Transportni sloj računalne mreže

Erceg, Roko

Master's thesis / Diplomski rad

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, School of Applied Mathematics and Informatics / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet primijenjene matematike i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:126:076104

Sveučilište Josipa Jurja Strossmayera u Osijeku
Fakultet primijenjene matematike i informatike

Sveučilišni diplomski studij matematike, smjer:
matematika i računarstvo

Roko Erceg

Transportni sloj računalne mreže

Diplomski rad

Osijek, 2023.

Sveučilište Josipa Jurja Strossmayera u Osijeku
Fakultet primijenjene matematike i informatike

Sveučilišni diplomski studij matematike, smjer:

matematika i računarstvo

Roko Erceg

Transportni sloj računalne mreže

Diplomski rad

Mentor: izv. prof. dr. sc. Domagoj Matijević

Osijek, 2023.

Sadržaj

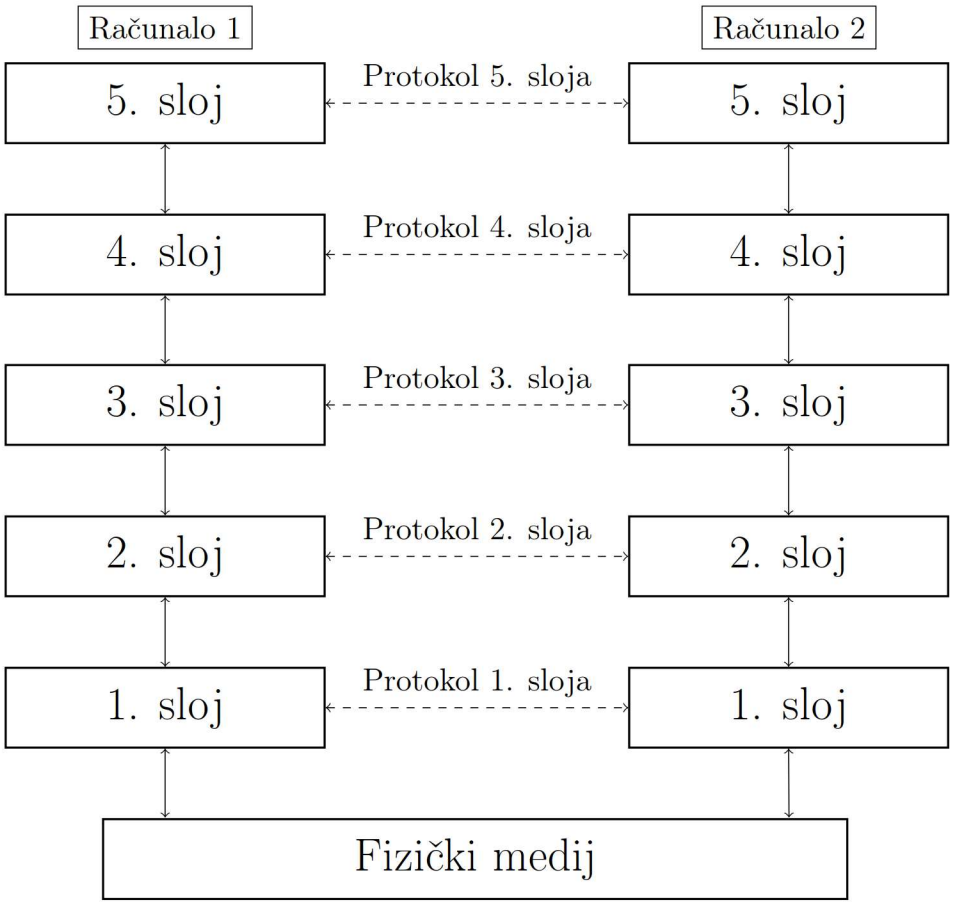
1 Uvod

U današnje vrijeme ne možemo zamisliti svijet bez računalnih mreža. Računalne mreže prisutne su u skoro svim aspektima našeg modernog života i omogućuju nam pouzdan prijenos podataka preko velikih udaljenosti. U samom središtu trenutne mrežne infrastrukture nalazi se transportni sloj računalne mreže. Ovaj sloj brine o efikasnosti, pouzdanosti i sigurnosti prijenosa podataka između ogromne količine računala povezanih putem mreže. Unutar transportnog sloja imamo dva važna protokola, to su TCP i UDP. Glavni cilj ova dva vrlo različita protokola jest dostava podataka do točno određenog procesa na udaljenom računalu.

U ovom radu kratko ćemo se osvrnuti na položaj samog transportnog sloja unutar mrežne arhitekture. No, većinu vremena provest ćemo u analiziranju funkcionalnosti transportnog sloja te protokola transportnog sloja. Zanimat će nas na koji se način ovi protokoli nose s problemima koji proizlaze iz nesavršenosti računalne mreže. U zadnjem poglavlju ovog rada dat ćemo primjer jedne implementacije koja u pozadini koristi UDP protokol (programski kod dostupan je na sljedećoj adresi: `https://github.com/rerceg/ReliableUDP`). Ideja ove implementacije je proširenje funkcionalnosti UDP protokola. Klasični UDP protokol vrlo je jednostavan i ne brine o stvarima poput uspješne dostave podataka do drugog uređaja. U ovom primjeru nadograđujemo klasični UDP protokol s određenim funkcionalnostima koje su prisutne kod TCP protokola poput potvrde primljenih segmenata, kontrole toka podataka te ponovnog slanja izgubljenih ili korumpiranih segmenata.

2 Hijerarhijska arhitektura

Računalne mreže najčešće se organiziraju kao stog od nekoliko slojeva od kojih je svaki sloj specijaliziran za određeni skup zadataka. Ti slojevi se razlikuju od mreže do mreže. Razlike uključuju broj i imena slojeva te funkcionalnost određenog sloja. Ono što im je zajedničko jest da svaki sloj pruža određene usluge sloju iznad sebe. Te usluge su dostupne "gornjem" sloju bez da on mora znati na koji način su te usluge implementirane u "donjem" sloju. Komunikaciju putem mreže između dva računala možemo promatrati kao komunikaciju između svaka dva sloja posebno. Prilikom komunikacije i-tog sloja na jednom računalu i i-tog sloja na drugom računalu, pravila koja se koriste u toj komunikaciji nazivamo protokolom sloja i. Protokol možemo definirati kao dogovor između sudionika komunikacije o načinu i pravilima komunikacije.



Slika 1: Primjer komunikacije putem mreže s 5 slojeva

Iako protokoli služe za komunikaciju između istih slojeva na udaljenim uređajima, tu komunikaciju bismo mogli nazvati virtualnom. Razlog tome je što slojevi ne komuniciraju direktno, odnosno podaci ne teku direktno iz i-tog sloja jednog računala u i-ti sloj drugog računala. Svaki sloj prosljeđuje podatke u sloj direktno "ispod" njega. Ispod zadnjeg sloja, sloja 1, nalazi se fizički medij (npr. optički kabel, bakrena žica, radiovalovi). Putem fizičkog medija odvija se stvarna komunikacija. Na slici 1 iscrtkanim linijama ilustrirana je virtualna komunikacija, dok je normalnim linijama prikazan stvarni put podataka.

Dva susjedna sloja komuniciraju putem sučelja (eng. interface). To sučelje definirano je od strane "nižeg" sloja i definira koje njegove metode su dostupne za korištenje "višem" sloju. Ovdje pronalazimo određene principe koji su nam poznati iz objektno orijentiranog

programiranja. "Niži" sloj ne dopušta korištenje svih svojih metoda. Svaki sloj obavlja točno određeni skup funkcija i daje na znanje "višem" sloju koje funkcije su mu na raspolaganju. Dobro definirano sučelje omogućava zamjenu trenutnog sloja s nekim slojem s potpuno drugačijom implementacijom ili čak s potpuno drugačijim protokolom (sve dok sučelje ostaje nepromijenjeno). Protokol u i-tom sloju može biti zamijenjen bez da ostali slojevi znaju za tu promjenu. Na primjer, možemo zamijeniti bakrene žice s optičkim kabelima i sve što trebamo zamijeniti u našoj arhitekturi jest najniži sloj, tj. sloj koji direktno komunicira s fizičkim medijem.

Arhitektura mreže je zajednički naziv za skup korištenih slojeva i protokola. Skup protokola koje koristi određeni sustav naziva se stog protokola. Korisnu analogiju za višeslojnu komunikaciju možemo pronaći u [1]. Zamislimo da imamo dva filozofa (procesi u trećem sloju), jedan govori samo engleski, drugi govori samo hrvatski. Budući da nemaju zajednički jezik, svaki od njih će trebati pomoć prevoditelja (proces u drugom sloju). Svaki od dva prevoditelja nakon toga kontaktira tajnicu (proces u prvom sloju). Prvi filozof daje svoju poruku (putem sučelja između trećeg i drugog sloja) na engleskom svome prevoditelju. Dva prevoditelja su se prethodno dogovorili da će komunicirati na njemačkom jeziku (protokol drugog sloja). Prvi prevoditelj prevodi poruku s engleskog na njemački i prosljeđuje ju tajnici (putem sučelja između drugog i prvog sloja). Tajnica šalje poruku putem elektronske pošte (protokol prvog sloja). Kada druga tajnica primi poruku, ona ju prosljeđuje prevoditelju koji prevodi poruku s njemačkog na hrvatski. Drugi filozof dobiva poruku na hrvatskom (kroz sučelje između drugog i trećeg sloja). Možemo primijetiti da protokol ne ovisi o ostalim protokolima sve dok sučelje ostaje nepromijenjeno. Npr. prevoditelji mogu odlučiti koristiti francuski umjesto njemačkog jezika. Ako se obojica slože za taj jezik, ishod će biti isti.
U stvarnosti komunikacija teče na sljedeći način. Želimo da poruka iz petog sloja jednog računala stigne do petog sloja drugog računala. Peti sloj daje poruku četvrtom sloju koji na poruku dodaje zaglavlje koje služi za identificiranje poruke, pošiljatelja, primatelja itd. Poruka se zatim predaje trećem sloju. Ovisno o veličini poruke, u trećem sloju može doći do segmentacije te poruke ako je ona prevelika. Svakom dijelu poruke dobivene iz četvrtog sloja nadodajemo zaglavlje iz trećeg sloja. Sljedeća postaja je drugi sloj. U drugom sloju naša poruka dobiva još jedno zaglavlje, ali također dodajemo neke informacije na kraj poruke. Zadnja postaja na računalu pošiljatelja jest prvi sloj koji poruku stavlja na fizički medij. Kada poruka stigne do primaoca, ona kreće od prvog sloja i kreće se prema višim slojevima. Zaglavlja se procesuiraju u svakom sloju i poruka nastavlja prema gornjim slojevima bez njih.

2.1 OSI model

OSI model (eng. Open Systems Interconnection) razvijen je od strane Međunarodne organizacije za normizaciju. Cilj je bio uspostaviti standard za protokole u pojedinim slojevima mreže. Kada opisujemo arhitekturu neke mreže, to obavezno podrazumijeva i stog protokola te sučelja pojedinih slojeva. Budući da OSI model ne navodi eksplicitno koja sučelja treba imati koji sloj te koje protokole treba koristiti, OSI model nije arhitektura mreže. Ovaj model ima sedam slojeva. Ukratko ćemo proći kroz svaki od slojeva, a u sljedećem ćemo se poglavlju detaljno posvetiti četvrtom sloju, odnosno transportnom sloju. OSI model opisuje za što bi svaki od sedam slojeva trebao biti zadužen.

Prvi sloj odozdo jest fizički sloj. Ovdje nas zanima na koji način ćemo prenositi bitove s jednog uređaja na drugi putem određenog fizičkog medija. Ono što nam treba biti cilj je da kada jedan uređaj pošalje bit vrijednosti 1 (odnosno 0), uređaj koji prima podatke treba

dobiti bit vrijednosti 1 (odnosno 0). Prilikom dizajniranja ovog sloja trebamo odrediti koja vrsta električnog signala će predstavljati bit vrijednosti 1, a koja bit vrijednosti 0. Također, potrebno je odrediti koliko dugo svaki bit traje, može li komunikacija istovremeno teći u oba smjera, kako uspostavljamo i prekidamo vezu između pošiljaoca i primaoca i slično.

Sljedeći je podatkovni sloj. Podatkovni sloj prima paket iz gornjeg sloja (mrežni sloj) i dijeli ga na više podatkovnih okvira (eng. data frames). Zatim, svaki se od podatkovnih okvira prosljeđuje fizičkom sloju. Podatkovni sloj na strani primaoca mora poslati potvrdu za primljene podatkovne okvire. Na ovaj način osiguravamo ispravnost i cjelovitost primljenih podataka. Ukoliko na primjer dođe do gubitka određenog podatkovnog okvira, pošiljatelj može ponovno poslati taj podatkovni okvir. Na ovaj način se štitimo od eventualnih grešaka koje se mogu dogoditi prilikom korištenja fizičkog medija. U ovom se sloju također bavimo sinkronizacijom pošiljatelja i primatelja, tj. brinemo da pošiljatelj šalje podatkovne okvire onom brzinom kojom ih primatelj može procesuirati.

Sljedeći je, kako smo već i spomenuli, mrežni sloj. U ovom sloju bavimo se problemom usmjeravanja paketa od izvora do odredišta. Rute kojima se šalju paketi mogu biti dobivene na osnovu statičkih tablica koje se rijetko mijenjaju. Također, ruta može biti određena prilikom uspostave veze ili čak prilikom slanja svakog pojedinog paketa. Uska grla unutar mreže mogu se stvoriti ukoliko mnogo korisnika u isto vrijeme šalju podatke putem iste podmreže (eng. subnet). Još jedan zadatak ovog sloja, uz pomoć "viših" slojeva, jest kontrola zakrčenja (eng. congestion). Najjednostavnije rješenje je često i najbolje rješenje. Tako je i prilikom prevelikog zakrčenja mreže rješenje vrlo jednostavno, potrebno je smanjiti opterećenje mreže na način da se smanji frekvencija ubacivanja novih paketa u mrežu. Paket koji putuje od izvora do odredišta može prolaziti kroz različite mreže. Još jedan zadatak mrežnog sloja jest da riješi potencijalne probleme koji mogu nastati zbog različitosti tih mreža (npr. različito adresiranje ili različita najveća dopuštena veličina paketa).

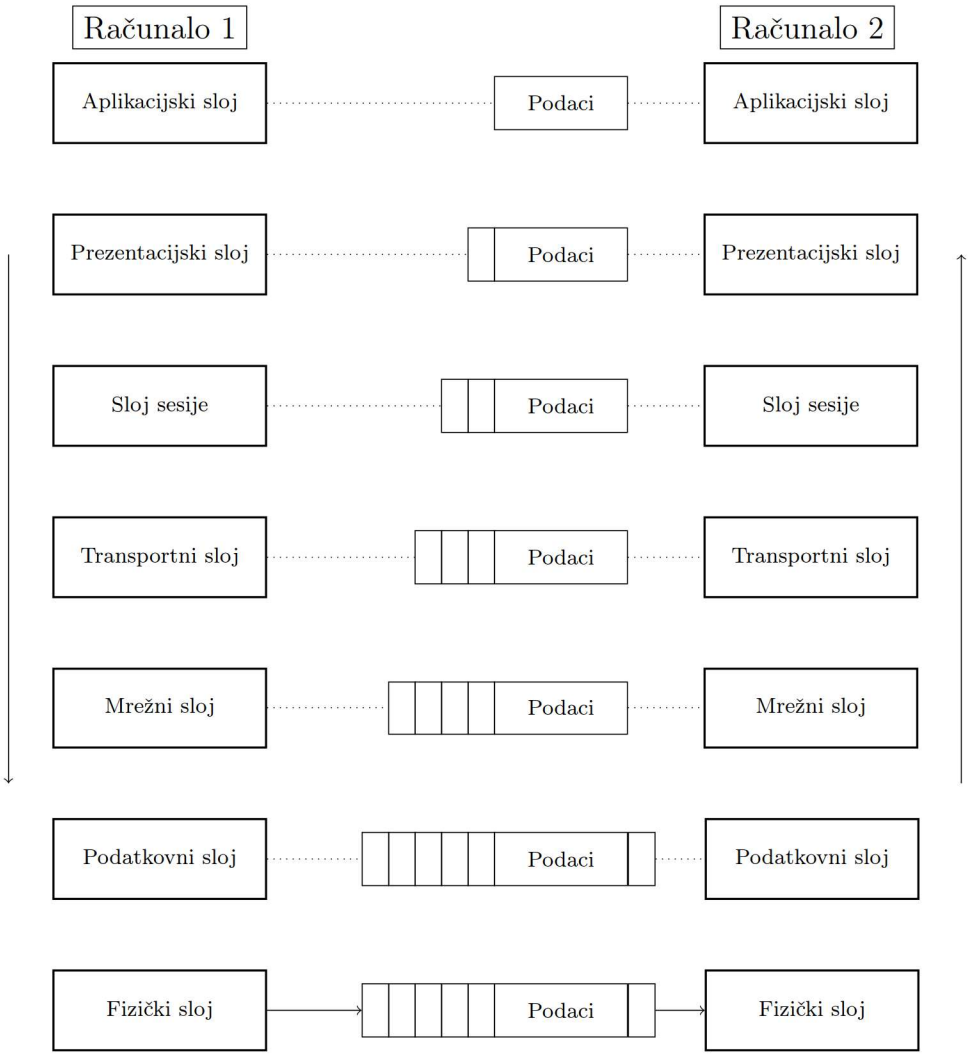
Sljedeći sloj na redu jest transportni sloj. Transportni sloj prima podatke od "viših" slojeva, dijeli ih na manje dijelove i prosljeđuje ih mrežnom sloju. Također se brini o tome da podaci stignu do odredišta i to bez greške. Ovaj sloj je, za razliku od "nižih" slojeva prisutan samo na izvorišnom i odredišnom računalu. Prva tri sloja su prisutna i u ruterima na putu između izvora i odredišta. Transportni sloj pošiljatelja komunicira (virtualno) s transportnim slojem primaoca i nije mu važno koji protokoli ili koji hardver se nalazi na putu do odredišta. Transportni sloj može pružati "gornjim" slojevima spomenutu uslugu, tj. osiguran prijenos poruka u pravilnom redoslijedu bez grešaka (tj. sa zanemarivom stopom pogreške). Usluga koju transportni sloj također može pružati "gornjim" slojevima jest transport izoliranih poruka bez garancije o redoslijedu pristizanja poruka na odredištu. Vrsta usluge se dogovara prilikom uspostave veze.

Slijedi sloj sesije. Ovaj sloj dopušta korisnicima na različitim računalima uspostavu sesija. Na ovaj način računala pamte čiji je red za slati poruku, koje operacije se trenutno smiju izvršavati. Također, u slučaju prekida veze, lagano je nastaviti razgovor od zadnje primljene poruke prije prekida.

Predzadnji sloj je prezentacijski sloj. Ovaj sloj je zadužen za sintaksu i semantiku informacija koje će biti dostavljene primaocu. Budući da računala mogu imati različite načine prikaza i spremanja različitih vrsta podataka, potreban je jedinstven način za zapis podataka koje želimo poslati putem mreže. U ovom sloju definiramo kodiranje i dekodiranje struktura podataka koje primamo iz zadnjeg sloja. Jedan primjer jest JSON.

Posljednji sloj je aplikacijski sloj. Ovaj sloj implementiran je od strane mrežne aplikacije. Aplikacija proizvodi podatke koje je potrebno prenijeti do udaljenog računala putem računalne mreže. Ovaj sloj je zadužen za stvaranje i prikaz podataka koji se šalju preko mreže.

Postoje mnogi protokoli aplikacijskog sloja, npr. HTTP, FTP, DNS.



Slika 2: OSI referentni model

Na slici 2 vidimo tok podataka od aplikacijskog sloja na računalu 1 do aplikacijskog sloja na računalu 2 (Aplikacijski sloj(R1) - Prezentacijski sloj (R1) - Sloj sesije (R1) - Transportni sloj (R1) - Mrežni sloj (R1) - Podatkovni sloj (R1) - Fizički sloj (R1) - Fizički medij - Fizički sloj (R2) - Podatkovni sloj (R2) - Mrežni sloj (R2) - Transportni sloj (R2) - Sloj sesije (R2) - Prezentacijski sloj (R2) - Aplikacijski sloj (R2)). Između aplikacijskog sloja i fizičkog sloja odvija se određena manipulacija inicijalnih podataka, dodaju se kontrolna zaglavlja, poruka se enkodira, segmentira i slično. Podaci se tada stavljaju na fizički medij i nakon niza usputnih čvorova stižu do odredišnog računala. Kako se podaci "penju" kroz slojeve tako im se skidaju i procesuiraju određena zaglavlja. Do aplikacijskog sloja primaoca stiže inicijalna poruka stvorena na računalu pošiljatelja.

2.2 TCP/IP model

Protokoli koji se vežu uz OSI model više nisu u upotrebi, iako je sam model i dalje relevantan. Međunarodna organizacija za normizaciju je objavila standarde za sve slojeve OSI modela, iako oni nisu dio samog referentnog modela. Dio OSI modela je u širokoj primjeni, dok su protokoli vezani uz model već neko vrijeme van upotrebe. Kod TCP/IP modela imamo suprotnu situaciju. Sam model nije od prevelike koristi, ali protokoli opisani u ovom modelu su u velikoj primjeni.

TCP/IP model nazvan je po svoja dva primarna protokola. Jedan od glavnih ciljeva ovoga modela bio je povezivanje više različitih mreža na način na koji krajnjem korisniku ne daje do znanja da se radi o različitim arhitekturama mreže. Drugi bitan cilj ovog modela jest da bude robustan. Američko ministarstvo obrane bojalo se da bi Sovjetski Savez mogao onesposobiti rutere i čvorišta na kojima se spajaju različite mreže. Želja im je bila da mreža može preživjeti gubitak određenog hardvera bez prekida protoka podataka. Također, mreža se planirala koristiti u razne svrhe, uključujući prijenos datoteka te prijenos govora u stvarnom vremenu. Bilo je bitno osmisliti protokole koji će biti u određenom smislu fleksibilni.

TCP/IP referentni model sastoji se od četiri sloja. Prvi i najniži sloj jest podatkovni sloj. Funkcija ovog sloja jest prijenos paketa između mrežnih slojeva (drugi sloj u TCP/IP modelu) uređaja koji se nalaze u istoj lokalnoj mreži.

Drugi sloj je mrežni sloj i otprilike se podudara s mrežnim slojem iz OSI referentnog modela. Ovaj sloj brine o putovanju paketa kroz mrežu. Paketi mogu doći u drugom redoslijedu od onoga kojim su poslani. Mrežni sloj ne brine o tom poretku već je to zadatak "viših" slojeva u modelu. Zadatak mrežnog sloja jest omogućiti korisniku stavljanje paketa u mrežu i određivanje putanje do odredišnog uređaja. Službeni protokol ovog sloja je IP (eng. Internet Protocol). Ovaj protokol opisuje format u kojem paketi trebaju biti kako bi bili uspješno dostavljeni do odredišta. Također, imamo pomoćni protokol ICMP (eng. Internet Control Message Protocol) koji služi za dojavu grešaka i mrežnu dijagnostiku.

Sljedeći sloj je transportni sloj. Kao i kod OSI modela, transportni sloj na izvoru i odredištu su u direktnoj komunikaciji, tj. zaglavlje koje je stvoreno i dodano na izvorišnom uređaju analizira se prvi puta tek na odredišnom uređaju i to u transportnom sloju. TCP/IP definira dva protokola za transportni sloj, TCP i UDP. TCP (eng. Transmission Control Protocol) je pouzdani protokol koji stvara virtualnu vezu između izvora i odredišta. Ovaj protokol omogućuje prijenos niza bajtova do odredišta bez grešaka ili gubljenja bajtova. Dolazni podaci iz "višeg" sloja dijele se u segmente određene veličine, dodaju im se zaglavlja i prosljeđuju se mrežnom sloju. Na odredištu segmenti se povezuju u originalnu poruku i poruka se prosljeđuje "gornjem" sloju. Ovaj protokol također vodi računa o brzini slanja segmenata i na taj se način brine da ne preplavi primaoca s više segmenata nego što on može procesuirati. Drugi protokol koji je napravljen za ovaj sloj jest UDP (eng. User Datagram Protocol). Ovaj protokol je, za razliku od TCP protokola, nepouzdan te ne stvara virtualnu vezu između izvora i odredišta. Također, UDP ne brine o segmentiranju niti o brzini slanja poruke. Ovaj protokol je za aplikacije koje žele same brinuti o spomenutim funkcionalnostima. UDP je često korišten za jednostavne klijentsko-poslužiteljske aplikacije te aplikacije kojima je bitnija brzina od pouzdanosti.

Zadnji sloj TCP/IP referentnog modela jest aplikacijski sloj. TCP/IP nema sloj sesije niti prezentacijski sloj već je njihova funkcionalnost, ukoliko je potrebna, dodana u aplikacijski sloj. Dakle, ovaj sloj obuhvaća "gornja" tri sloja OSI referentnog modela.

3 Transportni sloj

Transportni sloj nalazi se "iznad" mrežnog sloja. On nadograđuje mrežni sloj u određenom smislu i time omogućava prijenos podataka s programa na izvorišnom uređaju do točno određenog programa na odredišnom uređaju. Prilikom izvršavanja te usluge, transportni sloj brine da su svi podaci točno stigli do odredišta.

3.1 Usluge transportnog sloja

Korisnik transportnog sloja najčešće je proces u aplikacijskom sloju. Transportni sloj koristi usluge mrežnog sloja kako bi pružio usluge slojevima iznad sebe. Program ili hardver zadužen za obavljanje zadataka u transportnom sloju naziva se transportni entitet. Ovaj se entitet najčešće nalazi unutar jezgre operacijskog sustava ili u određenom programskom paketu koji je vezan uz mrežnu aplikaciju.

Korisnici mreže najčešće nemaju gotovo nikakvu kontrolu nad mrežom koju koriste. Ukoliko mreža često gubi pakete ili ukoliko određeni ruter prestane raditi tada nailazimo na problem ukoliko nemamo dobro definiran i robustan protokol u transportnom sloju. Transportni entitet može otkriti gubitak ili oštećenje određenih podataka i po potrebi ponovno poslati te podatke. Također, ukoliko usred prijenosa podataka dođe do prekida mreže, transportni entitet može biti programiran da prilikom nove uspostave veze provjeri s transportnim entitetom na odredištu koje podatke je zadnje primio. Na ovaj način prijenos podataka može nastaviti tamo gdje je stao i ne moramo trošiti mrežne resurse na ponovno slanje dijela podataka. Transportni sloj nam omogućava postojanje (prividno) pouzdane mreže.

Budući da aplikacijski sloj koristi samo sučelje transportnog sloja, a ne i sučelje mrežnog sloja, aplikacija može raditi na različitim vrstama mreža. Jedino što je potrebno jest zamijeniti jedan mrežni sloj s drugim koji će imati jednako sučelje prema transportnom sloju kao i prvi mrežni sloj.
Sučelje transportnog sloja prema aplikacijskom sloju treba biti jednostavno za korištenje s dobro definiranim i dokumentiranim funkcionalnostima. Razlog tome je što će prilikom izrade većine mrežnih aplikacija upravo to sučelje biti korišteno za slanje podataka putem mreže. Većina aplikacija neće graditi svoj transportni entitet već će koristiti postojeći hardver i softver. Iz tog razloga, većina programera neće direktno koristiti sučelje mrežnog sloja, ali sučelje transportnog sloja je druga priča. Najčešće operacije sučelja transportnog sloja su operacije za uspostavu i prekid veze, te za slanje i primanje segmenata podataka.

Pretpostavimo da imamo jednog poslužitelja i jednog klijenta koji žele međusobno razmjenjivati podatke. Slijed te komunikacije mogao bi teći na sljedeći način. Poslužitelj izvršava operaciju *Slušaj* koja blokira izvršavanje poslužiteljskog koda sve dok se ne pojavi klijent koji želi komunicirati s poslužiteljem. Klijent tada izvršava operaciju *Poveži*. Ova operacija blokira izvršavanje klijentskog koda i šalje inicijalni segment poslužitelju. Taj segment će biti enkapsuliran u paket mrežnog protokola koji će zatim biti enkapsuliran u jedan ili više podatkovnih okvira (u podatkovnom sloju) koji se šalju preko fizičkog medija. Kada podatkovni okvir stigne do poslužitelja, podatkovni sloj poslužitelja analizira zaglavlje podatkovnog okvira i ukoliko je taj podatkovni okvir namijenjen za računalo na kojemu se nalazi poslužitelj, podaci iz podatkovnog okvira (bez zaglavlja) prosljeđuju se mrežnom sloju. Mrežni sloj također procesuira zaglavlje dobivenog paketa i prosljeđuje ostatak tog paketa transportnom entitetu. Sada je inicijalni segment za povezivanje stigao do transportnog sloja poslužitelja. Transportni entitet prvo provjeri je li poslužitelj pokrenuo operaciju *Slušaj*. Ako je, tada poslužitelj biva odblokiran i šalje klijentu segment kojim označava da prihvaća uspostavu

veze. Nakon što taj segment stigne do klijenta, veza je uspostavljena i tada podaci mogu teći u oba smjera. Poslužitelj i klijent koriste operacije *Pošalji* i *Primi* za slanje i primanje podataka. Ukoliko je red na klijentu da primi podatke tada on izvršava blokirajuću operaciju *Primi* i čeka da poslužitelj izvrši operaciju *Pošalji*. Tada klijent može procesuirati dobivenu poruku i po potrebi poslati odgovor. Iako to nije vidljivo korisniku transportnog sloja, svaka poruka u ovom primjeru će biti i potvrđena od strane primatelja te poruke. Transportni entiteti također sami brinu o eventualnom gubitku segmenata i o redoslijedu segmenata. Krajnji korisnik ne mora brinuti o tim stvarima. Korisniku se čini kao da postoji cijev koja povezuje izvor i odredište i kada se podaci ubace na jednoj strani cijevi oni sigurno izlaze na odredištu na drugoj strani cijevi u pravilnom redoslijedu. Pri završetku komunikacije uspostavljena veza mora biti prekinuta kako bi se oslobodili računalni resursi. Postoje asimetrični i simetrični način prekida veze. Kod asimetričnog načina prekida veze, bilo koja strana može izvršiti operaciju *Prekini* i veza će tada biti zatvorena. Kod simetričnog načina svaka strana posebno mora zatvoriti vezu. Ukoliko klijent izvrši operaciju *Prekini* tada on nema više podataka za slati, ali i dalje može primati podatke. Kada i druga strana, u ovom slučaju poslužitelj, izvrši operaciju *Prekini*, veza će biti zatvorena.

3.2 Elementi transportnih protokola

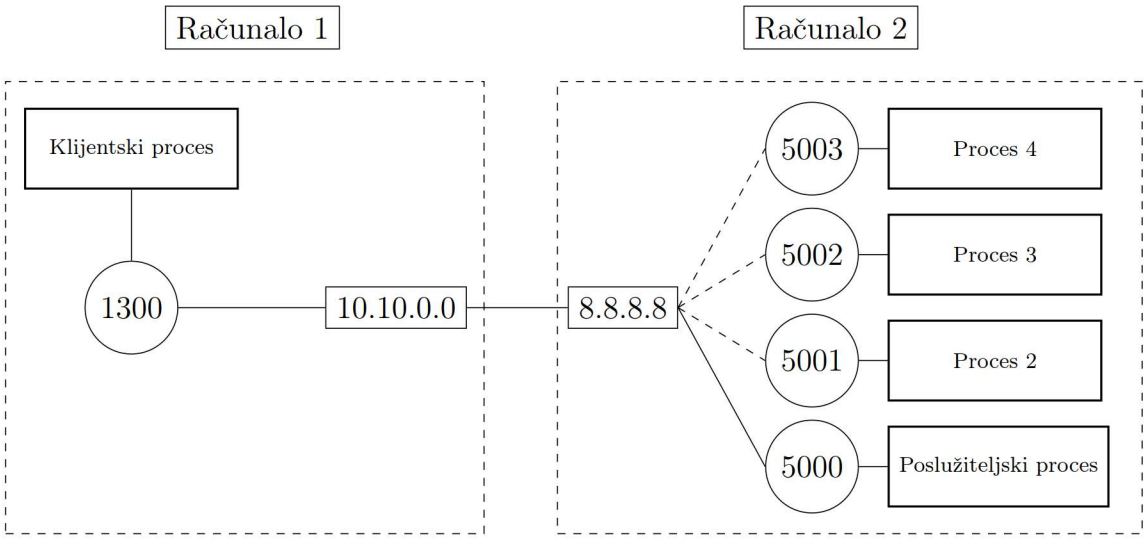
U određenim aspektima, protokoli transportnog sloja podsjećaju na protokole podatkovnog sloja. Oba se sloja bave otkrivanjem i ispravljanjem grešaka. Također, oba sloja segmentiraju podatke dobivene iz "gornjeg" sloja na manje segmente i šalju ih dalje, pri tom imajući na umu brzinu procesuiranja primaoca. Razlika nastaje kada pogledamo okolinu u kojoj ova dva sloja djeluju. U podatkovnom sloju, dva rutera direktno su spojena fizičkim medijem, dok transportne slojeve komunicirajućih uređaja dijeli cijela mreža. U transportnom sloju moramo na određeni način reći s kojim uređajem želimo komunicirati. Za rutere koji su spojeni npr. optičkim kabelom to nije potrebno kada govorimo o komunikaciji u podatkovnom sloju. Uspostava i prekid veze također su kompliciraniji u transportnom sloju. Također, treba imati na umu da se paketi koje pošaljemo putem mreže mogu zagubiti. Isto tako, moguće je da paketi putuju različitim rutama unutar mreže. Rezultat toga je da paketi mogu doći do odredišta u različitom poretku od onog u kojem su poslani.

3.2.1 Adresiranje

Najprije ćemo se baviti načinom odabira procesa na udaljenom uređaju s kojim želimo komunicirati. Računalo unutar mreže najčešće ima samo jednu mrežnu pristupnu točku (IP adresa ukoliko se koristi IP protokol). Usprkos tome, na jednom računalu možemo imati više procesa koji trebaju moći istovremeno komunicirati putem mreže. Iz tog razloga, transportni entitet koristi različite priključne točke (eng. ports) za svaki od tih procesa kako bi paket koji pristigne na mrežnu pristupnu točku proslijedio točno određenom procesu.

Pretpostavimo da želimo komunicirati s poslužiteljem koji se nalazi na udaljenom računalu. Neka je poslužitelj priključen na priključnu točku 5000. Neka na udaljenom računalu postoji i nekoliko drugih procesa koji su priključeni na priključne točke 5001, 5002 i 5003. Sva četiri procesa imat će jednaku IP adresu (npr. 8.8.8.8). Ukoliko želimo komunicirati s poslužiteljom, izvršit ćemo operaciju *Poveži* te ćemo navesti njegovu IP adresu (8.8.8.8) i pristupnu točku 5000. Prilikom izvršavanja naredbe *Poveži*, klijentskom procesu će također biti dodijeljena priključna točka (npr. 1300). Paket stiže na mrežnu pristupnu točku 8.8.8.8 te je proslijeđen transportnom entitetu koji tada analizira dobiveni segment. Provjerom

zaglavlja, transportni entitet zna da je dobiveni segment namijenjen za proces na priključnoj točki 5000. Kada poslužitelj šalje odgovor klijentu, on ga šalje na klijentsku IP adresu (koja se nalazi u zaglavlju paketa koji je stigao na mrežnu pristupnu točku poslužitelja, npr. 10.10.0.0), te navodi priključnu točku 1300 (navedenu u zaglavlju segmenta koji je pristigao do transportnog entiteta) kao adresu odredišnog procesa.



Slika 3: Priključne točke

Imamo jedno nerazjašnjeno pitanje. Kako klijentski proces zna na kojoj se priključnoj točki nalazi poslužiteljski proces? Jedna mogućnost je ta da se poslužiteljski proces uvijek prijavljuje na priključnu točku 5000 i da je to već opće poznato svim klijentima unutar mreže. Ovaj slučaj je u redu kada se radi o manjem broju procesa koji se nikada ne mijenjaju. Ukoliko imamo procese koji nemaju dobro poznate priključne točke i koji postoje samo određeno vrijeme, potreban nam je drugi način za poznavanje priključnih točaka. Jedna mogućnost jest da imamo poseban proces koji se naziva *Portmapper*. Kako bismo pronašli priključnu točku određenog procesa, potrebno je povezati se na *Portmapper* i poslati upit za točno određeni proces (najčešće ASCII vrijednost poput "CoolMailServer"). *Portmapper* tada šalje odgovor u kojemu se nalazi broj priključne točke traženog procesa. Kako bi ovaj način bio funkcionalan, novi procesi moraju se registrirati kod procesa *Portmapper* tako što će poslati svoje ime i priključnu točku. Također, broj priključne točke procesa *Portmapper* mora biti poznat svim uređajima u mreži.

3.2.2 Uspostava veze

Već smo spomenuli uspostavu veze, ali nismo spominjali eventualne komplikacije na koje možemo naići prilikom uspostave iste. Zato što mreža nije u potpunosti pouzdana, nije dovoljno da jedna strana zatraži otvaranje veze i jednostavno sačeka da druga strana prihvati vezu. Problemi nastaju prilikom gubitka, kašnjenja i dupliciranja poslanih paketa.

Ukoliko ne bismo imali dobro definirane protokole koji su otporni na situacije koje nastaju kada je mreža pod velikim naporom imali bismo mnogo problema. Zamislimo da korisnik pošalje paket za uspostavu veze s bankom i pošalje poruku u kojoj traži da se određena svota novca prebaci na drugi račun. Neka paketi zalutaju unutar mreže i odaberu najdužu moguću rutu. Pošiljatelj će u jednom trenutku, kada istekne određeno vrijeme, ponovno poslati pakete. Neka ovaj puta paketi odaberu najkraću moguću rutu i stignu do odredišta. Novac

je prebačen na odgovarajući račun. Pošiljatelj zatvara vezu. Sada napokon do odredišta stižu paketi za koje je pošiljatelj mislio da su izgubljeni. Banka ne zna da su ovo stari paketi i ponovno uspostavlja vezu i prebacuje novac s korisnikovog računa. Ovo je jedan fiktivan primjer, ali ilustrira da protokoli moraju dobro funkcionirati u svim slučajevima. U ovom primjeru problem je bio taj što je primatelj mislio da su zakašnjeli dupli paketi zapravo novi paketi. Ne možemo utjecati na to hoće li mreža izgubiti pakete ili hoće li paketi kasniti i biti duplicirani, ali problem možemo izbjeći pravilnim protokolom u transportnom i mrežnom sloju.

Potrebna nam je strategija koja će onemogućiti paketima da neodređeno vrijeme kruže mrežom. Najučinkovitije rješenje je da svakom paketu kojeg dodajemo u mrežu dodamo određeno vrijeme života. To vrijeme života može biti određeno brojem "skokova", odnosno dozvoljenim brojem prosljeđivanja paketa (od jednog rutera do drugog). Ovaj princip implementiran je u IP protokolu mrežnog sloja.

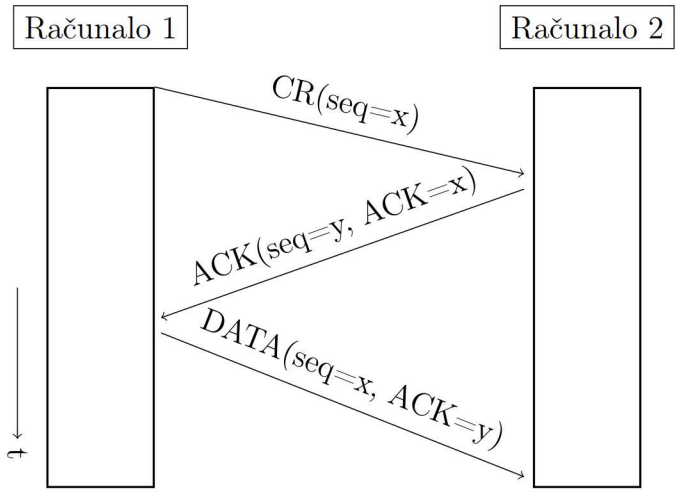
Osim što moramo voditi računa o tome da nema starih paketa u mreži, moramo voditi i računa o tome da nema starih potvrda dospjelih paketa u mreži. Neka je T višekratnik maksimalnog vremena života paketa unutar mreže. Ukoliko čekamo T vremena nakon što smo poslali paket putem mreže, možemo biti sigurni da se ni paket ni potvrda o primitku paketa više ne nalaze u mreži.

Sada kada imamo osiguranje da paketi neće neodređeno vrijeme kružiti mrežom i onda se pojaviti u krivom trenutku, možemo razviti strategiju za odbacivanje duplikata segmenata. Segmente ćemo označavati slijednim brojevima (eng. sequence numbers) koji se neće ponavljati idućih T sekundi. Za raspon slijednih brojeva moramo znati T i brzinu slanja paketa u mreži. Samo jedan segment s istim slijednim brojem može biti unutar mreže u određenom trenutku, odnosno ako se pojavi još jedan segment s istim slijednim brojem tada znamo da su ta dva segmenta jednaka. Prateći slijedne brojeve, na odredištu možemo odbaciti potencijalne duplikate. Na ovaj način nemoguće je da u mreži postoje dva različita segmenta s istim slijednim brojem.

Uz mogućnost razlikovanja starih od novih segmenata, sada možemo razmišljati o uspostavi veze. Algoritam koji koristimo naziva se trostruko rukovanje (eng. 3-Way Handshake) i prikazan je na slici 4. Razlog iz kojega nam je potreban ovaj algoritam jest što na odredištu ne pamtimo slijedne brojeve iz prijašnjih veza. Iz tog razloga nećemo znati je li zahtjev za uspostavljanje veze novi ili možda neki stari zagubljeni segment. Prvi korak jest da jedna strana odabere slijedni broj x i pošalje segment sa zahtjevom za uspostavu veze (eng. connection request) s tim slijednim brojem. Druga strana, nakon primitka segmenta za uspostavu veze, šalje potvrdni segment. Potvrdni segment sadrži inicijalni slijedni broj primaoca, y, te potvrdu slijednog broja pošiljatelja, x. Inicijalni pošiljatelj, nakon primitka potvrdnog segmenta, šalje svoj potvrdni segment u kojemu potvrđuje slijedni broj y, ali uz taj segment, osim same potvrde, može odmah slati i podatke. Na ovaj način, eventualni stari paketi neće moći slučajno uspostaviti vezu jer s obje strane moramo potvrditi slijedne brojeve, a oni se mijenjaju prilikom svake nove veze.

3.2.3 Zatvaranje veze

Sada ćemo pričati o zatvaranju veze koje je jednostavnije od uspostave veze, ali i dalje ima situacija (zbog nepouzdane mreže) na koje treba obratiti pažnju. Asimetrično zatvaranje veze (dovoljno je da jedna strana odluči zatvoriti vezu) može dovesti do gubitka podataka. Ukoliko prvi uređaj odluči zatvoriti vezu, a drugi uređaj je upravo stavio jedan paket u mrežu, taj paket nikada neće stići do prvog uređaja. Iz tog razloga, najčešće se koristi simetrično zatvaranje veze.



Slika 4: Uspostava veze

Prilikom simetričnog zatvaranja veze, svaki smjer se zatvara posebno, tj. dvosmjernu vezu možemo gledati kao dvije jednosmjerne veze. Ukoliko prvi uređaj odluči poslati segment za prekid veze, on i dalje može primati segmente od drugog uređaja, ali ih ne može slati. Ovaj princip dobro radi kada svaka strana ima konačan broj segmenata koje treba poslati i kada znamo da smo uspješno poslali sve segmente. Ako je to slučaj, jedan uređaj može jednostavno reći "Poslao sam sve segmente. Jesi li i ti poslao sve segmente?". Ako drugi uređaj odgovori "I ja sam poslao sve segmente.", tada veza može biti zatvorena. Ali ovaj protokol neće uvijek raditi kako treba. Da bismo ilustrirali problem, opisat ćemo poznati problem "problem dvije vojske". Neka se bijela vojska nalazi u dolini između dva brežuljka. Na oba brežuljka se nalazi plava vojska. Bijela vojska je brojnija od obje plave vojske posebno, ali zajedno, plave vojske su brojnije od bijele vojske. Ako jedna plava vojska napadne bijelu vojsku izgubit će bitku, ali ako obje plave vojske zajedno napadnu bijelu vojsku onda će plave vojske dobiti bitku. Ukoliko će plave vojske zajedno napasti, bitno je da to učine u isto vrijeme, tj. moraju sinkronizirati svoje napade. Jedino sredstvo komunikacije jest da pošalju glasnika na drugi brežuljak, ali glasnik mora proći kroz dolinu u kojoj se nalazi bijela vojska pa postoji mogućnost da bude zarobljen od strane neprijatelja (izgubljen paket zbog nepouzdane mreže). Pretpostavimo da zapovjednik prve plave vojske pošalje glasnika s porukom "Predlažem da napadnemo sutra u zoru, slažeš li se?". Neka glasnik uspješno prenese poruku zapovjedniku druge plave vojske. Zapovjednik druge plave vojske šalje odgovor koji uspješno stiže do zapovjednika prve plave vojske, ali imamo jedan problem. Zapovjednik druge plave vojske ne zna je li njegov odgovor uspješno stigao do prve plave vojske i iz tog razloga druga plava vojska neće napasti bijelu vojsku sutra u zoru. Ako na ovaj protokol primijenimo princip trostrukog rukovanja, imamo sljedeću situaciju. Zapovjednik prve plave vojske mora potvrditi da je primio odgovor druge plave vojske. Ukoliko potvrda uspješno stigne do druge plave vojske, sada će zapovjednik prve plave vojske oklijevati. Razlog je taj što zapovjednik prve plave vojske ne zna je li njegova potvrda uspješno stigla na drugu stranu. Ovako se možemo vrtjeti u krug, sve dok ne shvatimo da ne postoji protokol koji će sa stopostotnom sigurnošću raditi. Ukoliko plave vojske zamijenimo s dva uređaja koja komuniciraju, a napad na bijelu vojsku zamijenimo sa zatvaranjem veze, vidimo da ukoliko nijedna strana ne zatvori vezu sve dok nije sigurna da će druga strana zatvoriti vezu, veza nikada neće biti zatvorena.

Ukoliko ne zahtijevamo da se obje strane usuglase oko zatvaranja veze te ako uključimo korisnika transportnog sloja u tu odluku, onda imamo lakši problem za riješiti. Koristit ćemo princip trostrukog rukovanja. Neka prvi uređaj pošalje segment koji sadrži zahtjev za zatvaranje veze i zatim pokrene odbrojavanje. Kada taj segment stigne do drugog uređaja, on pošalje svoj segment sa zahtjevom za zatvaranje veze. Osim toga, drugi će uređaj pokrenuti odbrojavanje, za slučaj da se njegov segment izgubi. Kada zahtjev za zatvaranje veze stigne do prvog uređaja, on šalje potvrdni segment prema drugom uređaju i zatvara vezu sa svoje strane. Kada potvrdni segment stigne do drugog uređaja, on također zatvara vezu sa svoje strane. Ako je potvrdni segment izgubljen, prilikom isteka mjerača, drugi uređaj zatvara vezu sa svoje strane. Ako se dogodi da je drugi segment sa zahtjevom za zatvaranje veze (onaj koji šalje drugi uređaj) izgubljen, tada prilikom isteka mjerača na prvom uređaju, proces se iznova započinje. Ovaj proces možemo ponavljati određenih N puta te ako nijedan od tih pokušaja nije uspješan, prvi uređaj će nakon N pokušaja zatvoriti vezu sa svoje strane. Prilikom isteka mjerača na drugom uređaju, veza će biti zatvorena s obje strane. Ovaj princip neće raditi u jednom slučaju, a to je kada je prvi zahtjev za zatvaranjem veze izgubljen, kao i sljedećih N ponovnih pokušaja. Prvi uređaj će zatvoriti vezu sa svoje strane, dok drugi uređaj neće uopće znati da je prvi uređaj pokušao zatvoriti vezu. Na ovaj način ostajemo s poluotvorenom vezom. Rješenje može biti da svaka strana automatski zatvori vezu ukoliko je druga strana neaktivna određeno vrijeme. Sada moramo brinuti o nekoliko novih mjerača vremena. Očito je da je zatvaranje veze kompliciraniji problem nego što bismo na početku pomislili. Razlog je naravno nepouzdanost mreže.

3.2.4 Kontrola toka

Nakon što znamo kako uspostaviti i zatvoriti vezu, sada je na redu da proučimo kako se veza koristi. Kontrola toka služi za reguliranje brzine slanja paketa kako ne bismo preopteretili primaoca. Također, ne želimo slati manjom brzinom od optimalne jer na taj način ne iskorištavamo sve dostupne mrežne resurse.

Transportni protokoli većinom koriste tehniku kliznog prozora (eng. sliding window). Ova se tehnika koristi kada želimo slati više od jednog segmenta prije nego što dobijemo potvrdu o primitku prijašnjeg segmenta. Neka je veličina prozora jednaka pet. To znači da pošiljatelj može poslati pet segmenata prije nego što mora stati i sačekati potvrdu koja kaže da su segmenti uspješno primljeni. Potvrde za segmente očekujemo u onom poretku u kojemu smo ih poslali, tj. ako smo poslali segmente 1, 2 i 3 tim redom, tada očekujemo da ćemo prvo dobiti potvrdu za segment 1 pa za segment 2 i na kraju za segment 3. Ako dobijemo potvrdu koju očekujemo, prozor se "pomiče" za jedno mjesto i novi segment biva poslan. Ako npr. dobijemo potvrdu za segment 2, a još nismo dobili potvrdu za segment 1, tada se prozor ne pomiče. Postoji više varijacija tehnike kliznog prozora. Jedna je da kada određeno vrijeme ne primimo potvrdu o primitku sljedećeg po redu segmenta, da tada pošaljemo cijeli trenutni prozor ponovno (svih pet segmenata, bez obzira jesmo li za neke od njih već dobili potvrdu). Također, postoji varijacija u kojoj u slučaju da ne primimo potvrdu za sljedeći po redu segment, samo se taj segment, koji još nije potvrđen, ponovno šalje.

Druga stvar o kojoj treba brinuti je pohranjivanje primljenih segmenata u međuspremnike. Budući da jedan uređaj može imati istovremeno više otvorenih veza, potreban je međuspremnik (ili više međuspremnika) dovoljne veličine tako da mogu biti pohranjeni svi segmenti iz svih kliznih prozora iz svih otvorenih veza. Na strani pošiljatelja također imamo međuspremnik. Taj međuspremnik služi za čuvanje segmenata koji su već poslani, ali još nisu potvrđeni (za slučaj da ih treba ponovno poslati). Transportni entitet na odredištu,

ovisno o implementaciji, može na različite načine inicijalizirati međuspremnike. Ako na odredištu više nema mjesta u međuspremniku, segment se odbacuje. Ovo nije problem jer je pošiljatelj spreman ponovno poslati segmente koji nisu potvrđeni. Međuspremnici na odredištu mogu biti inicijalizirani dinamički ovisno o brzini pristizanja segmenata. Također, moguće je prilikom uspostave veze odmah inicijalizirati dovoljno međuspremnika za sve segmente u jednom kliznom prozoru i na taj način osigurati da uvijek imamo mjesta za pristigle segmente. Uređaj na odredištu može imati sve međuspremnike jednake veličine, međuspremnike različitih veličina ili npr. jedan veliki cirkularni međuspremnik. Ukoliko će svi segmenti biti slične veličine, ima smisla imati međuspremnike iste veličine i svaki segment spremati u jedan međuspremnik. Ako ćemo imati mnogo segmenata s različitim veličinama, međuspremnici jednake veličine će najvjerojatnije ili bespotrebno trošiti memorijske resurse (ako za veličinu međuspremnika odaberemo najveću moguću veličinu segmenta) ili će zakomplicirati implementaciju (ako veličina međuspremnika bude premala za većinu segmenata, jedan segment ćemo morati spremiti u više međuspremnika). Ako koristimo međuspremnike različitih veličina, nema bespotrebnog trošenja memorijskih resursa, ali implementacija postaje kompleksnija. Moguće je i imati jedan cirkularni međuspremnik u koji ćemo spremati sve pristigle segmente. Ovaj model je jednostavan, elegantan i ne ovisi o veličini segmenata. Jedina mana jest to što je memorija kvalitetno iskorištena jedino kada kroz uspostavljene veze dolazi velika količina podataka.

Veze se stalno uspostavljaju i zatvaraju i samim time potreba za međuspremnicima na odredištu stalno se mijenja. Transportni protokol bi trebao omogućiti primaocu način na koji može prenijeti informaciju pošiljatelju o broju dostupnih međuspremnika. Pošiljatelj bi na osnovu te informacije trebao povećati, smanjiti ili ostaviti nepromijenjenu veličinu kliznog prozora. Pošiljatelj bi mogao na početku komunikacije zatražiti određeni broj međuspremnika od primaoca. Zauzvrat, primalac će omogućiti onoliko međuspremnika koliko si ih trenutno može priuštiti i o tome će ga obavijestiti.

3.3 Zakrčenost mreže

Kada više transportnih entiteta na više različitih uređaja šalju previše paketa putem mreže u prekratkom vremenskom razdoblju, tada mreža postaje zakrčena. Rezultat zakrčene mreže su zakašnjeli i izgubljeni paketi. Kako bismo izbjegli ovu situaciju, potrebno je imati dobro dizajnirane mrežne i transportne slojeve. Zakrčenost se događa u ruterima i iz tog razloga je mrežni sloj prvo mjesto na kojemu primjećujemo zakrčenost. No, budući da je zakrčenost izazvana prebrzim slanjem paketa, za što je zadužen transportni sloj, jedino efikasno rješenje jest da transportni protokol u tom trenutku smanji brzinu kojom šalje pakete. U nastavku ćemo opisati koji je cilj algoritama za kontrolu zakrčenosti mreže te kako uređaji mogu kontrolirati brzinu slanja paketa putem mreže.

3.3.1 Željena mrežna propusnost

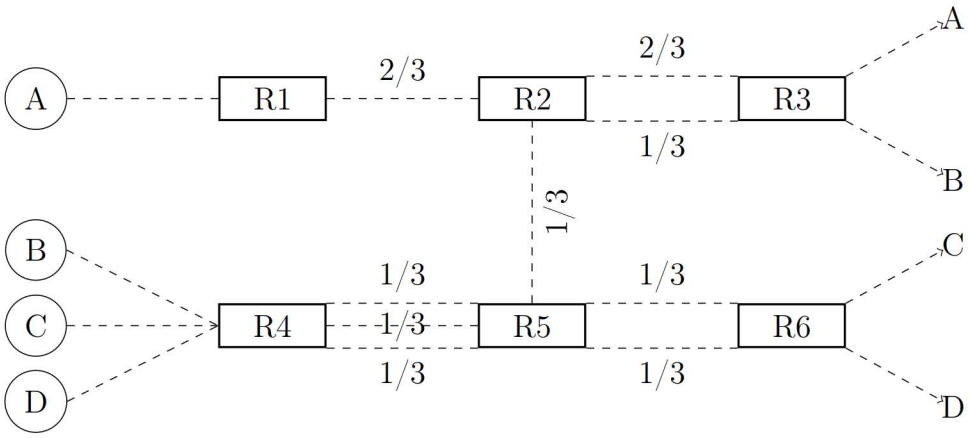
Algoritmi za kontrolu zakrčenosti ne rade samo na sprječavanju zakrčenosti već je potrebno i svakoj vezi omogućiti određenu mrežnu propusnost (eng. bandwidth). Alokacija mrežne propusnosti trebala bi biti poštena prema svim transportnim entitetima koji međusobno dijele mrežne resurse.

Dobra alokacija mrežne propusnosti koristi puni kapacitet mreže. Stopa korisnih paketa (bez kontrolnih ili ponovno poslanih paketa) koja stiže na odredište je funkcija koja ovisi o količini prometa unutar mreže. Prilikom inicijalnog povećanja prometa unutar mreže, stopa korisnih paketa također raste. Kako se količina prometa približava kapacitetu, povećanje

stope korisnih paketa se usporava. Razlog je što promet često dolazi u rafalima i ponekad se može iznenada nakupiti kod jednog rutera i preopteretiti ga. Kada je međuspremnik rutera pun, on odbacuje novopristigle pakete. Neki paketi mogu samo biti usporeni, ali ukoliko je transportni protokol loše dizajniran i pretpostavi da su ti paketi zapravo izgubljeni onda imamo problem. Transportni entitet će ponovno poslati pakete koji nisu zapravo izgubljeni nego su samo "upali u gužvu" negdje unutar mreže. Na taj se način bespotrebno povećava količina prometa unutar mreže i samim time stvaraju se novi zastoji što rezultira ponovnim slanjem paketa što dovodi do još većeg zastoja. Na ovaj način dolazimo do zakrčenosti koja ozbiljno smanjuje propusnost mreže te stopa korisnih paketa dodatno opada. Najbolje performanse postižu se kada povećavamo količinu mrežne propusnosti sve dok ne primijetimo značajnije kašnjenje paketa. Ovaj će se trenutak dogoditi prije nego što dostignemo puni kapacitet. Možemo pratiti omjer količine prometa i kašnjenja paketa. Ovaj omjer na početku raste zajedno s količinom prometa. Inicijalno kašnjenje je malo i unutar konstantnih granica (jer je uvijek potrebno određeno vrijeme da paket stigne do odredišta). Omjer postiže svoj maksimum prilikom povećanja vremena kašnjenja. Količina podataka pri najvećoj vrijednosti omjera predstavlja najefikasniju količinu podataka koju transportni entitet može slati putem mreže.

Svakom transportnom entitetu treba osigurati određeni dio ukupne mrežne propusnosti. Prva pomisao nam može biti da svatko dobije jednak udio na korištenje, ali imamo nekoliko stvari o kojima treba voditi brigu. Potrebno je odrediti što znači podijeliti mrežnu propusnost pošteno. Ako imamo N tokova koji od izvora do odredišta moraju prijeći samo preko jedne veze između dva rutera, onda možemo svakom toku dodijeliti $\frac{1}{N}$ mrežne propusnosti te veze. Što ako N tokova ima različite, ali preklapajuće mrežne putanje? Neka jedan tok prelazi preko tri mrežne veze, dok ostali tokovi prelaze samo preko jedne veze i neka se na toj vezi svi tokovi preklapaju. Mrežni tok koji koristi tri mrežne veze koristi više mrežnih resursa pa može imati smisla da tom toku osiguramo manje mrežne propusnosti nego ostalim tokovima. Potrebno je pronaći ravnotežu između poštenja i efikasnosti. Poštenje ćemo definirati na način koji ne ovisi o dužini putanje kojom tok putuje mrežom. Budući da mrežni tokovi putuju različitim putanjama i prelaze različite mrežne veze s različitim kapacitetima, postoji mogućnost da određeni tok prolazi kroz neko usko grlo kasnije u svojoj putanji. Tada će taj tok koristiti manje mrežne propusnosti nego ostali tokovi s kojima dijeli mrežnu vezu. Kada bismo tom toku pokušali dati više mrežne propusnosti, samo bismo usporili ostale tokove s kojima dijeli mrežnu vezu, dok se tok koji prolazi kroz usko grlo ne bi ubrzao. Koristimo definiciju max-min poštenja. Alokacija mrežne propusnosti je max-min poštena ako ne možemo povećati mrežnu propusnost nijednom toku bez da tu propusnost oduzmemo drugom toku.

U primjeru sa slike 5 imamo max-min poštenu alokaciju za četiri mrežna toka, A, B, C i D. Imamo šest različitih rutera kroz koje prolaze četiri različita mrežna toka. Svaka mrežna veza između dva rutera ima jednak kapacitet (1). Tokovi B, C i D se natječu za donju mrežnu vezu između rutera R4 i R5. Svaki će tok dobiti $\frac{1}{3}$ mrežne propusnosti na toj mrežnoj vezi. Tokovi A i B natječu se za mrežnu vezu između rutera R2 i R3. Kako B ima alokaciju od $\frac{1}{3}$, tok A dobiva preostale $\frac{2}{3}$. Sve ostale mrežne veze (R1-R2, R5-R6) imaju nepopunjene kapacitete, ali nijedan tok ne može dobiti više propusnosti bez da tu propusnost oduzmemo od nekog drugog toka. Max-min alokacija može biti izračunata ako imamo informacije o cijeloj mreži. Možemo zamisliti da svi tokovi kreću s propusnosti 0 i zatim im postepeno povećavamo propusnost. Kada neki tok naiđe na usko grlo, tada prestajemo povećavati propusnost za taj tok.



Slika 5: Max-min alokacija mrežne propusnosti

Algoritam za kontrolu zakrčenosti trebao bi brzo konvergirati prema optimalnoj alokaciji mrežne propusnosti. Kao što smo spomenuli, količina prometa nikada nije konstantna. Tijekom trajanja jedne veze, mnoge druge veze su stvorene i zatvorene. Algoritam treba cijelo vrijeme prilagođavati mrežnu propusnost za sve tokove.

3.3.2 Upravljanje brzinom slanja

Sada ćemo objasniti na koji način možemo upravljati brzinom slanja paketa tako da dobijemo željenu mrežnu propusnost. Brzina slanja ovisi o kontroli toka i o zakrčenosti mreže. Ako je međuspremnik na odredištu prepunjen, tada bez obzira na to što mreža može podnijeti veću brzinu slanja, pošiljatelj treba usporiti. Ako pošiljatelj ne uspori, doći će do gubitka paketa pa samim time (zbog ponovnog slanja izgubljenih paketa) i do većeg opterećenja u mreži. Ako imamo slučaj zakrčenosti mreže, odnosno popunjenosti kapaciteta određene mrežne veze, svi transportni entiteti koji koriste mrežnu vezu u pitanju moraju smanjiti brzinu kojom šalju pakete. U ovom slučaju, povećanje kapaciteta međuspremnika na odredištu ne igra nikakvu ulogu u povećavanju propusnosti mreže. Oba ova slučaja imaju jednaki rezultat, gubitak paketa. Ono što se razlikuje jest uzrok pa samim time trebamo i dva rješenja za navedene probleme. Budući da pošiljatelju ova dva slučaja izgledaju jednako (vidimo samo gubitak paketa), on će morati upotrijebiti oba rješenja kako bi uspješno riješio problem.

Već smo spomenuli kontrolu toka koja koristi klizni prozor promjenjive veličine. Ova metoda uspješno rješava prvi problem. Sada ćemo se pozabaviti rješenjem problema zakrčenosti mreže. Postoje protokoli s kojima ruteri mogu pošiljatelju javiti da postoji problem. Jedan od takvih protokola jest XCP (eXplicit Congestion Protocol). Ruter obavještava pošiljatelja kojom brzinom smije slati pakete. Također, imamo protokole koji pošiljatelja obavještavaju o zakrčenju mreže, ali ne specificiraju koliko treba usporiti. Neki protokoli implicitno zaključuju da je došlo do zakrčenja mreže. Jedan od takvih načina je mjerenje vremena kašnjenja paketa. Ukoliko je došlo do značajnog porasta u vremenu kašnjenja paketa, tada smo vrlo vjerojatno na pragu zakrčenosti. Ipak, najzastupljenija metoda signalizacije jest gubitak paketa. Ako imamo eksplicitni signal, kao na primjer XCP protokol koji pošiljatelju javlja kojom brzom treba slati pakete, tada jednostavno koristimo predloženu brzinu. No, u ostalim slučajevima nemamo egzaktnu brzinu kojom bismo trebali slati pakete već do nje dolazimo postepenim povećavanjem i smanjivanjem brzine.

AIMD (eng. Additive Increase Multiplicative Decrease) kontrolni zakon jest princip ko-

jim možemo doći do efikasne i poštene brzine slanja paketa. Prije nego što objasnimo na koji način on radi, razmotrimo nekoliko drugih pokušaja postizanja optimalne brzine slanja paketa. Neka imamo dva pošiljatelja koja dijele jednu mrežnu vezu i oba uređaja aditivno povećavaju broj paketa koji šalju preko veze. Npr. neka svake sekunde povećavaju brzinu slanja za 1Mbps. U jednom trenutku, oba uređaja dobivaju upozorenje o zakrčenosti mreže. Tada, oba uređaja moraju smanjiti svoju brzinu slanja paketa. Ako bi to smanjenje brzine bilo aditivno (-1Mbps), tada bi oba uređaja jednostavno oscilirala oko točke zakrčenosti. Ovo nije problem ako imamo samo spomenuta dva uređaja i ako su oba uređaja krenula s jednakom brzinom slanja paketa. No, ako dodamo novi uređaj koji kreće s mnogo manjom brzinom slanja paketa, tada ova raspodjela mrežne propusnosti ne bi bila poštena. Razlog je taj što bi inicijalna dva uređaja oscilirala oko veće brzine nego novo dodani uređaj.

Pretpostavimo da dva pošiljatelja koriste multiplikativno povećavanje brzine slanja paketa. Neka oba uređaja povećavaju brzinu slanja za 10% svake sekunde. Kada pošiljatelji dobiju upozorenje o zakrčenosti mreže, brzina se smanjuje za 10%. Oba uređaja ponovno osciliraju oko točke zakrčenosti i ponovno ne uspijevamo doći do optimalne brzine slanja koja je poštena i efikasna.

Sada pretpostavimo da koristimo sljedeću tehniku. Brzina slanja paketa aditivno se povećava. Prilikom zakrčenja mreže, brzina slanja paketa multiplikativno se smanjuje. Ova tehnika naziva se AIMD kontrolni zakon. Bez obzira na inicijalnu brzinu slanja paketa svih uređaja, ovaj algoritam efikasno i pošteno konvergira prema optimalnoj radnoj točki. Osim toga, vodimo se činjenicom da je jednostavnije dovesti mrežu do zakrčenja, nego oporaviti se od istoga. Iz tog razloga povećanje brzine slanja treba biti polagano, dok smanjenje brzine treba biti agresivno. AIMD kontrolni zakon koristi se u TCP protokolu. Postoje drugi transportni protokoli koji također koriste istu mrežu. Ako ti protokoli koriste drugačiju tehniku za otklanjanje zakrčenosti mreže, tada može doći do nejednake (nepoštene) raspodjele mrežnih resursa.

4 Primjeri protokola transportnog sloja

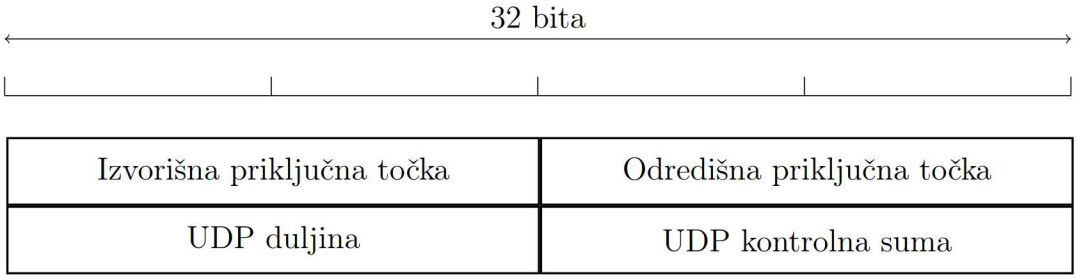
4.1 UDP

UDP (eng. User Datagram Protocol) je nepouzdani protokol koji ne stvara virtualnu vezu između izvora i odredišta. Kada bi IP protokol u svojem zaglavlju specificirao za koji proces je paket namijenjen, tada vjerojatno ne bismo imali potrebu za UDP protokolom. Skoro pa jedina funkcionalnost koju UDP pruža jest dostava paketa do točno određenog procesa (pomoću specificiranja odredišne priključne točke). Ako aplikacija želi određenu pouzdanost ili npr. kontrolu toka, tada sama mora implementirati te funkcionalnosti.

UDP prenosi segmente koji se sastoje od dva dijela, zaglavlja i korisnog tereta (eng. payload). Zaglavlje je konstantne veličine koja iznosi 8 bajtova. Ovo zaglavlje vidimo na slici 6. Prva dva bajta sadrže izvorišnu priključnu točku, odnosno priključnu točku koju koristi aplikacija koja šalje UDP segment. Sljedeća dva bajta specificiraju priključnu točku na odredištu, odnosno priključnu točku koju koristi aplikacija kojoj je namijenjen UDP segment. Na priključne točke možemo gledati kao poštanske sandučiće u koje aplikacije primaju pakete. Ako odredišna aplikacija treba poslati odgovor pošiljatelju, to može jer zna na kojoj se priključnoj točki (izvorišna priključna točka iz zaglavlja) nalazi aplikacija pošiljatelja. Sljedeća dva bajta predstavljaju duljinu UDP segmenta. Ova duljina predstavlja ukupnu duljinu segmenta, odnosno duljinu zaglavlja i korisnog tereta. Najmanja vrijednost ovog atributa je 8 bajtova (segment sadrži samo zaglavlje, bez korisnog tereta). Najveći broj koji možemo spremiti unutar dva bajta je 65535, ali najveća vrijednost ovog atributa je nešto manja. Najveća vrijednost UDP duljine iznosi 65515 jer je maksimalna veličina IP paketa (u kojemu je enkapsuliran UDP segment) 65535. Nakon što oduzmemo 20 bajtova koji su potrebni za zaglavlje IP paketa, ostajemo sa 65515 korisnog tereta u koji enkapsuliramo UDP segment. Zadnja dva bajta u UDP zaglavlju koriste se za otkrivanje grešaka koje mogu nastati u prijenosu paketa. Koristi se metoda koja se naziva kontrolna suma. Kada segment stigne na odredište, ukoliko je njegova kontrolna suma ispravna, korisni teret trenutnog segmenta prosljeđuje se procesu koji je priključen na odredišnu priključnu točku. U izračun kontrolne sume uključujemo zaglavlje (bez kontrolne sume), korisni teret te IP pseudo zaglavlje. Metoda radi na sljedeći način. Dva bajta koja predstavljaju kontrolnu sumu postavljaju se na vrijednost 0. Ako je duljina korisnog tereta neparan broj, na kraj dodajemo jedan bajt vrijednosti 0. Nakon toga, fiktivno podijelimo sve podatke koje koristimo za izračun na jednake dijelove duljine 16 bitova. Sve te dijelove zbrajamo i ako u zbroju imamo višak (eng. overflow, zbroj ima više od 16 bitova), taj višak pribrojimo trenutnoj sumi. Krajnji rezultat invertiramo i ta vrijednost je tražena kontrolna suma. Kada na odredištu računamo kontrolnu sumu, zbrajamo 16 bitne dijelove na jednaki način i na kraju usporedimo dobivenu vrijednost s kontrolnom sumom koju smo primili u zaglavlju i ako se vrijednosti razlikuju tada se primljeni segment odbacuje jer je došlo do greške prilikom prijenosa putem mreže.

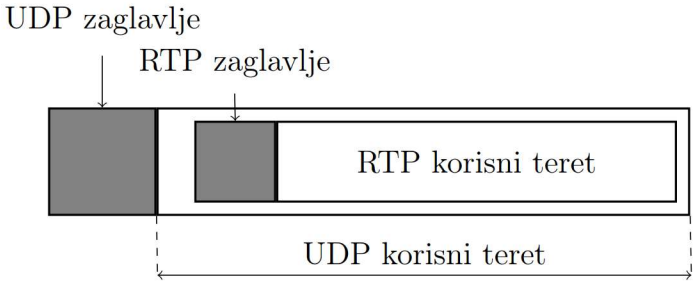
Kao što smo već spomenuli, UDP protokol ne brine o kontroli toka ni o zakrčenosti mreže. Također, ako je otkrivena greška u primljenom segmentu (pomoću kontrolne sume), UDP ne brine o tome da se taj segment ponovno pošalje. UDP protokol jednostavno služi za identificiranje izvorišnog i odredišnog procesa u komunikaciji i za otkrivanje grešaka koje mogu nastati prilikom prijenosa paketa. Ovaj protokol je idealan za one aplikacije koje žele imati potpunu kontrolu nad kontrolom toka, otkrivanjem i ispravljanjem grešaka i slično. Tada te aplikacije mogu implementirati svoj protokol "iznad" UDP protokola. Također, UDP protokol često je korišten s aplikacijama koje imaju strukturu klijent-poslužitelj. Klijent pošalje zahtjev poslužitelju na koji poslužitelj odgovara. Ako je jedna od tih poruka izgubljena,

klijent može ponoviti slanje svoje poruke kada zaključi da je prva poruka izgubljena.



Slika 6: UDP zaglavlje

UDP protokol često se koristi u multimedijskim aplikacijama koje rade u stvarnom vremenu poput videokonferencija, aplikacija za slušanje glazbe ili gledanje serija i filmova. Često korišten protokol za takvu vrstu aplikacija je RTP (eng. Real-time Transport Protocol). RTP koristi UDP za prijenos multimedijskog sadržaja. Multimedijske aplikacije mogu se sastojati od više različitih vrsta bitnih informacija poput zvuka, videa, teksta. Te informacije enkodiraju se u RTP pakete i prosljeđuju se UDP-u kako bi bili preneseni do odredišta. UDP dodaje svoje zaglavlje, a cijeli RTP paket nalazi se u korisnom teretu UDP segmenta. RTP protokol ne nalazi se u potpunosti ni u transportnom sloju ni u aplikacijskom sloju. Razlog je taj što je RTP implementiran u korisničkom prostoru (a ne u jezgri operativnog sustava) i usko je vezan uz mrežnu aplikaciju, ali budući da je generički protokol koji ne ovisi o konkretnoj aplikaciji nego samo pruža usluge prijenosa podataka, možemo ga smatrati i transportnim protokolom.



Slika 7: RTP paket unutar UDP segmenta

Budući da RTP koristi obični UDP protokol, nema garancije da će ti paketi stići na odredište. Također, paketi mogu kasniti ili može doći do greške prilikom putovanja mrežom. Svaki RTP paket označen je rednim brojem. Na osnovu toga broja, aplikacija može znati je li neki paket izgubljen. Ako je to slučaj, aplikacija najčešće samo preskoči izgubljeni segment. Razlog je taj što bi čekanje dok se izgubljeni paket ponovno pošalje i stigne bilo predugačko za aplikaciju koja radi u stvarnom vremenu. Korisnik će ovo iskusiti kao kratki zastoj ili nepravilnost u videu ili zvuku. Iz ovog razloga nema potvrda o primljenim paketima kao ni ponovnog slanja izgubljenih paketa. Kako bi bolje funkcionirao, RTP paket sadrži razne informacije poput vremenske oznake, vrste enkodiranog medija, oznake početka nove slike, identifikator medijskog izvora i slično. Pomoću ovih informacija, RTP protokolom možemo slati video i zvuk (ili više zvukova ako npr. imamo film na engleskom i francuskom jeziku) u isto vrijeme i primalac će bez problema moći rekonstruirati željene informacije iz primljenih paketa.

4.2 TCP

TCP (eng. Transmission Control Protocol) protokol je pouzdani protokol transportnog sloja koji na početku komunikacije između dva uređaja stvara virtualnu vezu između spomenutih uređaja. Kao što će biti vidljivo iz samog zaglavlja TCP segmenta, ovo je mnogo kompleksniji protokol od prethodno spomenutog UDP protokola. TCP brine o ponovnom slanju izgubljenih ili korumpiranih segmenata, također brine i o zakrčenosti mreže te o kontroli toka. Iz ovih razloga, TCP je jedan od najviše korištenih protokola na internetu.

TCP je dizajniran kako bi omogućio pouzdani prijenos podataka putem nepouzdane mreže. Podaci koji se šalju putem ovog protokola nalaze se u obliku niza bajtova. TCP je dizajniran tako da se može nositi s različitim mrežnim arhitekturama s različitom mrežnom propusnosti, veličinom paketa i slično. TCP entitet će primiti podatke od neke aplikacije iz "višeg" sloja te će ih podijeliti u segmente koji nisu veći od 65515 bajtova (u praksi segmenti ne budu veći od 1460 bajtova zbog ograničenja protokola "nižih" slojeva). Tada svaki od tih segmenata biva poslan kao jedan IP paket. Kada paketi stignu na odredište, TCP entitet ih procesuira i prosljeđuje procesu kojemu su namijenjeni. TCP brine da segmenti stignu do odredišta kao i kojom brzinom segmenti trebaju biti poslani. Također, budući da segmenti mogu doći u različitom redoslijedu od onoga u kojemu su poslani, TCP entitet na odredištu zadužen je za rekonstruiranje originalnog niza bajtova.

TCP veza može se koristiti u oba smjera u isto vrijeme. U toj vezi sudjeluju točno dva uređaja, odnosno TCP ne podržava grupnu komunikaciju. TCP ne mora nužno slati podatke istog trenutka kada ih dobije od strane aplikacije. Podaci mogu biti spremljeni u međuspremnik i poslani u onom trenutku kada ih se skupi dovoljno za puni klizni prozor. Postoje i mehanizmi kojima možemo reći da želimo da se podaci odmah pošalju. Više o tome ćemo vidjeti kada budemo analizirali TCP zaglavlje.

Prije nego što krenemo u analizu zaglavlja još ćemo spomenuti MTU (eng. Maximum Transfer Unit). TCP segment sastoji se od zaglavlja veličine barem 20 bajtova (postoji opcionalni dio zaglavlja) te segment može ili ne mora sadržavati korisni teret. Veličina segmenta ograničena je s maksimalnom dozvoljenom veličinom korisnog tereta u IP paketu (65515 bajtova) te s još jednom vrijednosti koju nazivamo MTU. Svaka mrežna veza ima MTU vrijednost, odnosno gornju granicu za veličinu paketa koji prelazi preko te mrežne veze. Za Ethernet ta vrijednost je 1500 bajtova. Ako je paket veći od MTU vrijednosti, tada će on biti fragmentiran i svaki dio će biti zasebno poslan prema odredištu. Na odredištu fragmentirani se dijelovi ponovno sastavljaju u originalni paket. Ova situacija nepotrebno stvara povećanu količinu prometa unutar mreže te komplicira administraciju paketa. Iz ovog razloga, moderne TCP implementacije provode istraživanje putanje kako bi se pronašla minimalna MTU vrijednost na trenutnoj vezi. Kada je poznata minimalna MTU vrijednost, tada se podešava maksimalna veličina TCP segmenta kako bi se izbjegla fragmentacija.

4.2.1 TCP zaglavlje

Na slici 8 vidimo zaglavlje TCP segmenta. Kao što smo gore spomenuli, TCP zaglavlje sastoji se od fiksnih 20 bajtova. Zaglavlje se može proširiti do maksimalnih 60 bajtova (40 opcionalnih bajtova). Analizirat ćemo svaki dio zaglavlja posebno.

Izvorišna i odredišna priključna točka imaju jednaku funkciju kao i kod UDP zaglavlja, a to je identificiranje izvorišnog i odredišnog procesa. Izvorišna i odredišna točka zajedno s izvorišnom i odredišnom IP adresom jedinstveno određuju TCP vezu.

Sljedeći na redu je slijedni broj koji zauzima četiri bajta. Ovaj brojač služi za praćenje poslanih bajtova u nizu bajtova koji šaljemo. Ukoliko segment koji šaljemo sadrži 500 baj-

tova korisnog tereta, tada će slijedni broj idućeg segmenta s korisnim teretom biti za 500 veći od prethodnog (ako dođemo do broja 4294967295 onda ponovno krećemo od nule).

Potvrdni broj služi kako bi jedna strana potvrdila drugoj strani da je primila određeni segment. Potvrdni broj specificira sljedeći slijedni broj koji očekujemo. Npr. ako primimo segment sa slijednim brojem 500 i on sadrži 499 bajtova korisnog tereta tada ćemo poslati potvrdni broj 1000. Potvrdni broj ne mora se slati nakon svakog primljenog segmenta jer on funkcionira kao kumulativna potvrda pa s jednim potvrdnim brojem možemo potvrditi nekoliko primljenih segmenata.

Duljina zaglavlja, kao što možemo i pretpostaviti, predstavlja duljinu zaglavlja u trenutnom segmentu, ali potrebno je objasniti mjernu jedinicu. Vrijednost duljine označava broj riječi duljine 32 bita u zaglavlju. Ova vrijednost potrebna je kada u zaglavlju imamo i opcionalne bajtove. Kada nema opcionalnih bajtova, vrijednost ovog parametra uvijek je 5 (5*(32 bita) = 160 bitova = 20 bajtova).

Iduća četiri bita trenutno nisu iskorištena. U prvim implementacijama TCP protokola postojalo je šest neiskorištenih bitova, ali s vremenom su dva iskorištena kako bi se protokol poboljšao. Preostala četiri bita ostaju rezervirana za potencijalnu buduću upotrebu.

Slijedi osam bitova koji služe kao zastavice. Prve dvije zastavice su CWR i ECE. Ove dvije zastavice koriste se kako bi signalizirale zakrčenost mreže prilikom korištenja proširenja za eksplicitne obavijesti o zakrčenju. Ovo je proširenje definirano za IP i TCP protokol. ECE zastavicu postavlja TCP primatelj na potvrdni segment koji šalje pošiljatelju. TCP ne odlučuje sam kada će aktivirati ovu zastavicu već to čini kada dobije informaciju o zakrčenju od IP protokola. Kada pošiljatelj dobije potvrdni segment s aktiviranom ECE zastavicom to mu služi kao znak da treba usporiti brzinu slanja segmenata. Tada pošiljatelj postavlja zastavicu CWR na jedinicu kako bi obavijestio primatelja da je usporio brzinu slanja segmenata te da primatelj može prestati slati potvrdne segmente s aktiviranom ECE zastavicom.

Sljedeća zastavica je URG zastavica. Ova zastavica postavljena je na vrijednost jedan kada se koristi polje u zaglavlju koje označava hitni pokazivač. URG zastavica služi kako bi pošiljatelj signalizirao primatelju da mu šalje podatke koje je potrebno odmah procesuirati. Hitni pokazivač označava gdje u trenutnom segmentu završavaju hitni podaci. URG zastavica većinom više nije u upotrebi.

ACK zastavica služi kako bismo znali da trenutni segment sadrži (validni) potvrdni broj. Ako je ACK zastavica postavljena na vrijednost nula, dio zaglavlja u kojemu se nalazi potvrdni broj ignorira se.

PSH zastavica signalizira primaocu da treba proslijediti bajtove, koje trenutno ima u svom međuspremniku (uključujući i trenutni segment), odgovarajućem aplikacijskom procesu.

Sljedeća je RST zastavica. Ova zastavica koristi se kako bi se TCP veza ponovno pokrenula. Mogući razlog za aktiviranje ove zastavice je da je jedan od uređaja imao određenu grešku u izvršavanju i morao je ponovno pokrenuti svoj sustav. Ova zastavica također se može koristiti za odbijanje pokušaja uspostave veze.

Posljednje dvije zastavice koriste se pri uspostavi i zatvaranju TCP veze. SYN zastavica, zajedno s ACK zastavicom koristi se pri uspostavi veze. Više o ovoj zastavici vidjet ćemo kada budemo govorili o uspostavi veze. FIN zastavica koristi se za zatvaranje veze. Kada jedan uređaj postavi FIN zastavicu na vrijednost jedan, to znači da on nema više podataka za poslati, ali i dalje može primati podatke.

Nakon osam zastavica slijede dva bajta koja predstavljaju veličinu prozora koji se koristi pri kontroli toka. Primatelj pomoću veličine prozora u zaglavlju potvrdnog segmenta obavještava pošiljatelja o tome koliko novih segmenata može poslati u tom trenutku. Ako je

ova vrijednost postavljena na nulu, to znači da primatelj još nije uspio procesuirati sve do sada primljene segmente i da trenutno ne želi primati nove segmente. Kada primatelj bude spreman primiti nove segmente tada šalje novi potvrdni segment u kojemu specificira novu veličinu prozora.

Kontrolna suma služi, kao i kod UDP protokola, za otkrivanje grešaka koje mogu nastati prilikom prijenosa segmenta putem mreže.

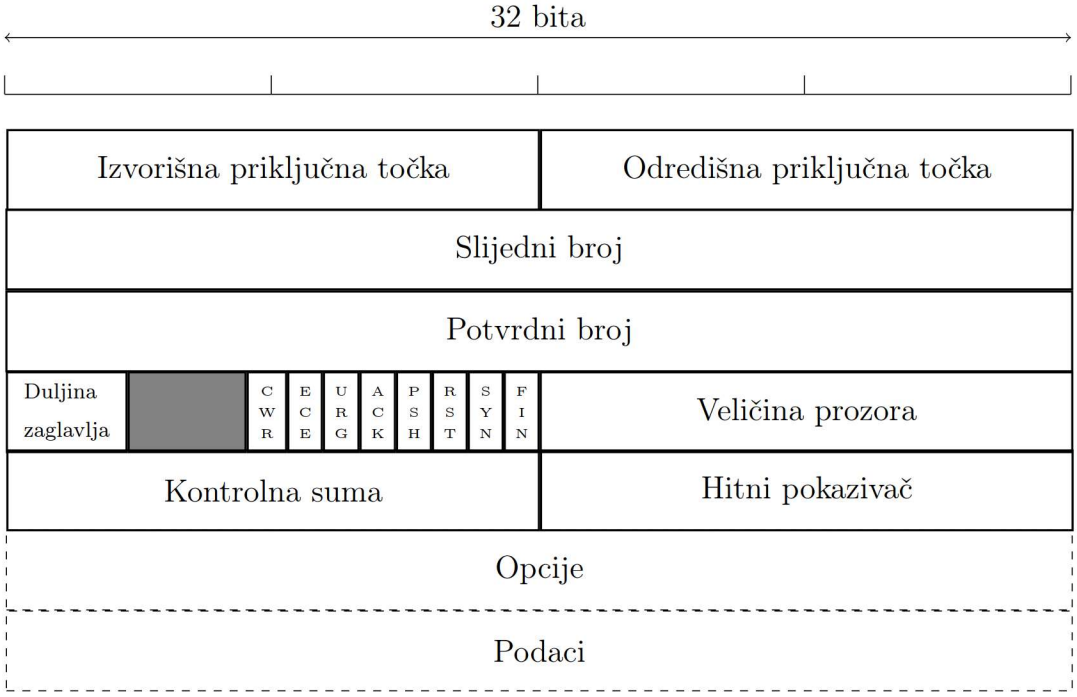
Nakon fiksnih 20 bajtova TCP zaglavlja slijede opcionalne opcije. Opcije služe kako bismo dobili neke dodatne funkcionalnosti koje nemamo u fiksnom zaglavlju. Postoji mnogo opcija, od kojih se nekoliko često koriste. Opcije nemaju fiksnu duljinu, ali njihova duljina (u bitovima) mora biti višekratnih broja 32 (na kraj dodajemo nule po potrebi). Gornja granica za veličinu opcija je 40 bajtova. Opcije mogu biti uključene u segment u različitim stadijima komunikacije (uspostava veze, korištenje veze). Spomenut ćemo nekoliko najčešće korištenih opcija.

MSS (eng. Maximum Segment Size) opcija koristi se kako bi svaki uređaj obavijestio drugi uređaj kolika je najveća veličina segmenta (bez računanja veličine zaglavlja) koju je voljan prihvatiti. Zbog velikog zaglavlja (barem 20 bajtova) efikasnije je u svaki segment ubaciti što više korisnog tereta. Razlog za manji MSS može biti ako npr. imamo jedan slabiji uređaj koji ne može dovoljnom brzinom procesuirati velike segmente. Ova opcija dogovara se prilikom uspostave veze. Svaki od uređaja reći će drugom uređaju svoj MSS. Ako uređaji ne specificiraju ovu opciju prilikom uspostave veze, tada se za MSS uzima vrijednost od 536 bajtova. Ranije smo spomenuli MTU vrijednost. Ako znamo MTU vrijednost za trenutnu vezu, pomoću MSS opcije možemo osigurati da nijedan IP paket ne bude veći od MTU vrijednosti.

Još jedna često korištena opcija je skaliranje veličine kliznog prozora. Ako se na putanji TCP veze nalaze mrežne veze koje imaju veliku mrežnu propusnost tada može biti efikasno koristiti veći klizni prozor. Originalni klizni prozor ograničen je (budući da je u zaglavlju predstavljen s dva bajta) s vrijednošću od 65535. Pomoću ove opcije možemo proširiti veličinu kliznog prozora. Na početku veze, dva uređaja dogovaraju faktor za skaliranje. Neka je x dogovoreni faktor, a W trenutna veličina kliznog prozora (vrijednost iz zaglavlja). Stvarna veličina kliznog prozora tada je $2^x \times W$ bajtova.

Timestamp opcija nosi u sebi vremensku oznaku i prisutna je u svakom segmentu ako je dogovorena prilikom uspostave veze. Ova opcija koristi se kako bi se izračunalo vrijeme koje je potrebno da paket ode od izvora do odredišta i natrag. Na taj način ova opcija pomaže pri zaključivanju o gubitku određenog segmenta. Također, ova opcija može biti korisna na brzim vezama. Možemo ju koristiti kao dodatak na slijedni broj (koji je inicijalno veličine 32 bita) i tako razlikovati nove od starih segmenata ako se dogodi da slijedni brojevi krenu ponovno od nule (nakon četiri gigabajta podataka iskoristit ćemo sve slijedne brojeve).

Posljednja opcija koju ćemo spomenuti je SACK (eng. Selective ACKnowledgement) opcija. Obični način za potvrdu segmenata potvrđuje segmente kumulativno. To znači da ako primimo segmente 1, 2, 4 i 5, poslat ćemo potvrdu za segment 1 i segment 2 (ili samo za segment 2), ali nećemo potvrditi da smo primili segmente 4 i 5 jer nismo još primili segment 3. SACK opcija omogućuje primaocu da potvrdi određeni skup segmenata. Na ovaj način pošiljatelj zna točno koje je sve segmente primalac primio, a koji su segmenti možda izgubljeni pa tako može ponovno poslati samo izgubljene segmente. Na ovaj način nema bespotrebnog slanja duplih segmenata.

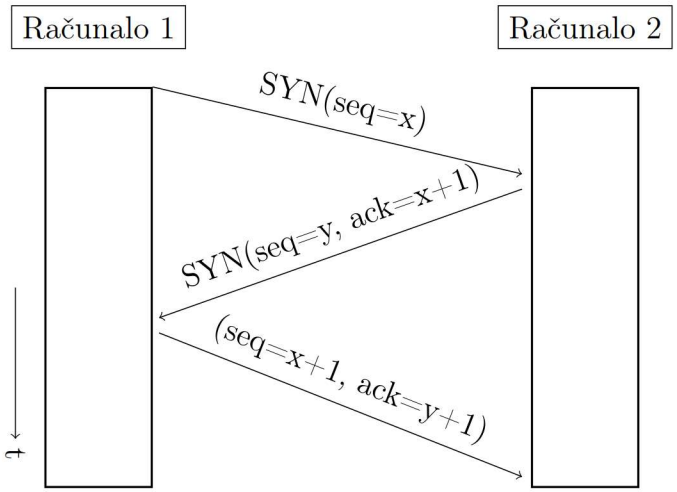


Slika 8: TCP zaglavlje

4.2.2 Uspostava i zatvaranje veze

TCP koristi prethodno opisanu strategiju trostrukog rukovanja kako bi uspostavio vezu između dva uređaja. Prvo što je potrebno je da se jedan od uređaja priključi na neku priključnu točku i čeka dolazeći zahtjev za uspostavom veze. Uređaj koji želi inicirati uspostavu veze specificira IP adresu i priključnu točku uređaja s kojim želi uspostaviti vezu. Također, moguće je specificirati neke opcije poput MSS vrijednosti ili SACK opcije. Kada pozovemo operaciju *Poveži*, TCP entitet poslat će na specificiranu adresu segment s aktiviranom SYN zastavicom i isključenom ACK zastavicom te će stati i čekati odgovor drugog uređaja. Drugi uređaj, onaj koji čeka nadolazeći zahtjev za uspostavom veze, prima spomenuti segment i procesuira njegovo zaglavlje. TCP entitet tog uređaja provjerava postoji li proces koji aktivno sluša nadolazeće zahtjeve na priključnoj točki specificiranoj u zaglavlju dobivenog segmenta. Ako ne postoji, TCP entitet šalje inicijalnom pošiljatelju segment s aktiviranom RST zastavicom kako bi ga obavijestio da je njegov pokušaj za uspostavom veze odbijen. Ako postoji proces koji je priključen na specificiranoj priključnoj točki, tada će segment biti proslijeđen tom procesu. Proces koji je primio zahtjev za uspostavom veze može taj zahtjev prihvatiti ili odbiti. Ako je zahtjev prihvaćen, tada šaljemo potvrdni segment s potvrdnim brojem koji je za jedan veći od slijednog broja primljenog segmenta. Kada uređaj koji je inicirao uspostavu veze primi potvrdni segment, on šalje svoj potvrdni segment i kada on pristigne na odredište, veza će biti uspostavljena. Taj zadnji potvrdni segment može u korisnom teretu sadržavati i prvu seriju bajtova namijenjenih za aplikaciju na drugoj stani veze. Ilustraciju uspostave TCP veze vidimo na slici 9

Slijedni brojevi trebali bi biti različiti od veze do veze, tj. prilikom uspostave veze nećemo uvijek krenuti sa slijednim brojem 0. Na ovaj način osiguravamo se od zakašnjelih paketa koji mogu imati jednaki slijedni broj, ali pripadati staroj vezi. TCP entiteti najčešće koriste nasumično generirane inicijalne slijedne brojeve i na taj se način ujedno štite i od potencijalnih napadača koji bi mogli preoteti trenutnu vezu i krivotvoriti segmente.



Slika 9: Uspostava TCP veze

Za zatvaranje TCP veze koriste se već spomenute FIN zastavice. TCP veze su dvosmjerne veze, ali kada razmatramo njihovo zatvaranje, možemo jednu dvosmjernu vezu promatrati kao dvije jednosmjerne veze. Kako bi jedan uređaj zatvorio svoju stranu TCP veze, on šalje segment s aktiviranom FIN zastavicom. Ovaj postupak znači da taj uređaj nema više podataka koje želi poslati drugom uređaju, ali i dalje može primati podatke. Kada uređaj koji je poslao segment s aktiviranom FIN zastavicom dobije potvrdu za taj segment, jedna strana veze je zatvorena. Kada i drugi uređaj napravi jednaki postupak (segment s FIN zastavicom poslan i potvrđen), veza je uspješno zatvorena. Kada smo govorili o općenitom načinu za zatvaranje veze u transportnom sloju, spomenuli smo problem dvije vojske. TCP se osigurava protiv ovog problema korištenjem mjerača vremena. Ako segment s aktiviranom FIN zastavicom ne bude potvrđen u vremenu koje je jednako dvostrukom očekivanom životu jednog paketa, uređaj koji je poslao segment zatvara svoju stranu veze. Drugi će uređaj nakon određenog vremena primijetiti da mu nitko s druge strane ne šalje nikakve segmente i on će zatvoriti svoju stranu veze.

4.2.3 Klizni prozor

TCP koristi dinamički klizni prozor kojim kontrolira tok segmenata između dva uređaja. Kada jedan uređaj primi segment određene veličine on ga mora spremiti u međuspremnik koji je predvidio za trenutnu vezu. Taj segment ostaje u međuspremniku sve dok ga aplikacijski proces nije spreman pročitati. To znači da, iako je segment uspješno stigao do odredišta i ako pošiljatelj dobije potvrdu da je segment uspješno stigao do odredišta, to ne znači nužno da je uređaj na odredištu spreman za primanje novih segmenata. Ako je međuspremnik na odredištu pun, pošiljatelj će dobiti informaciju o tome iz veličine prozora (koja se nalazi u zaglavlju) potvrdnog segmenta. Uređaj može postaviti veličinu prozora na vrijednost 0 i time obavijestiti pošiljatelja da trenutno ne može primiti nove segmente. Kada uređaj bude spreman za primanje novih segmenata, on će poslati segment koji će imati jednaki potvrdni broj kao prošli segment (onaj s veličinom prozora 0), ali će za veličinu prozora imati pozitivnu vrijednost. Ako uređaj ima prazan međuspremnik veličine 4000 bajtova i upravo je dobio segment veličine 2000 bajtova, tada će on poslati potvrdni segment koji će za veličinu prozora imati vrijednost 2000.

Neka je pošiljatelj dobio informaciju od primaoca da trenutno ne treba slati nove segmente (veličina prozora je 0). Pošiljatelj tada može slati segmente samo u dvije situacije.

Prva je, ako pošiljatelj želi poslati segment s aktiviranom URG zastavicom, odnosno ako šalje hitne podatke. Druga situacija je kada pošiljatelj šalje takozvanu prozorsku sondu (eng. window probe). Ovo je poseban segment koji je koristan u situaciji kada je segment koji bi nam trebao omogućiti ponovno slanje (segment s pozitivnom veličinom prozora) izgubljen. Kada ovaj segment stigne na odredište, on tjera primaoca da ponovno pošalje segment u kojemu će izjaviti koji idući segment očekuje i kolika je trenutna veličina prozora. Bez ove opcije, postoji mogućnost da bi nastupio takozvani *deadlock*. Primatelj bi zauvijek čekao sljedeći segment, a pošiljatelj bi čekao segment s pozitivnom veličinom prozora.

Postoje razlozi zbog kojih TCP pošiljatelj nije obvezan istog trenutka poslati podatke kada ih dobije od aplikacijskog procesa. Također, TCP primatelj nije obvezan slati potvrdne segmente istog trenutka kada primi podatke od pošiljatelja. Razlog je poboljšanje performansi. Pošiljatelj može čekati dok nema dovoljno podataka za popuniti cijeli klizni prozor ili barem dok nema dovoljno podataka za segment veličine prethodno dogovorene MSS vrijednosti. Budući da je TCP veza dvosmjerna, "primatelj" možda također ima podatke koje šalje "pošiljatelju". Iz tog razloga, "primatelj" može sačekati sa slanjem potvrdnog segmenta dok ne bude imao neke korisne podatke koje može poslati unutar korisnog tereta potvrdnog segmenta. Na ovaj način smanjuje se količina paketa u mreži i samim time imamo bolje performanse mreže.

Postoji algoritam koji je u širokoj primjeni u raznim implementacijama TCP protokola. Ovaj algoritam bavi se upravo situacijama u kojima pošiljatelj dobiva podatke iz aplikacijskog procesa u malenim dijelovima te odlučuje ne poslati podatke istog trenutka kada ih dobije. Ovaj algoritam zove se "Nagle's algorithm". Algoritam funkcionira na sljedeći način. Kada aplikacija šalje podatke u malenim dijelovima potrebno je poslati prvi od tih malih dijelova i zatim čekati da taj prvi segment bude potvrđen. Za to vrijeme dok čekamo potvrdu, pošiljatelj sprema u međuspremnik nove (malene) podatke koje proizvodi aplikacijski proces. U trenutku kada je prvi segment potvrđen tada šaljemo sve podatke koje smo do sada prikupili u međuspremniku (ako veličina prozora to dozvoljava). Dalje, dok čekamo potvrdu podataka koje smo upravo poslali, nastavljamo prikupljati nove podatke u međuspremniku. Na ovaj način, u jednom trenutku možemo imati najviše jedan neefikasan segment (segment u kojemu veličina zaglavlja dominira ukupnom veličinom segmenta) u mreži. Ovo je vrlo koristan algoritam, ali postoje situacije u kojima je bolje ako je isti onemogućen. Jedan primjer kada je bolje isključiti ovaj algoritam je kada je aplikacijski proces interaktivna igrica. Ovakva aplikacija mnogo ovisi o responzivnosti. Ako upravljačke komande nisu poslane istog trenutka kada ih igrač unese, nego su spremljene u međuspremnik dok se ne skupi dovoljna količina podataka, igrači mogu imati loše igračko iskustvo.

U ovom ćemo dijelu spomenuti još jedan mogući problem koji može ozbiljno utjecati na efikasnost TCP komunikacije. Sindrom blesavog prozora (eng. silly window syndrome) pojava je do koje dolazi kada aplikacija na odredišnom uređaju procesuira jedan po jedan primljeni bajt. Neka je TCP međuspremnik na odredištu pun. Pošiljatelju je javljeno da je trenutna veličina prozora jednaka 0. Aplikacija pročita jedan bajt iz TCP međuspremnika i time u međuspremniku imamo mjesta za jedan novi bajt. TCP entitet primatelja šalje segment s ažuriranjem veličine prozora (nova vrijednost je sada 1). Pošiljatelj šalje novi segment s jednim bajtom korisnog tereta. Međuspremnik na odredištu je ponovno pun i pošiljatelj ponovno dobiva vrijednost nula za trenutnu veličinu prozora. Ovo dovodi do vrlo neefikasne komunikacije. Rješenje je da ne dopustimo primatelju da šalje ažuriranje veličine prozora koje će dopustiti slanje samo jednog novog bajta. Umjesto toga, primatelj treba sačekati dok aplikacijski proces ne oslobodi dovoljno prostora u međuspremniku za veću količinu podataka prije nego što pošalje ažuriranje veličine prozora.

4.2.4 Mjerači vremena

Kako bi TCP pravilno funkcionirao potrebno mu je nekoliko mjerača vremena s različitim zadaćama. U ovom dijelu imat ćemo nekoliko formula koje ćemo odmah na početku navesti, a potom ćemo postepeno objasniti gdje se one koriste:

$$SRTT = \alpha SRTT + (1 - \alpha)R, \tag{1}$$

$$RTTVAR = \beta RTTVAR + (1 - \beta)|SRTT - R|, \tag{2}$$

$$RTO = SRTT + 4 \times RTTVAR. \tag{3}$$

Prvi mjerač vremena je mjerač zadužen za ponovno slanje (izgubljenih) segmenata. Ovaj mjerač naziva se RTO (eng. Retransmission TimeOut). Kada TCP entitet pošalje neki segment ovaj mjerač se pokreće. Ako vrijeme istekne prije nego što dobijemo potvrdni segment, onda ponovno šaljemo taj segment i ponovno pokrećemo RTO mjerač vremena. Ako dobijemo potvrdni segment, mjerač se zaustavlja. Najveći problem pri implementaciji mjerača vremena jest podešavanje vremena nakon kojeg oni reagiraju.

Određivanje vremena koje je potrebno da segment stigne do odredišta i vrijeme koje je potrebno da potvrdni segment stigne nazad do pošiljatelja je kompliciranо. Na to vrijeme utječu faktori unutar mreže na koje ne možemo utjecati. Ako vrijednost RTO mjerača postavimo na premalu vrijednost, nepotrebno ćemo slati duplikate segmenata. Ako mjerač odreagira previše kasno, nećemo imati efikasan prijenos podataka u slučaju kada segmenti stvarno budu izgubljeni. Budući da se stanje unutar mreže konstantno mijenja, potreban nam je dinamički algoritam za određivanje idealnog vremena nakon kojeg RTO mjerač treba odreagirati. TCP koristi varijablu $SRTT$ (eng. Smoothed Round-Trip Time) koja služi kao trenutna procjena ukupnog vremena koje je potrebno da segment stigne na odredište i da potvrdni segment stigne do pošiljatelja. Kada pošiljatelj pošalje segment, pokreće se RTO mjerač vremena. U ovom slučaju mjerač nam koristi za dvije stvari. Prvo, u slučaju da potvrdni segment ne stigne prije isteka vremena, segment se ponovno šalje. Drugo, ako potvrdni segment stigne, tada kada zaustavimo mjerač znamo koliko je ukupno vremena ovaj puta trebalo segmentu da stigne do odredišta i potvrdnom segmentu da stigne do pošiljatelja. To vrijeme (u slučaju kada smo dobili potvrdni segment) označit ćemo s R. Varijabla $SRTT$ tada se ažurira prema formuli (1) gdje je α hiperparametar najčešće postavljen na vrijednost $\frac{7}{8}$. α igra ulogu u tome koliko brzo zanemarujemo vrijednosti iz prijašnjih iteracija. Kada imamo vrijednost SRTT sljedeće što trebamo jest odrediti koliko će biti novo vrijeme našeg RTO mjerača. Predložena je varijabla $RTTVAR$ na koju će utjecati razlika između vremenu R i varijable $SRTT$. Formula za ažuriranje vrijednosti varijable $RTTVAR$ dana je u formuli (2) gdje je β obično $\frac{3}{4}$ (i ima jednaku ulogu kao α u (1)). Na kraju mjerač RTO postavljamo na vrijednost iz formule (3). Broj četiri u formuli (3) možemo interpretirati na način da će većina potvrdnih segmenata stići unutar četiri standardne devijacije od očekivanog vremena dolaska. Neovisno o vrijednosti koju dobijemo pomoću formula (1), (2) i (3), RTO vrijednost će uvijek biti postavljena na najmanje jednu sekundu kako bismo izbjegli nepotrebna ponovna slanja zbog neočekivanih vrijednosti dobivenih prilikom mjerenja.

Postoji mogućnost da RTO mjerač odreagira, a da je zbog moguće zakrčenosti u mreži inicijalni segment i dalje na putu prema odredištu. Tada ponovno šaljemo segment, jer ne znamo da prvi segment nije izgubljen. Problem nastaje kada dobijemo potvrdni segment. Nemamo način na koji možemo odrediti je li to potvrdni segment od inicijalno poslanog segmenta ili od onog kojeg smo (bez potrebe) poslali nakon isteka mjerača. Ako bismo koristili vrijeme R koje smo počeli mjeriti nakon ponovnog slanja segmenta, tada bismo mogli

pokvariti do sada izračunate vrijednosti $SRTT$ i $RTTVAR$. Rješenje koje se koristi je sljedeće. $SRTT$ i $RTTVAR$ se ne ažuriraju ukoliko smo morali ponovno poslati segment, već vrijednost RTO mjerača postavljamo na dvostruko veću vrijednost od prijašnje.

Sljedeći mjerač koji se koristi u TCP implementacijama je mjerač ustrajnosti (eng. persistence timer). Njega koristimo u već spomenutoj situaciji kada jedan uređaj postavi vrijednost veličine prozora na nulu i time govori drugom uređaja da trenutno ne želi primati nove segmente. Kasnije, uređaj šalje segment u kojemu postavlja vrijednost veličine prozora na neku pozitivnu vrijednost, ali taj segment se izgubi. Sada oba uređaja čekaju da drugi uređaj pošalje neki segment. Kada mjerač ustrajnosti odreagira, uređaj (koji je primio segment s veličinom prozora 0) šalje prozorsku sondu koju smo spomenuli kada smo pričali o kliznom prozoru. Ako nakon toga ponovno dobijemo segment koji kaže da je veličina prozora i dalje nula, mjerač ustrajnosti se ponovno pokreće.

U nekim implementacijama postoji mjerač vremena koji koristimo kako bismo vezu održali aktivnom. Ako određeno vrijeme nijedna strana nije primila niti jedan segment, ovaj mjerač vremena će odreagirati i poslati segment kako bi provjerio je li drugi uređaj i dalje aktivan. Ako ne dobije odgovor, uređaj zatvara vezu sa svoje strane. Ovaj je mjerač koristan u situacijama kada jedan od uređaja ima iznenadni problem i mora ponovno pokrenuti svoj sustav. Takav mjerač mora biti resetiran svaki puta kada primimo segment.

4.2.5 Kontroliranje zakrčenosti mreže

Zadnje o čemu ćemo govoriti vezano uz TCP protokol jesu principi s kojima TCP protokol izbjegava zakrčenost mreže te način ponašanja protokola za vrijeme zakrčenosti mreže. Kada TCP dobije informaciju da je mreža pod velikim opterećenjem (najčešći signal su izgubljeni segmenti), njegov zadatak je da uspori brzinu kojom šalje segmente u mrežu. Ranije smo spomenuli AIMD kontrolni zakon kao princip s kojim možemo postići efikasnu brzinu slanja paketa. TCP koristi principe ovog zakona kako bi uspješno kontrolirao zakrčenost mreže.

Način na koji TCP implementira kontrolu zakrčenosti jest pomoću prozora zakrčenosti koji predstavlja koliko bajtova pošiljatelj smije imati unutar mreže u jednom trenutku. Veličina ovog prozora mijenja se koristeći AIMD kontrolni zakon. Veličina prozora zakrčenosti također ovisi i o veličini kliznog prozora. Uzima se manja od te dvije vrijednosti i ta vrijednost označava koliko segmenata pošiljatelj smije poslati. Budući da gubitak segmenta tretiramo kao znak zakrčenosti mreže, potrebno je imati dobro podešene mjerače vremena. O tom postupku smo maloprije detaljno govorili. No, vođenje računa o prozoru zakrčenosti te o izgubljenim segmentima nije jedini posao kojim se naša kontrola zakrčenosti treba baviti.

Želimo imati efikasnu iskorištenost mrežne propusnosti, ali isto tako želimo biti pošteni prema ostalim korisnicima mreže. Zamislimo da je pošiljatelj spojen na mrežnu vezu s velikom propusnošću. Tada pošiljatelj u vrlo kratkom roku može poslati veliku količinu paketa. Na putu do odredišta, paketi mogu naići na ruter koji koristi vrlo sporu mrežnu vezu s malom propusnošću. Međuspremnik tog rutera bit će jako brzo napunjen s našim paketima koji čekaju na prijenos putem spore mrežne veze. Iz tog razloga, drugi paketi (od drugih uređaja koji također koriste ovaj ruter na putanji svoje komunikacije) koji stižu do ovog rutera mogu biti odbačeni jer više nema mjesta u međuspremniku. Tehnika koja se koristi kako bismo izbjegli ovu situaciju zasniva se na ideji da paketi trebaju biti poslani brzinom koja odgovara brzini najsporije mrežne veze na putanji do odredišta. Na početku, pošiljatelj šalje nekoliko segmenata jedan za drugim. Segmenti putuju preko mnogo mrežnih veza različitih brzina. Kada segment stigne do primatelja, on šalje potvrdni segment. Razmak između pristizanja pojedinih segmenata otprilike je jednak vremenu koje je potrebno da je-

dan segment pređe preko najsporije mrežne veze na putu. To je upravo vremenski razmak koji trebamo imati između slanja dva segmenta. Budući da primatelj šalje potvrdni segment nakon svakog primljenog segmenta, potvrdni segmenti su također poslani u tom željenom razmaku. Pošiljatelj iz vremena pristizanja potvrdnih segmenata zaključuje kojom brzinom treba slati segmente. Ako pošiljatelj šalje pakete ovom frekvencijom, tada će oni biti poslani maksimalnom mogućom brzinom (ograničenom brzinom najsporije mrežne veze na putanji), a da pri tom ne prepune ni jedan međuspremnik na putu.

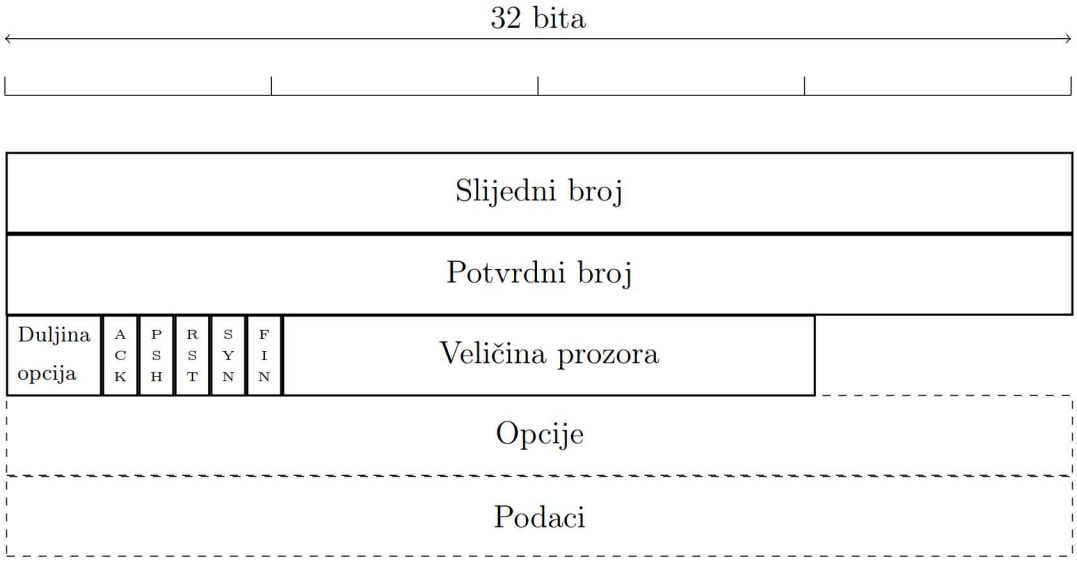
Sljedeća stvar na koju trebamo obratiti pažnju jest koliko je vremena potrebno da veličina prozora zakrčenosti naraste do optimalne veličine. Ako bismo koristili obični AIMD kontrolni zakon, na vezama koje imaju veliku mrežnu propusnost, trebalo bi mnogo vremena da postignemo optimalnu vrijednost. AIMD kontrolni zakon kaže da se veličina prozora povećava za jedan segment svakih *RTT* vremena. *RTT* predstavlja ukupno vrijeme koje je potrebno da segment stigne do odredište i da potvrdni segment stigne do pošiljatelja. Na ovaj način, veličina prozora povećava se relativno sporo. Rješenje koje se često koristi naziva se "spori start" (eng. slow start). Ovo je kombinacija aditivnog i multiplikativnog povećavanja veličine prozora. Nakon uspostave veze, pošiljatelj kreće s malim prozorom zakrčenosti veličine najviše četiri segmenta. Za svaki primljeni potvrdni segment (koji je stigao prije nego što je RTO mjerač vremena istekao), veličina prozora povećava se za jedan segment. Na ovaj način, svaki potvrđeni segment omogućava slanje dva nova segmenta (jedan koji će zamijeniti upravo potvrđeni segment te jedan koji možemo poslati zbog povećanja prozora). Koristeći obični AIMD zakon, prozor zakrčenosti povećavao se za veličinu jednog segmenta svakih *RTT* vremena. Koristeći algoritam sporog starta, svakih *RTT* vremena, prozor postaje dvostruko veći. Zato što veličina prozora brzo raste, u jednom trenutku ćemo zasititi mrežu i doći će do zakrčenja. Iz ovog razloga, postavljamo i ažuriramo graničnu vrijednost sporog starta. Ova granična vrijednost inicijalno je postavljena na neku vrijednost manju ili jednaku veličini kliznog prozora (maksimalna količina podataka koje primatelj može primiti bez gubitka). Spori start povećava veličinu prozora zakrčenosti sve dok ne detektiramo izgubljene segmente ili dok veličina prozora ne pređe postavljenu graničnu vrijednost. Zbog gubitka određenih segmenata, RTO mjerač reagira. U ovom trenutku postavljamo novu graničnu vrijednost koja će biti jednaka polovici trenutne veličine prozora zakrčenosti, a veličina prozora postavlja se na inicijalnu vrijednost. Ako pređemo graničnu vrijednost (koja je u ovoj situaciji manja od veličine kliznog prozora), prestajemo povećavati veličinu prozora na dosadašnji način i prebacujemo se na aditivno povećavanje. Poanta ovog algoritma je da držimo veličinu prozora zakrčenosti što bliže optimalnoj vrijednosti. Na taj način imamo dobru iskorištenost mreže i smanjujemo mogućnost gubitka paketa zbog zakrčenosti mreže.

Postoji nekoliko načina na koji možemo dodatno poboljšati performanse TCP veze. Prva je nešto što se naziva brza retransmisija. Ideja je da ne čekamo da nam RTO mjerač javi da je neki segment izgubljen već da sami dođemo do tog zaključka. Kada je jedan od segmenata izgubljen, primatelj neće moći poslati potvrdni broj za segmente koji dolaze nakon (po slijednom broju) izgubljenog segmenta. No, za svaki primljeni segment, primatelj šalje potvrdni segment. Ako je jedan segment izgubljen, pošiljatelj će slati potvrdne segmente s potvrdnim brojem izgubljenog segmenta (jer ne može povećati potvrdni broj iznad slijednog broja izgubljenog segmenta). Ovi duplikati potvrdnih segmenata mogu signalizirati pošiljatelju da je segment izgubljen. Budući da segmenti ponekad mogu stići na odredište u različitom poretku od onoga u kojemu su poslani, pošiljatelj ne reagira odmah. Nakon što pošiljatelj primi tri uzastopna duplikata potvrdnog segmenta, on zaključuje da je segment izgubljen. Budući da potvrdni broj označava sljedeći po redu slijedni broj koji primatelj očekuje, znamo točno koji segment trebamo ponovno poslati. Nakon što pošiljatelj ponovno

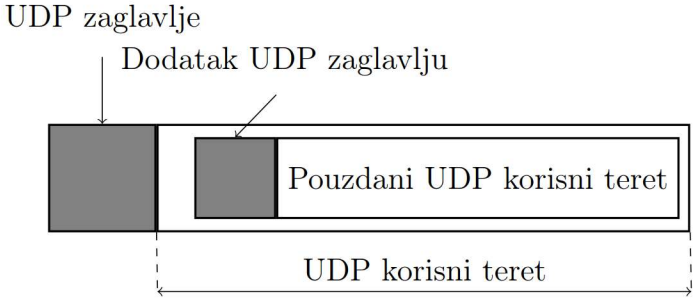
pošalje izgubljeni segment, granična vrijednost algoritma sporog starta postavlja se na polovicu trenutne vrijednosti prozora zakrčenosti. Veličina prozora zakrčenosti može se postaviti na inicijalnu vrijednost. Druga mogućnost je da veličinu prozora postavimo na spomenutu novo postavljenu graničnu vrijednost te da nakon toga počnemo aditivno povećavati veličinu prozora. Ovo je metoda takozvanog brzog oporavka. Na ovaj način izbjegavamo spori start osim na početku komunikacije ili u trenutku kada RTO mjerač istekne. Cilj nam je imati veličinu prozora, većinu vremena, što bliže optimalnoj vrijednosti. Ovi algoritmi mogu se dodatno poboljšati ako koristimo SACK opciju. Tada pošiljatelju možemo eksplicitno javiti koji su segmenti primljeni.

5 Pouzdani UDP

Za kraj, predstavit ćemo jednu implementaciju koja u pozadini koristi UDP protokol, ali kojoj smo sami dodali funkcionalnosti poput slijednih brojeva, potvrdnih segmenata, kontrole toka te ponovnog slanja izgubljenih segmenata. Ovu implementaciju nazivam pouzdani UDP jer brine da je svaki poslani segment uspješno dostavljen. Potpuna implementacija u programskom jeziku Python može se naći pod [8]. Ideja ja da poput ranije spomenutog RTP protokola, sami nadogradimo UDP zaglavlje s našim posebnim zaglavljem koje će se nalaziti u korisnom teretu UDP segmenta (po uzoru na RTP paket na slici 7). UDP zaglavlje sadrži izvorišnu i odredišnu priključnu točku te duljinu segmenta i kontrolnu sumu. Po uzoru na TCP zaglavlje, našem zaglavlju dodajemo slijedni broj (32 bita), potvrdni broj (32 bita), duljinu opcija (3 bita), zastavice ACK, PSH, RST, SYN, FIN (ukupno 5 bitova), veličinu prozora (16 bita) te opcionalne opcije. Svi dijelovi zaglavlja koji dijele imena s dijelovima TCP zaglavlja imaju jednaku funkcionalnost kao i kod TCP protokola. Duljina opcija korištena je umjesto duljine zaglavlja iako ima jednaku funkcionalnost. Pomoću ove vrijednosti znamo sadrži li naše zaglavlje ikakve opcionalne dijelove. Nakon zaglavlja, naš pouzdani UDP segment može imati korisni teret u kojemu su sadržani podaci koje šaljemo na drugi uređaj.



Slika 10: Pouzdani UDP - dodatak UDP zaglavlju



Slika 11: Segment pouzdanog UDP protokola

Sljedeće dvije Python datoteke, "senderMain.py" i "receiverMain.py", daju primjer kako koristiti ReliableUDP klasu. Datoteka "senderMain.py" specificira IP adresu primaoca te priključnu točku odredišnog procesa. U ovom slučaju želimo na drugi uređaj poslati fotografiju. Otvaramo željenu datoteku u binarnom zapisu te unutar varijable *img_bytes* spremamo binarni zapis te datoteke. Instanciramo objekt klase *ReliableUDP* te postavljamo željene parametre (objasnit ćemo ih kasnije). Pozivanjem metode *connect* uspostavljamo vezu s udaljenim uređajem na specificiranoj adresi. Ova aplikacija prvo šalje ime datoteke koju ćemo poslati, a zatim tu datoteku. Ime šaljemo radi ekstenzije jer budući da datoteku šaljemo u binarnom zapisu, pošiljatelj neće znati koju vrstu datoteke je primio. Ime datoteke potrebno je također pretvoriti u binarni zapis što činimo pozivom metode *bytes* ('utf8' specificira koju vrstu kodiranja želimo koristiti). Ime datoteke te samu datoteku šaljemo na jednaki način, a to je pozivom metode *sendall*.

```python
#senderMain.py
from ReliableUDP import ReliableUDP

RECEIVER_IP = "192.168.1.21"
RECEIVER_PORT = 5000
if __name__ == '__main__':
    f = open("IMG_20191225_171736_400.jpg", "rb")
    img_bytes = f.read()
    f.close()
    reliable_udp = ReliableUDP(mss=10000, time_to_wait_for_connection=10)
    reliable_udp.connect((RECEIVER_IP, RECEIVER_PORT))
    reliable_udp.sendall(bytes("IMG_20191225_171736_400.jpg", 'utf8'))
    reliable_udp.sendall(img_bytes)
```

Datoteka "receiverMain.py" izvršava se na uređaju primatelja. Ovdje specificiramo lokalnu IP adresu te željenu priključnu točku. Ovdje također instanciramo objekt klase *ReliableUDP* sa željenim parametrima. Pozivom metode *bind* priključujemo se na specificiranu priključnu točku. Zatim je potrebno pozvati blokirajuću metodu *accept* i sačekati da druga strana pozove metodu *connect*. Nakon toga veza je uspostavljena i možemo primati podatke. Dva puta pozivamo metodu *recvall* kako bismo primili ime datoteke te samu datoteku. Zatim otvaramo novu datoteku koja će imati naziv koji smo spremili u varijablu *name* te u nju pohranjujemo primljene podatke.

```python
#receiverMain.py
from ReliableUDP import ReliableUDP

LOCAL_IP = "192.168.1.21"
LOCAL_PORT = 5000
if __name__ == '__main__':
    reliable_udp = ReliableUDP(mss=5450, time_to_wait_for_connection=10)
    reliable_udp.bind((LOCAL_IP, LOCAL_PORT))
    conn, addr = reliable_udp.accept()
    name = conn.recvall()
    data = conn.recvall()
    f = open(name.decode('utf8'), "wb")
    f.write(data)
    f.close()
```

Konstruktor klase *ReliableUDP* ima četiri opcionalna parametra. Prvi od njih je *n_of_segments_in_window* koji specificira maksimalnu količinu segmenata koju smo voljni primiti prije nego što zahtijevamo da pošiljatelj stane i sačeka potvrdu. *mss* parametar predstavlja najveću veličinu segmenta koju smo voljni primiti. Idući parametar služi ako želimo da primatelj ne šalje potvrdu istog trenutka kada primi segment. Pomoću parametra *send_ack_after_n_segments_recv* možemo specificirati koliko segmenata da primatelj sačeka prije nego što pošalje potvrdni segment. *time_to_wait_for_connection* parametar specificira koliko smo sekundi voljni čekati drugi uređaj prilikom uspostave veze.

Sada ćemo proći kroz sve varijable klase *ReliableUDP*. *lAddr* predstavlja lokalnu adresu uređaja, tj. uređeni par (IP adresa, priključna točka). *rAddr* predstavlja adresu primatelja. *sock* je zapravo UDP priključak koji koristimo za komunikaciju s udaljenim uređajem. Sljedeće dvije varijable, *_seq_num* i *_ack_num* predstavljaju slijedni i potvrdni broj. Slijedni broj inicijaliziramo na nasumično generirani broj između 0 i 4294967295 (binarni zapis gornje granice ovog broja je šesnaest jedinica). *_mss* postavljamo na vrijednost parametra konstruktora ili na 65496, koji god broj je manji. 65496 je najveća moguća veličina korisnog tereta u segmentu našeg pouzdanog UDP protokola. Razlog je taj što je maksimalna veličina korisnog tereta IP paketa 65515, veličina UDP zaglavlja iznosi 8 bajtova i veličina našeg dodatnog zaglavlja iznosi 11 bajtova (65515 - 8 - 11 = 65496). *_window_size* jednostavno predstavlja najveću količinu segmenata (ali u bajtovima) koju smo voljni primiti prije nego što pošiljatelj mora sačekati potvrdu. *_rWindow_size* predstavlja pošiljateljevu veličinu prozora (ovu veličinu ažuriramo svaki puta kada primimo novi segment od pošiljatelja). Varijablu *_congestion_window_factor* koristimo za kontrolu zakrčenosti. Ova vrijednost označava koliko segmenata trenutno smijemo poslati poštujući algoritam "sporog starta". U varijablu *_buffer* spremamo pristigle podatke. Budući da segmenti mogu pristići u krivom poretku, potreban nam je i *_temp_buffer*. Lista *_segments* sadržavat će podatke, podijeljene u segmente, koje želimo poslati primatelju. *_send_ack_timer* je mjerač vremena koji koristimo u situaciji kada je parametar *send_ack_after_n_segments_recv* postavljen na vrijednost veću od jedan. Vidjet ćemo gdje se koristi kada budemo analizirali metodu *recvall*. Za sljedeći mjerač vremena imamo varijablu *_send_duplicate_ack_timer_flag* koja nam govori je li mjerač aktiviran ili nije. Mjerač *_send_duplicate_ack_timer* reagirat će kada određeno vrijeme ne primimo sljedeći segment po redu. Pomoću njega ćemo obavijestiti pošiljatelja da nismo primili određeni segment. Mjerač vremena koji se koristi na pošiljateljevoj strani je *_resend_segments_timer*. Ovaj mjerač služi kako bi nas obavijestio o mogućim izgubljenim segmentima. Pošiljatelj i primatelj koriste varijablu *_resolving_timer_callback_flag* kako bi znali kada su usred funkcije koja je pozvana istekom jednog od mjerača vremena. *_ack_counter* predstavlja broj segmenata koji smo primili, ali za koje još nismo poslali potvrdni segment. Pošiljatelj koristi varijable *_next_segment* i *_n_of_acked_segments* kako bi pratio koji je zadnji segment unutar liste *_segments* potvrđen, odnosno koji idući segment treba poslati. Varijabla *_last_received_ack* služi kako bi pošiljatelj identificirao duplikate potvrdnih segmenata (signal gubitka segmenta). Također, koristi i listu *_expected_acks* koja predstavlja trenutne potvrdne brojeve koje pošiljatelj očekuje. Na ovaj način štitimo se od starih duplikata. Zadnja varijabla klase je *_connection_error_counter* koja predstavlja broj uzastopnih ponovnih slanja segmenata (radi isteka mjerača vremena). Nakon 10 uzastopnih neuspješnih slanja, pretpostavljamo da je drugi uređaj prekinuo vezu i mi prekidamo vezu.

```
1 def __init__(self, n_of_segments_in_window=20, mss=1461,
  ↪  send_ack_after_n_segments_recv=4, time_to_wait_for_connection=20):
```

```
        self.lAddr = None
        self.rAddr = None
        self._sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self._seq_num = randrange(4294967296)
        self._ack_num = None
        self._mss = min(mss, 65496)
        self._n_of_segments_in_window = n_of_segments_in_window
        self._window_size = min(n_of_segments_in_window*(self._mss + header_size),
        ↪  65535)
        self._rWindow_size = 0
        self._congestion_window_factor = 4
        self._slow_start_treshold = n_of_segments_in_window
        self._max_congestion_window_factor = None
        self._buffer = bytes()
        self._temp_buffer = {}
        self._segments = []
        self._send_ack_timer = Timer(0.5, self._timer_says_send_ack)
        self._send_duplicate_ack_timer_flag = False
        self._send_duplicate_ack_timer = Timer(0.8, self._timer_says_send_ack)
        self._resend_segments_timer = Timer(1, self._timer_says_resend_segments)
        self._resolving_timer_callback_flag = False
        self.time_to_wait_for_connection = time_to_wait_for_connection
        self._ack_counter = 0
        self._next_segment = 0
        self._n_of_acked_segments = 0
        self._last_received_ack = -1
        self._expected_acks = []
        self.send_ack_after_n_segments_recv = send_ack_after_n_segments_recv
        self._connection_error_counter = 0
```

Metoda *connect* služi za uspostave veze s drugim uređajem. Kao što smo vidjeli, drugi uređaj mora najprije pozvati metodu *bind* kako bi se priključio na određenu priključnu točku, a zatim pozivom metode *accept* čeka uspostavu veze. Za uspostavu veze koristimo princip trostrukog rukovanja i zastavice SYN i ACK (kao kod TCP protokola). Metoda *connect* prima parametar $rAddr$, a to je adresa drugog uređaja s kojim želimo uspostaviti vezu. Prije nego što pošaljemo segment koji inicira uspostavu veze, stvaramo jedan mjerač vremena. Svrha ovog mjerača vremena jest prekinuti izvršavanje blokirajuće metode *connect* u slučaju da ne dobijemo odgovor unutar $\_time\_to\_wait\_for\_connection$ sekundi. Inicijalni segment sadržavat će samo zaglavlje, bez korisnog tereta. Metoda $\_create\_header$ prima slijedni broj, potvrdni broj, zastavice, veličinu prozora te opcionalne opcije, a vraća bajtove koji predstavljaju željeno zaglavlje. Metoda $\_mss\_option$ vraća string nula i jedinica koji predstavlja binarni zapis opcije koja služi za informiranje drugog uređaja o našoj maksimalnoj veličini segmenta. Korištenjem metode UDP sučelja *sendto* šaljemo segment *header* na adresu $rAddr$. Nakon toga pokrećemo mjerač vremena za slučaj da ne dobijemo odgovor i pozivamo blokirajuću metodu UDP sučelja $recvfrom$ kojoj kao parametar prosljeđujemo koliko bajtova želimo pročitati s UDP priključka. Veličina našeg standardnog zaglavlja je 11, a dodatne opcije koje očekujemo zauzimaju dodatna četiri bajta što ukupno iznosi 15 bajtova. Metoda $recvfrom$ vraća podatke korisnog tereta standardnog UDP segmenta te adresu s koje je taj segment poslan. Nakon što uspješno primimo segment, zaustavljamo mjerač vremena pozivom metode *cancel*. Sljedeće što trebamo je provjeriti je li dobiveni

segment onaj koji očekujemo. Metoda *_second_shake_confirmation* prima dobivene podatke i adresu s koje su oni poslani te vraća vrijednost *True* ako je sve u redu s podacima. Sve je u redu s podacima ako vrijede sljedeće tvrdnje. Adresa s koje su pristigle podaci jednaka je adresi *rAddr*. Pošiljatelj je poslao opcije u kojima specificira *mss* vrijednost. Također, u primljenom segmentu aktivirane su SYN i ACK zastavice te je potvrdni broj za jedan veći od slijednog broja kojeg smo poslali u inicijalnom segmentu. U ovoj metodi također postavljamo vrijednost *_rWindow_size* na vrijednost koju nam je poslao drugi uređaj te vrijednost *_ack_num* na vrijednost za jedan veću od slijednog broja koji smo primili. Ako vrijede navedene tvrdnje tada povećavamo naš slijedni broj za jedan, stvaramo novo zaglavlje i šaljemo ga drugom uređaju. Nakon toga smatramo da je veza uspostavljena.

```
def connect(self, rAddr):
    self.rAddr = rAddr
    connection_failed_timer = Timer(self.time_to_wait_for_connection,
    ↪  self._timer_says_connection_failed)
    header = self._create_header(self._seq_num, 0, 2, self._window_size,
    ↪  self._mss_option())
    print("1st shake initialization")
    self._sock.sendto(header, rAddr)
    connection_failed_timer.start()
    data, addr = self._sock.recvfrom(15)
    connection_failed_timer.cancel()
    print("2nd shake recived")
    if self._second_shake_confirmation(data, addr):
        self._increment_seq_num(1)
        header = self._create_header(self._seq_num, self._ack_num, 16,
        ↪  self._window_size)
        print("3rd shake initialization")
        self._sock.sendto(header, self.rAddr)
        print("connected")
    else:
        print("Connection failed")
```

Metoda *accept* vraća *ReliableUDP* objekt (radi konvencije) te adresu pošiljatelja. Kao i kod metode *connect*, koristimo mjerač vremena kako ne bismo zapeli u blokirajućoj metodi ako ne primimo nikakav segment od pošiljatelja. Pozivom metode *recvfrom* čekamo inicijalni segment. Nakon što smo primili inicijalni segment, izvršavamo sličnu provjeru kao i kod metode *connect* pozivom metode *_first_shake_confirmation*. Provjeravamo *mss* opciju i SYN zastavicu. Također, postavljamo varijable *rAddr* i *_rWindow_size* na specificirane vrijednosti. Varijabla *_ack_num* postavlja se na vrijednost za jedan veću od primljenog slijednog broja. Ako smo primili ispravan inicijalni segment, stvaramo zaglavlje s aktiviranim SYN i ACK zastavicama (vrijednost 18 u binarnom zapisu jednaka je 10010, jedinice odgovaraju pozicijama ACK i SYN zastavica) te ga šaljemo na adresu *rAddr*. Ponovno ćemo koristiti mjerač vremena zbog blokirajuće *recvfrom* metode. Ako primimo potvrdni segment prije isteka mjerača vremena, tada radimo zadnju provjeru prije završetka procesa uspostave veze. U metodi *_third_shake_confirmation* provjeravamo je li segment primljen s adrese *rAddr*, je li aktivirana ACK zastavica te je li potvrdni broj za jedan veći od slijednog broja koji smo poslali u zadnjem segmentu. Ako vrijede navedene tvrdnje tada je proces uspostave veze završen te metoda *accept* vraća instancu trenutnog objekta klase *ReliableUDP* te adresu drugog uređaja. Sada smo spremni za razmjenjivanje podataka

između ova dva povezana uređaja.

```
def accept(self):
    connection_failed_timer = Timer(self.time_to_wait_for_connection,
    ↪  self._timer_says_connection_failed)
    connection_failed_timer.start()
    data, addr = self._sock.recvfrom(15)
    connection_failed_timer.cancel()
    print("1st shake recived")
    if self._first_shake_confirmation(data, addr):
        header = self._create_header(self._seq_num, self._ack_num, 18,
        ↪  self._window_size, self._mss_option())
        print("2nd shake initialization")
        self._sock.sendto(header, self.rAddr)
        connection_failed_timer = Timer(self.time_to_wait_for_connection,
        ↪  self._timer_says_connection_failed)
        connection_failed_timer.start()
        data, addr = self._sock.recvfrom(11)
        connection_failed_timer.cancel()
        print("3rd shake recived")
        if self._third_shake_confirmation(data, addr):
            print("connected")
            return (self, self.rAddr)
        else:
            print("Connection failed")
            return (self, None)
    else:
        print("Connection failed")
        return (self, None)
```

Metoda *sendall* prima podatke (u binarnom zapisu) te ih šalje na adresu specificiranu u varijabli $rAddr$. Najprije postavljamo određene klase varijable na njihove inicijalne vrijednosti (ovo je potrebno ako smo tijekom trenutne veze već koristili ovu metodu, tj. ako smo već slali neke podatke). Također, pozivom metode $\_convert\_data\_to\_segments$ stvaramo segmente koji u korisnom teretu sadržavaju podatke koje želimo poslati drugom uređaju (uskoro ćemo vidjeti kako ova metoda funkcionira). Budući da koristimo prozor zakrčenosti koji smo ranije opisali u TCP protokolu, vrijednost $\_rWindow\_size$ postavljamo na manju od dvije veličine prozora (klizni prozor i prozor zakrčenosti). Zatim, ulazimo u beskonačnu *while* petlju u kojoj šaljemo segmente koje smo stvorili pomoću metode $\_convert\_data\_to\_segments$. U *while* petlji naizmjenično šaljemo segmente pomoću metode $\_send\_next\_window$ te čekamo potvrdne segmente pomoću metode $\_wait\_for\_ack$. Kada metoda $\_wait\_for\_ack$ vrati vrijednost $True$, to znači da smo primili potvrdu da je zadnji segment uspješno primljen i tada izlazimo iz petlje.

```
def sendall(self, data):
    self._last_received_ack = self._seq_num
    self._convert_data_to_segments(data)
    self._next_segment = 0
    self._n_of_acked_segments = 0
    self._expected_acks = []
```

```
    self._connection_error_counter = 0
    self._congestion_window_factor = 4
    self._slow_start_treshold = self._n_of_segments_in_window
    self._rWindow_size = min(self._rWindow_size,
    ↪  self._congestion_window_factor*(self._mss + header_size))
    while True:
        if self._next_segment <= (len(self._segments) - 1) and
        ↪  (self._rWindow_size >=
        ↪  len(self._segments[self._next_segment][0])):
            self._send_next_window()
        if self._wait_for_ack():
            return
```

Metoda *_convert_data_to_segments* prima podatke u binarnom zapisu, te stvara segmente koje pohranjuje u listu *_segments*. Svaki segment u korisnom teretu sadrži jedan dio podataka (maksimalne veličine *_mss*). Lista se zapravo sastoji od uređenih parova oblika (segment, slijedni broj segmenta). Samo u posljednjem segmentu postavljamo zastavicu PSH na vrijednost jedan, ostali segmenti nemaju aktiviranu ni jednu zastavicu. PSH zastavicu koristimo kako bi primatelj znao da je primio zadnji segment trenutnih podataka. Za svaki segment prvo stvaramo zaglavlje na koje nadodajemo dio podataka te taj segment sa svojim slijednim brojem stavljamo na kraj liste. Nakon kreiranja svakog segmenta povećavamo vrijednost varijable *_seq_num* pozivanjem metode *_increment_seq_num* koja brine da je slijedni broj manji ili jednak vrijednosti 4294967295.

```
def _convert_data_to_segments(self, data):
    size = len(data)
    n_of_segments = math.ceil(size / self._mss)
    self._segments = []
    for i in range(n_of_segments):
        curr_segment_size = self._mss if size >= self._mss else size
        size -= self._mss
        flags = 8 if i == (n_of_segments - 1) else 0
        curr_header = self._create_header(self._seq_num, 0, flags,
        ↪  self._window_size)
        segment_w_h = curr_header + data[i*self._mss:(i*self._mss) +
        ↪  curr_segment_size]
        self._segments.append((segment_w_h, self._seq_num))
        self._increment_seq_num(curr_segment_size)
```

Pomoću metode *_send_next_window* šaljemo idući prozor segmenata. Budući da šaljemo novi prozor segmenata, prvo je potrebno zaustaviti mjerač vremena koji se brine da je prošli prozor uspješno dostavljen. Zatim ulazimo u *while* petlju koja ima dva uvjeta. Prvi brine da ne pokušamo pristupiti elementu liste na poziciji nakon zadnjeg elementa liste. Drugi uvjet brine da ne pošaljemo veću količinu segmenata od trenutne veličine kliznog prozora. Prilikom svake iteracije smanjujemo trenutnu veličinu prozora i šaljemo idući po redu segment. Inkrementiramo brojač segmenata te prelazimo na niz uvjetnih naredbi. Cilj ovih uvjetnih naredbi je dodavanje očekivanog potvrdnog broja u listu *_expected_acks*. Imamo nekoliko uvjeta jer je potrebno brinuti da ne dodamo potvrdne brojeve koji se već nalaze u listi, te brinemo da ne pokušamo pristupiti elementu nakon zadnjeg elementa liste. Na kraju ponovno pokrećemo mjerač vremena koji je zadužen za ponovno slanje segmenata u slučaju

da ne dobijemo potvrdu o primitku.

```
def _send_next_window(self):
    self._resend_segments_timer.cancel()
    self._resend_segments_timer = Timer(1, self._timer_says_resend_segments)
    while self._next_segment <= (len(self._segments) - 1) and
    ↪  (self._rWindow_size >= len(self._segments[self._next_segment][0])):
        self._rWindow_size -= len(self._segments[self._next_segment][0])
        self._sock.sendto(self._segments[self._next_segment][0], self.rAddr)
        self._next_segment += 1
        if self._last_received_ack != ((self._segments[self._next_segment -
        ↪  1][1] + len(self._segments[self._next_segment - 1][0]) -
        ↪  header_size) % 4294967296):
            if self._next_segment < (len(self._segments) - 1):
                self._expected_acks.append(
                ↪  self._segments[self._next_segment][1])
            elif self._next_segment == (len(self._segments) - 1):
                self._expected_acks.append(
                ↪  (self._segments[self._next_segment][1] +
                ↪  len(self._segments[self._next_segment][0]) - header_size)
                ↪  % 4294967296)
            elif len(self._segments) == 1:
                self._expected_acks.append((self._segments[0][1] +
                ↪  len(self._segments[0][0]) - header_size) % 4294967296)
    self._resend_segments_timer.start()
```

Metoda $\_wait\_for\_ack$ je blokirajuća metoda koja se poziva nakon što pošaljemo puni prozor segmenata. U ovoj metodi čekamo potvrdne segmente. Ako smo primili potvrdu za zadnji segment iz liste $\_segments$ tada ova metoda vraća vrijednost $True$, inače metoda vraća vrijednost $False$. Python *select* modul omogućava nam da dobijemo informaciju o tome da postoje podaci koji su spremni za čitanje na UDP priključku koji trenutno koristimo. Ovaj modul koristimo kako bismo prije čitanja primljenih podataka provjerili jesmo li trenutno u izvršavanju funkcije koja je pozvana zbog isteka mjerača vremena. Budući da postoje varijable koje koristimo i prilikom slanja i prilikom primanja podataka (poput $\_next\_segment$), ne želimo da se ove dvije operacije izvršavaju u isto vrijeme kako ne bismo poremetili određene vrijednosti. Iz tog razloga, prije čitanja podataka pomoću metode $recvfrom$, provjeravamo jesmo li trenutno u izvršavanju ponovnog slanja segmenata. Ako nismo, onda čitamo primljene podatke. Prvo što provjeravamo je jesu li primljeni podaci poslani s adrese uređaja s kojim trenutno komuniciramo. Ako nisu, odbacujemo ih. Ako jesu, zaustavljamo mjerač vremena zadužen za ponovno slanje trenutnog prozora i raspakiravamo primljeni segment. Metoda $\_unpack\_data$ prima i procesuira segment, zatim vraća slijedni broj, potvrdni broj, zastavice, opcije te korisni teret. Prilikom procesuiranja segmenta, u ovoj metodi ažuriramo vrijednost parametra $\_rWindow\_size$ pomoću veličine prozora prisutne u zaglavlju primljenog segmenta. U nastavku metode $\_wait\_for\_ack$ imamo dvije mogućnosti.

Prva mogućnost je kada je trenutni primljeni potvrdni broj jednak zadnjem primljenom potvrdnom broju. U ovoj situaciji pretpostavljamo da je segment sa slijednim brojem koji je jednak trenutnom potvrdnom broju izgubljen. Varijabla $\_n\_of\_acked\_segments$ govori nam, osim broja do sada potvrđenih segmenata, indeks prvog po redu segmenta unutar liste $\_segments$ koji još nije potvrđen. Varijablu $\_next\_segment$ postavljamo na vrijednost

varijable *_n_of_acked_segments*. Prepolovljavamo veličinu prozora zakrčenosti i postavljamo novu graničnu vrijednost za "spori start". Budući da smo primili segment od drugog uređaja, znamo da veza funkcionira pa postavljamo *_connection_error_counter* na nulu.

Ako trenutni primljeni potvrdni broj nije jednak zadnjem primljenom potvrdnom broju, tada provjeravamo nalazi li se on unutar liste *_expected_acks*. Ako se ne nalazi, zanemarujemo ga. Ako se nalazi unutar te liste, tada pamtimo njegovu poziciju u listi. Budući da koristimo razliku između zadnjeg i trenutnog primljenog potvrdnog broja kako bismo brojali potvrđene segmente, potrebno je provjeriti koji od ova dva broja je veći. Ne možemo očekivati da će uvijek novi potvrdni broj biti veći od starog jer nakon slijednog broja 4294967295 dolazi slijedni broj 0. Ažuriramo broj potvrđenih segmenata i zadnji primljeni potvrdni broj. Ako je primljeni potvrdni broj jednak varijabli *_seq_num* to znači da je zadnji segment iz liste *_segments* potvrđen (jer smo varijablu *_seq_num* povećavali prilikom stvaranja svakog segmenta u metodi *_convert_data_to_segments*). U ovom slučaju ažuriramo varijablu *_ack_num* i vraćamo vrijednost *True*. Inače, ažuriramo listu potvrdnih brojeva koje očekujemo i postavljamo *_connection_error_counter* na nulu jer znamo da veza funkcionira. Tada, ovisno o trenutnoj veličini prozora zakrčenosti, isti povećavamo ili za jedan ili za broj segmenata koji su potvrđeni trenutnim potvrdnim segmentom.

Bez obzira na rezultat gornjeg uvjetnog grananja, na kraju metode *_wait_for_ack*, varijablu *_rWindow_size* postavljamo na manju od dvije veličine prozora (klizni prozor i prozor zakrčenosti) i vraćamo vrijednost *False*.

```
def _wait_for_ack(self):
    reader, _, _ = select.select([self._sock], [], [])
    if self._resolving_timer_callback_flag:
        return False
    data, addr = self._sock.recvfrom(self._mss + header_size)
    if addr != self.rAddr:
        return False
    self._resend_segments_timer.cancel()
    self._resend_segments_timer = Timer(1, self._timer_says_resend_segments)
    seq_num, rAck_num, _, _, _ = self._unpack_data(data)
    if self._last_received_ack == rAck_num:
        self._next_segment = self._n_of_acked_segments
        self._congestion_window_factor = max(4,
        ↪  self._congestion_window_factor/2)
        self._slow_start_treshold = self._congestion_window_factor
        self._connection_error_counter = 0
    else:
        try:
            i = self._expected_acks.index(rAck_num)
            if self._last_received_ack > rAck_num:
                self._last_received_ack -= 4294967296
            n_of_acked_segments_incr = round((rAck_num -
            ↪  self._last_received_ack)/self._mss)
            self._n_of_acked_segments += n_of_acked_segments_incr
            self._last_received_ack = rAck_num
            if rAck_num == self._seq_num:
                self._set_ack_num(seq_num)
                return True
```

```
27              self._expected_acks = self._expected_acks[(i+1):]
28              self._connection_error_counter = 0
29              if self._congestion_window_factor <
                ↪  self._max_congestion_window_factor:
30                  if self._congestion_window_factor > self._slow_start_treshold:
31                      self._congestion_window_factor += 1
32                  else:
33                      self._congestion_window_factor += n_of_acked_segments_incr
34          except:
35              pass
36      self._rWindow_size = min(self._rWindow_size,
        ↪  self._congestion_window_factor*(self._mss + header_size))
37
38      return False
```

Metoda *recvall* vraća bajt objekt koji predstavlja podatke koje šalje pošiljatelj. Na početku metode postavljamo $\_buffer$ i $\_temp\_buffer$ na inicijalne vrijednosti. $\_temp\_buffer$ je rječnik čiji su indeksi slijedni brojevi, a vrijednosti uređeni parovi pripadnih podataka te vrijednost koju tumačimo kao broj segmenata unutar tih podataka. Odnosno, kada primimo segment koji nema slijedni broj sljedećeg po redu segmenta kojeg očekujemo, taj segment spremamo u $\_temp\_buffer$ na sljedeći način $\_temp\_buffer[slijedni\_broj] = (korisni\_teret, 1)$. Druga vrijednost u uređenom paru (inicijalno postavljena na 1) potrebna nam je jer ćemo u određenim trenutcima pregledavati ovaj rječnik i usput ćemo spajati korisne terete koji po slijednom broju dolaze jedan iza drugoga (ako takvi postoje). U tom trenutku zbrajat ćemo tu drugu vrijednost uređenog para kako bismo pratili od koliko različitih segmenata je taj jedan korisni teret sastavljen.

Varijable $last\_seq$ i $last\_segment$ koristit će nam u trenutku kada zadnji segment (onaj s aktiviranom PSH zastavicom) stigne prije nekog drugog segmenta, tj. kada segmenti pristignu u krivom poretku ili kada su neki segmenti izgubljeni. $\_connection\_error\_counter$ također postavljamo na inicijalnu vrijednost. Odmah pokrećemo mjerač vremena jer postoji mogućnost da je prvi prozor podataka izgubljen ili da je drugi uređaj prekinuo vezu. Na ovaj način nećemo zapeti u blokirajućoj funkciji. Nakon toga ulazimo u *while* petlju.

Iz istog razloga kao i u metodi *sendall* koristimo modul *select*. Prije čitanja pristiglih podataka uvjeravamo se da trenutno ne izvršavamo funkciju koja je pozvana istekom mjerača vremena. Nakon što pročitamo pristigle podatke, provjeravamo je li ih poslao uređaj (i proces) s kojim smo uspostavili vezu, ako nije odbacujemo ih. Ako su podaci pristigli s poznate adrese, zaustavljamo mjerač vremena $\_send\_ack\_timer$. Ovaj mjerač vremena koristimo kada ne šaljemo potvrdni segment istog trenutka kada primimo segment od pošiljatelja već čekamo da primimo $send\_ack\_after\_n\_segments\_recv$ segmenata. No, kako ne želimo da pošiljatelj misli da su uspješno primljeni segmenti izgubljeni koristimo ovaj mjerač vremena koji nakon svog isteka šalje potvrdni segment, iako smo primili manje od $send\_ack\_after\_n\_segments\_recv$ segmenata. Svaki puta kada primimo segment s poznate adrese, postavljamo $\_connection\_error\_counter$ vrijednost na 0. Zatim, pozivamo već spomenutu metodu $\_unpack\_data$ i analiziramo vrijednosti koje vraća ta metoda.

Ako trenutni segment ima aktiviranu PSH zastavicu tada imamo dvije mogućnosti. Ako je to sljedeći po redu segment koji očekujemo to znači da smo primili sve segmente i tada pozivamo metodu $\_handle\_last\_segment\_recv$ kojoj prosljeđujemo korisni teret i slijedni broj zadnjeg segmenta. Ova metoda vraća sve do sada primljene podatke, tj. varijablu $\_buffer$ koju tada vraćamo kao rezultat metode *recvall*. Ako segment s aktiviranom PSH

zastavicom nije sljedeći segment koji očekujemo, njegov slijedni broj i korisni teret spremamo u ranije spomenute varijable *last_seq* i *last_segment*.
Kada segment nema aktiviranu PSH zastavicu imamo dvije opcije. Prva je opcija da je segment koji smo primili segment koji očekujemo (sljedeći po redu). U ovom slučaju, zaustavljamo mjerač *_send_duplicate_ack_timer* i nadodajemo korisni teret segmenta na kraj varijable *_buffer*. Ažuriramo *_ack_num* i povećavamo vrijednost varijable *_ack_counter*. Kada primljeni segment nije onaj koji očekujemo, spremamo ga unutar rječnika *_temp_buffer*. Ako mjerač vremena *_send_duplicate_ack_timer* nije pokrenut i ako trenutno nemamo segmenata za koje nismo poslali potvrdu (*_ack_counter* = 0), pokrećemo ga. U oba ova slučaja (kada nije aktivirana PSH zastavica), nakon navedenih radnji pozivamo metodu *_check_temp_buffer* koja provjerava nalazi li se unutar rječnika *_temp_buffer* segment s idućim po redu slijednim brojem. Ako takav segment postoji, on se briše iz rječnika i dodaje u varijablu *_buffer* uz ažuriranje vrijednosti varijabli *_ack_counter* i *_ack_num*. Na kraju još provjeravamo je li trenutni potvrdni broj (idući segment koji očekujemo) jednak vrijednosti *last_seq*. Ako je ovo istina, to znači da smo primili sve podatke i tada pozivamo metodu *_handle_last_segment_recv* i završavamo izvršavanje metode *recvall*.

Ako nismo još primili sve segmente tada na kraju iteracije *while* petlje provjeravamo vrijednost *_ack_counter* (broj primljenih, ali nepotvrđenih segmenata). Ako je ova vrijednost pozitivna, ali i dalje manja od varijable *send_ack_after_n_segments_recv* tada pokrećemo mjerač vremena *_send_ack_timer*. Ako je vrijednost *_ack_counter* veća ili jednaka vrijednosti *send_ack_after_n_segments_recv*, tada pozivamo metodu *_handle_sending_ack*.

```
def recvall(self):
    self._buffer = bytes()
    self._temp_buffer = {}
    last_seq = -1
    last_segment = None
    self._connection_error_counter = 0
    self._send_duplicate_ack_timer_flag = True
    self._send_duplicate_ack_timer = Timer(2, self._timer_says_send_ack)
    self._send_duplicate_ack_timer.start()
    while True:
        readers, _, _ = select.select([self._sock], [], [])
        if self._resolving_timer_callback_flag:
            continue
        data, addr = self._sock.recvfrom(self._mss + header_size)
        if addr != self.rAddr:
            continue
        self._send_ack_timer.cancel()
        self._connection_error_counter = 0
        seq_num, _, flags, _, payload = self._unpack_data(data)
        if flags[1] == '1' and seq_num == self._ack_num:
            return self._handle_last_segment_recv(payload, seq_num)
        elif flags[1] == '1' and seq_num != self._ack_num:
            last_seq = seq_num
            last_segment = payload
        else:
            if seq_num == self._ack_num:
```

```
                self._send_duplicate_ack_timer_flag = False
                self._send_duplicate_ack_timer.cancel()
                self._buffer += payload
                self._set_ack_num(seq_num + len(payload))
                self._ack_counter += 1
            else:
                if not self._send_duplicate_ack_timer_flag and
                ↪  self._ack_counter == 0:
                    self._send_duplicate_ack_timer = Timer(0.8,
                    ↪  self._timer_says_send_ack)
                    self._send_duplicate_ack_timer_flag = True
                    self._send_duplicate_ack_timer.start()
                self._temp_buffer[seq_num] = [payload, 1]

            self._check_temp_buffer()
            if self._ack_num == last_seq:
                    return self._handle_last_segment_recv(last_segment,
                    ↪  last_seq)

        if self._ack_counter < self.send_ack_after_n_segments_recv and
        ↪  self._ack_counter > 0:
            self._send_ack_timer = Timer(0.5, self._timer_says_send_ack)
            self._send_ack_timer.start()
        elif self._ack_counter >= self.send_ack_after_n_segments_recv:
            self._handle_sending_ack()
```

Metoda *_handle_sending_ack* zadužena je za slanje potvrdnog segmenta u regularnoj situaciji. Najprije zaustavljamo mjerač vremena zadužen za ponovno slanje potvrdnog segmenta. Zatim, postavljamo veličinu prozora na određenu veličinu, ova vrijednost ovisi o tome koliko segmenata potvrđujemo s trenutnim potvrdnim brojem. Inkrementiramo vlastiti slijedni broj te stvaramo zaglavlje potvrdnog segmenta. Brojač primljenih segmenata postavljamo na nulu i zatim šaljemo potvrdni segment. Na kraju pokrećemo mjerač vremena koji će ponovno poslati ovaj potvrdni segment ukoliko u međuvremenu ne primimo segment sa slijednim brojem koji je jednak potvrdnom broju ovog potvrdnog segmenta.

```
def _handle_sending_ack(self):
    self._send_duplicate_ack_timer_flag = False
    self._send_duplicate_ack_timer.cancel()
    self._send_duplicate_ack_timer = Timer(0.8, self._timer_says_send_ack)
    self._window_size = min(self._ack_counter * (self._mss + header_size),
    ↪  65535)
    self._increment_seq_num(1)
    ack_header = self._create_header(self._seq_num, self._ack_num, 16,
    ↪  self._window_size)
    self._ack_counter = 0
    self._sock.sendto(ack_header, self.rAddr)
    self._send_duplicate_ack_timer_flag = True
    self._send_duplicate_ack_timer.start()
```

Metoda *_handle_last_segment_recv* prima korisni teret i slijedni broj posljednjeg segmenta (zadnji segment u pošiljateljevoj listi *_segments*) te vraća *_buffer* tj. sve do sada

primljene podatke (uključujući podatke iz zadnjeg segmenta). Svi mjerači vremena se zaustavljaju. Inkrementiramo vlastiti slijedni broj, nadodajemo zadnje podatke na kraj varijable $\_buffer$ te računamo potvrdni broj zadnjeg potvrdnog segmenta. Stvaramo zaglavlje zadnjeg potvrdnog segmenta i šaljemo ga "pošiljatelju", zatim ažuriramo varijablu $\_ack\_num$ i vraćamo $\_buffer$.

```
def _handle_last_segment_recv(self, payload, seq_num):
    self._send_duplicate_ack_timer_flag = False
    self._send_ack_timer.cancel()
    self._send_duplicate_ack_timer.cancel()
    self._increment_seq_num(1)
    self._buffer += payload
    last_ack = seq_num + len(payload)
    if last_ack > 4294967295:
        last_ack -= 4294967296
    ack_header = self._create_header(self._seq_num, last_ack, 16,
    ↪  self._window_size)
    self._sock.sendto(ack_header, self.rAddr)
    self._set_ack_num(last_ack)
    return self._buffer
```

6 Zaključak

Za transportni sloj moglo bi se reći da je temelj modernih računalnih mreža. Zahvaljujući ovom temelju imamo pouzdan prijenos podataka između udaljenih uređaja povezanih unutar mreže. Glavni protokoli transportnog sloja, UDP i TCP, brinu o efikasnoj, poštenoj i robusnoj razmjeni podataka. TCP protokol naglašava pouzdanost, ispravnost i kontrolu toka, dok se UDP protokol fokusira na brzinu prijenosa podataka i jednostavnost uz maleni omjer kontrolnih informacija prema korisnom teretu.

Detaljnom analizom transportnog sloja i protokola koji se ovdje koriste postaje jasna potreba za kontinuiranim unaprjeđivanjem ovog dijela računalne mreže kako bi se osigurao kvalitetan mrežni promet. Neki problemi, poput zakrčenosti mreže, i dalje nisu u potpunosti riješeni. Kao što smo vidjeli, protokoli poput TCP protokola imaju ključnu ulogu u rješavanju ovog problema i osiguravanju brzog, sigurnog, poštenog i pouzdanog toka podataka.

Literatura

[1] A. S. Tanenbaum, D. J. Wetherall, Computer Networks, Pearson Education, Harlow, 2014.

[2] F. Halsall, Computer Networking and the Internet, Pearson Education, Harlow, 2005.

[3] Postel, J. "User Datagram Protocol." Www.rfc-Editor.org, 1980, www.rfc-editor.org/rfc/rfc768.

[4] Postel, J. Transmission Control Protocol. Sept. 1981, https://doi.org/10.17487/rfc0793.

[5] M. Allman, et al. TCP Congestion Control. Sept. 2009, https://doi.org/10.17487/rfc5681.

[6] V. Paxson, and M. Allman. Computing TCP's Retransmission Timer. 1 June 2011, https://doi.org/10.17487/rfc6298.

[7] Socket- low-level networking interface Python documentation, https://docs.python.org/3/library/socket.html.

[8] https://github.com/rerceg/ReliableUDP

Sažetak

U ovom radu bavimo se transportnim slojem računalne mreže. Analiziramo usluge transportnog sloja, te protokole transportnog sloja. U prvom poglavlju pojašnjavamo gdje se unutar mrežne arhitekture nalazi transportni sloj. U drugom poglavlju detaljno opisujemo usluge i elemente transportnog sloja te probleme s kojima se suočavamo u ovom sloju. U trećem poglavlju dajemo dva primjera protokola transportnog sloja, UDP i TCP. Detaljno opisujemo kako ovi protokoli obavljaju zadatke transportnog sloja i kako izgleda razmjena podataka između dva transportna sloja na dva različita uređaja. U zadnjem poglavlju dajemo primjer nadogradnje UDP protokola kako bi isti imao određene dodatne funkcionalnosti poput pouzdanosti i kontrole toka.

Ključne riječi

OSI referentni model, TCP/IP, transportni sloj, protokol, veza, TCP, UDP, zaglavlje, kontrola toka, zakrčenost mreže, mjerač vremena, mrežna propusnost, pouzdanost

Computer networks: Transport layer

Summary

In this paper, we deal with the transport layer of the computer network. We analyze transport layer services and transport layer protocols. In the first chapter, we explain where is the transport layer located within the network architecture. In the second chapter, we describe in detail the services and elements of the transport layer and the problems we face in this layer. In the third chapter, we give two examples of transport layer protocols, UDP and TCP. We describe in detail how these protocols perform the tasks of the transport layer and what the data exchange looks like between the two transport layers on two different host devices. In the last chapter, we give an example of upgrading the UDP protocol so that it has certain additional functionalities such as reliability and flow control.

Keywords

OSI reference model, TCP/IP, transport layer, protocol, connection, TCP, UDP, header, flow control, network congestion, timer, bandwidth, reliability

Životopis

Rođen sam 6. listopada 1999. godine u Osijeku. Pohađao sam Osnovnu školu Ljudevita Gaja u Osijeku. Nakon osnovne škole upisujem III. gimnaziju Osijek. Nakon završene srednje škole, upisujem preddiplomski studij Matematike i računarstva na Odjelu za matematiku na Sveučilištu Josipa Jurja Strossmayera u Osijeku. Godine 2021. stječem naziv sveučilišni prvostupnik matematike i računarstva. Na jesen iste godine upisujem sveučilišni diplomski studij matematike, smjer Matematika i računarstvo, na Odjelu za matematiku.