:

$$b_k = \frac{d_k}{t_k - t_{k-1}} \quad (4)$$

$b_k - izmjerena\ brzina\ prijenosa\ podataka\ u\ intervalu \langle t_{k-1}, t_k \rangle$

$d_k - broj\ potvrđenih\ bitova\ primitkom\ potvrde\ segmenta\ u\ trenutku\ k$

$\hat{b_k} - filtrirana\ brzina\ u\ trenutku\ k\ (filtriramo\ frenkvencije\ veće\ od\ 1/T)$

$T - vremenska\ konstanta\ koja\ određuje\ filtrirane\ frekvencije$

Kako bi zadržao svojstva filtra, TCP Westwood za svakih T/m (m >= 2) vremena koje prođe bez novih potvrda računa novu filtriranu brzinu temeljenu na pretpostavci da je dk = 0, odnosno bk = 0. Nadalje, u slučajevima ponovljenih potvrda segmenata, uzima se u obzir da je uspješno prenesen segment, no nije moguće saznati koliko bitova je preneseno. Dodatna komplikacija je odgođeno slanje potvrda koje je prisutno kod standardnih implementacija TCP-a. Stoga je za procjenu brzine prijenosa potrebno pratiti broj istovjetnih potvrda prije primitka nove te odgođeno slanje potvrda.

Faze polaganog starta i faza izbjegavanja zakrčenja su nepromijenjene u odnosu na TCP Reno, razlike između Renoa i Westwooda vidljive su nakon n istovjetnih potvrda i isteka maksimalnog dozvoljenog vremena bez potvrda. U slučaju istovjetnih potvrda Westwood postavlja:

$$SSTRESH = \frac{BWE * RTTmin}{segsize} \quad (5)$$

U izrazu (5) BWE je filtrirana procjena brzine prijenosa, RTTmin najmanja RTT vrijednost tokom veze, a segsize srednja veličina segmenta izražena u bitovima. Ako je cwnd veći od SSTRESH vrijednosti, cwnd se postavlja na istu vrijednost kao i SSTRESH. Sa ovako postavljenim vrijednostima Westwood započinje fazu brzog oporavka.

U slučaju gubitka segmenta nakon isteka maksimalnog vremena čekanja bez potvrde, Westwood postavlja cwnd na vrijednost 1, a SSTRESH na vrijednost dobivenu formulom (4). Dodatno se vrši provjera je li SSTRESH manji od 2 te ako je postavlja se na vrijednost 2. U oba slučaja (faza oporavka i polagani start) Westwood tumači zadnju procijenjenu brzinu prijenosa kao graničnu brzinu konekcije. Stoga u fazi polaganog starta postavlja upravo tu vrijednost kao prag kako bi se što prije vratio na najveću moguću brzinu prijenosa. Iz tog razloga u fazi brzog oporavka nastavljamo sa istom vrijednosti.

Sva mjerenja performansi TCP Westwood obavljena su na ns-2 simulatoru [12]. Usporedna mjerenja su napravljena na TCP Reno i TCP Reno sa selektivnim potvrdama (SACK), oba sa odgođenim potvrdama te UDP-om. Obavljeno mjerenje usporednog prometa UDP-a i TCP Westwooda pokazalo je da je Westwood iskoristio gotovo u potpunosti preostalu propusnost. Uz činjenicu da su daljnja mjerenja pokazala da je TCPW jednako, ako ne i više TCP pravedan (engl. fair) od TCP Renoa, možemo zaključiti da je Westwood vrlo uspješna modifikacija TCP Renoa. Jedina simulacija koja je pokazala srednje uspješne rezultate jest TCP friendliness mjerenje. TCPW je pokazao odstupanje od pravedne podijele širine pojasa za 16% u svoju korist naspram TCP Renoa. Ovaj rezultat autori Westwooda ocijenili su zadovoljavajućim [12].

3.2.3. TCP Prairie

Prairie [4] je vrlo sličan Westwoodu te kao i TCPW koristi procjenu brzine prijenosa za određivanje SSTRESH i cwnd vrijednosti. Za razliku od Westwooda napravljen je na osnovu TCP New-Renoa, a ne Renoa. Prairie uzima u obzir usnopljenost TCP prijenosa, stoga prilikom izračuna brzine prijenosa Prairie potvrđene podatke ne dijeli sa vremenski period od zadnje potvrde, već sve potvrđene podatke dijeli sa vremenom koje proteče između dva snopa potvrda. Osim navedenih razlika Prairie se ponaša poput Westwooda, procijenjene vrijednosti koristi za postavljanje primjerenije SSTRESH i cwnd vrijednosti u trenucima detekcije istovjetnih potvrda ili isteka maksimalnog vremena bez potvrda.

Prairie koristi EWMA (engl. Exponentially Weighted Moving Average) filtar kako bi procijenio brzinu prijenosa. EWMA više važnosti pridaje uzorcima sa većom promjenom vrijednosti, odnosno uzorak koji se značajno razlikuje od do tada procijenjene vrijednosti imat će veći utjecaj na procjenu brzine prijenosa. Stoga Prairie točnije procjenjuje trenutnu brzinu prijenosa. Bolja procjena za posljedicu ima veću učinkovitost (engl. TCP effectiveness) te veću pravednost prema TCP New-Reno u odnosu na Westwood [4].

3.2.4. TFRC sa kontrolom zakrčenja algoritmom kabla sa znakovima

TFRC (engl. TCP friendly rate control) [14] je mehanizam kontrole zakrčenja, koja se u uvjetima zakrčenja ravnopravno natječe za propusnost sa TCP prometom. Treba uzeti u obzir da se pod ravnopravnim smatra omjer tokova do vrijednosti 2. TFRC primatelj prati gubitak paketa i tu informaciju šalje pošiljatelju, koji na temelju toga i RTT računa brzinu prijenosa:

$$X = \frac{s}{R \cdot \sqrt{2 \cdot b \cdot p/3} + (t_{RTO} \cdot (3\sqrt{(3 \cdot b \cdot p/8)} \cdot p \cdot (1 + 32p^2)))} \quad (6)$$

U izrazu (6) X predstavlja brzinu slanja, s veličina paketa u oktetima, a p vjerojatnost gubitka paketa izražen vrijednošću između 0 i 1. Ova brzina osnova je odlučivanja o veličini prozora zakrčenja. U osnovi TFRC i dalje koristi AIMD (engl. Additive Increase – Multiplicative Decrease) algoritam. Korištenje AIMD algoritma može dovesti do kratkotrajnog zakrčenja, a prednosti sporog povećanja prozora ne utječu na kvalitetu veze.

Aplikacija izračunava brzinu prijenosa potrebnu za vezu te ju dojavljuje TFRC-u. Zatražena brzina treba biti manja od one koju je TFRC izračunao. Pošiljatelj nastavlja sa slanjem podataka brzinom zatraženom od strane kodera. Razlika između TFRC izračunate brzine i brzine slanja pohranjuje se u kablu sa znakovima. Znakovi se kasnije mogu iskoristiti u slučaju zakrčenja, kako bi osigurali stalnu brzinu prijenosa.

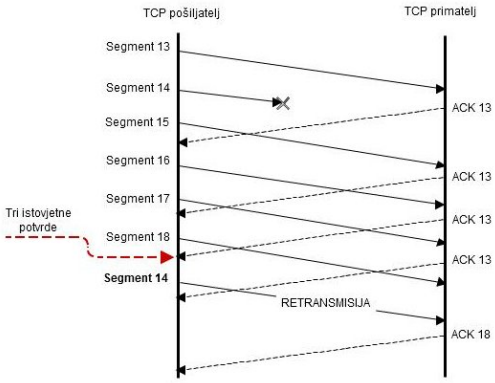
3.2.5. Ostali radovi

Osim navedenih postoje brojni radovi na temu poboljšanja TCP kontrole zakrčenja te je jedna od zanimljivih ideja korištenje primatelja za kontrolu toka podataka. Pod pretpostavkom da primatelj ima bolju informaciju o gubitku segmenata, čini se logičnijim da on kontrolira tok. To je u najjednostavnijem slučaju moguće ostvariti korištenjem odgode slanja potvrde [2] ili mijenjanjem veličine vlastitog prozora kako bi usporili tok u slučaju zakrčenja [5].

3.3. Potvrda segmenata

TCP pruža uslugu pouzdanog prijenosa svih podataka s očuvanim poretkom upotrebljavajući, između ostalog, retransmisiju nepotvrđenih segmenata. Ovo svojstvo je u većini slučajeva poželjno, no ukoliko prenosimo podatke koji imaju određeni "rok trajanja", ne želimo trošiti vrijeme na segmente koji nisu više aktualni.

U slučaju gubitka višestrukih segmenata, TCP nakon tri istovjetne potvrde ima informaciju samo o prvom nepotvrđenom segmentu. Taj slučaj prikazan je na slici 3.5. Prilikom slanja segmenta 14 dolazi do greške i taj segment ne stiže do primatelja. Pošiljatelj nastavlja slati segmente 15, 16, 17 i 18, nakon čega prima treću potvrdu segmenta 13. U tom trenutku detektira gubitak segmenta 14. Svaki izgubljeni segment uzrokuje ulazak u fazu brzog oporavka ili čak odlazak u fazu polaganog starta. Takvo ponašanje uzrokuje varijaciju kašnjenja koja značajno utječe na kvalitetu prijenosa podataka u stvarnom vremenu.



Slika 3.5 Gubitak segmenta

3.3.1. Protokol TCP za upotrebu u ugrađenim uređajima

TCP za upotrebu u ugrađenim uređajima je jednostavna izmjena mehanizma potvrde segmenata, s ciljem poboljšanja svojstava protokola TCP prilikom prijenosa podataka u stvarnom vremenu [10]. Sve izmjene su ograničene isključivo na primateljsku stranu. TCP mjeri vrijeme i periodično pokušava pročitati podatke iz međuspremnika. Ukoliko se predviđeni podaci nalaze u međuspremniku, ne postoji razlika u odnosu na originalni protokol TCP. U trenutku kada primatelj traži podatke u međuspremniku koji nisu primljeni, TCP aplikaciji ne vraća nikakve podatke, preskače preko podataka koji nedostaju, te potvrđuje njihov primitak.

U slučaju da su potvrđeni segmenti koji nikada nisu ni poslani, pošiljatelj implementiran na Embedded Xinu [22] operacijskom sustavu, pretpostaviti će da je došlo do greške te će poslati potvrdu koja umjesto idućeg nepotvrđenog segmenta, nosi informaciju o sljedećem segmentu koji šalje. U tom slučaju primatelj vraća vrijednost potvrđenih okteta u stanje prije slanja lažnih potvrda.

3.3.2. TCP-RTM

TCP-RTM je najopsežnija TCP modifikacija sa ciljem prilagodbe prijenosu podataka u stvarnom vremenu. Izmijenjene su i pošiljateljska i primateljska strana. Sve izmjene su napravljene na TCP Reno implementaciji koja se nalazi u Linux jezgri verzije 2.2.12 sa SACK-om. TCP-RTM osim izmjene prijenosnog protokola, dodatno zahtjeva podršku aplikacijskog sloja. Sve to zajedno čini ga nepraktičnim za širu primjenu, no zanimljiva je analiza mogućih poboljšanja. Na primateljskoj strani napravljene su sljedeće izmjene:

1. TCP-RTM uvodi polje u kojemu se čuvaju segmenti koje još nije moguće potvrditi. Polje nazivamo polje nesljednih segmenata (engl. out-of-order).Ukoliko proces traži nove segmente, a u polju nesljednih segmenata postoje segmenti, prvi kontinuirani niz segmenata se prebacuje iz polja u ulazni međuspremnik iz kojeg aplikacija čita podatke. Pokazivač koji pokazuje na zadnje potvrđeni segment pomiče se iza prebačenog niza.

2. U slučaju da proces pokušava pročitati nove podatke, a polje nesljednih segmenata je prazno, prvi segment koji dođe s indeksom većim od trenutno potvrđenoga, prosljeđuje se procesu i pokazivač zadnje potvrđenog segmenta se uvećava. Na pošiljateljskoj strani napravljene su sljedeće izmjene:

1. Ukoliko se međuspremnik (u kojem pošiljatelj čuva podatke koje dobiva od aplikacije) popunio do kraja, a aplikacija pokušava poslati nove podatke, jednostavno odbacujemo najstariji segment i dodajemo nove podatke.
2. Pošiljatelj od primatelja dobiva informaciju o izgubljenim segmentima. Pošto je primateljska strana u mogućnosti preskočiti nepotvrđene segmente, primatelj mu to dojavljuje SNACK opcijom (engl. Selective Negative ACK).

TCP-RTM uvodi uokvirenje – način rada koji se aktivira prilikom otvaranja spojne točke (engl. socket). U tom načinu rada, svaki podatak koji se šalje putem tako nastale spojne točke smješta se u zaseban TCP segment. Aplikacija radi vlastito uokviravanje podataka prije slanja. Okviri mogu nositi dodatnu informaciju o indeksu okvira, što daje mogućnost aplikaciji da vodi računa o izgubljenim okvirima te da primjereno reagira.

Streaming aplikacije uobičajeno koriste međuspremnik kojim rješavaju problem varijacije kašnjenja. Podaci koji pristižu spremaju se u međuspremnik te čitaju nakon određene odgode. Aplikacije koje koriste TCP-RTM, umjesto aplikacijskog međuspremnika koriste TCP-RTM međuspremnik. Aplikacije počinju s čitanjem podataka iz međuspremnika nakon izrazom određenog vremena:.

$$T = (3/2 * RTT + 3 * period) + delta \quad (7)$$

$T - minimalno\ vrijeme\ čekanja\ nakon\ inicijalizacije\ toka$
$period - vrijeme\ koje\ prođe\ između\ slanja\ dva\ TCP\ segmenta$
$delta - dodatno\ vrijeme$

Pri čemu takvo definiranje vremena odgode omogućava da se, u slučaju gubitka jednog TCP segmenta, taj segment pošalje ponovno prije sljedećeg pokušaja čitanja segmenta . U slučaju gubitka segmenta pošiljatelj treba dobiti tri istovjetne potvrde prije retransmisije, što znači da treba proći RTT uvećano za tri vremenske konstante koje odgovaraju vremenu između slanja TCP segmenata. Nakon toga prođe još 1/2 RTT dok segment ne stigne na primateljsku stranu.

Na razini aplikacije definira se aplikacijski puls (engl. Application-level Heartbeat), odnosno period hbp = 3 * period. U slučaju kada primatelj nema novih segmenata aplikacija svakih hbp vremena šalje poruku, koja može nositi informaciju o izgubljenim paketima, no ne mora nositi nikakvu informaciju osim potvrde. Ta poruka osigurava da u slučaju gubitka potvrde pošiljatelj ne čeka istek maksimalnog vremena bez potvrde kako bi započeo fazu polaganog starta.

Simulacija obavljena na Nisnet simulatoru pokazala je da je korištenjem samo TCP-RTM međuspremnika , na vezi koja ima gubitke do 4%, moguće suzbiti varijaciju kašnjenja. Prilikom većih gubitaka, brza retransmisija više ne rješava problem gubitka te dolazi do većih varijacija kašnjenja.

TCP-RTM, implementiran u Linux jezgri sa SACK-om, korišten je za evaluaciju RTM

proširenja. Testovi su obavljani na Internetu, ispitujući kvalitetu audio poziva u stvarnim uvjetima. Rezultati su pokazali da za 5% ili više gubitaka paketa metoda aplikacijskog pulsa značajno smanjuje varijaciju kašnjena, no primateljska modifikacija dodatno poboljšava rezultate za 50%. Gubitak aplikacijskih okvira raste proporcionalno gubitku TCP paketa.

Primateljska modifikacija omogućava RTM-u da smanji varijaciju kašnjenja i odbaci neaktualne segmente, no istovremeno utječe na kontrolu zakrčenja pošto pošiljatelj dobiva potvrde za segmente koji nikada nisu primljeni. Simulacija koja izvedena na Nisnet simulatoru pokazao je kako razlika u veličini prozora zakrčenja raste zajedno sa smanjivanjem propusnosti, u korist RTM-a. To znači da TCP-RTM ne možemo smatrati pravednim u odnosu na standardni TCP.

3.3.3. PRTP-ECN

Temeljna ideja PRTP-ECN [1] je žrtvovanje pouzdanosti u korist smanjene varijacije kašnjenja i bolje propusnosti. Ovo je vrlo jednostavna i mala TCP nadogradnja, sa izmjenama isključivo na primateljskoj strani. Primatelj odlučuje ima li potrebe za retransmisijom segmenata koji nisu dostavljeni. Odluku o tome donosi na temelju mjere pouzdanosti koja je u osnovi omjer eksponencijalno skalirane sume svih pristiglih segmenata i onih koji su trebali stići.

(8)

$af^k - težinski\ koeficjent$
$p_k - zastavica\ koja\ ukazuje\ je\ li\ segment\ primljen, ima\ vrijednosti\ 0\ ili\ 1$
$b_k - broj\ bitova\ sadržanih\ u\ segmentu$

U trenutku kada PRTP-ECN prima segment izvan poretka, donosi odluku hoće li ga potvrditi, odnosno hoće li preskočiti segmente koji još nisu pristigli. Odluku donosi na temelju mjere pouzdanosti izračunate izrazom (8). Pouzdanost se računa za slučaj da potvrdi segment i uspoređuje se s minimalno dozvoljenom pouzdanošću. Mogući rizik kod ovakvog pristupa jest da program ne primijeti zakrčenje mreže. U slučaju da dođe do zakrčenja, a primatelj potvrdi segmente koji su izgubljeni, pošiljatelj nema informaciju o zakrčenju.

4. Modifikacija protokola TCP za podršku kvaliteti usluge

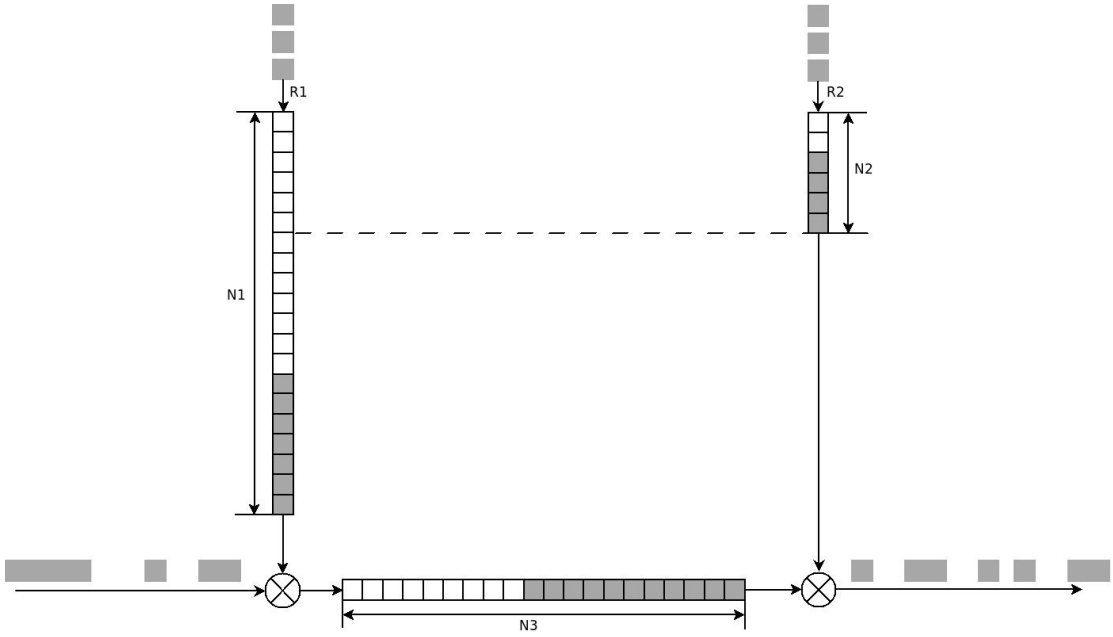
U ovom radu predlaže se modifikacija koja predviđa izmjene samo na primateljskoj strani. Temeljna ideja proširenja je dodati TCP protokolu podršku za kvalitetu usluge koja bi jamčila aplikaciji maksimalnu varijaciju kašnjenja, maksimalno kašnjenje segmenata, minimalnu pouzdanost prijenosa i minimalnu propusnost. Minimalna zahtijevana pouzdanost prijenosa podataka određena je korištenim koderom i njegovom otpornošću na gubitak segmenata. Maksimalna varijacija kašnjenja i kašnjenje ovise o željenoj kvaliteti usluge koju aplikacija pruža. Na primjer, u slučaju video poziva poželjno je da kašnjenje ne bude veće od 150ms i varijacija kašnjenja manja od 30ms. Ukoliko koristimo sporije kodere zahtjevi postaju još stroži. Primatelj preuzima kontrolu oblikovanja prometa pomoću TCP prozora, odnosno maksimalnog broja okteta koje je spreman prihvatiti.

Aplikacije koje koriste TCP za prijenos podataka u stvarnom vremenu koriste jedan od tri pristupa određivanju brzine prijenosa:

1. Koriste uvijek istu brzinu prijenosa, bez obzira na stanje veze. Ovo je najjednostavniji pristup koji funkcionira u slučajevima kada je propusnost dovoljno velika
2. Prije početka prijenosa slanjem ispitnih podataka određuju dostupnu propusnost na temelju kojeg određuju brzinu prijenosa. Ovaj pristup oduzima određeno vrijeme prije samog prijenosa podataka, a i stanje veze se može promijeniti tokom prijenosa
3. Brzina prijenosa podataka se mijenja tokom prijenosa ovisno o stanju veze.

Aplikacija stoga u trenutku definiranja spojne točke raspolaže barem informacijom o početnoj brzini prijenosa podataka. Ostali parametri kvalitete usluge također su poznati u trenutku definiranja spojne točke. Stoga preko spojne točke proširenom TCP protokolu dojavljujemo željenu kvalitetu usluge. Ako se ugovorena kvaliteta usluge ne može ispoštovati veza se prekida i dojavljuje se greška.

Dvostruki kabao sa znakovima je proširena verzija prethodnog algoritma. U proširenoj verziji koriste se dva kabla. Kablovi su povezani međuspremnikom na čijem se ulazu, odnosno izlazu, nalaze. Na slici 4.1 prikazana je shema algoritma dvostrukog kabla. R1 i R2 predstavljaju brzine punjenja prvog i drugog kabla. Kablovi su dubine N1 i N2, a N3 je kapacitet međuspremnika.



Slika 4.1 Dvostruki kabao sa znakovima

Uvjet algoritma je :

$$R_2 \geq R_1 \tag{9}$$

U slučaju da uvjet 9 nije ispoštovan, količina podataka u međuspremniku bi konstantno rasla. Jedan kabao kontrolira punjenje međuspremnika, a drugi pražnjenje. Dubinama kabla reguliraju se maksimalni snopovi punjenja, odnosno pražnjenja.

4.1. Maksimalno kašnjenje i varijacija kašnjenja

Algoritam dvostrukog kabla sa znakovima koristimo na primateljskoj strani proširenog TCP protokola kako bi oblikovali tok podataka. Oba kabla se pune znakovima onom brzinom kojom bi htjeli primati podatke, odnosno prosljeđivati ih aplikaciji. Međuspremnik se puni segmentima neovisno o tome stižu li po redu, dok iz međuspremnika propuštamo samo posložene segmente. Ovako dajemo vremena TCP protokolu da pošalje segmente koji nedostaju te ih posložimo u međuspremniku prije no što ih proslijedimo aplikaciji.

Oba kabla punimo istom brzinom no njihove se dubine razlikuju zbog funkcije koje vrše. Pretpostavimo da u idealnom slučaju podaci veličine jednog okteta stižu u segmentima u pravilnim vremenskim razmacima. Tada se znakovi u prvom kablu generiraju istovremeno sa primitkom svakog segmenta. U takvim uvjetima će popunjenost kabla biti konstantna vrijednost određena vremenom koje segmenti potroše na putovanje. No ukoliko segmenti kasne znakovi se počinju gomilati u kablu. Popunjenost kabla tada predstavlja minimalno kašnjenje sljedećeg okteta. Zato dubinu prvog kabla računamo sljedećim izrazom:

$$N_1 = R * d \quad (10)$$

$d - maksimalno\, dozvoljeno\, kašnjenje\, paketa$
$R - ugovorena\, brzina\, prijenosa\, podataka$

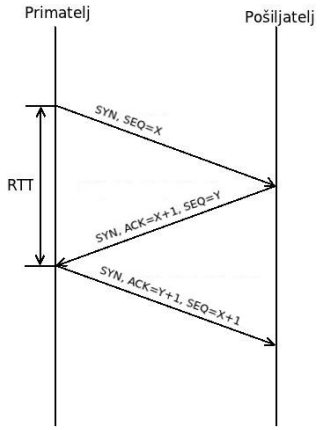
U uvjetima kada je međuspremnik uvijek pun podacima drugi kabao ga prazni brzinom generiranja znakova R2. U slučaju da se međuspremnik isprazni ili da nedostaje sljedeći segment u nizu, drugi kabao se počinje puniti znakovima. Ako ne dođe do prelijevanja znakova, maksimalna varijacija u kašnjenju podataka odgovarati će dubini drugog kabla. Stoga dubinu drugog kabla računamo izrazom:

$$N_2 = J * R \quad (11)$$

$J - maksimalna\, dozvoljena\, varijacija\, kašnjenje\, paketa$
$R - ugovorena\, brzina\, prijenosa\, podataka$

Na primjer, ugovorena je brzina prijenosa R=1.6kb/s=200B/s, maksimalno kašnjenje D=100ms i maksimalna varijacija kašnjenja J=30ms. Tada brzina generiranja znakova iznosi 200 znakova u sekundi. Dubina prvog kabla je N1=20 znakova, a drugog N2=6 znakova (Slika 4.1)

Prvi generator znakova pokrećemo u trenutku slanja prvog segmenta na strani pošiljatelja. Na primateljskoj strani ne možemo znati kada je segment poslan no možemo pretpostaviti na osnovu vremena primitka segmenta i izmjerenom vremenu povrata tokom sinkronizacije u tri koraka (slika 4.2). Drugi generator pokrećemo T1=D-J vremena nakon prvog. Vrijeme T1 koristimo za punjenje međuspremnika kojim pokušavamo smanjiti varijacije kašnjenja na dozvoljenu. U slučaju da se prvi kabao napunio znakovima do samoga vrha znamo da brzina prijenosa podataka nije dovoljna i prekidamo vezu. No ukoliko je drugi kabao pun, a u međuspremniku postoje podaci koji nisu u slijedu moguće je podatke podatke proslijediti aplikaciji ukoliko to ugovorena pouzdanost dozvoljava.



Slika 4.2 Procjena RTT-a tokom sinkronizacije u tri koraka

Kako su sve izmjene na primateljskoj strani, kontrolu zakrčenja ostvarujemo postavljanjem veličine kliznog prozora, koju prijavljujemo pošiljatelju svakom potvrdom segmenta. Veličinu prozora postavljamo tako da odgovara manjoj vrijednosti između broja znakova u prvom kablu i preostalog mjesta u međuspremniku. Ovom tehnikom ne utječemo negativno na zakrčenje jer pošiljatelj ionako uzima u obzir manju od vrijednosti veličine prozora i cwnd-a.

Nakon što pošiljatelj uđe u fazu izbjegavanja zakrčenja, broj okteta koji se šalje u jednom snopu konvergira stalnoj vrijednosti, odnosno prosječna brzina prijenosa podataka teži ugovorenoj. Ovako implementirana kontrola zakrčenja smanjuje varijaciju kašnjenja i prilagođena je prijenosu podataka u stvarnom vremenu. U slučaju zakrčenja pošiljatelj će smanjiti cwnd i zanemariti veličinu prozora koju mu primatelj šalje.

Informacija koju šaljemo pošiljatelju je broj okteta koji želimo da pošalje u određenom trenutku. No segmentu koji šaljemo treba neko vrijeme kako bi došao do pošiljatelja. Zato u trenutku čitanja segmenta ta informacija više nije važeća. Kako bi se riješio taj problem, prilikom slanja segmenta na osnovu procijenjene RTT vrijednosti, dodatno se uvećava prozor za odgovarajući broj okteta. U slučaju da je RTT procijenjen na 80ms, a ugovorena brzina je 20kB/s, tada se broju znakova iz kabla dodaje 800 znakova koji će se generirati u vremenu dok segment stigne do pošiljatelja.

Ako primateljska aplikacija obrađuje podatke u kratkim segmentima, moguća je situacija u kojoj primatelj u potvrdi šalje malu vrijednost kliznog prozora pošiljatelju. Bez ikakve optimizacije pošiljatelj bi tada slao segmente od samo nekoliko okteta podataka. TCP/IP zaglavlja takvih paketa značajno su veća od korisnih podataka koje prenose. To nazivamo "Sindromom besmislenih prozora", i obično se rješava primjenom Nagelovog algoritma na pošiljateljskoj strani ili na primateljskoj strani rješenjem koje je predložio David D. Clark. Nagelov algoritam [19] se može ilustrirati pseudokodom:

```
ako (postoje podatci koje treba poslati) {
          ako (prozor >= MSS i podaci koje šaljemo >= MSS) {
                    pošalji segment MSS veličine
          } inače {
                    ako (postoje nepotvrđeni podatci u redu) {
                              stavi podatke u red na čekanje nove potvrde
                    } inače {
                              pošalji podatke odmah
                    }
          }
}
```

Algoritam 4.1 Nagelov Algoritam

Prilikom određivanja vrijednosti veličine primateljskog prozora mora se pretpostaviti da se na pošiljateljskoj strani primjenjuje Nagelov algoritam. Stoga prozor određujemo na temelju izraza:

$$\bar{R} = round(T/MSS) * MSS$$
$$R = max(MSS, \bar{R}) \quad (12)$$

$MSS - maksimalna\ veličina\ segmenta$
$T - broj\ znakova\ u\ kablu$
$R - veličina\ prozora$

Ukoliko je ugovorena brzina prijenosa manja od MSS/RTT prvi kabao sa znakovima će na temelju izraza (12) propuštati MSS okteta po RTT, no drugi kabao će prazniti međuspremnik ugovorenom brzinom. Nakon dovoljno vremena preostali broj dostupnih okteta u međuspremniku će se smanjiti ispod maksimalne veličine segmenta. U tom trenutku primatelj počinje slati stvarni broj preostalih okteta u međuspremniku umjesto broja znakova u prvom kablu.

4.2. Potvrda segmenata i pouzdanost

Ako je aplikacija sposobna detektirati gubitak segmenata i oporaviti se od istog, onda je moguće definirati stupanj pouzdanosti koji zadovoljava potrebe aplikacije. Jedan od mogućih načina definiranja pouzdanosti je dozvoljeni postotak gubitka podataka. No takav način mjerenja pouzdanosti, u slučajevima video i audio prijenosa u stvarnom vremenu, ne odgovara kvaliteti pružene usluge.

Stoga je primjerenije koristiti mjeru pouzdanosti koja eksponencijalno s vremenom skalira utjecaj gubitka podatka. Može se koristiti izraz za izračunavanje pouzdanosti, korištena u protokolu PRTP-ECN. Ako međuspremnik sadrži segmente u trenutku prelijevanja znakova u drugom kablu, računa se koliko bi potvrđivanje sljedećeg očekivanog segmenta smanjilo pouzdanost. U slučaju da je ta vrijednost iznad ugovorene možemo potvrditi segment koji nije primljen.

Izraz u ovom obliku nije praktičan za korištenje u izmijenjenom protokolu TCP jer zahtjeva praćenje koliko okteta je sadržavao koji segment. Nešto jednostavniji izraz sa sličnim svojstvima bio bi:

$$cr_n = af^{d_n} \cdot (cr_{n-1} + d_n)$$
$$ca_n = af^{d_n} \cdot (ca_{n-1} + d_n) + b_n \quad (13)$$
$$Crl_n = \frac{cr_n}{ca_n}$$

$cr_n - eksponencijalno\ vrednovana\ suma\ svih\ primljenih\ okteta$
$ca_n - eksponencijalno\ vrednovana\ suma\ svih\ potvrđenih\ okteta$
$af - konstanta$
$d_n - broj\ primljenih\ okteta\ od\ zadnjeg\ izračunavanja\ pouzdanosti$
$b_n - broj\ okteta\ koje\ preskačemo$
$Crl_n - pouzdanost$

U trenutku kada se mora preskočiti određen broj okteta dostupni su svi podaci osim dozvoljenog broja okteta koji se preskaču. Ovakav izraz omogućava da se pouzdanost

računa samo u trenucima kada se preskaču okteti te da se iz nje jednostavno izračuna koji broj okteta se može preskočiti:

$$b_n = \frac{cr_n}{Crl_n} - af^{d_n} \cdot (ca_{n-1} + d_n) \tag{14}$$

4.3. Dozvoljene vrijednosti postavki kvalitete usluge

	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.	16.
Brzina prijenosa	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
Varijacija kašnjenja	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
Pouzdanost	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
Maksimalno kašnjenje	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0

Konfiguracije 9-15, nisu valjane budući da kašnjenje ili varijacija kašnjenja nemaju nikakvog smisla bez brzine prijenosa podataka, koja nam je potrebna kako bi smo znali koliko segmenti kasne. Pouzdanost je kao kvalitetu usluge moguće definirati u kombinaciji sa nekim drugim uvjetom kvalitete. U slučaju nemogućnosti ispunjenja navedenoga uvjeta može se narušiti pouzdanost veze kako bi održali ostale ugovorene vrijednosti kvalitete usluge. U slučaju bilo koje od nedozvoljenih kombinacija vrijednosti kvalitete usluge jednostavno se zanemaruje konfiguracija i TCP protokol funkcionira bez ikakvih modifikacija.

Ukoliko nije definirano maksimalno dozvoljeno kašnjenja (konfiguracije 2 i 4), ono se može postaviti na proizvoljnu vrijednosnu. Zato takvim slučajevima maksimalno kašnjenje funkcijski određujemo na osnovu izraza:

$$D = 3/2 \cdot RTT' + delta \tag{15}$$

$D - maksimalno\,kašnjenje$
$RTT' - Vrijeme\,do\,povratka\,povrde\,izmjereno\,tokom\,rukovanja$
$delta - dodatno\,vrijeme$

Ovako definiramo maksimalno kašnjenje kako bi TCP protokolu dali vremena da u slučaju gubitka segmenta obavi retransmisiju. U trenutku definiranja kvalitete usluge ne raspolažemo informacijom o RTT-u, zato koristimo vrijeme izmjereno tokom TCP rukovanja u tri koraka.

Ako nije definirana maksimalna vrijednost varijacije kašnjenja (konfiguracije 5 i 7), ona se

određuje na osnovu maksimalnog kašnjenja. U takvim slučajevima maksimalna dozvoljena varijacija kašnjenja jednaka je upravo maksimalnom kašnjenju. Na taj način ne moramo mijenjati logiku oporavka u slučaju nemogućnosti ispunjenja ugovorene kvalitete usluge.

4.4. Oporavak u slučaju isteka maksimalnog vremena kašnjenja

Zbog blokiranja čitanja podataka iz međuspremnika, za vrijeme punjenja međuspremnika i u slučajevima kada aplikacija prebrzo prazni međuspremnik, treba voditi brigu o tome da aplikacija zna kada može čitati nove podatke. Kako se provjera dostupnosti podataka ne bi morala izvršavati periodično, dovoljno je tijekom algoritma oporavka od isteka maksimalnog kašnjenja provjeriti postoje li dostupni podatci. U slučaju da postoje, obavještava se aplikacija, koja u normalnim okolnostima čita dostupne podatke te se izračunava novo vrijeme isteka maksimalnog dozvoljenog kašnjenja i završava oporavak.

Ako u međuspremniku ne postoje dostupni podatci, možemo pretpostaviti kako je do isteka vremena došlo uslijed greške u prijenosu ili loše kvalitete veze. U tom slučaju izračunavamo broj okteta koji smijemo preskočiti da ne narušimo pouzdanost veze ispod ugovorene. Za broj okteta jednak 0 prekidamo vezu jer ugovorene vrijednosti kvalitete veze nije moguće ispoštovati.

Ako je moguće preskočiti određenu količinu podataka istovremeno sa preskakanjem podataka potrebno je maknuti odgovarajući broj znakova u oba kabla. Znakove uklanjamo zato što preskakanjem simuliramo primitak i čitanje podataka od strane aplikacije. U rijetkim slučajevima kada u međuspremniku postoje podatci koje aplikacija može čitati, ali to ne radi, čitanje podataka se simulira. U tom slučaju brišu se samo znakovi iz drugog kabla. Algoritam oporavka od greške možemo opisati pseudokodom:

```
ako (postoje podaci u međuspremniku) {
          obavijesti aplikaciju
          ako (aplikacija pročitala podatke iz međuspremnika){
                    izađi iz petlje
          }
}
osvježi pouzdanost veze
izračunaj maksimalni broj okteta koje je moguće preskočiti
pokušaj preskočiti pakete
ako(preskakanje okteta uspjelo){
          izračunaj novu pouzdanost
          ako(moguće potvrditi nove podatke) {
                    pošalji novu potvrdu
          }
} inače {
          prekini vezu
}
```

Algoritam 4.2 Oporavk u slučaju preljeva znakova

Na početku algoritma 4.2 vrši se provjera postoje li podaci u međuspremniku koje aplikacija može čitati. Ukoliko postoje dostupni podaci obavještava se aplikacija. Ovu

provjeru je potrebno vršiti pošto kontrolu varijacije kašnjenja provodimo blokiranjem čitanja iz međuspremnika. Ako je aplikacija pročitala nešto nastavljamo dalje sa izvođenjem programa. U suprotnom slučaju izračunavamo maksimalan broj okteta koji možemo preskočiti i pokušavamo ih preskočiti. Ako to nije moguće prekidamo vezu.

5. Simulacija

5.1. NS-3

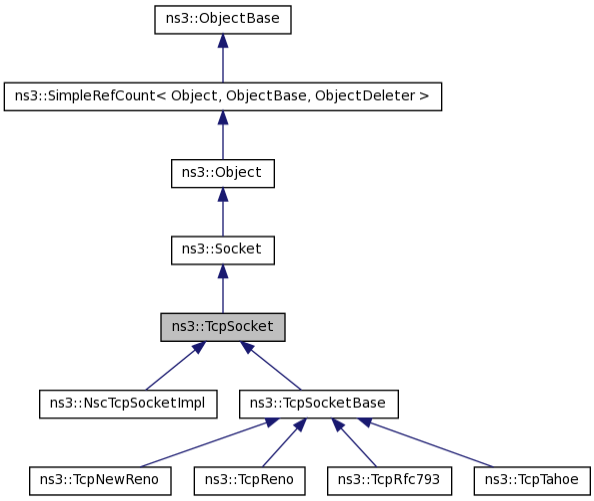
NS-3 (engl. N*etwork simulator 3*) je simulacijsko okruženje pomoću kojeg se ispituje ponašanje mreže i mrežne okoline izvođenjem različitih scenarija. Kratica *ns* odnosi se poglavito na mrežne simulatore diskretnih događaja, i to ns-2 i ns-3. Oni nude podršku za mrežne protokole te se koriste kao razvojni alat za simulaciju protokola usmjeravanja u ožičenim i bežičnim mrežma.

Razvoj ns-a započeo je 1989. kao dio rada na mrežnom simulatoru REAL, pisanog u programskom jeziku C. Ns-2 napisan je u u programskom jeziku C++ i objektnoj verziji skriptnog jezika Tcl (Otcl) pomoću kojeg implementira sučelje simulacije. Korisnik opisuje mrežnu topologiju pomoću Otcl skripti koju potom program mrežnog simulatora ns-2 simulira pomoću postavljenih parametara. Mrežni simulator ns-3 izgrađen je u programskim jezicima C++ i Phyton te podržava izvođenje skripti u oba jezika. Premda je logički nasljednik mrežnog simulatora ns-2, ne predstavlja njegovo proširenje jer je napisan iznova korištenjem drugih programskih jezika. Njegov razvoj započeo je 2006, a prvo izdanje izašlo je dvije godine kasnije. Predviđa se da će ns-3 nakon dodatnih proširenja (2011. izašlo je 11. izdanje ns-3) potpuno pokriti svu funkcionalnost svog prethodnika te ga posve zamijeniti [21].

Prioriteti prilikom razvoja mrežnog simulatora ns-3 bili su fleksibilno korisničko sučelje (nema više potrebe za učenjem skriptnih jezika), modularni pristup arhitekturi, skalabilnost i recikliranje koda. Korisnik opisuje scenarij simulacije pišući C++ ili Phyton program. Korisnikov program instancira skup simulacijskih modela koji potom pokreću opisan scenarij simulacije. U privitku se nalaze programski odsječci ErrorMeasurements.cc i ConcurrentMesurements.cc koji definiraju scenarij simulacija. U ns-3 koriste se povratni pozivi za događaje (engl. Callback-driven events). Svaki događaj u scenariju je predstavljen pozivom funkcije u točno određenom trenutku (zadanom u scenariju). Kako bi simulacije bile vjerne stvarnim slučajevima, prilikom razvoja mrežnog simulatora ns-3 naglasak je stavljen na na mogućnost interakcije simulacije i stvarne fizičke mreže ili komunikaciju mrežnih simulacija kroz stvarnu mrežu. Tu funkcionalnost pruža NSC (engl. Network Simulation Cradle) API, dio ns-3 projekta koji omogućava korištenje stvarnog TCP/IP sloja u simulacijskim scenarijima. Objekti koji predstavljaju pakete u simuliranoj mreži mogu se pohraniti u međuspremnicima kao nizovi okteta koji se potom mogu serijalizirati i poslati na mrežno sučelje prema stvarnoj mreži. Osnovna sučelja čvorova u mrežnom simulatoru ns-3 napravljeni su po uzoru na mrežnu arhitekturu Linuxa.

5.1.1. Arhitektura NS-3 TCP modela

Najvažnije klase NS-3 TCP modela su *ns3::Socket*, *ns3::TcpSocket*, *ns3::TcpSocketBase* te klase kontrole zakrčenja *ns3::TcpNewReno*, *ns3::TcpReno*, *ns3::TcpTahoe* i *ns3::TcpRfc793*. Njihov hijerarhijski dijagram klasa prikazan je na slici 5.1. Klasa *ns3::NscTcpSocketImpl* nasljeđuje *ns3::TcpSocket* i implementira poveznicu TCP modela i NSC-a.

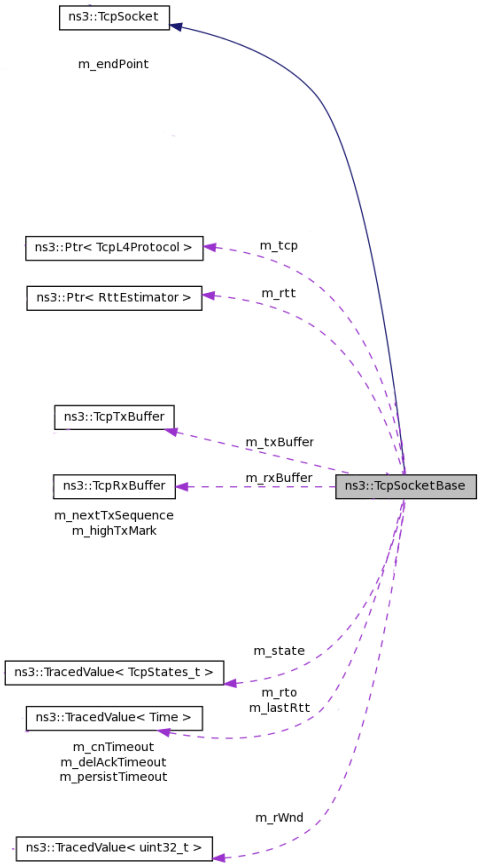


Slika 5.1 Nasljeđivanje i najvažnije ns3 klase

ns3::Socket sadrži funkcionalnost koja približno odgovara BSD Socket API-ju. Za razliku od njega, ne sadrži blokirajuće pozive, već se sve operacije izvršavaju asinkrono. Umjesto običnog pokazivača na podatke *ns3::Socket* koristi klasu *ns3::Packet* za enkapsulaciju paketa u spremnicima okteta. *ns3::TcpSocket* bazna je klasa za sve implementacije protokola TCP. Ova klasa sadrži zajedničke atribute koje koriste izvedene klase.

ns3::TcpSocketBase implementira sučelje prema višim slojevima i pruža osnovu za sve mehanizme kontrole zakrčenja. *ns3::Reno*, *ns3::Tahoe*, *ns3::Rfc793*, *ns3::NewReno* sadrže funkcionalnost kontrole zakrčenja.

Najvažnija klasa u predloženoj izmjeni protokla TCP je *ns3::TcpRxBuffer* slika 5.2. *TcpRxBuffer* je međuspremnik koji čuva poredane oktete koji su stigli. Brojem okteta koju međuspremnik može maksimalno primiti kontrolira se klizni prozor čiju vrijednost TcpSocketBase postavlja u zaglavlju segmenta prilikom slanja. Istovremeno se blokira čitanje podataka od strane aplikacije dok se ne završi punjenje međuspremnika.



Slika 5.2 Dijagram ns3 klasa

Prilikom izmjene klase *ns3::TcpRxBuffer* dodana je klasa ns3::TokenBucket. Klasa se brine o količini dostupnih znakova u kablu, preostalom vremenu prije prelijevanja znakova i pokretanju algoritma oporavka po isteku maksimalnog dozvoljenog vremena kašnjenja. Zaglavlje klase nalazi se u privitku rada.

5.2. NetAnim

NetAnim je animator temeljen na više-platformskom Qt okruženju. NetAnim može animirati simulaciju na temelju zapisa (engl. trace) tijekom simulacije modela. Osim animiranja simulacije, NetAnim sadrži jednostavne alate za statističku analizu simulacije. Svojstva NetAnim 2.0 aplikacije koja se koristi u ovome radu su:

Osnovna svojstva:

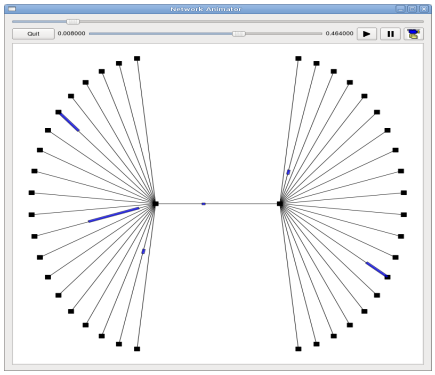
1. Čitanje XML datoteka koje generiraju objekti ns3 AnimationInterface klasa

2. Mogućnost praćenja žičnog i bežičnog prijenosa podataka
3. Prikaz animacije čvorova koji podržavaju model pokretljivosti
4. Praćenje pokretljivosti čvorova
5. Uzimanje snimka trenutnog stanja animacije
6. Napredne funkcije prikaza animacije poput interakcije sa elementima animacije, uvećanja ili smanjenja slike, prikaz svojstava elemenata.

Napredna svojstva (zahtjeva integraciju sa ns3 bibliotekama):

1. Animiranje u stvarnom vremenu za vrijeme izvođenja simulacije
2. Statistički alati za analizu svih elemenata simulacije
3. Analiza tablica usmjeravanja i IPv4 analiza poput statistike primljenih i poslanih paketa, brojača izgubljenih paketa
4. Analiza primljenih i poslanih podataka na aplikacijskoj razini
5. Analiza međuspremnika i redova.

Na slici 5.3 nalazi se primjer NetAnim simulacije prijenosa podataka.



Slika 5.3 NetAnim simulacija

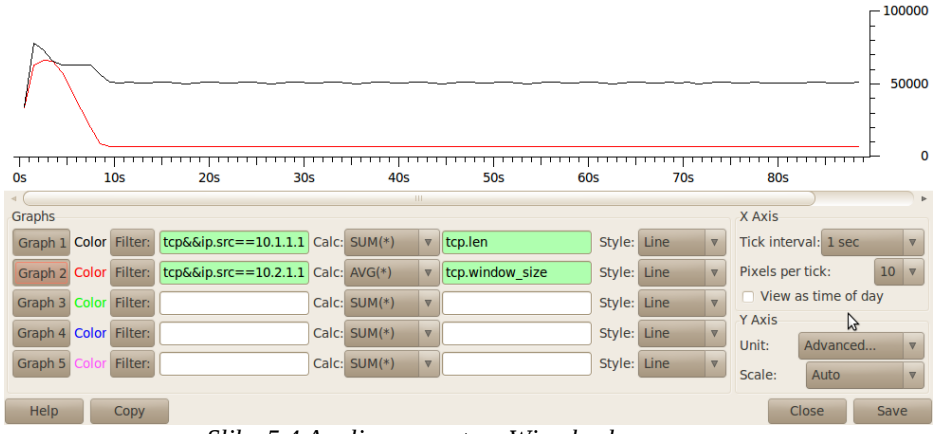
5.3. Mjerenja

Sva mjerenja u ovom radu napravljena su na simulacijama u ns-3 simulatoru. Zapis (engl. trace) simulacija moguće je zapisati u tekstualnom i pcap [23] (engl. packet capture) formatu. U ovom radu korišten je isključivo pcap format. Tragovi su analizirani korištenjem analizatora mrežnih protokola, Wireshark. Glavne odlike Wiresharka su:

1. Snimanje paketa ili analiza već snimljenog prometa
2. Snimanje prometa više vrsta mrežnih protokola uključujući: Ethernet, IEEE 802.11 i PPP
3. Snimljeni problem moguće je pregledavati u grafičkom sučelju ili u konzolnoj aplikaciji TShark

4. Snimljene datoteke je moguće programski mijenjati
5. Prikaz podataka moguće je mijenjati korištenjem filtera
6. Dodaci za analizu novih protokola
7. Detektiranje i praćenje VoIP poziva

Osim navedenih funkcionalnosti Wireshark ima mogućnost i napredne analize pojedinih polja TCP zaglavlja i podataka koje TCP segmenti prenose. Rezultate analiza moguće je pohraniti graf kao sliku ili csv datoteku. Kako bi analizirali pcap zapis prometa dovoljno je zapis otvoriti u Wireshark programu te u padajućem izborniku *Statistics* odabrati *IO Graphs*. Nakon toga se otvara prozor kao na slici 5.4, unutar kojega može definirati koji podaci nas zanimaju. Po završetku postavljanja filtera i funkcija koje želimo analizirati, podatke sačuvamo ili kao sliku odabirom gumba *Save*, ili kao csv listu vrijednosti klikom na *Copy:*



Slika 5.4 Analize prometa u Wireshark programu

Sve simulacije započinjemo pokretanjem izvršnih datoteka koje se nalaze u putanji *%putanja_projekta%\examples\QoSExtension*. *%putanja_projekta%* putanja je do eclipse projekta unutar kojega je definiran ns3 model. Dvije su datoteke koje možemo pokrenuti ErrorMeasurements i ConcurrentMeasurements. Njihovi glavni odsječci izvornog koda nalaze se u privitku na kraju ovog rada. Izvršne datoteke čitaju konfiguracijske datoteke iz tekućeg direktorija na temelju kojih se izvršavaju simulacije. Konfiguracije `error.config` i `concurrent.config` definiraju parametre dviju simulacija, poput kvalitete usluge, brzine kojom aplikacije šalju podatke, propusnosti i pouzdanost veza te korištene verzije TCP protokol. Primjeri konfiguracijskih datoteka se nalaze u privitku. Rezultat izvršavanja su pcap datoteke koje se pohranjuju u *%putanja_projekta %\examples\QoSExtension\mesurements*. Pošto se kompajliranje projekta izvršava wafom, alatom pisanom u programskom jeziku Python pokretanje izvršnih datoteka unutar projekta izgleda ovako:

```
./waf –run ErrorMeasurements
```

Nakon izvršavanja sadržaj putanje u kojemu se pohranjuju zapisi sadrži datoteke:

```
ls -l
total 20
drwxr-xr-x 2 4096 2011-09-09 10:25 concurent0
drwxr-xr-x 2 4096 2011-09-09 10:25 concurent1
drwxr-xr-x 2 4096 2011-09-09 19:46 error0
drwxr-xr-x 2 4096 2011-09-09 19:49 error1
drwxr-xr-x 2 4096 2011-09-09 19:49 error2
```

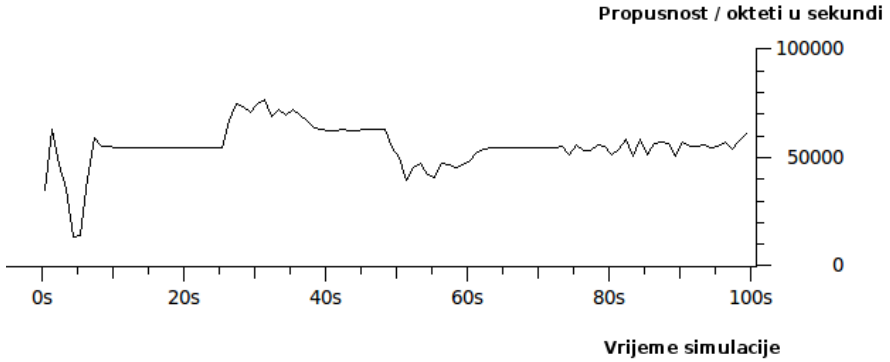
Datoteke čiji naziv započinje sa error sadrže rezultate simulacija na prvoj konfiguraciji, a datoteke koje počinju sa concurrent na drugoj. Broj na kraju naziva označava redni broj konfiguracije iz konfiguracijske datoteke. Unutar svake od putanja nalaze se pcap zapisi svih čvorova mrežne topologije korištene u simulaciji.

5.3.1. Utjecaj na konkurentni prijenos podataka

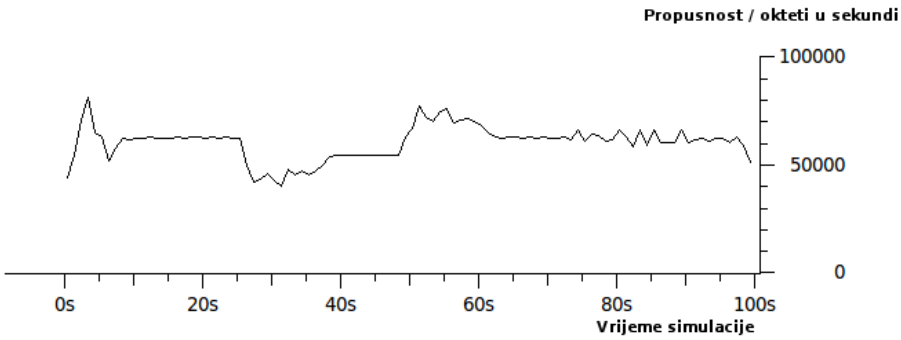
Utjecaj modificiranog TCP protokola simuliran je na konfiguraciji prikazanoj na slici 5.5. Između čvorova 3 i 5 podaci se prenose koristeći neizmijenjeni TCP protokol. U prvoj simulaciji između čvorova 2 i 4 također koristimo TCP protokol bez ikakvih izmjena. Veze između listova odnosno vanjskih čvorova i unutrašnje veze 0-1 imaju propusnost od 10Mbps. Unutrašnja veza ima propusnost od 1Mbps. Kašnjenje od listova do unutrašnje veze, odnosno čvorova 0 i 1, iznosi 15ms, a kroz središnju vezu 10ms. Aplikacije vezane za čvorove 2 i 3 generiraju pakete veličine 536 okteta brzinom od 500kbps, koje aplikacije vezane za čvorove 4 i 5 čitaju odmah po primitku. U ovakvim uvjetima za očekivati je da niti jedna aplikacija neće uspjeti prenositi podatke onom brzinom kojom oni pristižu.

Slika 5.5 Topologija mrežne arhitekture u prvoj simulaciji

Na slikama 5.6 i 5.7 prikazani su rezultati mjerenja propusnosti. Vidljiva je faza polaganog starta prilikom kojeg dolazi do naglog pada propusnosti. Taj pad posljedica je zakrčenja uslijed prekomjernog slanja podatka dvaju tokova. Od 7 sekunde oba toka se nalaze u fazi izbjegavanja zakrčenja, no njihove propusnosti variraju. Od 65 sekunde možemo reći da su tokovi stabilni sa manjim varijacijama.

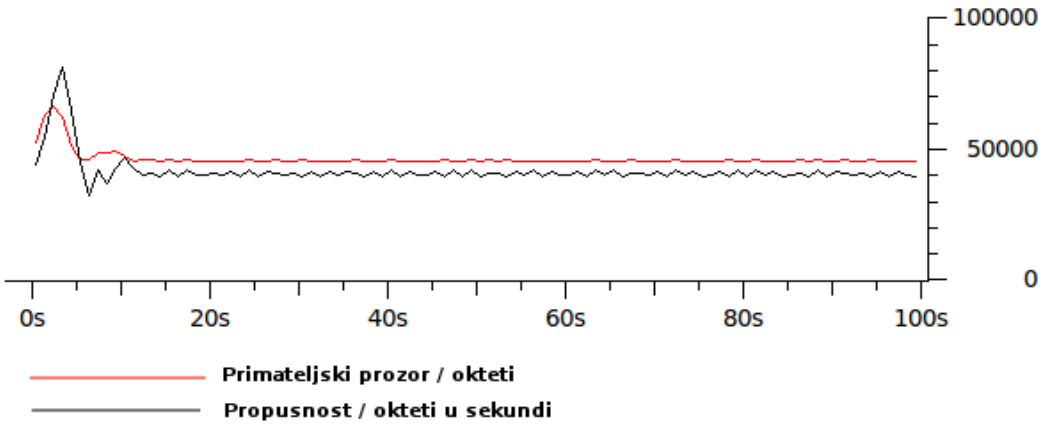


Slika 5.6 Propusnost prvog toka podataka



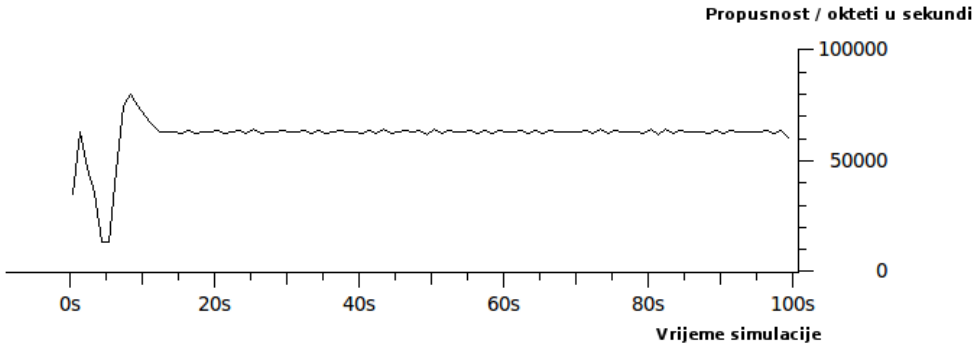
Slika 5.7 Propusnost drugog toka podataka

U slijedećoj je simulaciji između čvorova 2 i 4 korišten modificirani TCP protokol sa definiranom kvalitetom usluge. Parametri kvalitete usluge su postavljeni na brzinu prijenosa od 40 000 okteta u sekundi odnosno 320kbps, maksimalno kašnjenje 300ms, varijaciju kašnjenja 100ms i koeficijent pouzdanost 0.7. Propusnost i veličina rwnd-a prikazani su na slici 5.8.



Slika 5.8 Propusnost i rwnd izmjenjenog protokola TCP

Propusnost između čvorova 3 i 5 nalazi se na grafu na slici 5.9:



Slika 5.9 Propusnost neizmijenjenog protokola TCP

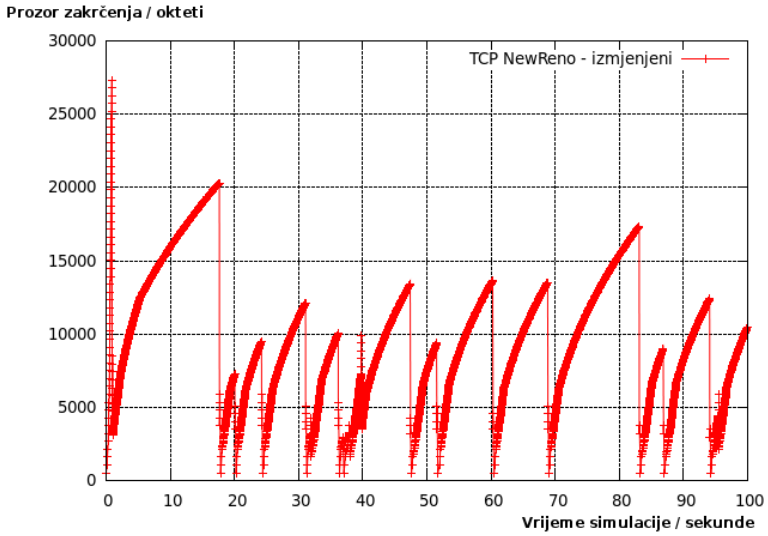
Za razliku od prvog mjerenja, propusnosti tokova stabilizirale su se već u 15 sekundi. Posljedica je to definiranja kvalitete usluge sa brzinom prijenosa na toku između čvorova 2 i 4. Prvih 5 sekundi graf je sličan prethodnom grafu propusnosti za mjerenja bez definirane kvalitete usluge. Na samom početku polaganog starta broj znakova u kablu sa znakovima raste brže od prozora zakrčenja. Stoga je tok kontroliran prozorom zakrčenja a ne primateljskim prozorom. U trenutku kada prozor zakrčenja naraste na vrijednost veću od primateljskog prozora tok počinje kontrolirati primatelj i propusnost se stabilizira na ugovorenu vrijednost. Pošto je ugovorena brzina prijenosa 320kbps, propusnost veze dovoljna je da se tok između čvorova 1 i 5 stablizira na brzinu slanja kojom aplikacija šalje podatake.

5.3.2. Utjecaj na kontrolu zakrčenja

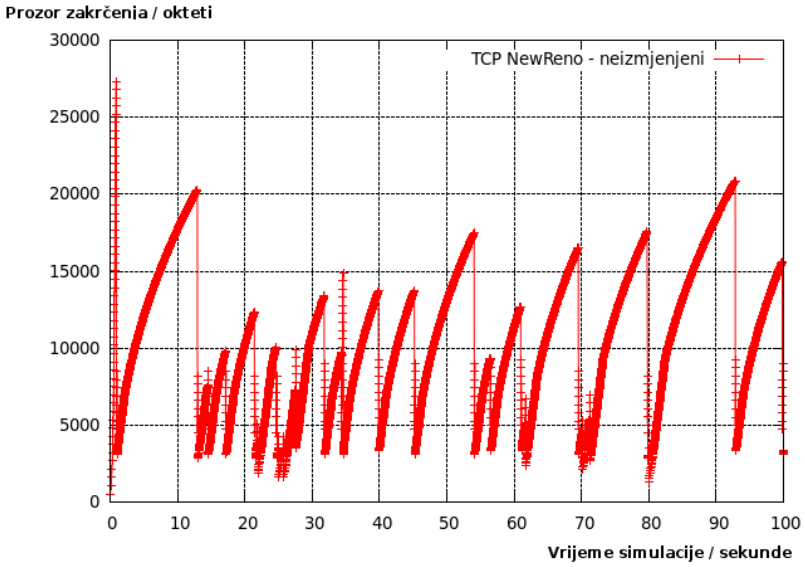
Utjecaj izmjena na mijenjanje prozora zakrčenja, odnosno izmjene faza TCP protokola tokom prijenosa podataka se mjeri jednostavnom simulacijom prijenosa između dviju aplikacije. Na primateljskoj strani mjerimo rezultate za dva slučaja, neizmijenjeni i

modificirani TCP protokol. Veza kojom se segmenti prenose simulira slučajni gubitak segmenata. Kvalitetu veze izražavamo faktorom učestalosti greške i njenom vjerojatnošću. U mjerenjima za tri verzije TCP protokola, New Reno, Reno i Tahoe, korištena je vjerojatnost pogreške od 50%, a faktora učestalosti greške 5e-6. Učestalost je izražena faktorom koji je proporcionalan sa vremenom trajanja između dvije greške.

Za TCP Reno rezultati simulacije identični su onima za TCP New Reno. Na slikama 5.10 i 5.11 usporedno pratimo pošiljateljski prozor zakrčenja tokom prijenosa podataka. Vidljivo je smanjenje broja odlazaka u fazu polaganog starta.

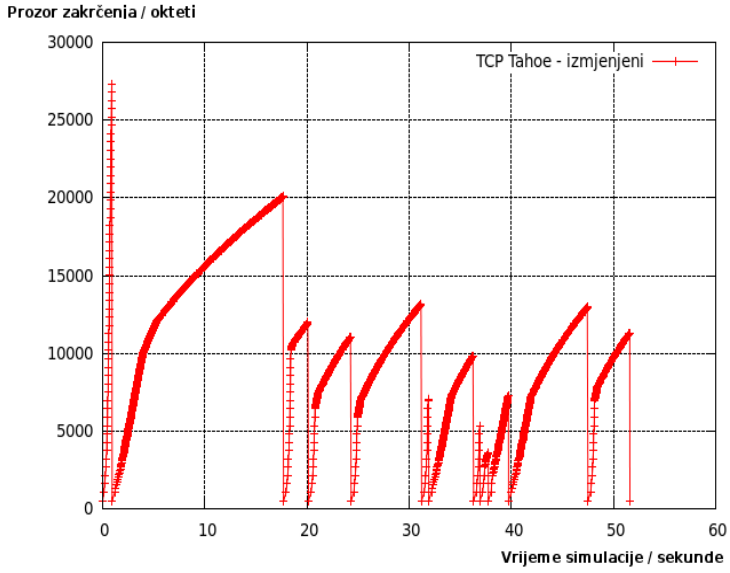


Slika 5.10 Izmijenjeni TCP NewReno protokol

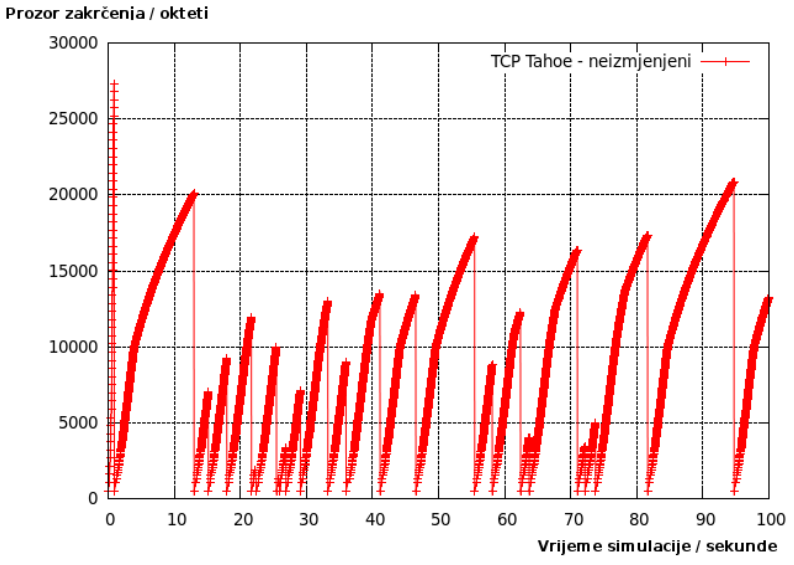


Slika 5.11 Neizmijenjeni TCP NewReno protokol

Istovjetna simulacija obavljena je i za TCP Tahoe. Usporedni grafovi prozora zakrčenja nalaze se na slikama 5.12 i 5.13. Vidljivo je da je utjecaj modifikacije još izraženiji za TCP Tahoe:



Slika 5.12 Izmijenjeni TCP Tahoe protokol

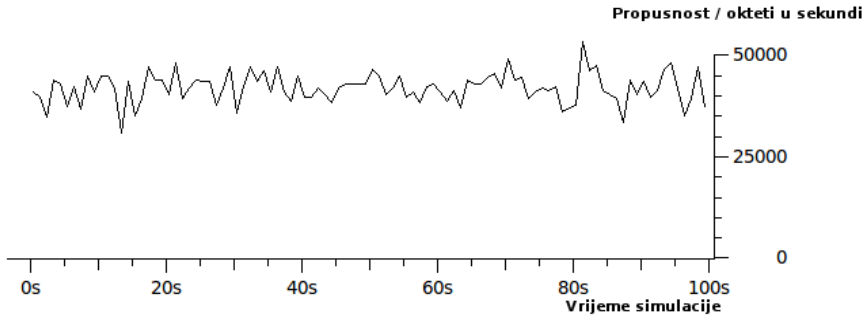


Slika 5.13Neizmjenjeni TCP Tahoe protokol

U sve četiri simulacije prenesena je ista količina podataka.

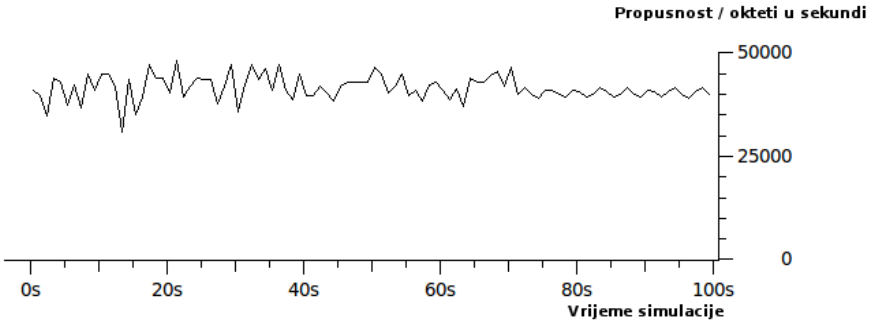
5.3.3. Brzina prijenosa kao kvaliteta usluge

U sljedećoj simulaciji uspoređen je tok podataka sa i bez definirane brzine prijenosa. U prvom slučaju (Slika 5.14) simulira se prijenos podataka između aplikacije koja generira podatke prosječnom brzinom od 360kbps. Rezultati na slici 14. koriste se kao temeljni scenarij za usporedbu sa rezultatima drugih simulacija.



Slika 5.14 Propusnost neizmijenjenog protokola TCP u prvoj simulaciji

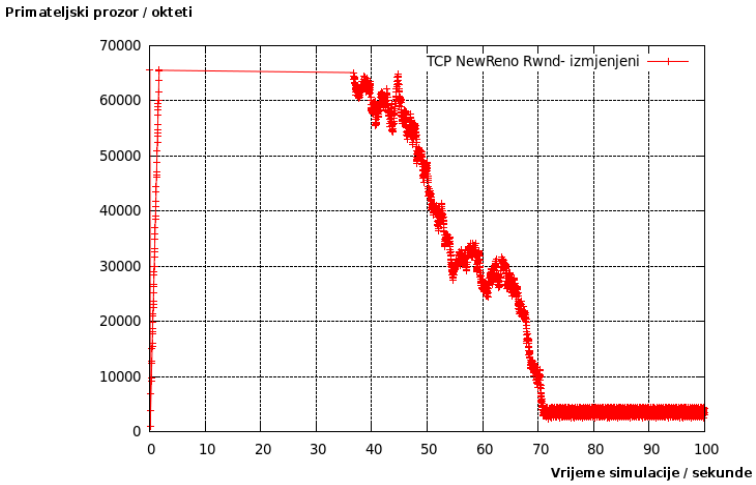
Uz definiranu brzinu prijenosa od 320kbps, za isti tok dobivamo rezultate prikazane na



Slika 5.15 Propusnost neizmijenjenog protokola TCP u drugoj simulaciji

slici 5.15. Vidljivo je značajno smanjenje varijacije propusnosti. Stabilniji prijenos posljedica je stalne brzine prijenosa konkurentskog toka podataka.

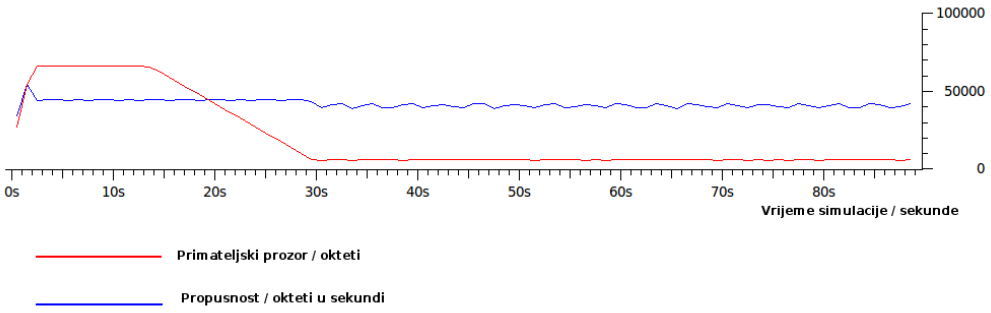
Rezultati mjerenja brzine prijenosa za simulaciju prikazani na slici 5.15 odgovaraju promjeni veličine primateljskog prozora što je prikazano na slici 5.16.



Slika 5.16 Primateljski prozor izmijenjenog TCP protokola

5.3.4. Konvergencija ravnotežnom stanju

Važan aspekt proširenja TCP protokola je vrijeme koje je potrebno kako bi prijenos podataka postigao stabilno stanje. Stabilno stanje definirano je kao stanje u kojemu primateljski prozor, kao i brzina prijenosa podataka, teže stalnoj vrijednosti. Slika 5.17 prikazuje prijenos podataka sa definiranom brzinom prijenosa kao kvalitetom usluge. Brzina prijenosa je 40000 okteta u sekundi odnosno 320kbps. Prijenos podataka odvija se na vezi kojom se istovremeno prenose podaci neizmijenjenog TCP protokola brzinom 500kbps. Propusnost veze je 1Mpbs. Maksimalno dozvoljeno kašnjenje je postavljeno na 20s kako bi se osiguralo da prijenos neće biti prekinut uslijed nemogućnosti ostvarenja definiranih parametara kvalitete veze.



Slika 5.17 Primateljski prozor i brzina slanja podataka tokom konkurentnog prijenosa

Iz simulacije je vidljivo da je tek nakon 30 sekundi postignuto stabilno stanje prijenosa podataka. Zbog toga u navedenim uvjetima ne možemo računati na kašnjenja koja bi dopuštala prijenos podataka u stvarnom vremenu. Ovaj problem moguće je riješiti "zagrijavanjem" veze, odnosno slanjem slučajnih podataka do postizanja stabilnog prijenosa. Takvo riješenje zahtjevalo bi izmjene i na pošiljateljskoj strani.

Na slici 5.17 kao i na slikama iz prethodnih simulacija vidljiv je "šiljak" na početku grafa

propusnosti. Ta prijelazna pojava posljedica je faze polaganog starta tokom koje je kontrola toka, zbog malih vrijednosti prozora zakrčenja, na pošiljateljskoj strani. To znači da je broj nepotvrđenih segmenata na mreži određen veličinom prozora zakrčenja. Stoga se znakovi počinju gomilati u kablu, do trenutka kada prozor zakrčenja ne naraste na vrijednost veću od vrijednosti primateljskog prozora,te omogući naglo pražnjenje kabla. Nakon toga vrijednost primateljskog prozora pada na znatno nižu vrijednost i time smanjuje propusnost, koja počinje težiti brzini prijenosa ugovorenoj kvalitetom usluge.

6. Zaključak

U ovom radu predstavljena je analiza mogućih proširenja TCP protokola, s ciljem bolje prilagodbe za prijenos podataka u stvarnom vremenu. Na primjerima radova i analiza postojećih prilagodbi TCP protokola ocjenjuju se različita rješenja 3 glavna problema: (i) neučinkovitog iskorištenja propusnosti veze, (ii) nemogućnost razlikovanja zakrčenja veze od slučajnog gubitka paketa i (iii) agresivno smanjivanje prozora u slučaju zakrčenja. Moguće izmjene možemo kategorizirati u dvije glavne skupine: izmjene kontrole zakrčenja i sustava potvrđivanja TCP segmenata. Cilj je odrediti optimalni kompromis između pouzdanosti, propusnosti i varijacije kašnjenja. Sve izmjene vrednuju se prema pravednosti raspodjele kapaciteta mreže, usklađenosti sa postojećim TCP implementacijama i učinka prijenosa podataka u stvarnom vremenu. Dodatni kriterij uspješnosti svakako je i opseg potrebnih izmjena za određeno rješenje, to jest odnose li se izmjene na primatelja, pošiljatelja ili oboje te jesu li potrebne izmjene aplikacija koje šalju podatke.

Kako bismo prilagodili TCP potrebama prijenosa podatka u stvarnom vremenu, nužno je promijeniti kontrolu zakrčenja. No, zbog kolapsa mreže koji može uslijediti uslijed zakrčenja, svaka izmjena se pomno ocjenjuje. Izmjenu možemo smatrati realnom mogućnošću ukoliko ju je moguće integrirati sa postojećom infrastrukturom i protokolima. TCP kompatibilnost omogućava komunikaciju između izmijenjene strane i TCP protokola. Svojstvo kompatibilnosti omogućava postupnu zamjenu. Zbog toga što veliku većinu Internet prometa čini TCP, vrlo je važno provjeriti kako pojedina izmjena utječe na usporedni TCP promet. Ukoliko modifikacija onemogućava da se TCP ravnopravno natječe za propusnost, njeno korištenje uzrokovalo bi pogoršanje usluga koje koriste TCP.

Pouzdani prijenos podataka nije nužan zahtjev prilikom prijenosa podataka u stvarnom vremenu, no brojne TCP primjene oslanjaju se na to svojstvo. Stoga se nameće potreba za pružanjem kvalitete usluge kako bi se TCP mogao koristiti i za prijenos podataka u stvarnom vremenu. Kvaliteta usluge može definirati parametre poput pouzdanosti, maksimalnog kašnjenja, razine varijacije kašnjenja itd.

Na temelju obavljenih simulacija vidljivo je da predložena kontrola zakrčenja na primateljskoj strani smanjuje varijaciju kašnjenja i ograničava propusnost tako da odgovara ugovorenoj brzini prijenosa. Najveći problem predloženog proširenje je veliki vremenski period potreban kako bi prijenos podataka postigao stabilno stanje u kojem brzina prijenosa i primateljski prozor teže stalnoj vrijednosti. Moguće rješenje tog problema je početno slanje nasumičnih podataka kako bi postigli stabilno stanje veze. No to bi zahtjevalo izmjene i na pošiljateljskoj strani protokola TCP. Ukoliko kašnjenje nije problem što je slučaj kod gledanja video sadržaja i slušanja glazbe preko Interneta, predložena kvaliteta usluge zadovoljava sve zahtjeve.

Glavni cilj ovog rada je istražiti mogućnosti poboljšanja TCP protokola, odnosno proširenja funkcionalnosti istog. Na temelju simulacija može se zaključiti kako za taj zadatak nisu potrebni kompleksni zahvati na mrežnoj opremi niti na obje strane koja komuniciraju. Upravo te karakteristike rješenja čine ga realnom opcijom. Rezultati simulacija predloženih izmjena indikativni su pokazatelji uspješnosti rješenja, no takve rezultate treba dodatno potvrditi mjerenjima na stvarnim uređajima i tokovima podataka.

7. Literatura

1. K.-J. Grinnemo, A. Brunstrom. A Simulation Based Performance Analysis of a TCP Extension for Best Effort Multimedia Applications. *Proceedings of the 35th Annual Simulation Symposium*, San Diego, California, USA, 2002.

2. P. H. Hsiao, H. T. Kung, K-S. Tan,Video over TCP with receiver based delay control. NOSSDAV, Port Jefferson, New York, USA, 2001.

3. O. Núñez Mori , J. Rochol, TCP HolyWood: A New Approach to Improve Throughput and Reduce Jitter. *Dissertação de Mestrado*, Biblioteca de Informática da UFRGS, Porto Alegre, Brazil, 2005.

4. N. Parvez, E. Hossain, TCP Prairie: A sender-only TCP modification based on adaptive bandwidth estimation in wired-wireless networks. Computer Communications (Elsevier), 2004.

5. S. Karandikar, S. Kalyanaraman, P. Bagal, B. Packer, TCP Rate Control. Computer Communications Review, 2000.

6. M. Maldonado, S. A. Baset, H. Schulzrinne, TCP-Friendly Rate Control with Token Bucket for VoIP Congestion Control. Department of Computer Science, Columbia University, New York, USA, 2005.

7. C. Zhang, V. Tsaoussidis, TCP-Real: Improving Real-time Capabilities of TCP over Heterogeneous Networks. 11th IEEE/ACM NOSSDAV,ACM Press, Port Jefferson, New York, 2001.

8. S. Liang, D. Cheriton, TCP-RTM: Using TCP for Real Time Applications. Stanford University Distributed Systems Group, URL:http://dsg.stanford.edu/sliang/rtm.pdf (4/4/2011).

9. B. Mukherjee, T. Brecht ,Time-lined TCP for the TCP-friendly Delivery of Streaming Media. Proceedings of the 2000 International Conference on Network Protocols, IEEE Computer Society Washington , DC, USA, 2000.

10. A. Gember, D. Brylow. Real-Time TCP for Embedded Devices. ACM Student Research Competition Poster Session, ACM SIGCSE, Chattanooga, TN, 2009.

11. S. Floyd, Congestion Control Principles, RFC 2914, 2000.

12. C. Casetti, M. Gerla, S. Mascolo, M.Y. Sanadidi, R. Wang TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links, *Proceedings of ACM Mobicom 2001 Conference*, Rome, Italy, 2001.

13. S. Mascolo, A. Grieco, G. Pau, M. Gerla, C. Casetti, End-to-End Bandwidth Estimation in TCP to Improve Wireless Link Utilization. European Wireless Conference, Florence, Italy, 2002.

14. S. Floyd, M. Handley, J. Padhye, and J. Widmer. *TCP friendly rate control (TFRC): Protocol specification*, RFC 3448, 2003.

15. J. Padhye,V. Firoiu, D. Towsley, J. Kurose. Modeling TCP throughput: A simple model and its empirical validation, Proc. of ACM SIGCOMM, Vancouver, Canada,1998.

16. L. Brakmo and L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. IEEE Journal on Selected Areas in Communication, 1995.

17. Tian Bu, Yong Liu, Don Towsley. On the TCP-Friendliness of VoIP Traffic, *Proceedings of INFOCOM*, 2006.

18. H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson. *RTP: A Transport Protocol for Real-Time Applications*, RFC 3550 ,2003

19. John Nagle. Congestion Control in IP/TCP Internetworks, RFC 896, 1984.

20. Dario Bonfiglio et al. “Revealing Skype Traffic: When Randomness Plays with You,” ACM SIGCOMM Computer Communication Review, Volume 37:4 (SIGCOMM 2007), p. 37-48

21. Issariyakul, Teerawat , Hossain, Ekram: Introduction to Network Simulator NS2, Springer, USA, 2009

22. Dennis Brylow: An experimental laboratory environment for teaching embedded operating systems, Proceedings of the 39th SIGCSE, New York, USA, 2008

23. Glen Turner: URL: http://www.iana.org/assignments/media-types/application/vnd.tcpdump.pcap, 2011

8. Privitak

ConcurrentMeasurements.cc

```
// Create the router point-to-point link helpers
PointToPointHelper pointToPointRouter;
pointToPointRouter.SetDeviceAttribute("DataRate", StringValue(routerBandwidth));
pointToPointRouter.SetChannelAttribute("Delay", StringValue(routerDelay));

// Create the leaf point-to-point link helpers
PointToPointHelper pointToPointLeaf;
pointToPointLeaf.SetDeviceAttribute("DataRate", StringValue(leafBandwith));
pointToPointLeaf.SetChannelAttribute("Delay", StringValue(leafDelay));

//create dumbbell network configuration helper
PointToPointDumbbellHelper d(2, pointToPointLeaf, 2, pointToPointLeaf, pointToPointRouter);

// Install Internet Stack on dumbbell configuration
InternetStackHelper stack;
d.InstallStack(stack);

// Assign IP Addresses to all nodes
d.AssignIpv4Addresses(Ipv4AddressHelper("10.1.1.0", "255.255.255.0"),
  Ipv4AddressHelper("10.2.1.0", "255.255.255.0"),
  Ipv4AddressHelper("10.3.1.0", "255.255.255.0"));

//enable pcap tracing and save to measurements folder
stack.EnablePcapIpv4All(str.str());

//port on which application will send data
uint16_t sinkPort = 8080;

//sink/receiver applications container
ApplicationContainer sinkApps;
//setup all left/sending nodes
for (uint32_t i = 0; i < d.RightCount(); i++) {
 //setup tcp socket type, Ren, New Reno or Tahoe
 if (i == 0) {
  Config::SetDefault ("ns3::TcpL4Protocol::SocketType", StringValue (ModifiedTCPSocketType));
 } else {
  Config::SetDefault ("ns3::TcpL4Protocol::SocketType", StringValue (NonModifiedTCPSocketType));
 }

 //create tcp socket
 Ptr<Socket> ns3TcpSocket = Socket::CreateSocket(d.GetRight(i),
  TcpSocketFactory::GetTypeId());

 if (i == 0) {
  //setup QoS parameters
  ns3TcpSocket->SetAttribute("QoSBitRate", UintegerValue(ModifiedTCPQoSBitrate));
  ns3TcpSocket->SetAttribute("QoSDelay", UintegerValue(ModifiedTCPQoSDelay));
  ns3TcpSocket->SetAttribute("QoSJitter", UintegerValue(ModifiedTCPQoSJitter));
  ns3TcpSocket->SetAttribute("QoSReliability", DoubleValue(ModifiedTCPQoSReliability));
 }

 //create application
 Ptr<CustomReceiverApp> app = CreateObject<CustomReceiverApp>();

 //setup application
 app->Setup(ns3TcpSocket,InetSocketAddress(Ipv4Address::GetAny(), sinkPort));

 //add application to node
 d.GetRight(i)->AddApplication(app);

 //add application to conatiner
 sinkApps.Add(app);

}

//setup start and stop time for all applications in container
sinkApps.Start(Seconds(0.0001));
sinkApps.Stop(Seconds(100.));

//data send application container
ApplicationContainer senderApps;
```

```
//setup all right/receiving nodes
for (uint32_t i = 0; i < d.LeftCount(); i++) {

 //setup tcp socket type, Reno, New Reno or Tahoe
 if (i == 0) {
  Config::SetDefault ("ns3::TcpL4Protocol::SocketType", StringValue (ModifiedTCPSocketType));
 } else {
  Config::SetDefault ("ns3::TcpL4Protocol::SocketType", StringValue (NonModifiedTCPSocketType));
 }

 //create socket
 Ptr<Socket> ns3TcpSocket = Socket::CreateSocket(d.GetLeft(i),
      TcpSocketFactory::GetTypeId());
    Ptr<CustomSenderApp> app = CreateObject<CustomSenderApp>();

 //setup application
 if (i == 0) {
  app->Setup(ns3TcpSocket, InetSocketAddress(d.GetRightIpv4Address(i), sinkPort),
    536, 2000000, DataRate(ModifiedTCPApplicationBitrate));
 } else {
  app->Setup(ns3TcpSocket, InetSocketAddress(d.GetRightIpv4Address(i), sinkPort), 536,
    2000000, DataRate(NonModifiedTCPApplicationBitrate));
 }

 //add application to node
 d.GetLeft(i)->AddApplication(app);

 //add application to application container
 senderApps.Add(app);
}

//setup start and stop time for all applications in container
senderApps.Start(Seconds(0.01));
senderApps.Stop(Seconds(100.));

//setup routing tables for dumbbell network configuration
Ipv4GlobalRoutingHelper::PopulateRoutingTables();

//define simulation stop time
Simulator::Stop(Seconds(100));

//run simulation
Simulator::Run();

//destroy simulation
Simulator::Destroy();
```

concurrent.config

```
#leafDelay;
#leafBandwith;
#routerDelay;
#routerBandwidth;
#NonModifiedTCPApplicationBitrate;
#NonModifiedTCPSocketType;
#ModifiedTCPApplicationBitrate;
#ModifiedTCPSocketType
#ModifiedTCPQoSDelay(ms);
#ModifiedTCPQoSJitter(ms);
#ModifiedTCPQoSBitrate(Bps);
#ModifiedTCPQoSReliability(ms)
15ms;1Mbps;10ms;1Mbps;300kbps;ns3::TcpNewReno;500kbps;ns3::TcpTahoe;400;100;50000;0.7
15ms;1Mbps;10ms;1Mbps;250kbps;ns3::TcpReno;350kbps;ns3::TcpTahoe;400;100;40000;0.7
```

ErrorMeasurements.cc

```
//setup QoS parameters, because packet sink application is created after
//simulation start, it's easier to setup QoS parameters as default and
//later restart qos parameters on sender application
Config::SetDefault("ns3::TcpL4Protocol::SocketType", StringValue(TcpSocketType));
Config::SetDefault("ns3::TcpSocket::QoSBitRate", UintegerValue(ModifiedTCPQoSBitrate));
Config::SetDefault("ns3::TcpSocket::QoSDelay", UintegerValue(ModifiedTCPQoSDelay));
Config::SetDefault("ns3::TcpSocket::QoSJitter", UintegerValue(ModifiedTCPQoSJitter));
Config::SetDefault("ns3::TcpSocket::QoSReliability", DoubleValue(ModifiedTCPQoSReliability));

//create node container, and nodes
NodeContainer nodes;
nodes.Create(2);

// Create the point-to-point link helper, define bandwidth and delay
PointToPointHelper pointToPoint;
pointToPoint.SetDeviceAttribute("DataRate", StringValue(linkBandwith));
pointToPoint.SetChannelAttribute("Delay", StringValue(linkDelay));

//create network configuration
NetDeviceContainer devices;
devices = pointToPoint.Install(nodes);

pointToPoint.EnablePcap("qosTrace", devices.Get(0), true, false);

//define error rate model
Ptr<RateErrorModel> em = CreateObjectWithAttributes<RateErrorModel>(
 "RanVar", RandomVariableValue(UniformVariable(0., 1.)),
  "ErrorRate", DoubleValue(linkErrorModelInterval));

//attach error model to link
devices.Get(1)->SetAttribute("ReceiveErrorModel", PointerValue(em));

//Install Internet stack on devices
InternetStackHelper stack;
stack.Install(nodes);

//Assign IP addresses
Ipv4AddressHelper address;
address.SetBase("10.1.1.0", "255.255.255.252");
Ipv4InterfaceContainer interfaces = address.Assign(devices);

//enable pcap tracing and save to measurements folder
stack.EnablePcapIpv4All(str.str());

//port on which application will send data
uint16_t sinkPort = 8080;

//define packet sink application
Address sinkAddress(InetSocketAddress(interfaces.GetAddress(1), sinkPort));
PacketSinkHelper packetSinkHelper("ns3::TcpSocketFactory",
  InetSocketAddress(Ipv4Address::GetAny(), sinkPort));

//Add application to node
ApplicationContainer sinkApps = packetSinkHelper.Install(nodes.Get(1));

//setup receiving application start and stop time
sinkApps.Start(Seconds(0.));
sinkApps.Stop(Seconds(100.));

//create sending node socket
Ptr<Socket> ns3TcpSocket = Socket::CreateSocket(nodes.Get(0),
  TcpSocketFactory::GetTypeId());

//reset QoS parameters
ns3TcpSocket->SetAttribute("QoSBitRate", UintegerValue(0));
ns3TcpSocket->SetAttribute("QoSDelay", UintegerValue(0));
ns3TcpSocket->SetAttribute("QoSJitter", UintegerValue(0));

//create sending application
Ptr<MyApp> app = CreateObject<MyApp>();

//setup application
app->Setup(ns3TcpSocket, sinkAddress, 536, 2000000, DataRate(ApplicationBitrate));
```

```
//add application to node
nodes.Get(0)->AddApplication(app);

//setup sending application start and stop time
app->SetStartTime(Seconds(0.001));
app->SetStopTime(Seconds(100.));

//define simulation stop time
Simulator::Stop(Seconds(100));

//run simulation
Simulator::Run();

//destroy simulation
Simulator::Destroy();
```

error.config

```
#linkDelay; linkBandwith; linkErrorModelInterval;ApplicationBitrate; TcpSocketType;
#ModifiedTCPQoSDelay(ms); ModifiedTCPQoSJitter(ms);
ModifiedTCPQoSBitrate(Bps);ModifiedTCPQoSReliability(ms)
15ms;1Mbps;0.000001;500kbps;ns3::TcpNewReno;400;100;50000;0.7
15ms;1Mbps;0.000001;500kbps;ns3::TcpReno;400;100;40000;0.7
15ms;1Mbps;0.000001;500kbps;ns3::TcpTahoe;400;100;50000;1
```