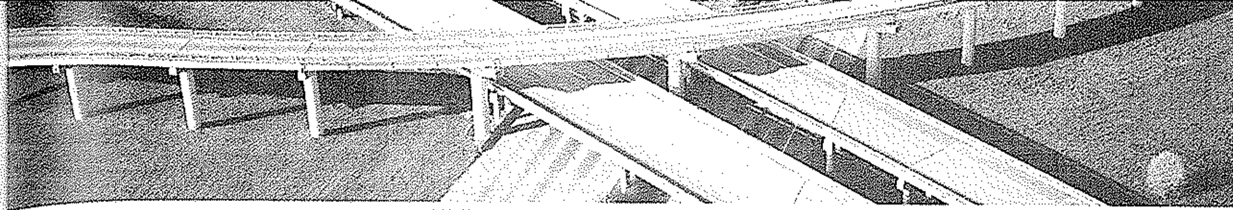


Transportni sloj

Smešten između aplikativnog i mrežnog sloja, transportni sloj predstavlja centralni deo slojevite mrežne arhitekture. Njegova najvažnija uloga je neposredno obezbeđivanje komunikacionih usluga procesima aplikacija koji se izvršavaju na različitim računarima. Pedagoški pristup koji koristimo u ovom poglavlju podrazumeva naizmenično izlaganje o principima transportnog sloja i o načinu na koji se ti principi realizuju u postojećim protokolima. Kao i obično, naglasak se stavlja na protokole interneta, a posebno na protokole transportnog sloja TCP i UDP.

Počinjemo opisivanjem odnosa između transportnog i mrežnog sloja. To nam omogućava da ispitamo jedan od ključnih zadataka transportnog sloja – proširivanje usluge isporuke između dva krajnja sistema, koja se obavlja na mrežnom sloju, tako što se ostvari usluga isporuke između dva procesa na aplikativnom sloju, koji se izvršavaju na tim krajnjim sistemima. Ovaj zadatak transportnog sloja prikazaćemo kada budemo opisivali transportni protokol interneta bez uspostavljanja veze, protokol UDP.

Potom se vraćamo opštijim temama i suočavamo se sa jednim od najosnovnijih problema umrežavanja računara – kako omogućiti pouzdanu komunikaciju preko medijuma na kome može doći do gubitaka ili oštećenja podataka. Nizom sve složenijih (i stvarnih!) primera sklopićemo skup tehnika koje transportni protokoli koriste za rešavanje ovih problema. Zatim ćemo prikazati kako su ova pravila ugrađena u TCP, transportni protokol interneta sa uspostavljanjem veze.

Potom prelazimo na drugi osnovni problem umrežavanja – kontrolu brzine prenosa na transportnom sloju – kako bi se izbeglo zagušenje u mreži, ili omogu-

ćio oporavak nakon zagušenja. Razmotrićemo uzroke i posledice zagušenja, kao i uobičajene tehnike za kontrolu zagušenja. Kada ovladamo znanjima koja se tiču kontrole zagušenja, proučavamo rešenje za kontrolu zagušenja, koje se koristi u protokolu TCP.

3.1 Uvod i usluge transportnog sloja

U prethodna dva poglavlja pomenuli smo ulogu transportnog sloja i usluge koje on nudi. Ukratko ćemo ponoviti ono što smo o transportnom sloju već naučili.

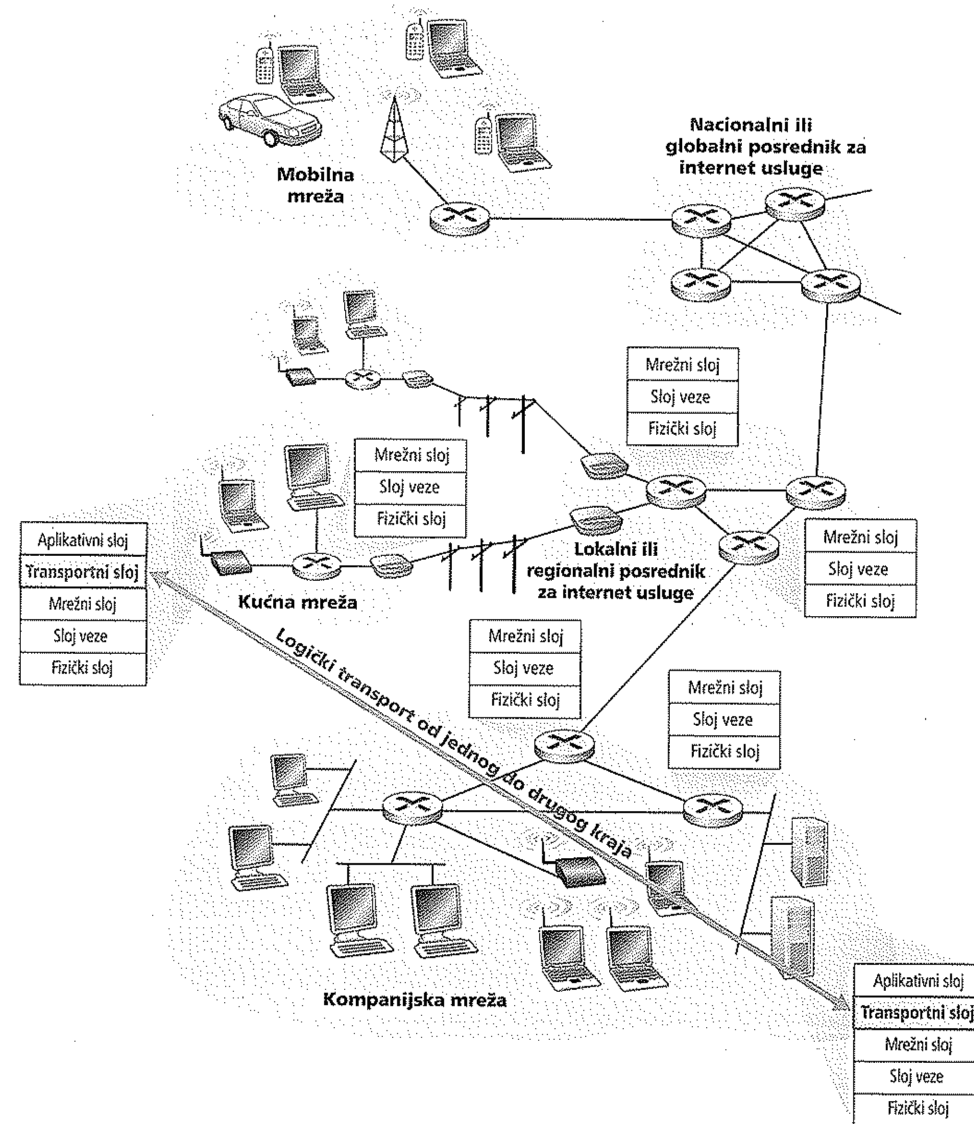
Protokol transportnog sloja obezbeđuje **logičku komunikaciju** između procesa aplikacija, koje se izvršavaju na različitim računarima. Pod *logičkom komunikacijom* podrazumevamo to što sa stanovišta aplikacije izgleda kao da su računari na kojima se procesi izvršavaju neposredno povezani, a u realnosti računari se mogu nalaziti na suprotnim krajevima planete, povezani preko niza rutera i različitih vrsta linkova. Procesi aplikacija koriste logičku komunikaciju koju obezbeđuje transportni sloj za međusobnu razmenu poruka, oslobođeni brige o detaljima fizičke infrastrukture koja se koristi za prenošenje tih poruka. Značenje logičke komunikacije prikazano je na slici 3.1.

Kao što se vidi na slici 3.1, protokoli transportnog sloja realizuju se na krajnjim sistemima, a ne u mrežnim ruterima. Na predajnoj strani, transportni sloj pretvara poruku aplikativnog sloja, koju dobija od procesa aplikacije pošiljaoca u pakete transportnog sloja, u internet terminologiji poznate pod nazivom **segmenti** transportnog sloja. To se vrši (po potrebi) razbijanjem poruke aplikacije na manje delove i dodavanjem zaglavlja transportnog sloja u sve te delove, tako da se dobiju segmenti transportnog sloja. Transportni sloj zatim prenosi te segmente do mrežnog sloja predajnog krajnjeg sistema, gde se ti segmenti enkapsuliraju u pakete mrežnog sloja (datagrame) i šalju ka odredištu. Važno je uočiti da mrežni ruteri obrađuju samo polja datagrama mrežnog sloja; to jest, ne ispituju polja segmenta transportnog sloja koji su enkapsulirani unutar datagrama. Na prijemnoj strani, mrežni sloj iz datagrama izvlači segment transportnog sloja i prenosi ga do transportnog sloja. Transportni sloj zatim obrađuje primljeni segment i podatke iz segmenta dostavlja prijemnoj aplikaciji.

Mrežnim aplikacijama dostupno je više protokola transportnog sloja. Na primer, internet ima dva protokola – TCP i UDP. Svaki od ovih protokolâ, aplikaciji koja ih koristi, obezbeđuje drugačije usluge transportnog sloja.

3.1.1 Odnos između transportnog i mrežnog sloja

Sećate se da u skupu protokola transportni sloj leži neposredno iznad mrežnog sloja. Dok protokol transportnog sloja obezbeđuje logičku komunikaciju između *procesâ* koji se izvršavaju na različitim računarima, protokol mrežnog sloja obezbeđuje logičku komunikaciju između *računarâ*. Razlika je jedva primetna, ali veoma značajna. Objasnimo ovu razliku sličnošću sa običnim domaćinstvima.



Slika 3.1 ◈ Transportni sloj nudi logičku, a ne fizičku, komunikaciju između procesa aplikacija

Uzmimo dve kuće, jednu na Istočnoj, a drugu na Zapadnoj obali, pri čemu u svakoj od tih kuća živi po dvanaestoro dece. Deca u kući na Istočnoj obali su rođaci dece u kući na Zapadnoj obali. Deca vole da se dopisuju – svako dete, svake nedelje piše po jedno pismo svakom rođaku, a svako pismo se otprema u zasebnoj koverti, običnom poštom. Tako se iz svake kuće svake nedelje šalje 144 pisma onoj drugoj kući. (Ova deca bi uštedela silne novce kada bi imala e-poštu!) U svakoj od ovih kuća, po jedno dete – Ana na Zapadnoj obali i Bil na Istočnoj – zaduženo je za prikupljanje i slanje pisama. Svake nedelje Ana obilazi svu svoju braću i sestre, prikuplja poštu i predaje je poštaru, koji svakoga dana obilazi kuću. Kada pisma stignu u kuću na Zapadnoj obali, Anin zadatak je da podeli poštu svojoj braći i sestrama. Na Istočnoj obali, iste te poslove obavlja Bil.

U ovom primeru, poštanska služba obezbeđuje logičku komunikaciju između dve kuće – poštanska služba prenosi pisma od kuće do kuće, a ne od osobe do osobe. S druge strane, Ana i Bil obezbeđuju logičku komunikaciju između rođaka – Ana i Bil preuzimaju i isporučuju pisma svojoj braći i sestrama. Obratite pažnju – sa stanovišta rođaka, Ana i Bil *jesu* poštanska služba, iako su samo deo (krajnji deo) isporuke pisama s kraja na kraj. Ovaj primer predstavlja lepo poređenje kojim objašnjavamo kako se transportni sloj odnosi prema mrežnom sloju:

poruke aplikacija = pisma u kovertama
procesi = rođaci
računari (takođe se zovu i krajnji sistemi) = kuće
protokol transportnog sloja = Ana i Bil
protokol mrežnog sloja = poštanska služba (uključujući i poštare)

Nastavimo li sa ovim poređenjem, uočavamo da Ana i Bil sav svoj posao obavljaju u vlastitim kućama; oni nisu uključeni, na primer, u razvrstavanje pisama u nekom usputnom poštanskom centru, niti u prenošenje pisama iz jednog centra u drugi. Isto važi za protokole transportnog sloja koji se nalaze u krajnjim sistemima. Unutar krajnjeg sistema transportni protokol prenosi poruke od procesa aplikacija do ivice mreže (tj. do mrežnog sloja) i obratno, ali ni na koji način ne utiče na način prenošenja poruke unutar same mreže. U suštini, kao što se vidi na slici 3.1, usputni ruteri ne koriste, niti prepoznavaju, informacije koje je transportni sloj dodao porukama aplikacije.

Nastavimo našu porodičnu sagu i pretpostavimo da, kad Ana i Bil odu na odmor drugi par rođaka – recimo, Suzan i Harvi – menja njih i skuplja i isporučuje pisma unutar njihovih kuća. Nažalost, Suzan i Harvi ne prikupljaju i ne isporučuju pisma baš isto kao Ana i Bil. Pošto su mlađi, Suzan i Harvi prikupljaju i dele pisma ređe, a povremeno i izgube poneko pismo (koje izgrizu njihovi psi). Stoga, Suzan i Harvi ne pružaju iste usluge (tj. isti model uslugâ) kao Ana i Bil. Slično tome, računarska mreža nudi različite transportne protokole i svaki od tih protokolâ aplikacijama nudi drugačiji model usluga.

Usluge koje su Ana i Bil u stanju da ponude očigledno su ograničene mogućim uslugama, koje nudi poštanska služba. Na primer, ako poštanska služba ne garantuje za koliko vremena će najduže da isporuči pisma između dve kuće (na primer, tri dana), tada Ana i Bil ni na koji način ne mogu garantovati koliko će rođaci najviše čekati na isporuku. Slično tome, usluge koje transportni protokol može da obezbedi obično su ograničene modelom usluga protokola mrežnog sloja, koji se nalazi ispod njega. Ako protokol mrežnog sloja ne garantuje kašnjenje ili propusni opseg za segmente transportnog sloja, koji se šalju između računara, onda ni protokol transportnog sloja ne može da garantuje kašnjenje niti propusni opseg za poruke aplikacija, koje se šalju između procesa.

Međutim, transportni protokol *može* da ponudi neke usluge čak i kada mrežni protokol od koga zavisi ne nudi odgovarajuću uslugu na mrežnom sloju. Na primer, kao što ćemo videti u ovom poglavlju, transportni protokol može da aplikaciji ponudi uslugu pouzdanog prenosa podataka, čak i kada mrežni protokol ispod njega nije naročito pouzdan, tj. čak i kada mrežni protokol gubi, oštećuje i duplira pakete. Drugi primer (koji obrađujemo u poglavlju 8 prilikom razmatranja bezbednosti mrežâ) bio bi da transportni protokol može da koristi šifrovanje kojim se garantuje da poruke aplikacijâneće čitati neovlašćena lica, čak i ako mrežni sloj ne može da garantuje poverljivost za segmenate transportnog sloja.

3.1.2 Kratak pregled transportnog sloja na internetu

Sećate se da internet, kao i svaka TCP/IP mreža, aplikativnom sloju nudi dva potpuno različita transportna protokola. Jedan od tih protokola je **UDP** (User Datagram Protocol), koji aplikaciji koja ga koristi nudi nepouzdanu uslugu bez uspostavljanja veze. Drugi protokol je **TCP** (Transmission Control Protocol), koji aplikaciji koja ga koristi nudi pouzdanu uslugu sa uspostavljanjem veze. Pri projektovanju mrežne aplikacije, programer mora da izabere jedan od ova dva transportna protokola. Kao što smo videli u odeljku 2.7, programer bira između protokola UDP ili TCP prilikom izrade soketa.

Da bi terminologija bila jednostavnija, kada govorimo o internetu, pakete transportnog sloja nazivamo *segmentima.* Pomenućemo, međutim, da se u internet literaturi (na primer u RFC dokumentima) za pakete transportnog sloja protokola TCP koristi izraz segmenti, dok se za pakete protokola UDP koristi izraz datagrami. Ista ta internet literatura takođe koristi izraz *datagram* i za paket mrežnog sloja! Smatramo da ćemo u uvodnoj knjizi o umrežavanju računara, kakva je ova, sprečiti zabunu, ako pakete oba protokola, TCP i UDP, zovemo segment, a ostavimo izraz *datagram* za pakete mrežnog sloja.

Pre nego što pređemo na kratak uvod u protokole UDP i TCP, biće na od pomoći kraće izlaganje o mrežnom sloju interneta. (Mrežni sloj podrobno je obrađen u poglavlju 4.) Protokol mrežnog sloja interneta se zove IP – skraćenica od Internet Protocol. Protokol IP obezbeđuje logičku komunikaciju između računara. Model

usluga koje nudi protokol IP je **usluga najboljeg pokušaja isporuke**. To znači da protokol IP čini „sve što može" da isporuči segmente između računara, ali *ne daje nikakvu garanciju*. Tačnije, ne garantuje isporuku segmenta, ne garantuje redosled isporuke segmenata i ne garantuje integritet podataka u segmentima. Zato se za protokol IP kaže da je **nepouzdana usluga**. Pomenućemo takođe da svaki računar ima bar jednu adresu mrežnog sloja, takozvanu IP adresu. U poglavlju 4 podrobno razmatramo IP adresiranje; za sada je jedino važno upamtiti da *svaki računar ima IP adresu*.

Nakon ovog kratkog prikaza modela usluga protokola IP, opisaćemo ukratko modele usluga koje pružaju protokoli UDP i TCP. Najvažniji zadatak protokola UDP i TCP je da uslugu isporuke protokola IP između dva krajnja sistema prošire na uslugu isporuke između dva procesa koji se izvršavaju na krajnjim sistemima. Proširivanje isporuke od računara do računara na isporuku od procesa do procesa naziva se **multipleksiranje** i **demultipleksiranje transportnog sloja**. Multipleksiranje i demultipleksiranje transportnog sloja razmatramo u sledećem odeljku. Protokoli UDP i TCP takođe obezbeđuju proveravanje integriteta, tako što u zaglavljima segmenata postoji polje za otkrivanje grešaka. Ove dve osnovne usluge transportnog sloja – isporuka podataka od procesa do procesa i provera grešaka – jedine su dve usluge koje pruža protokol UDP! Tačnije, slično protokolu IP, i protokol UDP je nepouzdana usluga – ne garantuje da će podaci koje šalje jedan proces stići neoštećeni (ili stići uopšte!) do odredišnog procesa. Protokol UDP podrobno razmatramo u odeljku 3.3.

S druge strane, protokol TCP nudi aplikacijama nekoliko dodatnih usluga. Prvo i najvažnije, nudi **pouzdan prenos podataka**. Koristeći kontrolu toka, redne brojeve, potvrdu prijema i tajmere (tehnike koje podrobno obrađujemo u ovom poglavlju), protokol TCP obezbeđuje da se podaci od predajnog procesa do prijemnog procesa isporuče tačno i u ispravnom redosledu. Na ovaj način, TCP pretvara nepouzdanu uslugu protokola IP između krajnjih sistema, u pouzdanu uslugu za prenos podataka između procesa. Protokol TCP takođe obezbeđuje **kontrolu zagušenja**. Kontrola zagušenja je pre svega usluga korisna internetu kao celini, kao usluga za opšte dobro, a ne toliko usluga koja se pruža aplikaciji, koja je poziva. Slobodno govoreći, kontrola zagušenja protokola TCP sprečava da neka TCP veza pretrpa podacima linkove i rutere, koji se nalaze između računara koji komuniciraju. Protokol TCP nastoji da svim TCP vezama, koje prolaze preko zagušenog mrežnog linka, dodeli jednak deo propusnog opsega. Ovo se postiže regulisanjem brzine kojom TCP veza na strani pošiljaoca šalje podatke u mrežu. Nasuprot tome, protokol UDP ne reguliše brzinu slanja podataka. Aplikacija koja koristi UDP transport podatke može da šalje proizvoljnom brzinom, dokle god to hoće.

Protokol koji obezbeđuje pouzdan prenos podataka i kontrolu zagušenja mora da bude složen. Biće nam potrebno nekoliko odeljaka za opisivanje pouzdanog prenosa podataka i kontrole zagušenja, kao i dodatni odeljci za opis samog protokola TCP. Ove teme obrađene su u odeljcima od 3.4 do 3.8. U ovom poglavlju naizmenično izlažemo: osnovne pojmove, a zatim njihovu primenu u protokolu TCP. Na primer, prvo razmatramo pouzdan prenos podataka u opštem slučaju, a zatim način na koji protokol TCP obezbeđuje pouzdan prenos podataka. Slično tome, prvo razmatramo kontrolu zagušenja u opštem slučaju, a zatim način na koji se u protokolu TCP obavlja kontrola zagušenja. Ali, pre nego što se upustimo u sve to, razmotrimo najpre multipleksiranje i demultipleksiranje transportnog sloja.

3.2 Multipleksiranje i demultipleksiranje

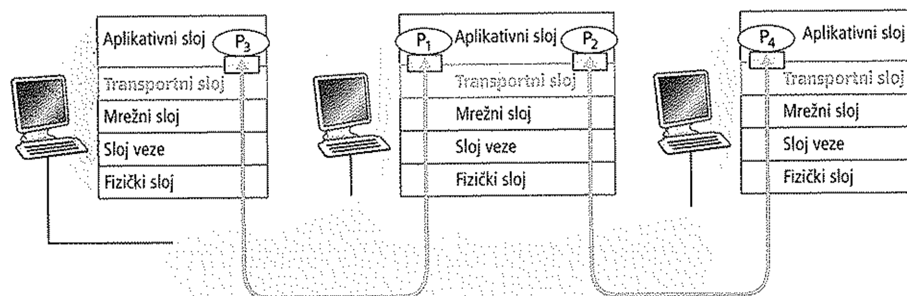
U ovom odeljku opisujemo multipleksiranje i demultipleksiranje transportnog sloja, to jest, proširivanje usluge isporuke od računara do računara, koju pruža mrežni sloj, šireći se na uslugu isporuke od procesa do procesa, za aplikacije koje se izvršavaju na računarima. Da bi naše razmatranje bilo jasnije, razmotićemo u okviru interneta ovu osnovnu uslugu transportnog sloja. Naglašavamo, međutim, da je usluga multipleksiranja i demultipleksiranja potrebna svim računarskim mrežama.

Na odredišnom računaru, transportni sloj prima segmente od mrežnog sloja koji se nalazi neposredno ispod njega. Transportni sloj zadužen je da isporuči podatke iz tih segmenata odgovarajućem procesu aplikacije koja se izvršava na određenom računaru. Pogledajmo jedan primer: uzmimo da sedite pred računarom i da preuzimate veb stranice, dok se istovremeno izvršavaju: jedna FTP sesija i dve Telnet sesije. Prema tome, pokrenuli ste četiri procesa mrežnih aplikacija – dva Telnet procesa, jedan FTP proces i jedan HTTP proces. Kada transportni sloj na vašem računaru primi podatke od mrežnog sloja ispod, on mora da usmeri primljene podatke jednom od ova četiri procesa. Sada ćemo videti kako se to radi.

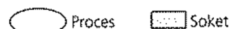
Pre svega, sećate se iz odeljka 2.7 da proces (kao deo mrežne aplikacije) ima jedan ili više **soketa** koji predstavljaju vrata kroz koja prolaze podaci iz mreže prema procesu i obratno. Prema tome, kako je prikazano na slici 3.2, transportni sloj na prijemnom računaru u suštini ne isporučuje podatke neposredno određenom procesu, već posrednom soketu. Pošto u svakom trenutku na prijemnom računaru može da postoji više soketa, svaki od njih ima jedinstven identifikator. Format indentifikatora zavisi od toga da li je reč o UDP ili TCP soketu, o čemu uskoro govorimo.

Razmotrimo sada kako prijemni računar usmerava dolazni segment transportnog sloja u odgovarajući soket. Svi segmenti transportnog sloja u tu svrhu imaju skup poljâ. Na prijemnom kraju, transportni sloj ispituje ova poljâ, da bi odredio prijemni soket i zatim usmerava segment u taj soket. Ovaj zadatak isporučivanja podataka iz segmenta transportnog sloja u odgovarajući soket naziva se **demultipleksiranje**. Zadatak prikupljanja delova podataka na izvornom računaru iz različitih soketa, enkapsuliranje svakog dela dodavanjem zaglavlja, (koje će se kasnije koristiti za demultipleksiranje) da bi se napravili segmenti i predavanje tih segmenata mrežnom sloju, naziva se **multipleksiranje**. Obratite pažnju na to da transportni sloj na srednjem računaru, na slici 3.2, mora da demultipleksira segmente koje dobija iz mrežnog sloja ispod sebe, da bi ih predao procesu P_1 ili P_2 iznad njega; to se postiže usmeravanjem podataka iz pristiglih segmenata na soket odgovarajućeg

procesa. Transportni sloj na srednjem računaru mora takođe da prikuplja odlazne podatke iz tih soketâ, da formira segmente transportnog sloja i te segmente preda naniže, ka mrežnom sloju. Iako smo multipleksiranje i demultipleksiranje upoznali u okviru transportnih protokola interneta, bitno je da shvatite da su oni važni uvek kada jedan protokol na nekom sloju (transportnom ili bilo kom drugom sloju) koristi više protokola iz narednog, višeg sloja.



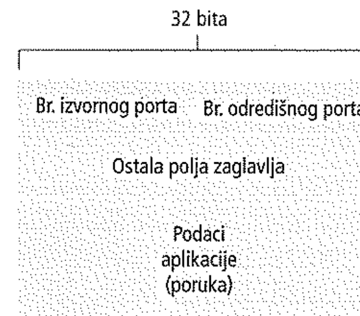
Legenda:



Slika 3.2 ◈ Multipleksiranje i demultipleksiranje na transportnom sloju

Da bismo bolje pojasnili demultipleksiranje, setite se poređenja sa domaćinstvom iz prethodnog odeljka. Svako dete prepoznajemo po imenu. Kada Bil primi gomilu pisama od poštara, on postupak demultipleksiranja obavlja tako što utvrđuje kome su pisma upućena i lično uručuje pisma svojoj braći i sestrama. Ana obavlja multipleksiranje kada prikuplja pisma od braće i sestara i predaje ih poštaru.

Pošto smo shvatili uloge multipleksiranja i demultipleksiranja na transportnom sloju, pogledajmo kako se to zaista odvija na računaru. Iz prethodnog razmatranja znamo da je za multipleksiranje na transportnom sloju potrebno: (1) da soketi imaju jedinstvene identifikatore i (2) da svaki segment sadrži posebna poljâ, koja naznačavaju soket u koji bi određeni segment trebalo isporučiti. Ta, posebna poljâ, prikazana su na slici 3.3 – **polje broja izvornog porta** i **polje broja odredišnog porta**. (UDP i TCP segmenti sadrže i druga poljâ, koja razmatramo u sledećim odeljcima ovog poglavlja.) Broj porta je 16-bitni broj, u rasponu od 0 do 65535. Brojevi portova od 0 do 1023 nazivaju se **dobro poznatim brojevima portova** i njihova upotreba je ograničena, što znači da su rezervisani za protokole opšte poznatih aplikacija, kao što su HTTP (koji koristi broj porta 80) i FTP (koji koristi broj porta 21). Spisak dobro poznatih brojeva portova može se naći u dokumentu RFC 1700 i u ažurnijoj verziji na adresi http://www.iana.org [RFC 3232]. Kada razvijamo novu aplikaciju (kao što je to učinjeno u odeljku 2.7), toj aplikaciji moramo dodeliti broj porta.



Slika 3.3 ◈ Polja brojeva izvornog i odredišnog porta u segmentu transportnog sloja

Sada bi trebalo da je jasno kako transportni sloj *može* da ostvari uslugu demultipleksiranja: svakom soketu na računaru dodeljuje se broj porta, a kada neki segment stigne na taj računar transportni sloj ispituje broj odredišnog porta u segmentu i taj segment usmerava na odgovarajući soket. Podaci iz segmenta prolaze kroz soket ka odgovarajućem procesu. Kao što ćemo videti, protokol UDP u suštini tako i radi. Međutim, kao što ćemo takođe videti, multipleksiranje i demultipleksiranje kod protokolaTCP je mnogo složenije.

Multipleksiranje i demultipleksiranje bez uspostavljanja veze

Sećate se iz odeljka 2.7.1 da *Python*, program koji se izvršava na računaru, pravi UDP soket linijom koda:

```
clientSocket=socket(socket.AF_INET,socket.SOCK_DGRAM)
```

Kada se na ovaj način napravi UDP soket, transportni sloj soketu automatski dodeljuje broj porta. Tačnije, transportni sloj dodeljuje broj porta, u rasponu od 1024 do 65535, koji trenutno ne koristi nijedan drugi UDP port na računaru; ili, možemo da dodamo red u naš *Python* program, nakon što kreiramo soket za povezivanje određenog broja porta, (recimo 19157) sa ovim UDP soketom pomoću metode `socket bind ()`:

```
clientSocket.bind(('',19157))
```

Kada programer piše kôd za serversku stranu „dobro poznatog protokola“ trebalo bi da joj dodeli odgovarajući, dobro poznati broj porta. Obično, klijentska strana

aplikacije dozvoljava da transportni sloj automatski (i transparentno) dodeljuje broj porta, dok se na serverskoj strani aplikacije dodeljuje tačno određeni broj porta.

Pošto smo UDP soketima dodelili brojeve portova, možemo precizno da opišemo UDP multipleksiranje i demultipleksiranje. Pretpostavimo da proces na računaru A, sa brojem UDP porta 19157, želi da pošalje deo podataka neke aplikacije procesu sa UDP portom broj 46428, na računaru B. Transportni sloj na računaru A pravi segment transportnog sloja koji obuhvata podatke iz odgovarajuće aplikacije, broj izvornog porta (19157), broj odredišnog porta (46428) i još dve vrednosti (o kojima govorimo kasnije, ali su nevažne za trenutnu priču). Transportni sloj zatim predaje dobijeni segment mrežnom sloju. Mrežni sloj enkapsulira ovaj segment u IP datagram i daje sve od sebe, (najbolji pokušaj) da taj segment isporuči prijemnom računaru. Ako segment stigne do prijemnog računara B, transportni sloj na prijemnom računaru ispituje broj odredišnog porta u segmentu (46428) i isporučuje segment u odgovarajući soket, određen portom 46428. Trebalo bi napomenuti da računar B možda izvršava više procesâ, od kojih svaki ima vlastiti UDP soket sa odgovarajućim brojem porta. Kako UDP segmenti pristižu sa mreže, računar B usmerava (demultipleksira) segmente do odgovarajućeg soketa, ispitujući brojeve odredišnog porta u segmentima.

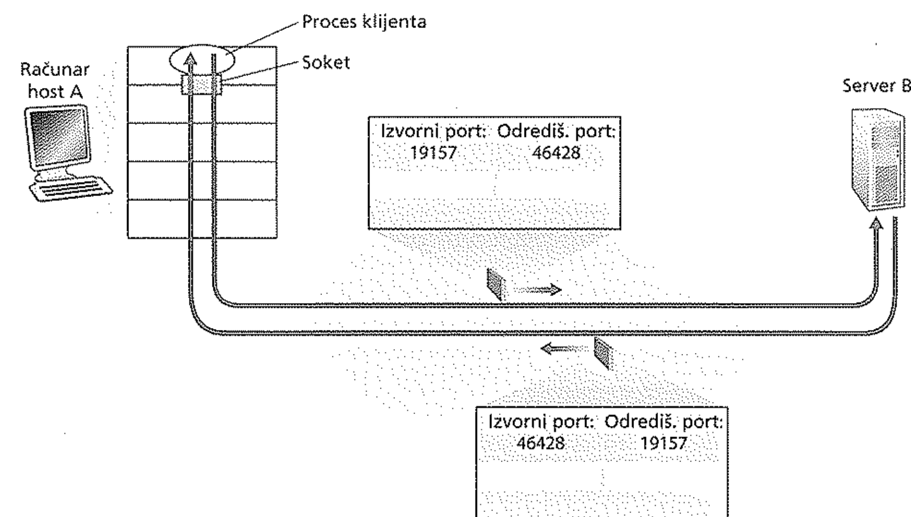
Važno je primetiti da je UDP soket potpuno određen dvodelnim podatkom koji se sastoji od odredišne IP adrese i broja odredišnog porta. Prema tome, ako dva UDP segmenta imaju različite izvorne IP adrese i/ili različite brojeve izvornog porta, a *istu* odredišnu IP adresu i broj *odredišnog* porta, ta dva segmenta biće usmerena ka istom odredišnom procesu preko istog odredišnog soketa.

Možda se sada čudite čemu služi broj izvornog porta. Kao što je prikazano na slici 3.4, u segmentu od A ka B broj izvornog porta služi kao deo „povratne adrese" – kada računar B želi da pošalje segment računaru A, odredišni port u segmentu od B ka A uzima vrednost izvornog porta iz segmenta od A ka B. (Potpuna adresa za odgovor sastoji se od IP adrese računara A i broja izvornog porta.) Kao primer za ovo može poslužiti program UDP servera koji smo proučavali u odeljku 2.7. U programu `UDPServer.py`, server koristi metod `recvfrom ()`, kojim izvlači broj (izvornog) porta klijentske strane iz segmenta koji je primio od klijenta; zatim šalje klijentu novi segment u kojem se dobijeni broj izvornog porta koristi kao broj odredišnog porta u ovom novom segmentu.

Multipleksiranje i demultipleksiranje sa uspostavljanjem veze

Da bismo shvatili TCP demultipleksiranje, moramo bolje da upoznamo TCP sokete i uspostavljanje TCP veze. Najočiglednija razlika između TCP soketa i UDP soketa je u tome što se TCP soket prepoznaje pomoću četvorodelne oznake (izvorna IP adresa, broj izvornog porta, odredišna IP adresa i broj odredišnog porta). Prema tome, kada TCP segment pristigne sa mreže na neki računar, taj računar koristi sve četiri vrednosti da bi usmerio (demultipleksirao) taj segment na odgovarajući soket. Tačnije, nasuprot UDP segmentima, dva pristigla TCP segmenta sa različitim izvornim IP adresama ili različitim brojevima izvornih portova usmeriće se (izuzimajući

TCP segmente koji nose početni zahtev za uspostavljanje veze) na dva različita soketa. Da bismo dobili još bolji uvid, razmotrimo ponovo primer programiranja TCP klijenta/servera iz odeljka 2.7.2:



Slika 3.4 ◈ Inverzija brojeva izvornog i odredišnog porta

- TCP serverska aplikacija ima „prijemni soket" koji čeka zahteve za uspostavljanje veze, koji stižu od TCP klijenata (slika 2.29) na broju porta 12000.
- TCP klijent kreira soket i šalje segment zahteva za uspostavljanje veze redom koda:

```
clientSocket= socket(AF_INET,SOCK_STREAM)
clientSocket.connect((serverName,12000))
```

- Zahtev za uspostavljanje veze nije ništa drugo nego TCP segment sa brojem odredišnog porta 12000 i posebnim bitom za uspostavljanje veze koji je postavljen u TCP zaglavlju (razmatramo u odeljku 3.5). Segment takođe sadrži broj izvornog porta koji je izabrao klijent.
- Kada operativni sistem računara na kome se izvršava serverski proces primi dolazni segment sa zahtevom za uspostavljanje veze sa odredišnim portom 12000, on pronalazi serverski proces koji čeka uspostavljanje veze na portu broj 12000. Serverski proces tada pravi novi soket:

```
connectionSocket, addr = serverSocket.accept()
```

- Takođe, transportni sloj na serveru beleži sledeće četiri vrednosti iz segmenta sa zahtevom za uspostavljanje veze: (1) broj izvornog porta iz segmenta, (2) IP adresu izvornog računara, (3) broj odredišnog porta iz segmenta i (4) vlastitu IP adresu. Novi, napravljeni soket veze identifikuje se pomoću te četiri vrednosti; svi, ubuduće pristigli segmenti čiji se izvorni port, izvorna IP adresa, odredišni port i odredišna IP adresa poklapaju sa ove četiri vrednosti, biće demultipleksirani na ovaj soket. Pošto je sada uspostavljena TCP veza, klijent i server mogu da šalju podatke jedan drugome.

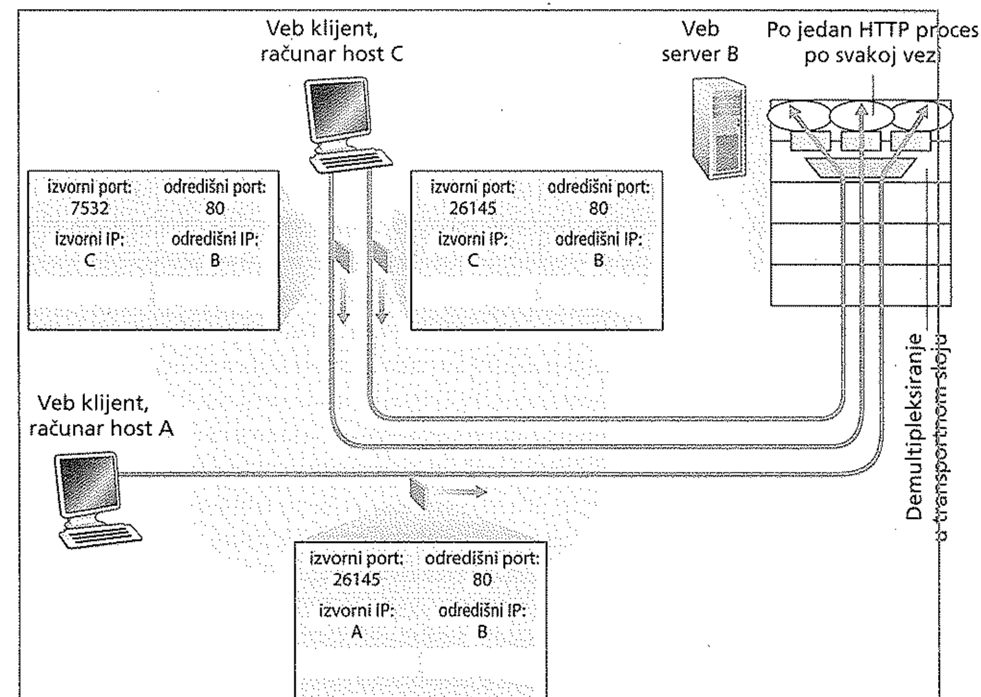
Serverski računar može da podrži više istovremenih TCP soketa, pri čemu su svi soketi pridruženi nekom procesu, a svaki soket prepoznaje se po sopstvenoj četvorodelnoj oznaci. Kada TCP segment stigne na računar, koriste se sva četiri poljâ (izvorna IP adresa, izvorni port, odredišna IP adresa, odredišni port), da bi se segment usmerio (demultipleksirao) u odgovarajući soket.

BEZBEDNOST PRE SVEGA

SKENIRANJE PORTOVA

Videli smo da serverski proces strpljivo čeka na otvorenom portu da klijent uspostavi vezu sa njim. Neki portovi su rezervisani za dobro poznate aplikacije (npr. veb, FTP, DNS i SMTP serveri); druge portove obično koriste popularne aplikacije (npr. *Microsoft* 2000 SQL server čeka zahteve na UDP portu 1434). Stoga, ukoliko ustanovimo da je na računaru neki port otvoren, možemo da povežemo taj port sa određenom aplikacijom, koja se izvršava na tom računaru. Ovo je veoma korisno administratorima sistema, koji obično žele da znaju koje se mrežne aplikacije izvršavaju na računarima u njihovoj mreži. Ali i napadači, kako bi „iskoristili gužvu", takođe žele da znaju koji portovi su otvoreni na ciljanom računaru. Ukoliko pronađu računar na kome se izvršava aplikacija sa poznatim bezbednosnim propustom (npr. SQL server koji čeka zahteve na portu 1434 omogućava napad, punjenje bafera, što omogućava da udaljeni korisnik izvršava proizvoljni kôd na nezaštićenom računaru, propust koji je iskoristio crv Slammer [CERT 2003-04]), onda je taj računar odlična meta za napad.

Određivanje na kojim portovima da određene aplikacije očekuju, relativno je jednostavan zadatak. U stvari, postoji više javno dostupnih programa, nazvanih skeneri portova, koji rade upravo to. Možda najviše korišćen među njima je program *nmap*, dostupan besplatno na adresi http://insecure.org/nmap, a koji je sastavni deo većine *Linux* distribucija. Za protokol TCP program *nmap* redom skenira portove, tražeći portove koji prihvataju TCP vezu. Za protokol UDP, program nmap takođe redom skenira portove, tražeći UDP portove koji odgovaraju na prenete UDP segmente. U oba slučaja, program *nmap* vraća spisak otvorenih, zatvorenih i nedostupnih portova. Računar na kome se izvršava program *nmap* može da pokuša da skenira bilo koji ciljani računar *bilo gde* na internetu. Program *nmap* ponovo obrađujemo u odeljku 3.5.6, kada razmatramo upravljanje TCP vezama.



Slika 3.5 ◈ Dva klijenta koji koriste isti broj odredišnog porta (80) za komunikaciju sa istom aplikacijom veb servera

Ovakav slučaj prikazan je na slici 3.5, na kojoj je računar C uspostavio dve HTTP sesije sa serverom B, a računar A je uspostavio jednu HTTP sesiju sa serverom B. Računari A i C i server B imaju vlastite jedinstvene IP adrese – A, C i B, respektivno. Računar C dodeljuje dva različita broja (26145 i 7532) izvornih portova svojim dvema HTTP vezama. Pošto računar A bira brojeve izvornih portova nezavisno od računara C, on takođe može da dodeli broj izvornog porta 26145 svojoj HTTP vezi. Ali, to nije problem – server B ipak može pravilno da demultipleksira dve veze sa istim brojem izvornog porta, zato što te dve veze imaju različite izvorne IP adrese.

Veb serveri i TCP

Pre zaključka trebalo bi reći još nešto o veb serverima i načinu na koji oni koriste brojeve portova. Uzmimo na primer računar na kome se izvršava veb server, kao što je veb server *Apache*, na portu 80. Kada klijenti (na primer pretraživači) šalju serveru segmente, *svi* segmenti imaju 80 kao odredišnibroj porta. Tačnije, segment za početno uspostavljanje veze i segmenti koji sadrže HTTP poruke zahteva imaju 80 kao odredišnibroj porta. Kao što smo upravo rekli, server razlikuje segmente od različitih klijenata po izvornim IP adresama i brojevima izvornih portova.

Na slici 3.5 je prikazan veb server koji pravi novi proces za svaku vezu. Kao što je prikazano na slici 3.5, svi ovi procesi imaju sopstvene sokete vezâ kroz koje pristižu HTTP zahtevi i šalju se HTTP odgovori. Napominjemo, međutim, da broj soketa nije uvek jednak broju procesâ. U stvari, savremeni veb serveri visokih performansi obično koriste samo jedan proces, a prave novu nît sa novim soketom veze za sve nove veze sa klijentima. (Nît se može posmatrati kao oslabljeni potproces.) Ako ste uradili prvi programerski zadatak u poglavlju 2, napravili ste veb server koji upravo tako radi. Sa takvim serverom, u bilo kom trenutku, moguće je imati više soketa veze (sa različitim identifikatorima), povezanih sa istim procesom.

Ako klijent i server koriste postojanu HTTP vezu, onda tokom trajanja te postojane veze klijent i server razmenjuju HTTP poruke kroz isti soket servera. Međutim, ako klijent i server koriste nepostojanu HTTP vezu, onda se nova TCP veza pravi i zatvara za svaki zahtev i odgovor, tako da se novi soket pravi i kasnije zatvara za svaki zahtev i odgovor. To često pravljenje i zatvaranje soketa može veoma loše da utiče na performanse opterećenog veb servera (mada se za ublažavanje ovog problema može koristiti niz trikova sa operativnim sistemom). Čitaoci zainteresovani za teme koje se tiču operativnog sistema u vezi sa postojanim i nepostojanim HTTP vezama trebalo bi da pročitaju [Nielsen 1997; Nahum 2002].

Pošto smo razmotrili multipleksiranje i demultipleksiranje transportnog sloja, prelazimo na razmatranje jednog od transportnih protokola interneta, protokola UDP. U sledećem odeljku videćemo da protokol UDP protokolu mrežnog sloja dodaje maltene samo uslugu multipleksiranja i demultipleksiranja.

3.3 Prenos bez uspostavljanja veze: protokol UDP

U ovom odeljku podrobnije razmatramo protokol UDP – kako radi i šta radi. Poželjno je da ponovo pročitate odeljak 2.1, u kome je dât prikaz modela uslugâ protokola UDP i odeljak 2.7.1 u kome se obrađuje programiranje soketa korišćenjem protokola UDP.

Za početak naše rasprave o protokolu UDP, pretpostavimo da želite da projektujete ogoljeni transportni protokol, bez suvišnih dodataka. Kako biste mogli to da uradite? Prvo biste mogli da razmotrite korišćenje naizgled, besmislenog transportnog protokola. Tačnije, ovaj protokol bi radio tako što bi se na strani pošiljaoca poruke preuzimale iz procesa određene aplikacije i prenosile pravo na mrežni sloj, dok bi se na strani primaoca poruke, pristigle iz mrežnog sloja, prenosile pravo u proces aplikacije. Međutim, kao što smo naučili u prethodnom odeljku, ipak moramo da uradimo bar malo više od toga! Najmanje što transportni sloj mora da obezbedi jeste usluga multipleksiranja i demultipleksiranja za prenošenje podataka između mrežnog sloja i odgovarajućeg procesa na aplikativnom sloju.

Protokol UDP, definisan u dokumentu [RFC 768], obavlja najmanje što transportni protokol uopšte može da uradi. Osim poslova multipleksiranja i demultipleksiranja i najosnovnije provere grešaka, on protokolu IP ne dodaje ništa više. U suštini, ako programer neke aplikacije izabere da koristi protokol UDP umesto protokola TCP, onda se ta aplikacija skoro neposredno obraća protokolu IP. UDP uzima poruke od procesa aplikacije, pridružuje joj polja sa brojevima izvornog i odredišnog porta koji se koriste za multipleksiranje i demultipleksiranje, dodaje još dva manja poljâ i tako dobijeni segment predaje mrežnom sloju. Mrežni sloj enkapsulira ovaj segment transportnog sloja u IP datagram i zatim na najbolji mogući način nastoji da taj segment isporuči prijemnom računaru. Ako segment stigne do prijemnog računara, UDP koristi broj odredišnog porta, kako bi podatke iz segmenta isporučio procesu odgovarajuće aplikacije. Napominjemo da kod protokola UDP pre slanja segmenta nema usklađivanja za uspostavljanje veze između predajnihi prijemnih entiteta transportnog sloja. Zato se za UDP kaže da je protokol *bez uspostavljanja veze.*

DNS je primer protokola aplikativnog sloja koji obično koristi UDP. Kada DNS aplikacija na nekom računaru želi da postavi upit, ona pravi DNS poruku upita i tu poruku predaje protokolu UDP. Bez prethodnog usklađivanja sa UDP entitetom koji se izvršava u odredišnom krajnjem sistemu, klijentska strana protokola UDP poruci upita dodaje poljâ zaglavlja i tako dobijeni segment predaje mrežnom sloju. Mrežni sloj enkapsulira ovaj UDP segment u datagram i šalje ga serveru imena. DNS aplikacija na računaru, koji je postavio upit, zatim čeka odgovor na upit. Ako ne dobije odgovor (možda zato što je mreža izgubila upit ili odgovor), DNS aplikacija pokušava da pošalje upit drugom serveru imena, ili obaveštava aplikaciju koja ju je pozvala, da ne može da dobije odgovor.

Možda vas čudi zašto programeri uopšte grade aplikacije, koristeći UDP umesto protokola TCP. Zar nije uvek bolje izabrati TCP, s obzirom da TCP nudi uslugu pouzdanog prenosa podataka, a UDP je ne nudi? Odgovor je ne, jer mnogim aplikacijama UDP više odgovara iz sledećih razloga:

- *Bolja kontrola na nivou aplikacije toga šta se šalje i kada se šalje.* Ako se koristi UDP, čim proces aplikacije preda podatke protokolu UDP, UDP će spakovati podatke u UDP segment i taj segment odmah predati mrežnom sloju. S druge strane, TCP ima mehanizam za kontrolu zagušenja koji zadržava TCP slanje na transportnom sloju, kada linkovi između izvorišnog i odredišnog računara postanu preterano zagušeni. TCP će takođe više puta da šalje isti segment, dokle god od odredišta ne dobije potvrdu da je segment primljen, bez obzira na to koliko dugo takva, pouzdana isporuka traje. Pošto aplikacije u realnom vremenu obično zahtevaju neku najmanju brzinu slanja i ne žele preterano da odlažu prenošenje segmenta, a mogu da podnesu manji gubitak podataka, model usluga protokola TCP nije naročito pogodan za potrebe takvih aplikacijâ. Kao što ćemo kasnije objasniti, ove aplikacije mogu da koriste UDP i da, u okviru same aplikacije, obave sve dodatne zadatke, koji su im neophodni, pored osnovne usluge isporuke segmenata, koju nudi protokol UDP.

- *Nema uspostavljanja veze.* Kao što ćemo objasniti kasnije, TCP, pre nego što počne sa prenošenjem podataka, koristi trostruko usaglašavanje. UDP jednostavno kreće bez ikakve ceremonijalne najave. Zato kod protokola UDP nema kašnjenja zbog uspostavljanja veze. To je verovatno glavni razlog zbog kojeg se DNS usluga izvršava preko protokola UDP, a ne preko protokola TCP – DNS usluga bi bila mnogo sporija, ukoliko bi se izvršavala preko protokola TCP. HTTP koristi TCP, a ne UDP, pošto je pouzdanost izuzetno značajna veb stranicama sa tekstom. Ipak, kao što smo ukratko napomenuli u odeljku 2.2, kašnjenje zbog uspostavljanja TCP veza kod protokola HTTP značajno doprinosi ukupnom kašnjenju koje postoji prilikom preuzimanja veb dokumenata.
- *Nema stanja veze.* TCP održava stanje veze na krajnjim sistemima. Ovo stanje veze obuhvata bafere za primanje i slanje, parametre za kontrolu zagušenja, redne brojeve i brojeve potvrde prijema. Videćemo u odeljku 3.5 da su ove informacije o stanju neophodne za realizaciju pouzdanog prenosa podataka i za obezbeđenje kontrole zagušenja. Nasuprot tome, UDP ne održava stanje veze i ne vodi računa o bilo kom od ovih parametara. Zato server namenjen određenoj aplikaciji obično može da podrži mnogo više aktivnih klijenata, ako se aplikacija izvršava preko protokola UDP, umesto preko protokola TCP.
- *Malo dodatno zaglavlje paketa.* TCP segment ima zaglavlje od 20 dodatnih bajtova, dok UDP ima samo 8 dodatnih bajtova.

Na slici 3.6 navedene su popularne internet aplikacije i transportni protokoli koje one koriste. Kao što bismo i očekivali: e-pošta, pristup udaljenim terminalima, veb i prenos datoteka izvršavaju se protokolom TCP – svim ovim aplikacijama neophodna je usluga pouzdanog prenosa podataka protokola TCP. Ipak, mnoge važne aplikacije izvršavaju se protokolom UDP, a ne protokolom TCP. UDP se koristi za ažuriranje RIP tabelâ rutiranja (pogledajte odeljak 4.6.1). S obzirom da se ažuriranja RIP tabela šalju povremeno (obično svakih 5 minuta), izgubljena ažuriranja se zamenjuju svežijim, tako da su izgubljena i zastarela ažuriranja beskorisna. UDP se takođe koristi za prenos podataka za upravljanje mrežom (SNMP; pogledajte poglavlje 9). U ovom slučaju UDP je bolji u odnosu na TCP, zato što aplikacije za upravljanje mrežom obično moraju da se izvršavaju, kada je mreža u vanrednom stanju – upravo kada je teško postići pouzdan prenos podataka uz kontrolu zagušenja. Osim toga, kako smo već pomenuli, DNS se izvršava preko protokola UDP, čime se izbegava kašnjenje zbog uspostavljanja TCP veza.

Kao što je prikazano na slici 3.6, oba protokola, i UDP i TCP, danas se koriste za multimedijalne aplikacije, kao što su: telefoniranje preko interneta, video konferencije u realnom vremenu i protok snimljenih audio i video zapisa. Ove aplikacije podrobnije razmatramo u poglavlju 7. Za sada samo napominjemo da sve te aplikacije mogu da podnesu manje gubitke paketâ, pa pouzdani prenos podataka nije apsolutno neophodan za uspešan rad aplikacije. Štaviše, aplikacije u realnom vremenu, kao što su internet telefoniranje i video konferencije, veoma loše podnose kontrolu zagušenja protokola TCP. Zato programeri multimedijalnih aplikacija često odluče

da im se aplikacije izvršavaju korišćenjem protokola UDP, umesto protokola TCP. Međutim, TCP se sve više koristi za prenos multimedijalnih zapisa. Na primer, u radu [Sripanidkulchai 2004] se tvrdi da se u skoro 75% protoka multimedijalnih zapisa uživo i na zahtev koristi TCP. U slučajevima kada je broj izgubljenih paketâ mali i kada neke organizacije blokiraju UDP saobraćaja iz bezbednosnih razloga (pogledajte poglavlje 8), TCP postaje sve zanimljiviji protokol za prenos multimedijalnih zapisa.

Aplikacija	Protokol aplikativnog sloja	Transportni protokol koji koriste
E-pošta	SMTP	TCP
Pristupanje udaljenim terminalima	Telnet	TCP
Veb	HTTP	TCP
Transfer datotekâ	FTP	TCP
Udaljeni server datotekâ	NFS	najčešćeUDP
Protok multimedijalnih zapisâ	najčešće vlasnički	UDPiliTCP
Internet telefoniranje	najčešće vlasnički	UDPiliTCP
Upravljanje mrežom	SNMP	najčešćeUDP
Protokol rutiranja	RIP	najčešćeUDP
Prevođenje imenâ	DNS	najčešćeUDP

Slika 3.6 ◆ Popularne internet aplikacije i transportni protokoli na kojima se zasnivaju

Iako se danas često koristi, izvršavanje multimedijalnih aplikacija preko protokola UDP je u izvesnoj meri sporno. Kao što smo već napomenuli, UDP nema kontrolu zagušenja. Međutim, kontrola zagušenja je neophodna, da bi se sprečilo da mreža dođe u stanje u kojem se malo šta korisno može uraditi. Kada bi svi velikom brzinom protoka pokrenuli prenos video zapisa u realnom vremenu, bez ikakve kontrole zagušenja, ruteri bi bili preplavljeni paketima, tako da bi samo manji broj UDP paketâ uspešno prešao put od izvora do odredišta. Štaviše, veliki broj izgubljenih paketâ od strane nekontrolisanih UDP pošiljalaca doveo bi do toga da TCP pošiljaoci (koji, kao što ćemo videti, suočeni sa zagušenjem *smanjuju* svoje brzine prenosa) značajno smanje svoje brzine. Prema tome, posledica nedostatka kontrole zagušenja kod protokola UDP može da bude veliki broj izgubljenih paketâ između UDP pošiljaoca i primaoca, kao i istiskivanje uspostavljenih TCP sesijâ, što predstavlja potencijalno veliki problem [Floyd 1999]. Mnogi istraživači predlažu nove mehanizme kojima bi se svim izvorima, pa i UDP izvorima, nametnula prilagodljiva kontrola zagušenja [Mahdavi 1997; Floyd 2000; Kohler 2006; RFC 4340].

Pre nego što pređemo na opis strukture UDP segmenta, napominjemo da *jeste* moguće da aplikacija dobije pouzdan prenos podataka, kada koristi UDP. To se može postići, ako se pouzdanost ugradi u samu aplikaciju (na primer, dodavanjem

mehanizama za potvrđivanje prijema i ponovno slanje, poput onih koje proučavamo u sledećem odeljku). Ali, ovo nije prost zadatak i programer bi se dugo bavio ispravljanjem grešaka. Ipak, ugrađivanje pouzdanosti neposredno, u samu aplikaciju, omogućava da ona zadrži „i jare i pare". Drugim rečima, procesi aplikacije mogu da ostvare pouzdanu komunikaciju, ne trpeći ograničenja brzine slanja, koje nameće mehanizam za kontrolu zagušenja protokola TCP.

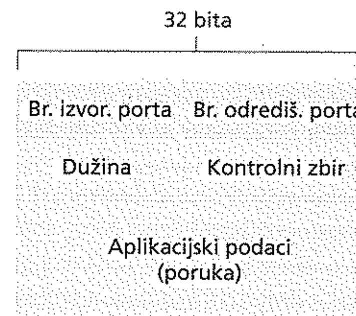
3.3.1 Struktura UDP segmenta

Struktura UDP segmenta, prikazana na slici 3.7, definisana je u dokumentu RFC 768. Podaci aplikacije zauzimaju polje podataka UDP segmenta. Na primer, za DNS, polje podataka sadrži, ili poruku upita, ili poruku odgovora. Za audio aplikaciju sa protokom u realnom vremenu, polje podataka popunjava se audio zapisima. UDP zaglavlje ima samo četiri polja, od kojih svako zauzima dva bajta. Kao što je rečeno u prethodnom odeljku, brojevi portova omogućavaju odredišnom računaru da podatke aplikacije prosledi odgovarajućem procesu koji se izvršava na odredišnom krajnjem sistemu (tj. da izvrši funkciju demultipleksiranja). Polje dužine navodi broj bajtova u UDP segmentu (zaglavlje plus podaci). Tačna vrednost dužine je potrebna, jer veličina polja sa podacima može da se razlikuje od jednog do drugog UDP segmenta. Kontrolni zbir služi prijemnom računaru, da proveri da li je u segmentu došlo do grešaka. Iskreno govoreći, kontrolni zbir izračunava se korišćenjem i nekoliko polja iz IP zaglavlja, a ne samo nad UDP segmentom. Zanemarićemo ovu pojedinost kako bismo videli šumu kroz drveće. U nastavku opisujemo izračunavanje kontrolnog zbira. Osnovna pravila koja se primenjuju za otkrivanje greške opisana su u odeljku 5.2. Polje dužine određuje dužinu UDP segmenta u bajtovima, uključujući i zaglavlje.

3.3.2 UDP kontrolni zbir

UDP kontrolni zbir služi za otkrivanje greške. Drugim rečima, kontrolni zbir se koristi kako bi se utvrdilo da li su bitovi unutar UDP segmenta promenjeni (na primer, zbog smetnji na linkovima, ili dok se segment nalazio u ruteru) prilikom prenosa od izvora do odredišta. UDP na strani pošiljaoca izračunava komplement jedinice za sumu svih 16-bitnih reči u segmentu, pri čemu se prekoračenjâ, do kojih dođe prilikom sabiranja, dodaju bitu najmanje težine. Ovaj rezultat se stavlja u polje kontrolnog zbira UDP segmenta. Ovde dajemo jednostavan primer izračunavanja kontrolnog zbira. Podrobniji opis izračunavanja možete naći u dokumentu RFC 1071, a primere sa stvarnim podacima u [Stone 1998; Stone 2000]. Kao primer, pretpostavimo da imamo sledeće tri 16-bitne reči:

0110011001100000
0101010101010101
1000111100001100



Slika 3.7 ◈ Struktura UDP segmenta

Zbir prve dve 16 bitne reči je:

0110011001100000
0101010101010101
1011101110110101

Dodavanjem treće reči na ovaj zbir dobijamo:

1011101110110101
1000111100001100
0100101011000010

Obratite pažnju na to da poslednje sabiranje daje prekoračenje, koje je dodato bitu najmanje težine. Komplement jedinice dobija se pretvaranjem svih 0 u 1, a svih 1 u 0. Prema tome, komplement zbira 0100101011000010 je 1011010100111101, što se uzima kao kontrolni zbir. Na prijemnom kraju, sabiraju se sve četiri 16-bitne reči, uključujući i kontrolni zbir. Ako u paketu ne postoje greške, jasno je da će zbir kod primaoca biti 1111111111111111. Ako je neki od bitova jednak 0, znamo da je u paketu došlo do greške.

Možda vas čudi što UDP uopšte koristi kontrolni zbir, kada mnogi protokoli sloja veze (uključujući i popularni protokol Eternet) takođe sadrže proveru grešaka. Razlog leži u tome što ne postoji garancija da svi linkovi od izvora do odredišta obezbeđuju proveru grešaka; to jest, neki od linkova možda koristi protokol sloja veze koji ne nudi proveru grešaka. Štaviše, čak i ukoliko se segmenti tačno prenose duž linkova, moguće je da se pogrešan bit pojavi dok se segment nalazi u memoriji rutera. Pošto se ne garantuju, niti pouzdan prenos između linkova, niti otkrivanje grešaka nastalih u memoriji, UDP mora da obezbedi otkrivanje grešaka na transportnom sloju, *od jednog do drugog kraja*, ako bi usluga prenosa podataka trebalo da obezbedi otkrivanje grešaka od jednog do drugog kraja. Ovo je primer primene slavnog **principa od jednog do drugog kraja** u projektovanju sistema [Saltzer 1984], koje kaže da se izvesni zadatak (otkrivanje grešaka, u ovom slučaju) mora primeniti od jednog do drugog kraja: „funkcionalnosti koje se obavljaju na nižim

nivoima možda su suvišne, ili manje vredne, kada se porede sa tim šta je potrebno uraditi da bi se ostvarile na višem nivou".

Pošto se smatra da protokol IP može da se izvršava preko ma kog protokola sloja 2, korisno je da transportni sloj obezbedi proveru grešaka za svaki slučaj. Iako UDP nudi proveru grešaka, on ništa ne preduzima da bi se nastala greška ispravila. U nekim verzijama protokola UDP oštećeni segment se jednostavno odbacuje; dok druge verzije aplikaciji prosleđuju oštećeni segment uz upozorenje.

Ovim zaključujemo opis protokola UDP. Uskoro ćemo videti da TCP svojim aplikacijama nudi pouzdan prenos podataka, kao i neke druge usluge, koje UDP ne nudi. Naravno, protokol TCP je zato mnogo složeniji od protokola UDP. Pre nego što pređemo na razmatranje protokola TCP, biće korisno da se vratimo za jedan korak i prvo razmotrimo osnovne principe pouzdanog prenosa podataka.

3.4 Principi pouzdanog prenosa podataka

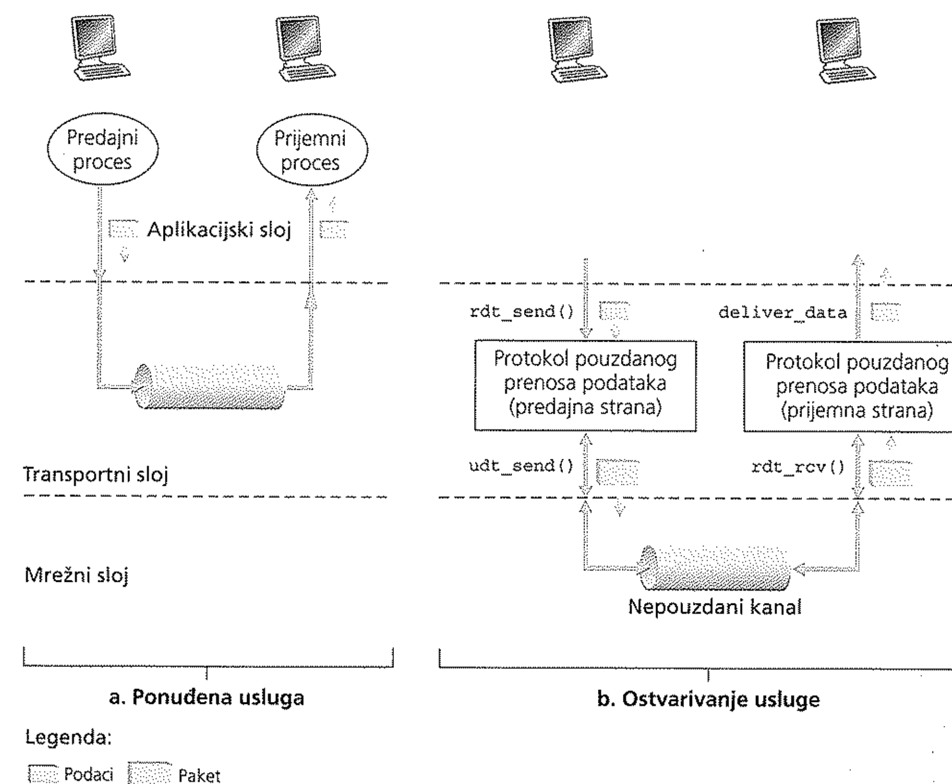
U ovom odeljku razmatramo problem pouzdanog prenosa podataka u opštem smislu. Ovo je korisno jer se problem realizacije pouzdanog prenosa podataka ne javlja samo na transportnom sloju već i na sloju veze i na aplikativnom sloju. Prema tome, taj problem je od ključnog značaja za umrežavanje. Zaista, kada bi trebalo napraviti spisak „prvih 10" najznačajnijih, opštih problema u vezi sa umrežavanjem, ovo bi bio kandidat za prvo mesto. U sledećem odeljku istražićemo protokol TCP i na tom posebnom primeru pokazati kako je u TCP ugrađena većina principa koje ćemo sada opisati.

Na slici 3.8 okvirno je prikazana zamisao pouzdanog prenosa podataka. Uopšteno, usluga koja se nudi entitetima na gornjem sloju je usluga pouzdanog kanala kroz koji se mogu prenositi podaci. Bitovi podataka koji se prenose pouzdanim kanalom se ne oštećuju (nema promena vrednosti iz 0 u 1 ili obrnuto), niti se gube i svi se isporučuju redosledom po kome su poslati. To je upravo model usluga koji protokol TCP nudi internet aplikacijama koje ga koriste.

Protokol pouzdanog prenosa podataka ima zadatak da ostvari ovako zamišljenu uslugu. Zadatak je otežan činjenicom da je sloj *ispod* protokola pouzdanog prenosa podataka možda nepouzdan. Na primer, TCP je protokol pouzdanog prenosa podataka koji se ostvaruje preko nepouzdanog (IP) mrežnog sloja s jednog na drugi kraj. Još uopštenije, sloj ispod dve krajnje tačke koje pouzdano komuniciraju može da se sastoji od samo jednog fizičkog linka (kao u slučaju protokola za prenos podataka na nivou linka) ili više međusobno povezanih mreža (kao u slučaju protokola transportnog sloja). Za sada je, međutim, dovoljno da taj niži sloj posmatramo jednostavno kao nepouzdan kanal od tačke do tačke.

U ovom odeljku postepeno razvijamo predajnu i prijemnu stranu protokola pouzdanog prenosa podataka, uzimajući u obzir sve složenije modele kanala, koji se nalazi nasloju ispod. Na primer, uzećemo u obzir koji mehanizmi protokola su potrebni kada kanal, koji se nalazi na sloju ispod, može da ošteti bitove ili izgubi cele pakete. Pretpostavka koju ćemo usvojiti u ovoj diskusiji je da će paketi biti isporučeni redom kojim se šalju, uz postojanje mogućnosti da se neki paketi izgube; odnosno kanal, koji se

nalazi na sloju ispod, neće promeniti raspored paketa. Na slici 3.8(b) prikazani su interfejsi našeg protokola za prenos podataka. Predajna strana protokola za prenos podataka se pokreće odozgo – pozivom `rdt_send()`. Time se prosleđuju podaci koji bi trebalo da budu isporučeni gornjem sloju na prijemnoj strani. (Ovde rdt znači *reliable data transfer* – pouzdani prenos podataka, a `_send` znači da se poziva predajna strana protokola `rdt`. Prvi korak pri razvoju protokola je biranje dobrog imena!) Na prijemnoj strani, kada paket pristigne iz prijemne strane kanala poziva se `rdt_rcv()`. Kada protokol `rdt` želi da isporuči podatke gornjem sloju, to radi pozivom `deliver_data()`. U buduće koristimo izraz „paket" umesto izraza „segment", koji se koristi natransportnom sloju. Pošto se teorija koju razvijamo u ovom odeljku odnosi uopšteno na sve računarske mreže, a ne samo na transportni sloj interneta, primereniji je opštiji izraz „paket".



Slika 3.8 ◇ Pouzdan prenos podataka: model usluga i ostvarivanje usluga

U ovom odeljku razmatramo samo **jednosmeran prenos podataka**, odnosno prenos podataka od predajne do prijemne strane. Pouzdan **dvosmeran** (tj. puni dupleksni) **prenos podataka** u suštini nije ništa teži, ali ga je znatno teže opisati. Iako razmatramo samo jednosmeran prenos podataka, važno je napomenuti da će predajna i prijemna strana našeg protokola ipak morati da prenose pakete u *oba* smera, kao što je naznačeno na slici 3.8. Uskoro ćemo videti da, osim razmene paketa koji sadrže podatke koje bi trebalo preneti, predajna i prijemna strana protokola `rdt`

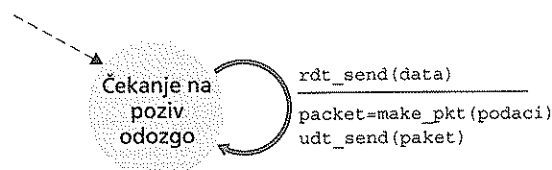
moraju takođe da razmenjuju i kontrolne pakete u oba smera. Obe strane protokola `rdt` šalju pakete drugoj strani pozivanjem `udt_send()` – pri čemu udt znači *unreliable data transfer* – nepouzdan transfer podataka.

3.4.1 Razvoj protokola za pouzdan prenosa podataka

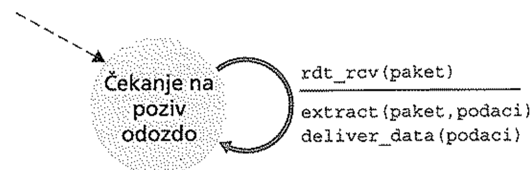
Razmotrićemo niz protokola, svaki složeniji od prethodnog, dok ne stignemo do besprekornog protokola za pouzdan prenos podataka.

Pouzdan prenos podataka preko savršeno pouzdanog kanala: protokol rdt1.0

Prvo razmatramo najjednostavniji slučaj, u kome je kanal koji se nalazi na sloju ispod potpuno pouzdan. Sâm protokol, koji ćemo nazvati `rdt1.0`,jednostavan je. Na slici 3.9 prikazane su definicije **mašine konačnog stanja** (finite-state machine, FSM) pošiljaoca i primaoca za protokol rdt1.0. FSM na slici 3.9(a) prikazuje rad pošiljaoca, dok FSM na slici 3.9(b) prikazuje rad primaoca. Važno je primetiti da za primaoca i pošiljaoca postoje *zasebne* mašine konačnog stanja. FSM pošiljaoca i primaoca na slici 3.9 imaju samo po jedno stanje. Strelice u FSM opisu označavaju prelazak protokola iz jednog stanja u drugo. (Pošto svaki FSM na slici 3.9 ima samo po jedno stanje, prelazak iz jednog stanja u drugo znači povratak u isto stanje; uskoro ćemo videti mnogo složenije dijagrame stanja.) Događaj koji izaziva prelazak prikazan je iznad horizontalne linije koja označava odgovarajući prelazak, a ispod horizontalne linije prikazano je šta se preduzima, kada taj događaj nastupi. Kada se povodom nekog događaja ništa ne preduzima, ili ako se nešto preduzima, a nema



a. rdt1.0: predajna strana



b. rdt1.0: prijemna strana

Slika 3.9 ◆ Protokol rtd1.0 za potpuno pouzdan kanal

događaja koji bi to izazvao, koristićemo simbol Λ ispod, odnosno iznad horizontalne linije, da bi se dodatno naglasilo da se ništa ne preduzima, ili da nema događaja. Početno stanje FSM se označava isprekidanom strelicom. Iako mašine konačnog stanja na slici 3.9 imaju samo po jedno stanje, uskoro ćemo videti mašine konačnog stanja sa više stanja, pa je važno prepoznati njihovo početno stanje.

Predajna strana protokola `rdt` prihvata podatke iz gornjeg sloja događajem `rdt_send(data)`, pravi paket koji sadrži te podatke (postupkom `make_pkt(data)`) i šalje paket u kanal. U praksi, događaj `rdt_send(data)` nastaje tako što aplikacija iz gornjeg sloja poziva neku proceduru (recimo, proceduru `rdt_send()`).

Na prijemnoj strani, `rdt` prima paket iz kanala ispod sebe događajem `rdt_rcv(data)`, izvlači podatke iz paketa (postupkom `extract (packet,data)`) i prenosi podatke gornjem sloju (postupkom `deliver_data(data)`). U praksi, događaj `rdt_rcv(paket)` nastaje tako što protokol nižeg sloja poziva odgovarajuću proceduru (recimo, proceduru `rdt_rcv()`).

U ovom jednostavnom protokolu, nema razlike između jedinice podataka i paketa. Osim toga, sav protok paketa je od pošiljaoca prema primaocu; sa savršeno pouzdanim kanalom nije potrebno da prijemna strana šalje povratnu informaciju pošiljaocu, jer do greške ne može doći! Obratite pažnju da smo, takođe, pretpostavili da je primalac u stanju da prihvata podatke istom brzinom kojom ih pošiljalac šalje. Prema tome, nema potrebe da primalac zahteva od pošiljaoca da uspori!

Pouzdan prenos podataka preko kanala sa greškama na bitovima: protokol rdt2.0

Realniji model kanala ispod je onaj u kome se bitovi u paketu mogu oštetiti. Takve greške obično nastaju u fizičkim komponentama mreže prilikom prenosa i prostiranja paketa, ili dok se paketi nalaze u baferima. I dalje pretpostavljamo da su svi paketi primljeni onim redosledom kojim su i poslati (mada su neki bitovi u paketima možda izmenjeni).

Pre nego što razvijemo protokol za pouzdanu komunikaciju preko takvog kanala, razmotrimo prvo kako bi ljudi postupili u takvoj situaciji. Zamislite da diktirate dugačku poruku preko telefona. Stvari bi obično tekle tako što bi sagovornik rekao: „U redu", nakon svake rečenice koju je čuo, razumeo i zapisao. Ako sagovornik ne bi čuo neku rečenicu, zatražio bi da je ponovite. Ovaj protokol za diktiranje poruka sastoji se od **pozitivnih potvrda** („u redu") i **negativnih potvrda** („molim, ponovite"). Te kontrolne poruke omogućavaju primaocu da saopšti pošiljaocu šta je pravilno, a šta pogrešnoprimio, pa bi trebalo ponoviti. U računarskim mrežama pouzdani protokoli za transfer podataka, koji se zasnivaju na takvom ponovnom slanju, poznati su kao **ARQ** (Automatic Repeat reQuest) **protokoli**.

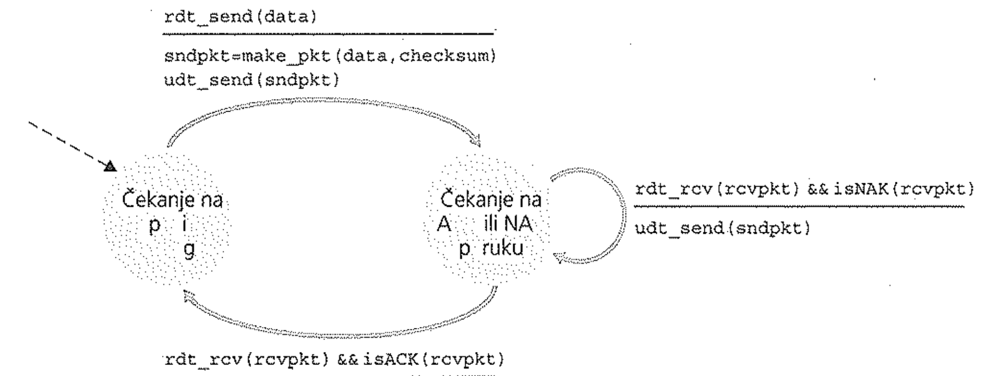
U suštini, da bi se izborili sa greškama na bitovima, potrebno je ARQ protokole dopuniti sa još tri dodatne mogućnosti:

- *Otkrivanje grešaka*. Prvo, potreban je mehanizam koji će dozvoliti primaocu da otkrije kada se greške na bitovima pojave. Sećate se iz prethodnog odeljka da protokol UDP baš u ovu svrhu koristi polje kontrolnog zbira. U poglavlju 5 detaljnije ćemo razmotriti tehnike za otkrivanje i ispravljanje grešaka; te tehnike omogućavaju primaocu da otkrije i po potrebi ispravi pogrešne bitove u paketu. Za sada, dovoljno je da znamo da je za te tehnike potrebno da pošiljalac pošalje primaocu dodatne bitove (pored bitova podataka, koje bi inače trebalo preneti); ti bitovi se smeštaju u polje kontrolnog zbira paketa podataka protokola `rdt2.0`.
- *Povratna informacija od primaoca*. Pošto se pošiljalac i primalac obično izvršavaju na različitim krajnjim sistemima, možda razdvojeni hiljadama kilometara, jedini način da pošiljalac sazna kako primalac gleda na stvari (u ovom slučaju, da li je paket ispravno primljen) jeste da mu primalac pošalje nedvosmislenu, povratnu informaciju. Pozitivne (ACK) i negativne (NAK) potvrde iz malopređašnjeg primera diktiranja su u stvari takve, povratne informacije. Naš protokol `rdt2.0` će vraćati slične ACK i NAK pakete od primaoca ka pošiljaocu. U suštini, ti paketi bi trebalo da sadrže samo jedan bit; na primer, vrednost 0 bi značila NAK, a vrednost 1 ACK.
- *Ponovno slanje*. Pošiljalac će ponovo poslati paket koji primalac primi sa greškom.

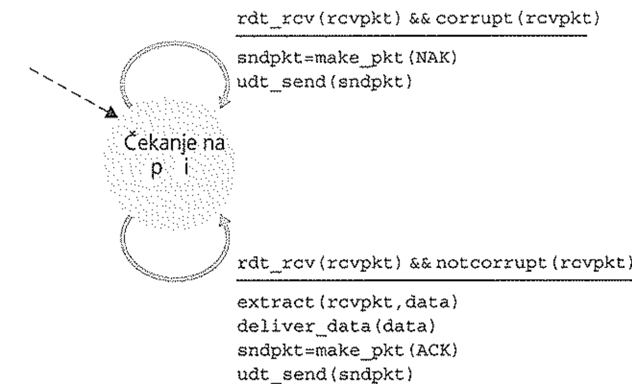
Na slici 3.10 je prikazan FSM opis protokola `rdt2.0`, protokola za prenos podataka u kome se koriste otkrivanje grešaka i pozitivne i negativne potvrde.

Predajna strana protokola `rdt2.0` ima dva stanja. U stanju prikazanom levo, predajna strana protokola čeka podatke iz gornjeg sloja koje bi trebalo da prosledi dole. Kada nastupi događaj `rdt_send (data)`, pošiljalac pravi paket (`sndpkt`) koji sadrži podatke koji se šalju i kontrolni zbir paketa (na primer, onako kako je u odeljku 3.3.2 opisano za slučaj UDP segmenta), a zatim šalje taj paket operacijom `udt_send(sndpkt)`. U stanju prikazanom desno, protokol pošiljaoca čeka da od primaoca dobije paket ACK ili NAK. Ako stigne paket ACK (na slici 3.10 ovaj događaj označen je sa `rdt_rcv(rcvpkt) && isACK (rcvpkt)`), pošiljalac zna da je poslednji preneti paket ispravno primljen i stoga se protokol vraća u stanje čekanja podataka iz gornjeg sloja. Ako primi NAK, protokol ponovo šalje poslednji paket i čeka da primalac vrati ACK ili NAK, kao odgovor na ponovo poslati paket podataka. Važno je primetiti da pošiljalac u stanju čekanja na ACK ili NAK paket *ne može* da preuzima još podataka iz gornjeg sloja, to jest, ne može da se desi događaj `rdt_send()`; to se dešava tek pošto pošiljalac dobije ACK i napusti ovo stanje. Prema tome, pošiljalac neće poslati nove podatke, sve dok se ne uveri da je primalac

ispravno primio tekući paket. Zbog ovakvog ponašanja, protokol kakav je `rdt2.0`, poznat je kao protokol **stani i čekaj** (engl. *stop-and-wait*).



a. rdt2.0: predajna strana



b. rdt2.0: prijemna strana

Slika 3.10 ◈ Protokol rtd2.0 za kanal sa greškama na bitovima

FSM prijemne strane protokola `rdt2.0` i dalje ima samo jedno stanje. Pošto paket pristigne, primalac odgovara sa ACK ili NAK, u zavisnosti od toga da li je primljeni paket oštećen ili ne. Na slici 3.10 oznaka `rdt_rcv(rcvpkt) && corrupt(rcvpkt)` odgovara događaju, kada se paket primi i utvrdi da sadrži grešku.

Protokol `rdt2.0` možda izgleda kao da dobro radi ali, nažalost, on ima fatalni nedostatak. Tačnije, nismo računali na mogućnost da ACK ili NAK paket može da bude oštećen! (Pre nego što nastavimo, razmislite kako bi to moglo da se popravi.) Nažalost, ovaj mali previd nije tako bezazlen, kao što se čini. U najmanju ruku,

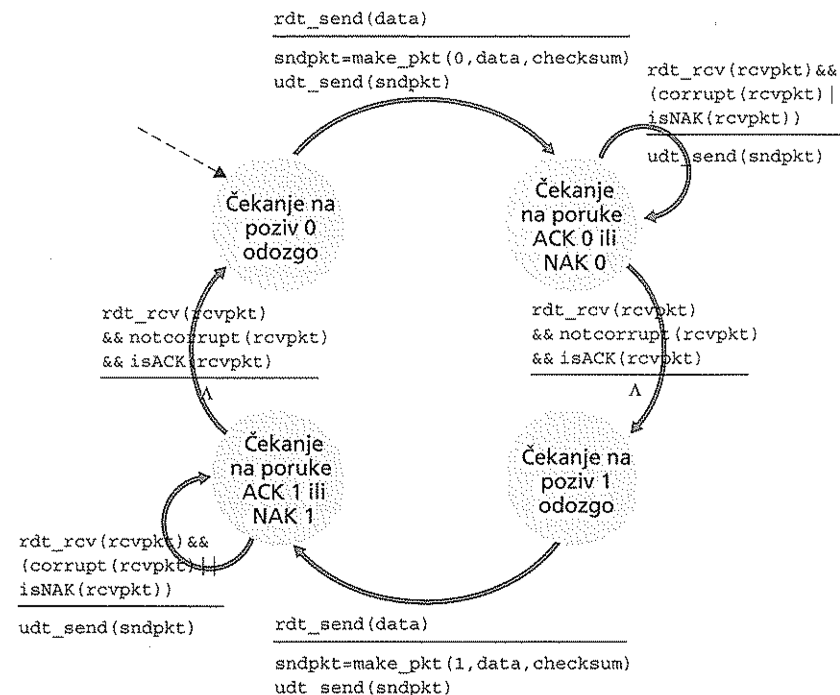
moraćemo paketima ACK i NAK da dodamo kontrolni zbir, da bismo otkrili takvu grešku. Mnogo je teže pitanje kako da se protokol oporavi od grešaka u ACK i NAK paketima. Problem je u tome što u slučaju greške u ACK ili NAK paketu, pošiljalac nema načina da sazna da li je primalac ispravno primio poslednji deo prosleđenih podataka.

Pogledajmo tri mogućnosti za postupanje sa neispravnim ACK ili NAK paketima:

- Kao prvo, pogledajmo šta bi ljudi mogli da urade u slučaju sa diktiranjem poruke. Ako govornik ne razume odgovor primaoca: „u redu", ili „molim ponovite", verovatno bi pitao: „Šta ste rekli?" (i na taj način bi u naš protokol uveo novu vrstu paketa od pošiljaoca ka primaocu). Primalac bi potom ponovio odgovor. Međutim, šta ako se govornikovo pitanje: „Šta ste rekli?", ne čuje dovoljno jasno? Primalac, budući da ne zna da li je rečenica koju nije jasno čuo deo diktata, ili je to zahtev da ponovi poslednji odgovor, verovatno bi odgovorio: „Šta ste *Vi* rekli?". Naravno, i taj odgovor može da bude oštećen. Očigledno je da smo pošli pogrešnim putem.
- Druga mogućnost je da se kontrolni zbir dopuni bitovima, tako da pošiljalac može ne samo da otkrije, već se i oporavi od grešaka na bitovima. Time se odmah rešava problem za kanal u kome se paketi mogu oštetiti, ali se ne gube.
- Treći pristup je da pošiljalac jednostavno ponovo pošalje paket podataka, ako primi oštećeni ACK ili NAK paket. Ovim pristupom se, međutim, u kanal od pošiljaoca ka primaocu uvode **duplikati paketa**. Osnovni problem sa duplikatima paketa je u tome što primalac ne zna da li je pošiljalac ispravno primio poslednji poslati ACK ili NAK. Prema tome, on ne može *a priori* da zna da li paket koji stiže sadrži nove ili ponovljene, prethodne podatke.

Jednostavno rešenje ovog, novog problema (koje je prihvaćeno skoro u svim postojećim protokolima za prenos podataka, uključujući i TCP) je da se paketu sa podacima doda još jedno polje, u koje pošiljalac stavlja **redni broj** i tako brojevima obeleži svoje pakete. Primalac bi onda samo trebalo da proveri taj redni broj i utvrdi da li primljeni paket predstavlja ponavljanje. Za ovaj jednostavan primer protokola stani i čekaj biće dovoljan redni broj od jednog bita,što će omogućiti primaocu da utvrdi da li pošiljalac ponovo šalje jednom već poslat paket (u tom slučaju redni broj novoprimljenog paketa je isti kao i redni broj poslednjeg primljenog paketa), ili je to novi paket (redni broj se menja, pomerajući se „napred" u aritmetičkom modulu broja 2). Pošto za sada pretpostavljamo da kanal ne gubi pakete, ACK i NAK paketi ne moraju i sami da navode redni broj paketa čiji prijem potvrđuju. Pošiljalac zna da je primljeni ACK ili NAK paket (ispravan ili neispravan) nastao kao odgovor na poslednji poslati paket podataka.

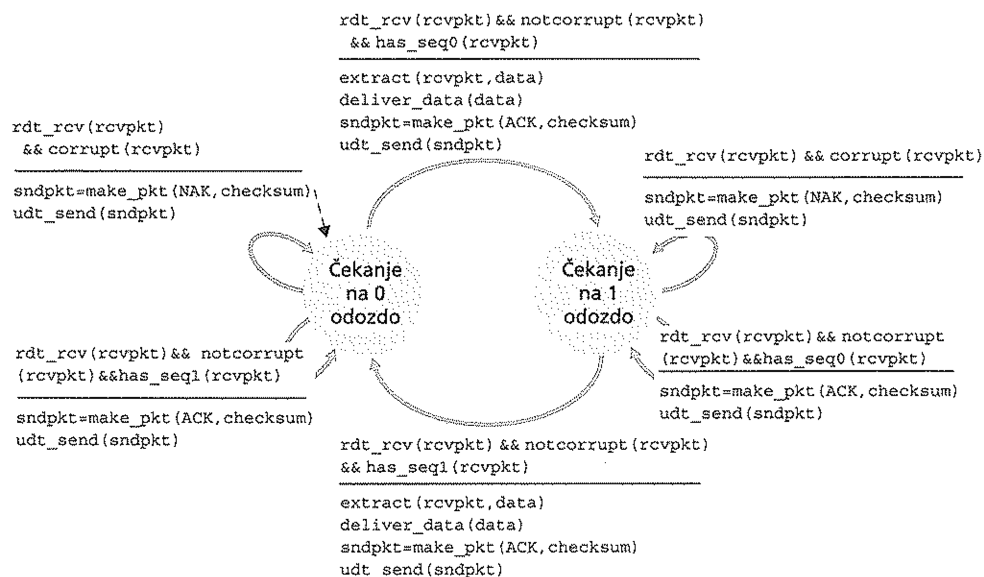
Na slikama 3.11 i 3.12 je prikazan FSM opis protokola `rdt2.1`, fiksne verzije protokola `rdt2.0`. FSM pošiljaoca i primaoca u protokolu `rdt2.1` i sada imaju dva puta više stanja nego ranije. To je zato što stanja protokola sada moraju da odražavaju da li bi paket koji (pošiljalac) trenutno šalje, ili koji (primalac) očekuje trebalo da ima redni broj 0 ili 1. Primetićete da je ono što se radi u stanjima kada se šalje ili očekuje paket sa brojem 0, slika u ogledalu onoga što se radi kada se očekuje ili šalje paket sa brojem 1; jedina razlika se odnosi na to kako se postupa sa rednim brojem.



Slika 3.11 ◈ Pošiljalac protokola rtd2.1

Protokol `rdt2.1` koristi i pozitivne i negativne potvrde od primaoca ka pošiljaocu. Kada primi paket van redosleda, pošiljalac šalje pozitivnu potvrdu da je primio taj paket. Kada primi oštećeni paket, pošiljalac šalje negativnu potvrdu prijema. Moguće je postići isti efekat kao sa NAK-om, ako se umesto NAK poruka pošalje ACK poruka za poslednji ispravno primljeni paket. Pošiljalac koji primi dve ACK poruke za isti paket (tj. primi **duplikat ACK** poruke) zna da primalac nije ispravno primio paket koji sledi iza paketa sa za koji su poslate dve ACK potvrde. Ovakav pouzdan protokol za prenos podataka bez NAK poruka kanalom sa greškama na bitovima, nazvali smo `rdt2.2` i prikazan je na slikama 3.13 i 3.14. Najvažnija razlika između protokola `rdt2.1` i `rdt2.2` je u tome što primalac sada mora da doda redni broj paketa čiji prijem potvrđuje ACK porukom (to se postiže uključivanjem argumenta `ACK,0` ili `ACK,1` u make_pkt() u FSM primaoca), a pošiljalac sada

mora da proveri redni broj paketa čiji je prijem potvrđen primljenom ACK porukom (ovo se postiže dodavanjem argumenta 0 ili 1 u isACK() u FSM pošiljaoca).

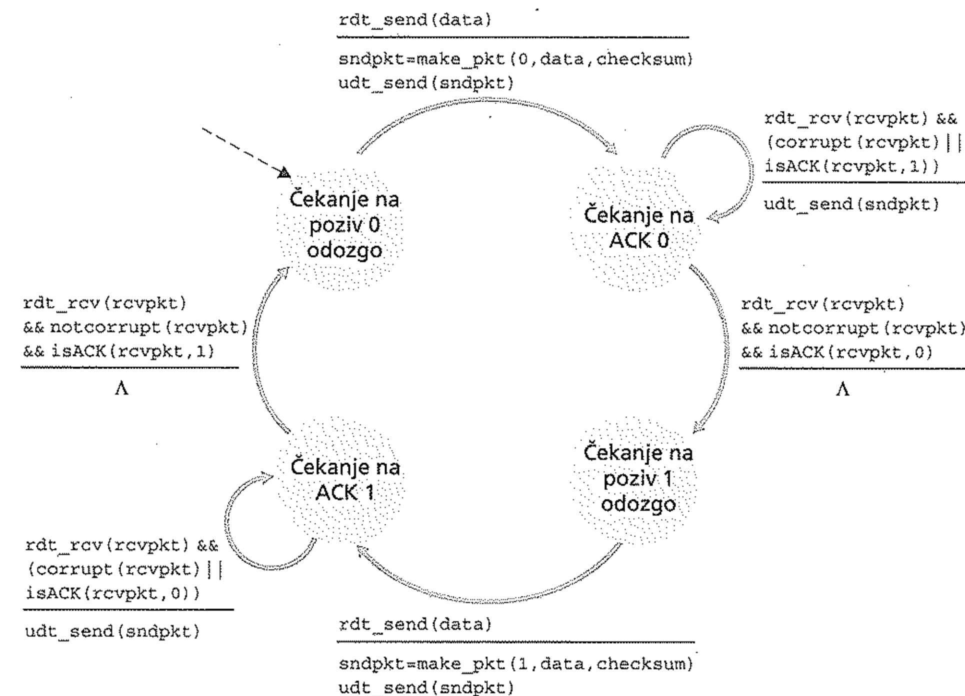


Slika 3.12 ◈ Primalac protokola rtd2.1

Pouzdani prenos podataka preko kanala sa gubicima i greškama na bitovima: protokol rdt3.0

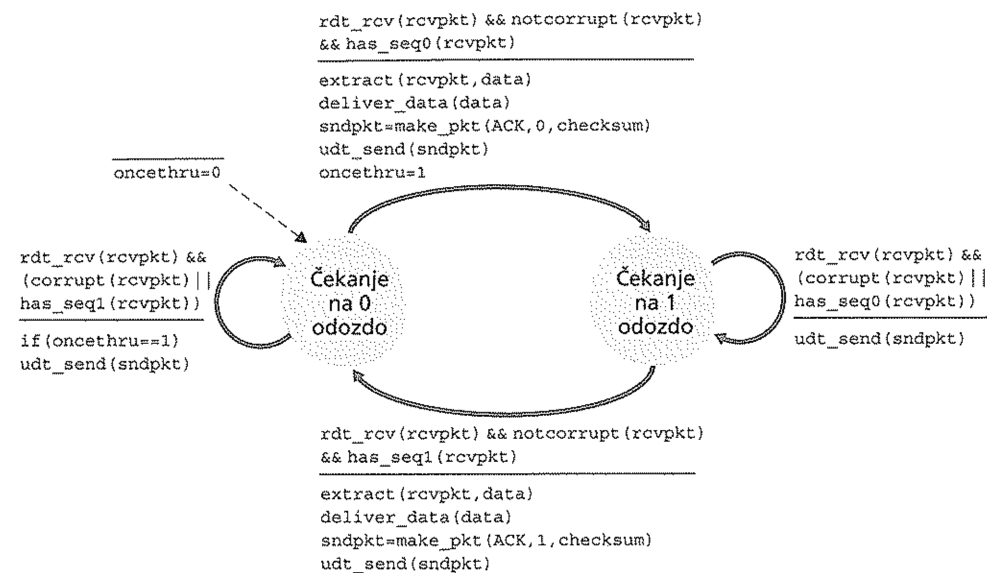
Pretpostavimo sada da, pored toga što oštećuje bitove, kanal kroz koji se vrši prenos podataka može i da *gubi* pakete, što u današnjim računarskim mrežama (uključujući i internet) nije neuobičajeno. Sada bi u protokolu trebalo rešiti još dva dodatna problema: kako otkriti gubitke paketa i šta uraditi u slučaju gubitka. Kontrolni zbirovi, redni brojevi, ACK paketi i ponovna slanja – tehnike razvijene za protokol rdt2.2 – omogućiće nam da rešimo ovo drugo pitanje. Rešavanje prvog pitanja zahtevaće dodavanje novog mehanizma u protokol.

Rešavanju gubitka paketa može se pristupiti na više načina (u vežbama na kraju poglavlja videćemo ih još nekoliko). Ovde ćemo za otkrivanje i rešavanje problema izgubljenih paketa zadužiti pošiljaoca. Pretpostavimo da pošiljalac pošalje paket podataka i da se izgubi bilo sâm paket, bilo primaočev ACK za taj paket. U oba slučaja pošiljalac od primaoca više neće dobiti nikakav odgovor. Ako je pošiljalac voljan da sačeka dovoljno dugo, kako bi bio *siguran* da se paket izgubio, onda prosto može ponovo da pošalje isti paket podataka. Pokušajte samostalno da pokažete da bi takav protokol zaista radio.



Slika 3.13 ◈ Pošiljalac protokola rtd2.2

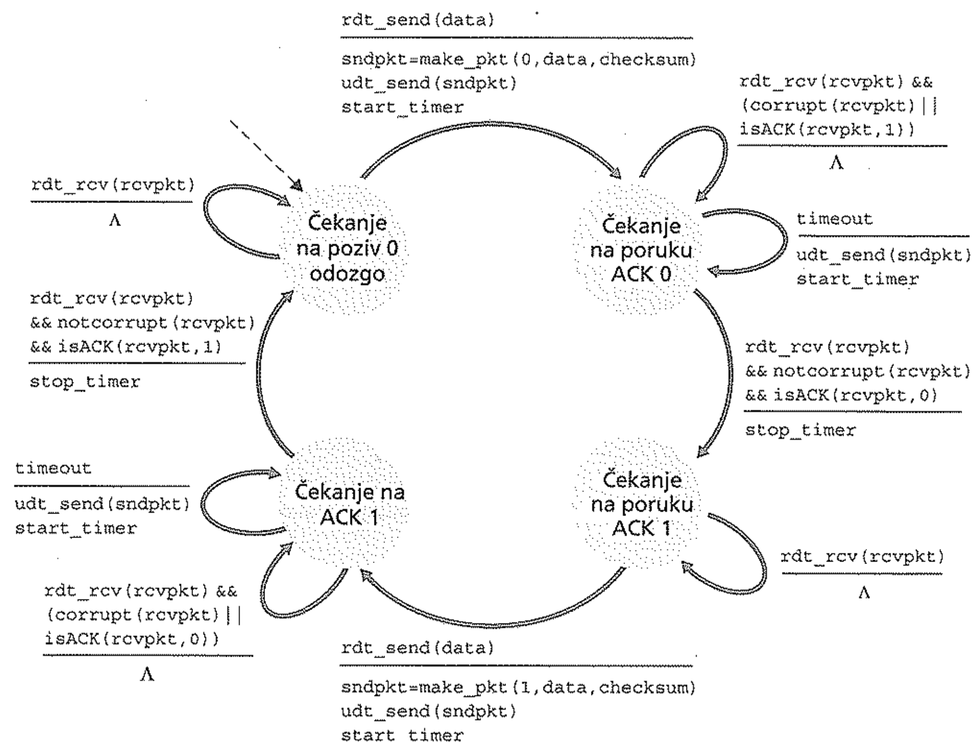
Koliko dugo bi trebalo pošiljalac da čeka, da bude siguran da se nešto izgubilo? Jasno je da mora da čeka bar onoliko koliko traje kašnjenje puta između pošiljaoca i primaoca i nazad (što obuhvata i privremeno čuvanje na usputnim ruterima), plus vreme potrebno da primalac obradi paket. U većini mreža veoma je teško i proceniti najduže kašnjenje u najgorem slučaju, a kamoli tačno ga odrediti. Štaviše, protokol bi u idealnom slučaju trebalo da se oporavi od gubitka paketa što je pre moguće; zato bi bilo previše čekati sa početkom ispravljanja greške dok ne prođe vreme potrebno u najgorem slučaju. U praksi je zato usvojeno rešenje da pošiljalac izabere neko „razumno" vreme u kojem je najverovatnije, mada ne i sigurno, došlo do gubitka paketa. Ako se za to vreme ne primi ACK poruka, paket se šalje ponovo. Primećujete da, ako paket izuzetno mnogo kasni, pošiljalac ga ponovo šalje, iako nije izgubljen ni sâm paket podataka, ni njegov ACK. Tako se u kanalu od pošiljaoca do primaoca javlja mogućnost **dupliranja paketa podataka**. Srećom, protokol rdt2.2 već ima mogućnost (to jest, redne brojeve) kojom rešava slučaj dupliranih paketa.



Slika 3.14 ◆ Primalac protokola rtd2.2

Sa stanovišta pošiljaoca, ponovno slanje je lek za sve. Pošiljalac ne zna da li je izgubljen paket podataka, da li je izgubljen ACK, ili da li paket ili ACK jednostavno isuviše kasne. U svim tim slučajevima rešenje je isto: ponovno slanje. Za primenu mehanizma ponovnog slanja na osnovu vremena, potreban je **tajmer** koji može da pošalje prekid pošiljaocu, po isteku zadatog vremena. Pošiljalac će morati da bude u stanju da: (1) pokrene tajmer uvek kad pošalje paket (bilo da je reč o prvom ili ponovljenom slanju), (2) odgovori na prekid od strane tajmera (odgovarajućim postupcima) i (3) da zaustavi tajmer.

Na slici 3.15 je prikazan FSM pošiljaoca za `rdt3.0`, protokol koji pouzdano prenosi podatke preko kanala koji može da ošteti i izgubi pakete; u domaćim zadacima biće zatraženo da prikažete FSM primaoca protokola `rdt 3.0`. Na slici 3.16 prikazano je kako protokol radi, kada nema gubitka i kašnjenja paketa i kako postupa u slučaju gubitka paketa. Na slici 3.16 vreme teče od vrha dijagrama prema dnu; obratite pažnju na to da je vreme prijema paketa obavezno veće od vremena slanja, zbog kašnjenja u prenosu i predaji. Na na slikama 3.16(b) do (d) uglaste zagrade na strani pošiljaoca označavaju vremena kada se tajmer uključuje i kasnije isključuje. Nekoliko najvažnijih osobina ovog protokola ispituju se u vežbama na kraju ovog poglavlja. Pošto redni brojevi paketa naizmenično menjaju vrednost između 0 i 1, protokol `rdt3.0` se ponekad naziva **protokol naizmeničnih bitova** (engl. *alternating-bit protocol*).



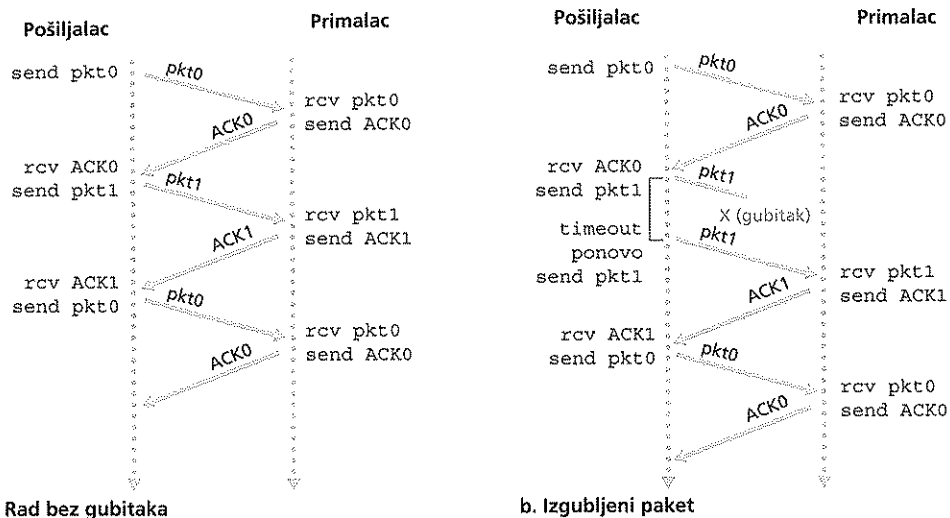
Slika 3.15 ◆ Pošiljalac u protokolu rtd3.0

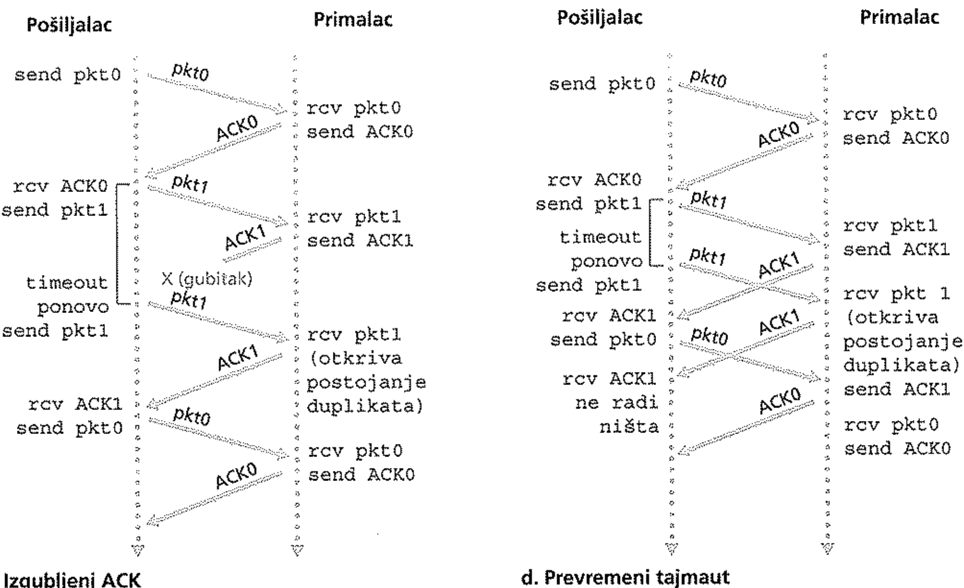
Sada smo na jednom mestu skupili ključne elemente protokola za transfer podataka. Kontrolni zbirovi, redni brojevi, tajmeri i pozitivne i negativne potvrde prijema paketa imaju ključnu ulogu i neophodni su za rad protokola. Sada imamo protokol za pouzdan prenos podataka!

Ra ijanje pr t k la i F M repre enta ija a je n sta ni pr t k l aplikati n g sl ja

3.4.2 Pouzdani cevovodni protokoli za prenos podataka

Protokol `rdt3.0` radi kako bi trebalo, ali malo je verovatno da bi iko bio zadovoljan njegovim performansama, pogotovo u današnjim mrežama velike brzine. Ključ problema performansi protokola `rdt3.0` leži u činjenici da je to protokol sa stani i čekaj.



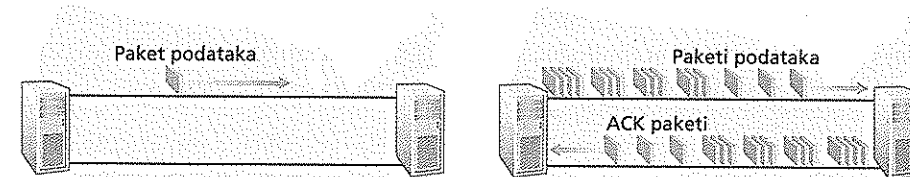


Slika 3.16 ◈ Rad protokola rtd3.0, protokola naizmeničnih bitova

Da biste shvatili do koje mere to stajanje i čekanje protokola stani i čekaj utiče na performanse, pogledajmo idealan slučaj sa dva računara od kojih se jedan nalazi na Zapadnoj obali SAD, a drugi na Istočnoj obali, kao što je prikazano na slici 3.17. RTT, vreme povratnog puta, između ova dva krajnja sistema približno iznosi 30 milisekundi. Pretpostavimo da su krajnji računari povezani kanalom sa brzinom prenosa R od 1 Gb/s (109 bitova u sekundi). Za paket veličine L od 1000 bajtova

(8000 bitova), uključujući polja zaglavlja i podatke, vreme potrebno da se paket zaista prenese linkom od 1 Gb/s je:

$$d_{\text{prenos}} = \frac{L}{R} = \frac{8000 \text{ bitova/paketu}}{10^9 \text{ bitova/sec}} = 8 \text{ mikrosekundi}$$



a. Rad protokola „stani i čekaj"

b. Rad cevovodnog protokola

Slika 3.17 ◈ Poređenje protokola stani i čekaj u odnosu na cevovodni protokol

Na slici 3.18(a) prikazano je da sa našim protokolom stani i čekaj, ako pošiljalac počne slanje paketa u $t = 0$, poslednji bit će ući u kanal na strani pošiljaoca za $t = L/R = 8$ mikrosekundi. Paket zatim polazi na put preko kontinenta od 15 milisekundi, s tim što poslednji bit paketa stiže primaocu za $t = RTT/2 + L/R =$ 15,008 milisekundi. Jednostavnosti radi, pretpostavimo da su ACK paketi izuzetno mali (da bismo mogli zanemariti vreme potrebno za njihov prenos) i da primalac može da pošalje ACK čim primi poslednji bit iz paketa podataka. U tom slučaju se ACK pojavljuje kod pošiljaoca nakon $t = RTT + L/R = 30{,}008$ milisekundi. U tom trenutku pošiljalac može da počne sa prenosom sledeće poruke. Prema tome, od 30,008 milisekundi pošiljalac se bavio slanjem samo tokom 0,008 milisekundi. Ako definišemo **iskorišćenost** pošiljaoca (ili kanala) kao deo vremena tokom koga je pošiljalac stvarno zauzet slanjem bitova u kanal, analiza slike 3.18(a) pokazuje da **iskorišćenost** protokola stani i čekaj nije baš za pohvalu. Iz ovoga sledi:

$$U_{\text{pošiljalac}} = \frac{L/R}{RTT + L/R} = \frac{0{,}008}{30{,}008} = 0{,}00027$$

Drugim rečima, pošiljalac je zauzet samo 2,7 stotih delova jednog procenta vremena. Drugačije posmatrano, pošiljalac može da šalje samo 1 000 bajtova za 30,008 milisekundi, što znači stvarnu propusnu moć od samo 267 kb/s – iako je na raspolaganju imao link od 1 Gb/s! Zamislite nesrećnog administratora mreže koji je upravo platio celo bogatstvo za link kapaciteta od 1 gigabita, a kroz njega uspeva da postigne protok od samo 267 kb/s! Ovo je očigledan primer kako mrežni protokoli mogu da ograniče mogućnosti koje pruža mrežni hardver u upotrebi. Osim toga, zanemarili smo vremena obrade protokola nižih slojeva kod pošiljaoca i primaoca, kao i kašnjenje zbog obrade i čekanja u redu do kojeg može doći na bilo kojem

usputnom ruteru između pošiljaoca i primaoca. Da smo i to uzeli u obzir, ukupno kašnjenje bilo bi još veće, a loše performanse još izraženije.

Rešenje navedenog problema sa performansama sasvim je jednostavno: umesto da radi tako što stoji i čeka, pošiljaocu se dozvoljava da šalje više paketa ne čekajući potvrde prijema, kako je prikazano na slici 3.17(b). Na slici 3.18(b) prikazano je da se **iskorišćenost** pošiljaoca u suštini utrostručava, ako pošiljalac može da pošalje tri paketa, pre nego što sačeka potrvde prijema. Pošto prolazak više paketa između pošiljaoca i primaoca može da se zamisli kao punjenje cevovoda, ova tehnika se naziva **cevovodnom obradom** (protočno slanje podataka, instrukcijski tok). Cevovodna obrada ima sledeće posledice na protokole za pouzdan prenos podataka:

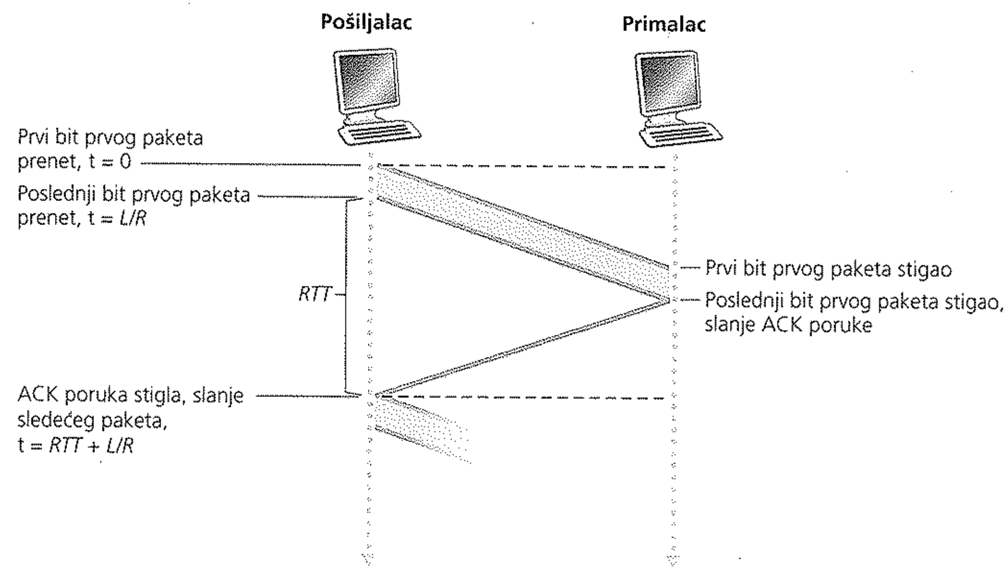
- Mora se povećati raspon rednih brojeva, zato što svi paketi u prolazu (ne računajući ponovna slanja) moraju da imaju jedinstven redni broj, a u prolazu se nalazi više paketa čiji prijem nije potvrđen.
- Predajna i prijemna strana protokola moraju da imaju memoriju za privremeno čuvanje više paketa – bafer. Pritom, pošiljalac će morati privremeno da čuva pakete koji su poslati, a njihov prijem još nije potvrđen. Privremeno čuvanje primljenih paketa ponekad je potrebno i kod primaoca, kao što ćemo opisati u daljem izlaganju.
- Potreban raspon rednih brojeva i veličina bafera zavisiće od toga kako protokol za prenos podataka postupa u slučaju izgubljenih i oštećenih paketa, kao i u slučaju paketa koji previše kasne. Razlikujemo dva osnovna pristupa za oporavak od grešaka cevovodne obrade: **Vrati-se-za-N** (Go-Back-N – GBN) i **selektivno ponavljanje** (eng. selective repeat).

3.4.3 Protokol vrati se za N - GBN

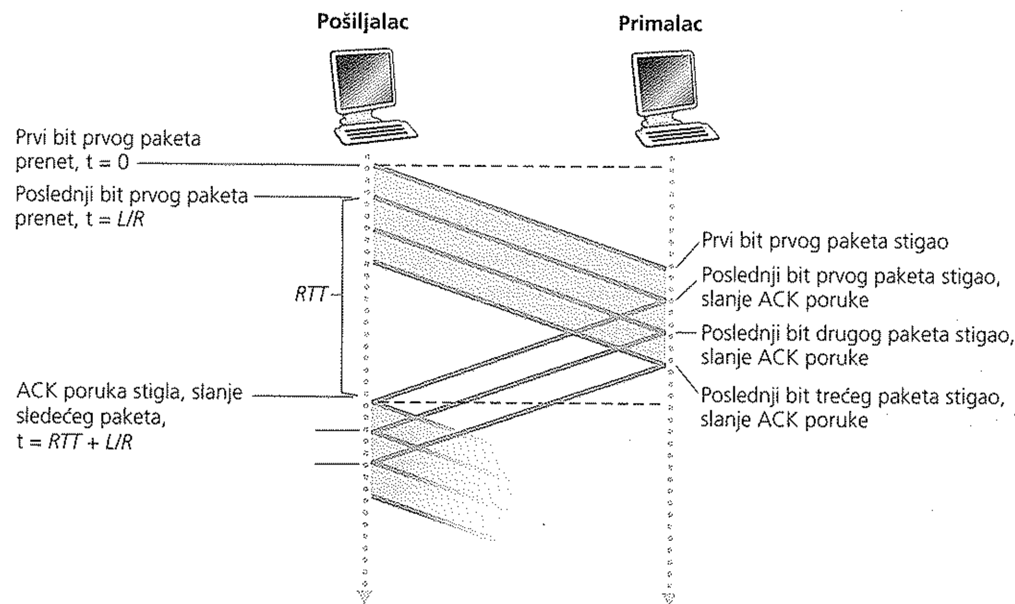
Protokol **GBN** (Go-Back-N – Vrati-se-za-N) dozvoljava pošiljaocu da pošalje više paketa (ako ih ima) bez čekanja na potvrdu, ali ograničava najveći dozvoljeni broj na N nepotvrđenih paketâ u cevovodu. U ovom odeljku podrobnije opisujemo protokol GBN. Međutim, pre nego što nastavite sa čitanjem, preporučujemo da se poigrate sa apletom GBN (čudesan aplet!) na veb stranici ove knjige.

Na slici 3.19 prikazan je raspon rednih brojeva kod protokola GBN – onako kako ih vidi pošiljalac. Ako definišemo `base` kao redni broj najstarijeg nepotvrđenog paketa, a `nextseqnum` kao najmanji neupotrebljeni redni broj (tj. redni broj sledećeg paketa koji bi trebalo poslati), možemo uočiti četiri intervala u rasponu rednih brojeva. Redni brojevi u intervalu `[0,base1]` odgovaraju paketima koji su već preneti i čiji je prijem potvrđen. Interval `[base,nextseqnum1]` odgovara paketima koji su poslati, ali čiji prijem još nije potvrđen. Redni brojevi u intervalu `[nextseqnum,base+N1]` koriste se za pakete koji mogu da se pošalju odmah, čim podaci stignu iz gornjeg sloja. Konačno, redni brojevi veći ili jednaki

od `base+N` ne mogu se koristiti, sve dok ne stigne potvrda prijema za nepotvrđeni paket koji se trenutno nalazi u cevovodu (tačnije, paket sa rednim brojem `base`).

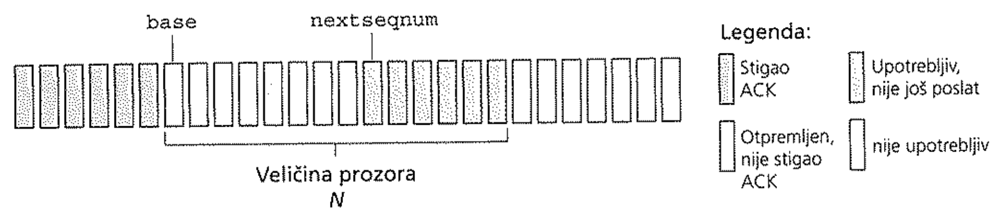


a. Rad stani i čekaj



b. Cevovodna obrada

Slika 3.18 ◆ Slanje po principu stani i čekaj i cevovodnom obradom

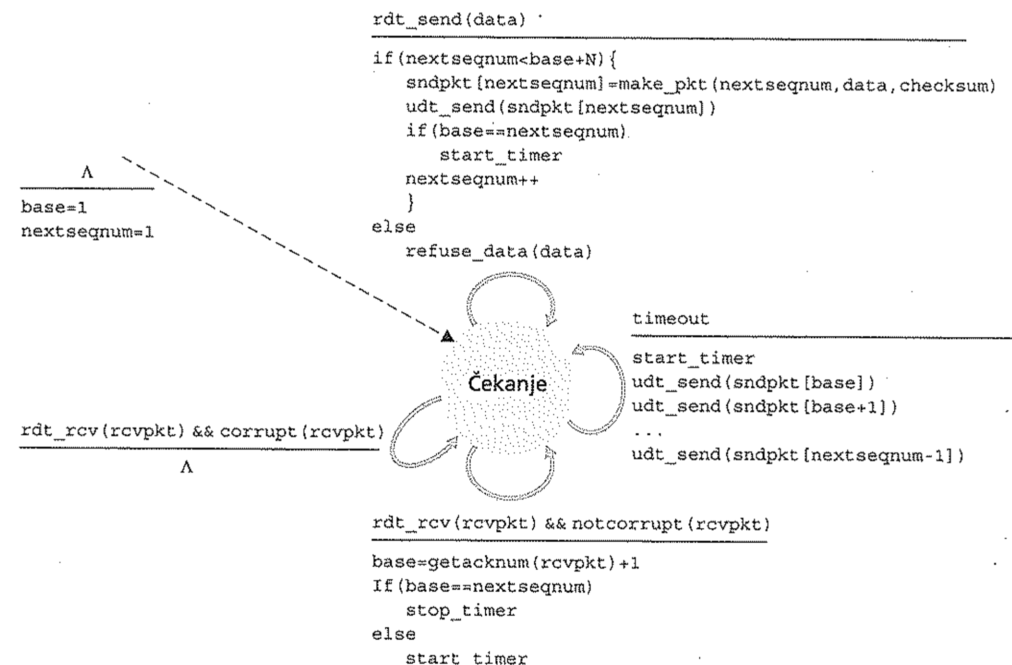


Slika 3.19 ◈ Redni brojevi onako kako ih vidi pošiljalac kod protokola GBN

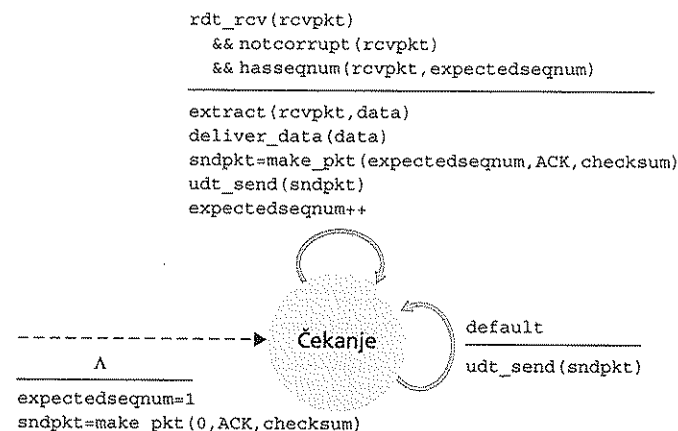
Kako je nagovešteno na slici 3.19, raspon dozvoljenih rednih brojeva za poslate, ali još nepotvrđene pakete, može se posmatrati kao prozor veličine *N* u rasponu rednih brojeva. Tokom rada protokola, ovaj prozor klizi unapred, duž raspoloživih rednih brojeva. Zbog toga se za *N* često kaže da je **veličina prozora**, a za sâm protokol GBN se kaže da je **protokol sa kliznim prozorom**. Možda se čudite zašto se broj nepotvrđenih paketa uopšte ograničava na neku vrednost *N*. Zašto nije dozvoljen neograničen broj takvih paketa? Videćemo u odeljku 3.5 da je kontrola toka jedan od razloga da se pošiljaocu nametne takvo ograničenje. Drugi razlog da se to uradi ispitujemo u odeljku 3.7, kada razmatramo kontrolu zagušenja u protokolu TCP.

U praksi, redni broj paketa prenosi se poljem fiksne dužine u zaglavlju paketa. Ako je *k* broj bitova u polju rednog broja paketa, raspon rednih brojeva je [0, 2k – 1]. Sa konačnim rasponom rednih brojeva, sve računanje vezano za redne brojeve mora da se obavlja po modulu broja 2k. (To jest, prostor rednih brojeva može se zamisliti kao prsten veličine 2k, gde redni broj 0 neposredno sledi iza rednog broj 2k1.) Sećate se da je protokol `rdt3.0` imao 1-bitne redne brojeve i raspon rednih brojeva [0,1]. Nekoliko problema na kraju ovog poglavlja istražuje posledice korišćenja konačnog raspona rednih brojeva. U odeljku 3.5 videćemo da protokol TCP ima 32bitno polje za redni broj, pri čemu ovi redni brojevi odbrojavaju bajtove u protoku bajtova umesto pakete.

Na slikama 3.20 i 3.21 dat je proširen FSM opis predajne i prijemne strane protokola GBN čiji se rad zasniva na ACK potvrdama, a ne koristi NAK potvrde. Ovaj FSM opis nazivamo *prošireni FSM,* zato što smo brojevima `base` i `nextseqnum` dodelili promenljive (slično kao promenljive u programskim jezicima) i dodali operacije nad tim promenljivama i uslovne aktivnosti u koje su uključene te promenljive. Primetićete da proširena FSM specifikacija počinje donekle da liči na specifikaciju programskog jezika. [Bochman 1984] sadrži izvrstan pregled dodatnih proširenja FSM tehnika kao i drugih tehnika nalik programskim jezicima za definisanje protokola.



Slika 3.20 ◈ Prošireni FSM opis GBN pošiljaoca



Slika 3.21 ◈ Prošireni FSM opis GBN primaoca

GBN pošiljalac mora da odgovori na tri vrste događaja:

- *Poziv odozgo.* Kada se odozgo pozove `rdt_send()`, pošiljalac prvo proverava da li je prozor pun, tj. da li postoji N nepotvrđenih paketa. Ako prozor nije pun, paket se pravi i šalje, a promenljive se ažuriraju na odgovarajući način. Ako je prozor pun, pošiljalac jednostavno vraća podatke gornjem sloju, što posredno znači da je prozor pun. Gornji sloj bi kasnije, prvom prilikom, trebalo da ponovo pokuša da pošalje. U praksi, verovatnije rešenje je da pošiljalac sačuva te podatke u privremenoj memoriji (neće ih poslati odmah), ili možda da ima mehanizam sinhronizacije (na primer, semafore ili oznake) koji gornjem sloju dozvoljava da pozove `rdt_send()`, samo ako prozor nije pun.
- *Prijem ACK poruke.* U našem GBN protokolu, potvrda prijema za paket sa rednim brojem n smatra se **kumulativnom potvrdom prijema**, koja označava da je primalac pravilno primio sve pakete sa rednim brojevima koji su manji i jednaki n. Vratićemo se uskoro ovom pitanju – prilikom opisa prijemne strane GBN protokola.
- *Istek vremena tajmera.* Naziv protokola „vrati-za-N" potiče od ponašanja pošiljaoca u slučaju izgubljenih paketa ili paketa koji previše kasne. Kao i u protokolu stani i čekaj, koristi se tajmer za oporavak od gubitka paketa sa podacima ili paketa sa potvrdom prijema. Posle isteka određenog vremena, pošiljalac ponovo šalje *sve* pakete koji su prethodno poslati, a čiji prijem nije još uvek potvrđen. Naš pošiljalac na slici 3.20 koristi samo jedan tajmer, koji možemo zamisliti kao tajmer najstarijeg poslatog, ali još uvek nepotvrđenog paketa. Ako stigne ACK poruka, ali još uvek ima poslatih i nepotvrđenih paketa, tajmer se ponovo pokreće. Ako više nema nepotvrđenih paketa, tajmer se zaustavlja.

Postupci primaoca u protokolu GBN takođe su jednostavni. Ako paket sa rednim brojem n primi ispravno i po redu (tj. poslednji podaci isporučeni gornjem sloju potiču iz paketa sa rednim brojem $n - 1$), primalac šalje ACK poruku za paket n i predaje deo podataka iz paketa gornjem sloju. U svim ostalim slučajevima, primalac odbacuje paket i ponovo šalje ACK poruku za poslednji paket primljen po redu. Obratite pažnju: uzimajući u obzir da se paketi isporučuju gornjem sloju jedan po jedan – ukoliko je paket k primljen i isporučen, to znači da su i svi paketi sa rednim brojevima manjim od k, takođe isporučeni. Prema tome, korišćenje kumulativnih potvrda prijema prirodno je za GBN protokol.

U našem protokolu GBN, primalac odbacuje pakete koji ne stižu po redu. Iako izgleda glupo i rasipno da se odbaci paket koji je pravilno primljen (iako nije po redu), za tako nešto postoji opravdanje. Ne zaboravite da primalac gornjem sloju mora redom da predaje podatke. Pretpostavimo sada da se očekuje paket n, a stiže paket $n + 1$. Pošto podaci moraju da se isporuče po redu, primalac *bi mogao* da (privremeno) sačuva paket $n + 1$ i isporuči ga gornjem sloju kad primi i isporuči paket n. Međutim, ako se paket n izgubio, i on i paket $n + 1$ biće kasnije ponovo poslati, zbog pravila ponovnog slanja protokola GBN na strani pošiljaoca. Prema tome, primalac može jednostavno da odbaci paket $n + 1$. Prednost ovakvog pristupa je jednostavnije privremeno čuvanje kod primaoca – primalac ne mora privremeno da čuva *nijedan* paket izvan redosleda. Prema tome, dok pošiljalac mora da održava gornju i donju granicu svog prozora i položaj `nextseqnum` unutar tog prozora, jedino što primalac mora da održava je redni broj sledećeg paketa po redu. Ova vrednost čuva se u promenljivoj `expectedseqnum` prikazanoj u FSM opisu primaoca na slici 3.21. Naravno, nedostatak odbacivanja ispravno primljenog paketa je i to što ponovno poslati paket može da se izgubi ili ošteti, što će zahtevati još ponovnih slanja.

Na slici 3.22 prikazan je rad protokola GBN za prozor veličine 4 paketa. Zbog ovakvog ograničenja veličine prozora, pošiljalac šalje pakete od 0 do 3, ali potom mora da čeka potvrdu prijema jednog ili više tih paketa, kako bi mogao da nastavi sa slanjem. Prijemom svake sledeće ACK poruke (na primer, `ACK0` i `ACK1`), prozor klizi unapred i pošiljalac može da pošalje nove pakete (pkt4 i pkt5, respektivno). Na strani primaoca, paket 2 je izgubljen i zato se paketi 3, 4 i 5 odbacuju, jer su van redosleda.

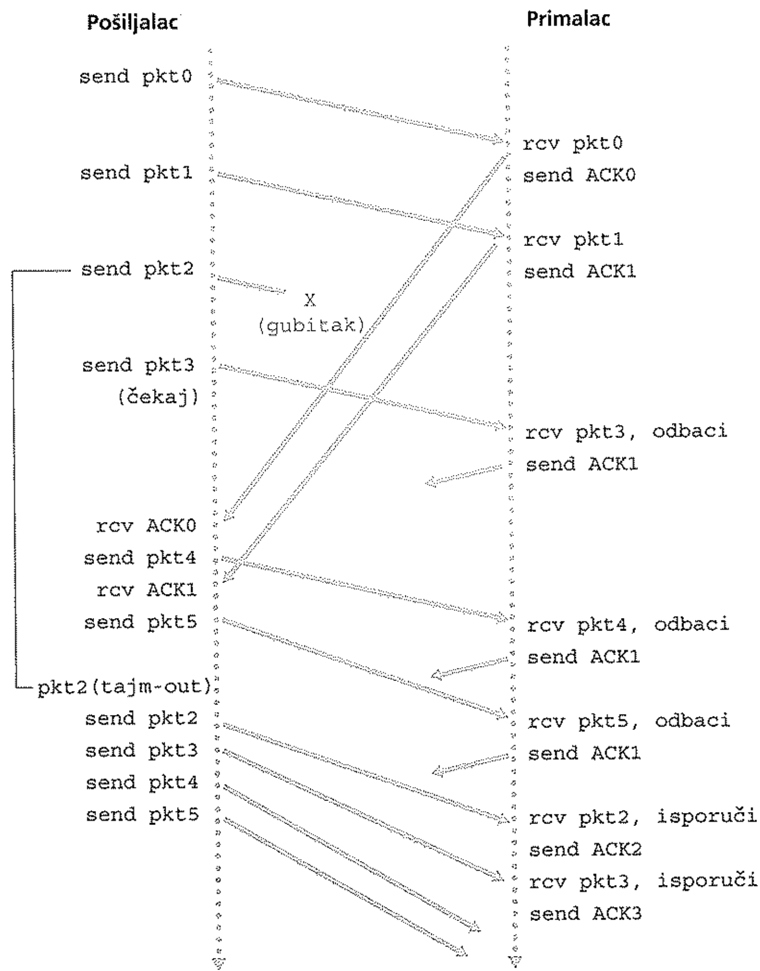
Pre nego što završimo razmatranje protokola GBN, vredi primetiti da bi stvarna realizacija ovog protokola u skupu protokola verovatno imala strukturu sličnu proširenom FSM opisu sa slike 3.20. Ta realizacija bi, takođe verovatno bila u obliku raznih procedura, kojima bi se utvrdili postupci koje bi trebalo preduzeti povodom različitih događaja, koji mogu da nastupe. U takvom **programiranju zasnovanom na događajima** pozivaju se (pokreću) različite procedure, bilo iz drugih procedurâ u skupu protokola, ili kao rezultat prekida. Kod pošiljaoca, ti događaji mogu da budu: (1) poziv entiteta iz gornjeg sloja da se pokrene `rdt_send()`, (2) prekid prouzrokovan istekom tajmera i (3) poziv iz donjeg sloja da se pokrene `rdt_rcv()`, kada stigne paket. Programerski zadaci na kraju poglavlja daće vam priliku da napravite takve rutine u zamišljenom, ali mogućem, mrežnom okruženju.

Ovde primećujemo da protokol GBN obuhvata skoro sve tehnike na koje ćemo naići pri razmatranju komponenti protokola TCP za pouzdan prenosa podataka u odeljku 3.5. U ove tehnike spadaju: upotreba rednih brojeva, kumulativnih potvrda prijema, kontrolnih zbirova i radnji vezanih za istek vremena ili ponovno slanje.

3.4.4 Selektivno ponavljanje

Protokol GBN dozvoljava pošiljaocu da paketima skoro potpuno „popuni cevovod" (sa slike 3.17) i tako izbegne probleme nedovoljnog iskorišćavanja kanala koje smo spominjali u vezi sa protokolom stani i čekaj. Međutim, u određenim okolnostima i sâm GBN protokol ima probleme sa performansama. Tačnije, kada su širina prozora i proizvod kašnjenja i propusnog opsega dovoljno veliki, u cevovodu može da se nađe mnogo paketa. Samo jedna greška u paketu može da dovede do toga da GBN ponovo šalje velik broj paketa, što je za većinu paketâ nepotrebno. Povećanjem verovatnoće za nastajanje grešaka u kanalu, cevovod može da se prepuni paketima koji

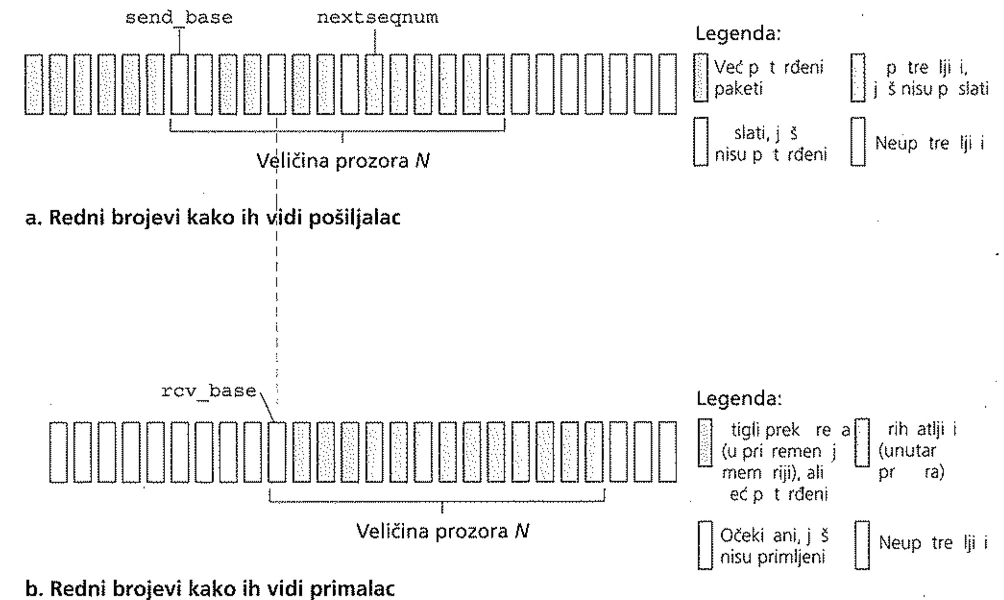
se ponovno šalju bez potrebe. Zamislite kada bi, u primeru sa diktiranjem poruke, za svaku nerazgovetnu reč trebalo ponoviti okolnih hiljadu reči (ako je, na primer, veličina prozora hiljadu reči). Diktiranje bi se usporilo zbog toliko ponavljanja.



Slika 3.22 ◈ Rad protokola GBN

Kao što im ime govori, protokoli sa selektivnim ponavljanjem (selective repeat, SR) izbegavaju nepotrebna ponovna slanja, tako što pošiljalac ponovo šalje samo one pakete za koje sumnja da ih je primalac pogrešno primio (odnosno, da su izgubljeni ili oštećeni). Ovakvo pojedinačno ponovno slanje prema potrebi zahteva podrazumeva da primalac *zasebno* potvrđuje ispravno primljene pakete. Veličina prozora N i ovde se koristi da bi se ograničio broj zaostalih, nepotvrđenih paketa

u cevovodu. Međutim, za razliku od protokola GBN, pošiljalac je već dobio ACK potvrde za neke pakete u prozoru. Na slici 3.23 prikazan je prostor rednih brojeva sa gledišta SR pošiljaoca. Na slici 3.24 prikazane su razne aktivnosti koje preduzima SR pošiljalac.



Slika 3.23 ◈ Prostor rednih brojeva sa stanovišta pošiljaoca i primaoca protokola SR

SR primalac će potvrditi ispravno primljeni paket, bez obzira na to da li je stigao po redu. Paketi izvan redosleda čuvaju se u privremenoj memoriji, dok ne stignu paketi koji nedostaju (tj. paketi sa nižim rednim brojevima), a tada se gornjem sloju redom isporučuje grupa paketa. Na slici 3.25 prikazano je šta sve preduzima SR primalac. Na slici 3.26 dat je primer kako se protokol SR ponaša u slučaju gubitka paketa. Obratite pažnju na to da na slici 3.26 primalac u privremenu memoriju prvo smešta pakete 3, 4 i 5, a da ih isporučuje gornjem sloju zajedno sa paketom 2, kada paket 2 konačno stigne.

Važno je primetiti da u koraku 2 na slici 3.25 primalac ponovo potvrđuje (umesto da zanemari) ranije primljene pakete sa rednim brojevima *manjim* od trenutnog broja u prozoru. Proverite sami da li je ovo ponovno potvrđivanje zaista potrebno. Za dati prostor rednih brojeva pošiljaoca i primaoca na slici 3.23, na primer, bez ACK potvrde za paket `send_base` koju primalac šalje pošiljaocu, pošiljalac bi kasnije možda ponovo poslao paket `send_base`, mada je jasno (nama, ali ne i pošiljaocu!), da je primalac već dobio taj paket. Kada primalac ne bi potvrdio prijem

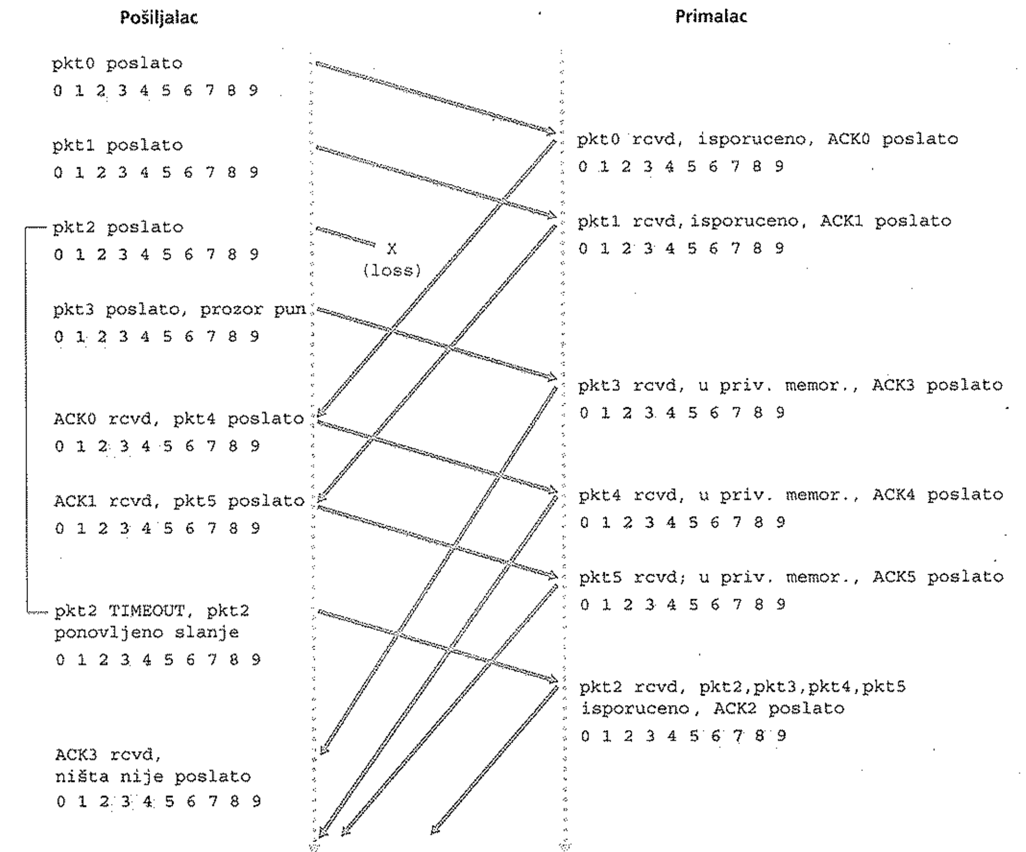
tog paketa, prozor pošiljaoca nikad se ne bi pomerio! Ovaj primer ilustruje jednu važnu osobinu protokola SR (kao i mnogih drugih protokola). Pošiljalac i primalac nemaju uvek isti uvid u to šta je pravilno primljeno, a šta nije. Za protokole SR to znači da se prozori pošiljaoca i primaoca ne poklapaju uvek.

1. *Podaci primljeni odozgo.* Kada primi podatke odozgo, SR pošiljalac proverava koji je sledeći redni broj dostupan za paket. Ukoliko je taj redni broj unutar prozora pošiljaoca, podaci se pakuju i šalju; inače se, ili smeštaju u privremenu memoriju, ili se vraćaju u gornji sloj za kasnije slanje, kao u protokolu GBN.
2. *Istek vremena.* I ovde se koriste tajmeri kao zaštita u slučaju izgubljenih paketa. Međutim, svi paketi moraju da imaju sopstvene softverske tajmere, jer se samo jedan paket ponovo šalje, kada istekne određeno vreme. Samo jedan hardverski tajmer može da se iskoristi kao zamena za više softverskih tajmera [Varghese 1997].
3. *Prijem ACK potvrde.* Ukoliko se primi ACK potvrda, SR pošiljalac obeležava da je određeni paket primljen, pod uslovom da je u prozoru. Ukoliko je redni broj tog paketa jednak `send_base`, početak prozora se pomera unapred, do sledećeg nepotvrđenog paketa sa najmanjim rednim brojem. Ukoliko se prozor pomeri i pojave se nepreneseni paketi sa rednim brojevima koji se sada nalaze unutar prozora, ti paketi se prenose.

Slika 3.24 ◈ Događaji i mere koje preduzima SR pošiljalac

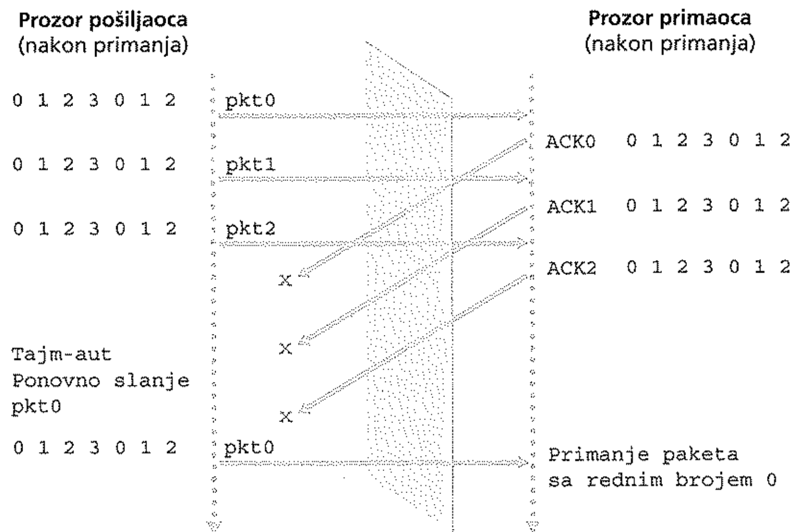
1. *Paket sa rednim brojem u opsegu* `[rcv_base, rcv_base+N-1]` *ispravno je primljen.* U ovom slučaju primljeni paket se nalazi unutar prozora primaoca i odgovarajući ACK paket vraća se pošiljaocu. Ukoliko taj paket nije ranije primljen, on se smešta u privremenu memoriju. Ukoliko taj paket ima redni broj jednak početku prozora primaoca (`rcv_base` na slici 3.22), tada se taj paket i svi paketi koji se nalaze u privremenoj memoriji sa uzastopnim rednim brojevima (koji počinju sa `rcv_base`) isporučuju gornjem sloju. Prozor primaoca se pomera unapred, za broj paketa koji su isporučeni gornjem sloju. Kao primer, pogledajte sliku 3.26. Kada se primi paket sa rednim brojem `rcv_base=2`, tada se on i paketi 3, 4 i 5 mogu isporučiti gornjem sloju.
2. *Paket sa rednim brojem u opsegu* `[rcv_base-N, rcv_base-1]` *ispravno je primljen.* U ovom slučaju mora da se napravi ACK potvrda, mada je to paket za koji je primalac već poslao potvrdu o ispravnom prijemu.
3. *Ostalo.* Paket se zanemaruje.

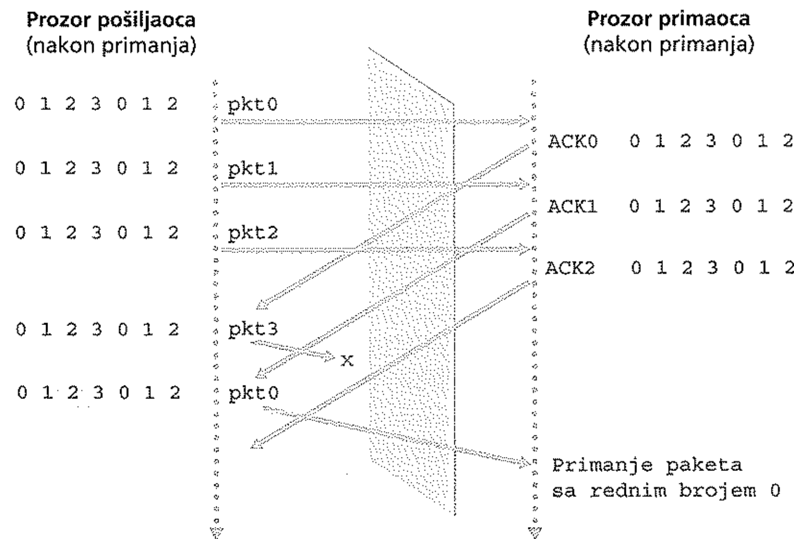
Slika 3.25 ◈ Događaji i mere koje preduzima SR primalac



Slika 3.26 ◈ Rad protokola SR

Neusklađenost između prozora pošiljaoca i primaoca ima značajne posledice u slučaju sa konačnim rasponom rednih brojeva. Pogledajmo šta bi se dogodilo, na primer, sa konačnim rasponom od 4 redna broja (0, 1, 2, 3) i veličinom prozora tri. Pretpostavimo da su poslati paketi od 0 do 2 pravilno primljeni i da je primalac potvrdio njihov prijem. U tom trenutku se prozor primaoca nalazi na četvrtom, petom i šestom paketu čiji su redni brojevi 3, 0 i 1, respektivno. Sada ćemo razmotriti dva slučaja. U prvom slučaju, prikazanom na slici 3.27(a), ACK potvrde za prva tri paketa su se izgubile i pošiljalac ponovo šalje te pakete. Primalac, znači, dobija paket sa rednim brojem 0 – kopiju prvog poslatog paketa.





Slika 3.27 ◆ Dilema SR primaoca sa prevelikim prozorom: novi paket ili ponovno slanje?

U drugom slučaju, prikazanom na slici 3.27(b), ACK potvrde za prva tri paketa pravilno su isporučene. Pošiljalac tada pomera svoj prozor unapred i šalje četvrti, peti i šesti paket sa rednim brojevima 3, 0 i 1. Paket sa rednim brojem 3 se gubi, ali paket sa rednim brojem 0 stiže – paket koji sadrži *nove* podatke.

Sada ćemo razmotriti kako to izgleda sa stanovišta primaoca na slici 3.27, pred kojim se nalazi figurativna zavesa između pošiljaoca i njega, jer primalac ne može da „vidi" šta se dešava na strani pošiljaoca. Sve što primalac vidi je niz poruka koje prima iz kanala i koje šalje u kanal. Što se njega tiče, oba slučaja, prikazana na slici 3.27,*istovetna* su. Ne postoji način da primalac razlikuje ponovno slanje prvog paketa od prvog slanja petog paketa. Jasno je da veličina prozora, koja je za jedan manja od veličine prostora rednih brojeva, nije dobra. Kolika bi trebalo da bude veličina prozora? U jednom problemu na kraju poglavlja imaćete zadatak da pokažete da za SR protokole veličina prozora mora biti manja, ili jednaka polovini veličine prostora rednih brojeva.

Na veb stranici ove knjige pronaći ćete aplet, koji u obliku animacije, prikazuje rad protokola SR. Isprobajte ono što ste radili sa apletom za protokol GBN. Da li se rezultati poklapaju sa onim što očekujete?

Ovim zaključujemo razmatranje protokola za pouzdan prenos podataka. Obradili smo *najvažnije* osnove i uveli niz mehanizama kojima se obezbeđuje pouzdan prenos podataka. U tabeli 3.1 ukratko su nabrojani ovi mehanizmi. S obzirom da ste videli sve te mehanizme i možete da razumete celinu, predlažemo da ponovo pregledate ovaj odeljak, kako biste uočili da smo ove mehanizme postupno dodavali, da bismo objasnili sve složenije (i realnije) modele kanala koji povezuju pošiljaoca i primaoca, ili da bismo poboljšali performanse protokola.

Zaključimo našu priču o protokolima za pouzdan prenos podataka razmatranjem još jedne pretpostavke za model kanala preko koga se vrši prenos. Sećate se pretpostavke da paketi unutar kanala između pošiljaoca i primaoca ne mogu da promene redosled. Ovo je sasvim realna pretpostavka, kada su pošiljalac i primalac povezani samo jednim fizičkim kablom. Međutim, kada je ovaj „kanal" koji povezuje dva računara u stvari mreža, može doći do promene redosleda paketâ. Ono po čemu se izmenjeni redosled paketa može uočiti jeste pojava stare kopije paketa sa rednim brojem ili brojem potvrde x, iako prozori pošiljaoca i primaoca ne sadrže broj x. Zbog promene redosleda paketa, kanal se može zamisliti kao da smešta pakete u privremenu memoriju i proizvoljno ih ispušta u *bilo kom* kasnijem trenutku. Pošto redni brojevi mogu da se koriste više puta, nešto mora da se preduzme protiv pojave duplikata paketâ. Rešenje koje se koristi u praksi služi da se osigura da se određeni redni broj ponovo ne koristi, sve dok pošiljalac ne bude „siguran" da u mreži nema više nijednog ranije poslatog paketa sa tim rednim brojem. To se postiže uz pretpostavku da paket ne može da „živi" u mreži duže od nekog, tačno određenog, maksi-

malnog vremena. U TCP proširenjima za mreže velike brzine [RFC 1323] prihvaćen je maksimalni životni vek paketa od približno 3 minuta. U knjizi [Sunshine 1978] je opisan metod korišćenja rednih brojeva na takav način da se potpuno izbegnu problemi promene redosleda paketa.

Mehanizam	Upotreba, komentari
Kontrolni zbir	Koristi se za otkrivanje bitskih grešaka prenošenih paketâ.
Tajmer	Koristi se za merenje isteka vremena i ponovno slanje paketa, moguće zato što je paket (ili njegov ACK) izgubljen unutar kanala. U slučajevima kada vreme istekne jer paket kasni, ali nije izgubljen (prerano isteklo vreme), ili kada primalac primi paket, ali se ACK potvrda koju on šalje izgubi, primalac može da dobije dve kopije istog paketa.
Redni broj	Koriste se da bi se rednim brojevima obeležili paketi podataka koji teku od pošiljaoca do primaoca. Redni brojevi koji nedostaju u primljenim paketima omogućavaju primaocu da ustanovi gubitak paketa. Paketi sa ponovljenim rednim brojevima omogućavaju primaocu da otkrije duplikate kopija paketa.
Potvrda prijema	Koristi je primalac, da bi saopštio pošiljaocu, da su neki paket ili grupa paketa primljeni ispravno. Potvrda prijema obično sadrži redni broj jednog ili više paketa, čiji se prijem potvrđuje. Potvrda prijema može da bude pojedinačna ili kumulativna, što zavisi od protokola.
Negativna potvrda prijema	Koristi je primalac, da bi saopštio pošiljaocu, da neki paket nije ispravno primljen. Negativna potvrda prijema obično sadrži redni broj paketa koji nije ispravno primljen.
Prozor, cevovodna obrada	Moguće je ograničiti pošiljaoca da šalje samo one pakete čiji de redni broj nalazi unutar određenog opsega. Omogućavanjem istovremenog slanja više paketa čiji prijem nije potvrđen, poboljšava se iskorišćenost pošiljaoca u odnosu na rad protokola stani i čekaj. Uskoro ćemo videti da se veličina prozora može postaviti na osnovu mogućnosti primaoca da prima i privremeno čuva poruke, ili na osnovu zagušenja u mreži, ili na osnovu oba navedena.

Tabela 3.1 ◈ Rezime mehanizama za pouzdan prenos podataka i njihova upotreba

3.5 Transport sa uspostavljanjem veze: protokol TCP

Pošto smo upoznali osnovna pravila na kojima se zasniva pouzdan prenos podataka, preći ćemo na TCP – protokol za pouzdan prenos podataka sa uspostavljanjem veze koji se koristi na transportnom sloju interneta. U ovom odeljku videćemo da se za obezbeđenje pouzdanog prenosa podataka, protokol TCP oslanja na većinu osnovnih principa koje smo razmotrili u prethodnom odeljku, uključujući otkrivanje grešaka, ponovno slanje, kumulativne potvrde, tajmere i poljâ u zaglavlju za redne

brojeve paketa i redne brojeve potvrda. Protokol TCP definisan je u dokumentima RFC 793, RFC 1122, RFC 1323, RFC 2018 i RFC 2581.

3.5.1 TCP veza

Za protokol TCP se kaže da je protokol **sa uspostavljanjem veze**, zato što pre početka slanja podataka od jedne aplikacije ka drugoj, ta dva procesa moraju prvo „da se rukuju" – tj. moraju jedan drugom da pošalju neke uvodne segmente, kako bi se uspostavili parametri prenosa podataka koji slede. Prilikom uspostavljanja TCP veze obe strane uspostavljaju početne vrednosti brojnih promenljivih TCP stanja (većinu njih opisujemo u ovom odeljku i u odeljku 3.7) koje se odnose na tu TCP vezu.

TCP „veza" nije TDM ili FDM kolo od jednog kraja do drugog, kakvo postoji u mreži sa komutacijom kanala. Nije ni virtuelno kolo (pogledajte poglavlje 1), jer se stanje veze u potpunosti nalazi na krajnjim sistemima. S obzirom da se protokol TCP izvršava samo na krajnjim sistemima, a ne na usputnim elementima mreže (ruterima i komutatorima sloja veze), oni ne održavaju stanje TCP veze. U suštini, usputni ruteri uopšte nisu svesni postojanja TCP veze; oni vide samo datagrame, a ne i uspostavljene veze.

ISTORIJSKA ČITANKA

VINTON SERF, ROBERT KAN I TCP/IP

Prve mreže sa komutiranjem paketa pojavile su se ranih sedamdesetih godina, pri čemu je mreža ARPAnet – preteča interneta – bila tek jedna od mnogih mreža. Sve te mreže imale su sopstvene protokole. Dva istraživača, Vinton Serf i Robret Kan, shvatili su značaj međusobnog povezivanja tih mreža i stvorili su protokol za međusobno povezivanje tih mreža pod nazivom TCP/IP (skraćenica od Transmission Control Protocol/Internet Protocol). Mada su Serf i Kan počeli tako što su ovaj protokol zamislili kao jedinstvenu celinu, on je kasnije podeljen na dva dela, TCP i IP, koji zasebno rade. Serf i Kan objavili su svoj rad o protokolu TCP/IP u maju 1974. godine u časopisu *IEEE Transaction on Communication Technology* [Cerf 1974].

Protokol TCP/IP, koji je suština savremenog interneta, razvijen je pre PC-ja i radnih stanica, pametnih telefona i tableta, pre širenja Eterneta, kablovskog interneta, lokalnih računarskih mreža, WiFi i ostalih tehnologija pristupnih mreža i pre veba, društvenih mreža i prenosa video zapisa. Serf i Kan su uvideli potrebu za mrežnim protokolom koji bi, sa jedne strane, obezbeđivao opštu podršku aplikacijama, čije vreme je tek dolazilo i koji bi, sa druge strane, omogućavao raznoraznim računarima i protokolima sloja veze da međusobno sarađuju.

Serf i Kan su 2004. godine primili nagradu udruženja ACM, koja se smatra „Nobelovom nagradom u računarstvu" za „pionirski rad u povezivanju mreža, uključujući dizajn i primenu osnovnih komunikacionih protokolâ na internetu, protokola TCP/IP, kao i za nadahnuto predvodništvo u umrežavanju".

TCP veza obezbeđuje **punu dupleksnu uslugu**: ako postoji TCP veza između procesa A na jednom računaru i procesa B na drugom računaru, onda podaci iz aplikativnog sloja mogu da teku od procesa A do procesa B, a u isto vreme i od procesa B do procesa A. TCP veza je takođe uvek **od tačke do tačke**, tj. između tačno određenog pošiljaoca i tačno određenog primaoca. Takozvano „višeznačno usmeravanje" (videti odeljak 4.7) – prenos podataka od jednog pošiljaoca do više primalaca u istoj operaciji slanja – nije moguće ostvariti TCP vezom. Za TCP vezu dva računara su dovoljna – treći je višak!

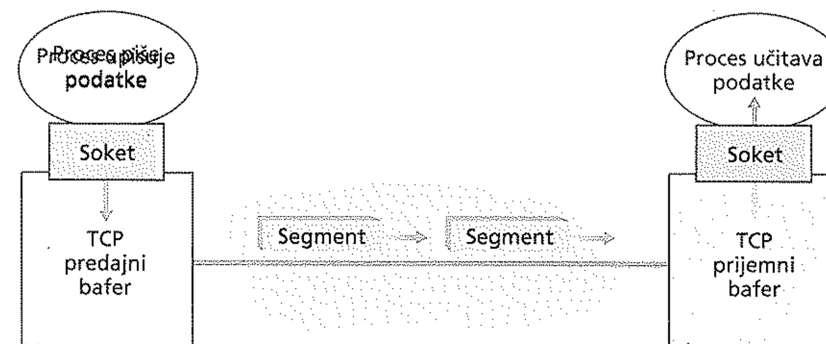
Pogledajmo sada kako se uspostavlja TCP veza. Pretpostavimo da proces koji se izvršava na jednom računaru želi da uspostavi vezu sa drugim procesom na drugom računaru. Sećate se da se proces koji pokreće vezu naziva *klijentski proces*, dok se drugi proces naziva *serverski proces*. Klijentski proces aplikacije prvo obaveštava transportni sloj klijenta da želi da uspostavi vezu sa procesom na serveru. Sećate se iz odeljka 2.7 da *Python*, klijentski program, to postiže izdavanjem komande:

```
clientSocket.connect((serverName,serverPort))
```

gde je servername naziv servera, a serverPort označava odgovarajući proces na serveru. Protokol TCP na klijentu zatim prelazi na uspostavljanje TCP veze sa protokolom TCP na serveru. Na kraju ovog odeljka nešto podrobnije ćemo razmotriti ovaj postupak uspostavljanja veze. Za sada je dovoljno da znamo da klijent prvo šalje poseban TCP segment; server odgovara drugim posebnim TCP segmentom; a na kraju klijent ponovo odgovara trećim posebnim segmentom. Prva dva segmenta ne sadrže nikakve korisne podatke, tj. nikakve podatke aplikativnog sloja; treći segment već može da sadrži takve podatke. Pošto se među računarima razmenjuju tri segmenta, ovaj postupak uspostavljanja veze često se naziva **trostruko usaglašavanje** (eng. three-way handshake).

Pošto se TCP veza uspostavi, dva procesa aplikacijâ mogu jedan drugom da šalju podatke. Razmotrimo slanje podataka od klijentskog procesa do serverskog procesa. Klijentski proces propušta niz podataka kroz soket (vrata procesa), kao što je opisano u odeljku 2.7. Kada ti podaci prođu kroz vrata, oni se nalaze u rukama protokola TCP, koji se izvršava na klijentu. Kao što je prikazano na slici 3.28, TCP usmerava ove podatke u **predajni bafer (privremenu memoriju)** te veze, koji je jedan od bafera, koji se rezervišu tokom početnog trostrukog usaglašavanja. S vremena na vreme, TCP zahvata delove podataka iz predajnog bafera. Zanimljivo je da u TCP specifikaciji [RFC 793] nema strogog pravila o tome kada bi TCP trebalo da pošalje podatke iz privremene memorije; kaže se da bi TCP trebalo da „šalje podatke u segmentima prema vlastitom nahođenju". Najveća količina podataka koja može da se zahvati i stavi u jedan segment ograničena je **najvećom veličinom segmenta** (Maximum Segment Size, **MMS**). MSS se obično postavlja tako što se prvo odredi dužina najvećeg okvira sloja veze koji može da pošalje lokalni računar pošiljalac, takozvani **MTU** (Maximum Transmision Unit) – **najveća jedinica prenosa** – i zatim, postavljanjem vrednosti MSS, tako da se osigura da TCP segment (kada se enkapsulira u IP datagram) i kad mu se doda TCP/IP zaglavlje (najčešće

40B), može da stane u jedan okvir sloja veze. Protokoli Eternet i PPP sloje veze imaju vrednosti MSS od 1.500 bajtova. Predložena su i rešenja u kojima se pronalazi MTU putanje – najveći okvir sloja veze koji se može poslati duž svih linkova, od izvorišta do odredišta [RFC 1191] – i postavljanjem MSS na osnovu vrednosti za MSU putanje. Obratite pažnju na to da je MSS maksimalna količina podataka aplikativnog sloja u segmentu, a ne maksimalna veličina TCP segmenta, uzimajući u obzir i zaglavlja. (Ovi nazivi malo zbunjuju, ali moramo se navići na njih, jer su već prilično ustaljeni.)



Slika 3.28 ◈ Predajna i prijemna privremena memorija TCP veze

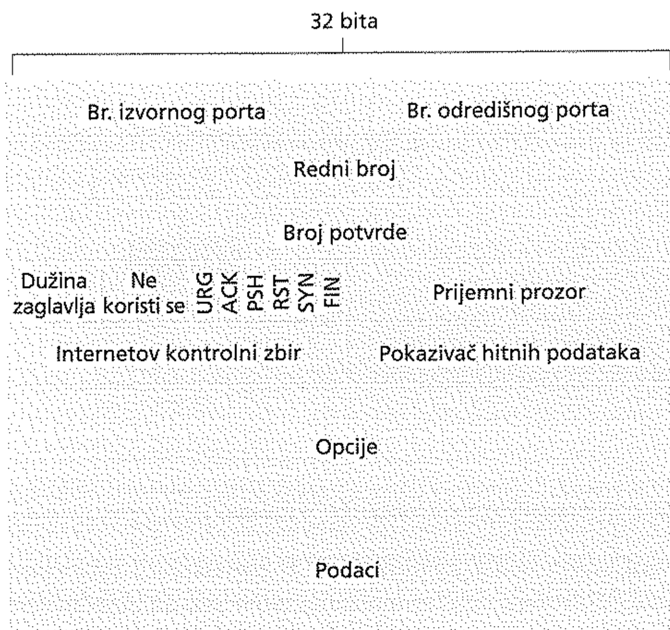
TCP dopunjava svaku celinu podataka klijenta TCP zaglavljem i tako pravi **TCP segmente**. Segmenti se predaju naniže, mrežnom sloju gde se pojedinačno enkapsuliraju u IP datagrame mrežnog sloja. IP datagrami se zatim šalju u mrežu. Kada TCP na drugom kraju primi segment, podaci iz segmenta smeštaju se u prijemni bafer TCP veze, kao što je prikazano na slici 3.28. Aplikacija čita niz podataka iz ovog bafera. Svaka strana veze ima svoj predajni i svoj prijemni bafer. (Predlažemo čitaocu da na internetu pogleda aplet kontrole toka na adresi http://www.awl.com/kuroseross, koji prikazuje animaciju predajnog i prijemnog bafera.)

Iz ovog opisa vidimo da se TCP veza sastoji od: bafera, promenljivih i soketa veze sa procesom na jednom računaru i još jednog skupa od bafera, promenljivih i soketa veze sa procesom na drugom računaru. Kao što je ranije spomenuto, vezi se u mrežnim elementima između dva računara (ruterima, komutatorima i repetitorima), ne dodeljuju privremene memorije ili promenljive.

3.5.2 Struktura TCP segmenta

Pošto smo ukratko razmotrili TCP vezu, proučimo strukturu TCP segmenta. TCP segment se sastoji od više poljâ zaglavlja i jednog polja podataka. Polje podataka sadrži komad podataka aplikacije. Kao što je već pomenuto, MSS ograničava maksimalnu veličinu polja podataka u segmentu. Kada TCP šalje veliku datoteku, kao što je slika sa neke veb stranice, on obično deli tu datoteku na komade veličine MSS (osim poslednjeg dela, koji je obično manji od MSS). Interaktivne aplikacije,

međutim, često prenose komade podataka koji su manji od MSS; na primer, kod aplikacija za daljinsko prijavljivanje, kao što je *Telnet*, polje podataka u TCP segmentu često je dugačko samo 1 bajt. Pošto je TCP zaglavlje obično dugačko 20 bajtova (12 bajtova više od UDP zaglavlja), segmenti koje šalje *Telnet* ponekad su dugački samo 21 bajt.



Slika 3.29 ◆ Struktura TCP segmenta

Na slici 3.29 prikazana je struktura TCP segmenta. Kao i kod UDP segmenta, zaglavlje sadrži **broj izvornog i odredišnog porta**, koji se koriste za multipleksiranje i demultipleksiranje podataka iz aplikacija gornjeg sloja i prema njima. Takođe, kao i u UDP segmentima, zaglavlje sadrži i **polje kontrolnog zbira**. Zaglavlje TCP segmenta sadrži još i sledeća polja:

- 32-bitno **polje rednog broja** i 32-bitno **polje broja potvrde**, koje TCP pošiljalac i primalac koriste pri ostvarivanju usluge pouzdanog prenosa podataka, što ćemo kasnije opisati;
- 16-bitno polje **prijemnog prozora** koristi se za kontrolu toka; uskoro ćemo videti da se ono koristi da bi se saopštilo koliko bajtova primalac pristaje da prihvati;
- 4-bitno **polje dužine zaglavlja** navodi dužinu TCP zaglavlja u 32-bitnim rečima; TCP zaglavlje može biti promenljive dužine zbog polja za TCP opcije (obično je polje opcija prazno, pa je dužina uobičajenog TCP zaglavlja jednaka 20 bajtova);

- neobavezno **polje opcija** promenljive dužine koje se koristi kada pošiljalac i primalac pregovaraju o maksimalnoj dužini segmenta (MSS), ili kao faktor za podešavanje prozora koji se koristi u mrežama velike brzine; definisana je i mogućnost unošenja vremenske oznake (više o tome možete naći u dokumentima RFC 845 i RFC 1323);
- **polje oznaka** sadrži 6 bitova; **bit ACK** se koristi da bi se naznačilo da li je vrednost koja se nalazi u polju broja potvrde važeća, odnosno da segment sadrži potvrdu prijema za segment koji je uspešno primljen. Bitovi **RST**, **SYN** i **FIN** koriste se prilikom uspostavljanja i prekidanja veze o čemu govorimo na kraju ovog odeljka. Postavljanje **PSH** bita znači da bi primalac trebalo odmah da prosledi podatke gornjem sloju. Konačno, bit **URG** se koristi da bi se označilo da u ovom segmentu ima podataka koje je gornji sloj na strani pošiljaoca označio kao „hitne" (urgentne). Položaj na kome se nalazi poslednji bajt ovih hitnih podataka naznačen je 16-bitnim **poljem pokazivača hitnih podataka**. TCP mora da obavesti entitet gornjeg sloja na prijemnoj strani da postoje hitni podaci i da mu preda pokazivač na kraj tih hitnih podataka. (U praksi se PSH, URG i pokazivač hitnih podataka ne koriste. Međutim, ta polja pominjemo, kako bi opis bio potpun.)

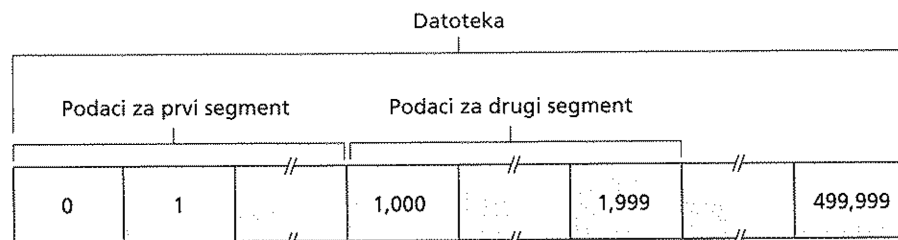
Redni brojevi i brojevi potvrda

Dva najvažnija polja u zaglavlju TCP segmenta su polje rednog broja i polje broja potvrde. Ta polja su bitan deo usluge pouzdanog prenosa podataka protokola TCP. Ali, pre nego što opišemo kako se ova polja koriste za pouzdan prenos podataka, prvo ćemo objasniti šta tačno TCP stavlja u njih.

TCP podatke vidi kao tok bajtova bez strukture, ali poređanih po određenom redu. Način na koji TCP koristi redne brojeve pokazuje da se redni brojevi odnose na neprekidni tok prenetih bajtova, a *ne* na niz prenetih segmenata. **Redni broj segmenta** je, prema tome, redni broj prvog bajta u segmentu unutar toka bajtova. Pogledajmo jedan primer. Uzmimo da proces na računaru A želi da pošalje tok podataka do procesa na računaru B preko TCP veze. TCP na računaru A, sâm po sebi, obeležava rednim brojevima sve bajtove u toku podataka. Pretpostavimo da se taj tok podataka sastoji od datoteke koja ima 500 000 bajtova, da MSS iznosi 1 000 bajtova, a da prvi bajt toka podataka ima redni broj 0. Kao što je prikazano na slici 3.30, TCP od tog toka podataka pravi 500 segmenata. Prvom segmentu dodeljuje se redni broj 0, drugom segmentu redni broj 1 000, trećem redni broj 2 000 i tako dalje. Redni brojevi se stavljaju u polje rednog broja u zaglavlju odgovarajućeg TCP segmenta.

Razmotrimo sada brojeve potvrda. Oni su nešto složeniji od rednih brojeva. Znate da TCP radi u punom dupleksnom režimu, tako da računar A može istovremeno da prima podatke od računara B, dok i sâm šalje podatke računaru B (u okviru iste TCP veze). Svaki segment koji stigne od računara B ima redni broj za podatke, koji se šalju od B prema A. *Broj potvrde koji računar A stavlja u svoj segment je*

redni broj sledećeg bajta, koji računar A očekuje od računara B. Dobro je pogledati nekoliko primera, kako bismo razumeli šta se ovde događa. Uzmimo da je računar A primio od računara B sve bajtove obeležene brojevima od 0 do 535 i uzmimo da se sprema da pošalje neki segment računaru B. Računar A očekuje bajt 536 i sve naredne bajtove iz toka podataka računara B. Zato računar A stavlja broj 536 u polje za broj potvrde segmenta, koji šalje računaru B.



Slika 3.30 ◈ Deljenje podataka iz datoteke na TCP segmente

Kao drugi primer, uzmimo da je računar A primio od računara B jedan segment koji sadrži bajtove od 0 do 535 i drugi segment koji sadrži bajtove od 900 do 1 000. Iz nekog razloga računar A još nije primio bajtove od 536 do 899. U ovom primeru, računar A i dalje čeka bajt 536 (i one iza njega) da bi ponovo napravio tok podataka od računara B. Prema tome, sledeći segment koji računar A šalje računaru B imaće u polju za broj potvrde vrednost 536. Pošto TCP potvrđuje samo bajtove do prvog nedostajućeg bajta u toku, za TCP se kaže da šalje **kumulativne potvrde prijema**.

Ovaj poslednji primer takođe nameće jedno važno, suštinsko pitanje. Računar A je primio treći segment (bajtove 900 do 1 000), pre nego što je primio drugi segment (bajtove 536 do 899). Prema tome, treći segment stigao je van redosleda. Suštinsko pitanje glasi: šta računar radi kada TCP vezom primi segmente van redosleda? Zanimljivo je da RFC dokumenti za protokol TCP ne postavljaju nikakva pravila što se toga tiče, prepuštajući odluku programerima TCP protokola. U osnovi postoje dve mogućnosti: ili će (1) primalac odmah da odbaci segmente primljene van redosleda (što, kao što smo rekli, može da pojednostavi dizajn primaoca), ili će (2) primalac da zadrži bajtove primljene van redosleda i da sačeka nedostajuće bajtove i njima popuni praznine. Jasno, ovo drugo mnogo je efikasnije, što se tiče propusnog opsega mreže i zato je prihvaćeno u praksi.

Na slici 3.30 pretpostavili smo da je početni redni broj jednak 0. U suštini, obe strane TCP veze nasumice biraju početni redni broj. To se radi da bi se smanjila verovatnoća da se segment, koji još uvek postoji u mreži od ranije, prekinute veze između dva računara, ne zameni sa važećim segmentom u kasnije uspostavljenoj vezi između ta ista dva računara (pri čemu se takođe koriste isti brojevi portova kao u staroj vezi) [Sunshine 1978].

Telnet: Kratak osvrt na redne brojeve i brojeve potvrda

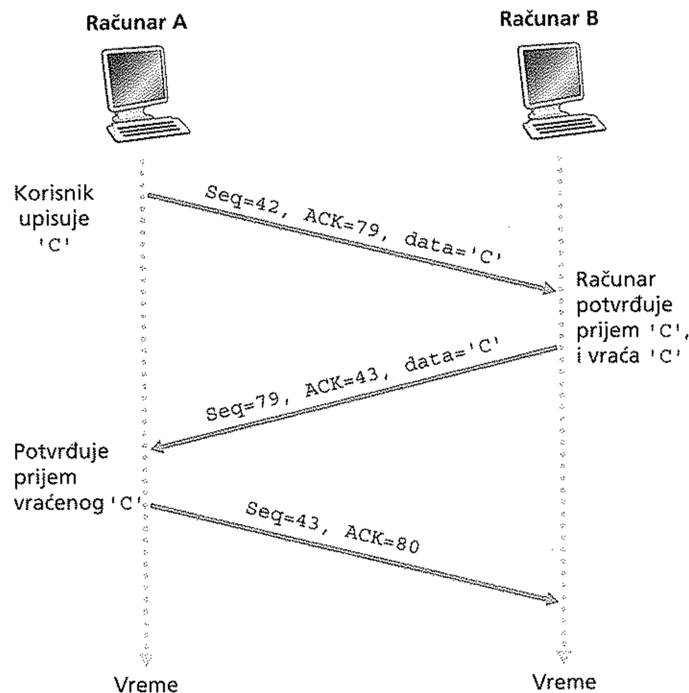
Telnet, definisan u dokumentu RFC 854, popularan protokol aplikativnog sloja, koji se koristi za daljinsko prijavljivanje. Izvršava se preko TCP protokola i predviđen je da radi između bilo koja dva računara. Za razliku od aplikacija za paketski prenos podataka, opisanih u poglavlju 2, *Telnet* je interaktivna aplikacija. Ovde razmatramo primer korišćenja Telneta, kao sjajan primer korišćenja rednih brojeva i brojeva potvrda u protokolu TCP. Napominjemo da mnogi korisnici radije koriste protokol SSH umesto *Telneta*, jer podaci koji se šalju *Telnet* vezom (uključujući i lozinke!) nisu šifrovani, zbog čega je *Telnet* osetljiv na prisluškivanje (što razmatramo u odeljku 8.7).

Pretpostavimo da računar A pokrene *Telnet* sesiju sa računarom B. Pošto računar A pokreće sesiju, njega smatramo klijentom, a računar B serverom. Svaki karakter koji korisnik otkuca (na strani klijenta) šalje se udaljenom računaru; udaljeni računar vraća kopiju karaktera i ona se prikazuje na ekranu korisnika *Telnet*-a. Ovakav „povratni eho" služi kao potvrda da su karakteri koje korisnik *Telnet*-a vidi već primljeni i obrađeni na udaljenom računaru. Svaki karakter tako dva puta prelazi celu mrežu, od trenutka kada korisnik pritisne taster do trenutka kada se karakter pojavi na njegovom monitoru.

Sada pretpostavimo da korisnik upiše jedno jedino slovo: „C", a zatim ode na kafu. Pogledajmo koji se TCP segmenti šalju između klijenta i servera. Kao što se vidi na slici 3.31, pretpostavka je da su početni redni broj kod klijenta 42, a kod servera 79. Sećate se da redni broj u segmentu predstavlja redni broj prvog bajta u polju podataka. Prema tome, prvi segment koji klijent pošalje imaće redni broj 42; prvi segment koji pošalje server imaće redni broj 79. Sećate se da je broj potvrde redni broj sledećeg bajta podataka, koji računar očekuje. Pošto se uspostavi TCP veza, a pre slanja podataka, klijent očekuje bajt 79, a server očekuje bajt 42.

Kao što je prikazano na slici 3.31, šalju se tri segmenta. Prvi segment koji se šalje od klijenta ka serveru u svom polju podataka sadrži jednobajtni ASCII prikaz slova „C". Ovaj prvi segment sadrži još i vrednost 42 u polju rednog broja, kao što smo upravo opisali. Isto tako, pošto klijent još nije primio nikakve podatke od servera, taj prvi segment će u polju broja potvrde imati 79.

Drugi segment šalje se od servera ka klijentu. On ima dvostruku svrhu. Prvo, služi kao potvrda da je server primio određene podatke. Stavljanjem 43 u polje potvrde, server obaveštava klijenta da je uspešno primio sve do bajta 42 i da sada očekuje bajtove od 43 nadalje. Druga svrha ovog segmenta je da vrati nazad slovo „C". Prema tome, drugi segment ima u polju podataka ASCII prikaz slova „C". Ovaj drugi segment nosi redni broj 79, početni redni broj toka podataka od servera ka klijentu za ovu TCP vezu, pošto je u pitanju prvi bajt podataka koji šalje server. Obratite pažnju na to da se potvrda podataka koji su stigli od klijenta ka serveru šalje u istom segmentu koji prenosi podatke od servera ka klijentu; za takvu potvrdu kažemo da se **šlepuje** (eng. piggybacked) na segmentu podataka od servera ka klijentu.



Slika 3.31 ◆ Redni brojevi i brojevi potvrda za jednostavnu *Telnet* aplikaciju preko TCP veze

Treći segment šalje se od klijenta ka serveru. Njegova jedina svrha je da se potvrdi prijem podataka od servera. (Sećate se da drugi segment sadrži podatke – slovo „C" – od servera ka klijentu.) Treći segment ima prazno polje podataka (tj. potvrda se ne šlepuje sa podacima od klijenta ka serveru). Segment sadrži 80 u polju broja potvrde, zato što je klijent primio tok bajtova, sve do rednog broja 79 i sada očekuje bajtove od 80 nadalje. Možda vas čudi što i ovaj segment sadrži redni broj iako u njemu nema podataka, ali pošto TCP ima polje rednog broja u njemu mora da bude neki redni broj.

3.5.3 Procena vremena povratnog puta i isteka vremena tajmera

Protokol TCP, slično našem protokolu rdt iz odeljka 3.4, za oporavak od izgubljenih segmenata, koristi mehanizam isteka vremena tajmera i ponavljanje slanja. Iako je suština ovog mehanizma jednostavna, prilikom primene u stvarnom protokolu, kakav je TCP, nastaju brojni, na prvi pogled teško uočljivi problemi. Možda je najočiglednije pitanje: koliko vremena bi trebalo da protekne pre ponovnog slanja. Jasno je da ovaj zastoj mora da bude duži od vremena povratnog puta (RTT) za tu vezu, tj. vremena koje protekne od trenutka kada se segment pošalje, pa dok ne stigne njegova potvrda. Inače bi dolazilo do nepotrebnih ponavljanja. Ali, koliko duže? Kako, pre svega, da se proceni RTT? Da li bi baš svakom nepotvrđenom segmentu trebalo pridružiti zaseban tajmer? Toliko pitanja! Razmatranja u ovom odeljku zasnivaju se na tekstu o protokolu TCP iz knjige [Jacobson 1988] i važećim IETF preporukama za upravljanje TCP tajmerima [RFC 6298].

Procena vremena povratnog puta

Naše proučavanje upravljanja TCP tajmerom počinjemo razmatranjem načina na koji TCP procenjuje vreme povratnog puta između pošiljaoca i primaoca. To se postiže na sledeći način: uzorak vremena povratnog puta RTT za neki segment, koji označavamo sa SampleRTT, jeste vreme od trenutka slanja segmenta (odnosno, predavanja IP protokolu) do prijema potvrde tog segmenta. Umesto merenja SampleRTT za svaki preneti segment u većini postojećih implementacija protokola TCP meri se samo po jedan uzorak SampleRTT. To jest, u nekom trenutku procenjuje se SampleRTT za samo jedan preneti, ali još uvek nepotvrđeni segment, tako da se dobija nova vrednost SampleRTT, približno jednom tokom svakog povratnog puta. Osim toga, TCP nikad ne izračunava SampleRTT za segment koji se šalje ponovo; SampleRTT se meri samo za segmente koji su poslati jednom [Karn 1987]. (U jednom zadatku pri kraju poglavlja traži se da odgovorite zašto.)

Očigledno je da se vrednost SampleRTT menja od segmenta do segmenta, u zavisnosti od zagušenja na ruterima i od različitog opterećenja krajnjih sistema. Zbog ovih razlika, pojedine vrednosti SampleRTT mogu biti neuobičajene. Da bi se procenio tipični RTT, prirodno je da se uzme neki prosek, dobijen od više vrednosti SampleRTT. TCP stalno izračunava prosek izmerenih vrednosti SampleRTT, nazvan EstimatedRTT. Čim dobije novi SampleRTT, TCP ažurira promenljivu EstimatedRTT, prema sledećem obrascu:

```
EstimatedRTT = (1 - α) • EstimatedRTT + α • SampleRTT
```

Gornja formula, napisana je u obliku iskaza u programskom jeziku – nova vrednost za EstimatedRTT je ponderisana kombinacija prethodne vrednosti EstimatedRTT i nove vrednosti SampleRTT. Preporučena vrednost za α je 0,125 (tj. 1/8) [RFC 6298], pa u tom slučaju gornji obrazac postaje:

```
EstimatedRTT = 0.875 • EstimatedRTT + 0.125 • SampleRTT
```

Obratite pažnju na to da je EstimatedRTT ponderisani prosek vrednosti SampleRTT. Kao što je opisano u domaćem zadatku na kraju poglavlja, ovaj ponderisani prosek daje veći značaj novijim uzorcima nego starijim. To je prirodno, jer noviji uzorci bolje odražavaju trenutno zagušenje mreže. U statistici se ovakav prosek naziva **eksponencijalno ponderisani klizni prosek** (exponential weighted moving average, **EWMA**). Ovaj prosek je „eksponencijalan", zato što značaj (pon-

der) date vrednosti SampleRTT eksponencijalno opada sa svakim sledećim ažuriranjem. U domaćim zadacima biće zatraženo da izvedete eksponencijalni izraz za EstimatedRTT.

Protokol TCP obezbeđuje pouzdan prenos podataka, koristeći pozitivne potvrde prijema i tajmere na način, po mnogo čemu sličan onome koji smo proučavali u odeljku 3.4. TCP potvrđuje prijem podataka koji su ispravno primljeni, a zatim ponovo prenosi segmente, kada se misli da su segmenti ili njima odgovarajuće potvrde prijema izgubljeni ili oštećeni. Pojedine verzije protokola TCP koriste i posredan NAK mehanizam - sa mehanizmom brzog ponovnog slanja, prijem tri potvrde prijema za određeni segment služi kao posredna NAK potvrda za sledeći segment, tako da se pre isteka određenog vremena ponovo pokreće slanje tog segmenta. TCP koristi redne brojeve koji primaocu omogućavaju da prepozna izgubljene ili duplirane segmente. Kao i u slučaju našeg protokola za pouzdan prenos podataka, rdt3.0, TCP ne može sa sigurnošću da tvrdi da li je neki segment, ili njegov ACK, izgubljen, oštećen, ili da li previše kasne. Na strani pošiljaoca, dužnost protokola TCP je ista: ponovno slanje segmenta koji je u pitanju.

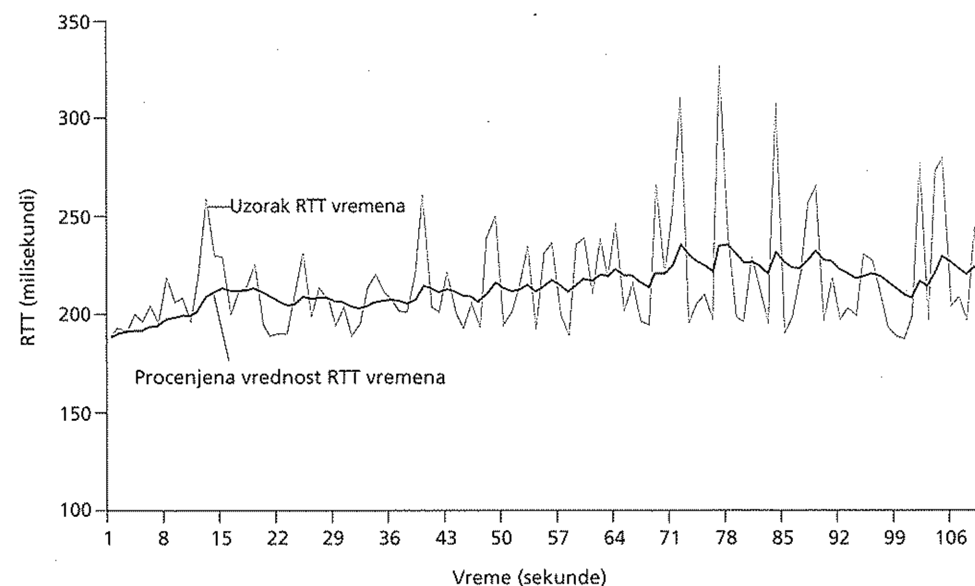
TCP takođe koristi cevovodnu obradu, što pošiljaocima omogućava da imaju više poslatih, ali još uvek nepotvrđenih segmenata u bilo kom trenutku. Ranije smo videli da cevovodna obrada može značajno da poveća propusnu moć, kada je odnos veličine segmenta u odnosu na kašnjenje povrtanog puta mali. Tačan broj nerešenih, nepotvrđenih segmenta, koje pošiljalac može da ima, određen je mehanizmima protokola TCP za kontrolu toka i kontrolu zagušenja. Kontrola toka protokola TCP razmatra se pri kraju ovog odeljka; kontrola zagušenja protokola TCP razmatra se u odeljku 3.7. Za sada je dovoljno da imate na umu da TCP pošiljalac koristi cevovodnu obradu.

Na slici 3.32 prikazane su vrednosti SampleRTT i EstimatedRTT za vrednost $\alpha = 1/8$ za TCP vezu između računara gaia.cs.umass.edu (u gradu Amherst u Masačusetsu) i računara fantasia.eurecom.fr (na jugu Francuske). Jasno je da se varijacije vrednosti SampleRTT izravnavaju izračunavanjem vrednosti EstimatedRTT.

Osim procene trajanja povratnog puta – RTT, vredno je posedovati i meru varijacije vrednosti RTT. Dokument [RFC 6298] definiše varijaciju vremena povratnog puta – DevRTT, kao procenu standardne devijacije SampleRTT od EstimatedRTT:

```
DevRTT = (1 - ß) • DevRTT + ß •| SampleRTT - EstimatedRTT |
```

Obratite pažnju na to da je DevRTT eksponencijalno ponderisani klizni prosek razlike između SampleRTT i EstimatedRTT. Ako vrednosti SampleRTT malo odstupaju, onda je DevRTT malo; a kada su ova odstupanja veća, DevRTT je veliko. Preporučena vrednost za β je 0,25.



Slika 3.32 ◆ Uzorci RTT vremena i procenjene vrednosti RTT vremena

Postavljanje i upravljanje vremenom tajmera pre ponovnog slanja

Sada, kada imamo vrednosti EstimatedRTT i DevRTT, koju bi vrednost trebalo koristiti kao vreme koje protekne pre ponovnog slanja u protokolu TCP? Jasno je da bi to vreme trebalo da bude veće ili jednako vrednosti EstimatedRTT, inače bi dolazilo do nepotrebnih ponovnih slanja. Ali, ne bi smelo da bude ni mnogo veće od EstimatedRTT; inače TCP ne bi dovoljno brzo ponovo poslao segment u slučaju gubitka, čime bi došlo do značajnog kašnjenja pri prenosu podataka. Zato je poželjno da ovo vreme bude jednako EstimatedRTT, uz neku rezervu. Rezerva bi trebalo da bude velika u slučaju značajnih odstupanja vrednosti SampleRTT, a manja u slučaju manjih odstupanja. Ovde u igru ulazi vrednost DevRTT. Sva ova razmatranja uzeta su u obzir u načinu na koji TCP određuje vreme koje protekne pre ponovnog slanja:

```
TimeoutInterval = EstimatedRTT + 4 • DevRTT
```

Dokument [RFC 6298] preporučuje početnu vrednost TimeoutInterval od 1 sekunde. Takođe, kada se pojavi istek vremena, vrednost TimeoutInterval će biti duplirana, kako bi se izbegla pojava isteka pre vremena za naredni segment, koji će uskoro biti potvrđen. Međutim, čim se segment preuzme, a vrednost EstimatedRTT ažurira, vrednost TimeoutInterval se ponovo računa, pomoću gore navedene formule.

3.5.4 Pouzdan prenos podataka

Sećate se da je usluga mrežnog sloja interneta (IP usluga) nepouzdana. IP ne garantuje isporuku datagrama, ne garantuje pravilan redosled isporuke datagrama i ne garantuje integritet podataka u datagramima. Kada se koristi usluga IP, datagrami mogu da preplave bafere rutera i da nikada ne stignu do odredišta; mogu da stižu izvan redosleda, a bitovi u datagramu mogu da se oštete (da se promene iz 0 u 1 i obratno). Pošto se segmenti transportnog sloja prenose preko mreže IP datagramima, svi ti problemi mogu da se jave i u segmentima transportnog sloja.

TCP ostvaruje **uslugu pouzdanog prenosa podataka** preko usluge nepouzdanog, najboljeg mogućeg pokušaja isporuke protokola IP. Usluga pouzdanog prenosa podataka protokola TCP obezbeđuje da tok podataka koji proces čita iz svog prijemnog bafera protokola TCP bude neoštećen, bez praznina, bez duplikata i u pravilnom redosledu; tj. da tok bajtova bude tačno onakav kakav je poslao krajnji sistem na drugoj strani veze. Način na koji TCP obezbeđuje pouzdan prenos podataka obuhvata većinu principa koje smo proučili u odeljku 3.4.

Kada smo razmatrali tehnike pouzdanog prenosa podataka, bilo je najlakše pretpostaviti da se svakom predatom, a još nepotvrđenom segmentu, pridružuje zaseban tajmer. Mada je ovo teoretski odlično, upravljanje većim brojem tajmera predstavljalo bi značajno programsko opterećenje. Zbog toga se u procedurama za upravljanje tajmerima u protokolu TCP [RFC 6298] preporučuje *samo jedan* tajmer za ponovno slanje, iako ima više poslatih, a još nepotvrđenih segmenata. Protokol TCP koji opisujemo u ovom odeljku, zasnovan je na ovoj preporuci sa samo jednim tajmerom.

Način na koji TCP obezbeđuje pouzdan prenos podataka opisaćemo u dva postepena koraka. Prvo predstavljamo krajnje pojednostavljeni opis TCP pošiljaoca, gde se za oporavak od izgubljenih segmenata koristi samo istek vremena tajmera; zatim dajemo potpuniji opis u kojem se, osim isteka vremena tajmera koriste i duplikati potvrda. U sledećem razmatranju, polazimo od pretpostavke da se podaci šalju samo u jednom smeru, od računara A do računara B i da računar A šalje veliku datoteku.

Na slici 3.33 prikazan je krajnje pojednostavljeni opis TCP pošiljaoca. Vidimo da kod TCP pošiljaoca postoje tri glavna događaja u vezi sa prvobitnim i ponovnim slanjem podataka: primanje podataka od aplikacije iznad, istek vremena tajmera i prijem ACK potvrde. Kada nastupi prvi glavni događaj, TCP prima podatke od aplikacije, enkapsulira ih u segment i taj segment predaje protokolu IP. Obratite pažnju na to da svaki segment sadrži redni broj, koji predstavlja broj prvog bajta podataka tog segmenta u toku bajtova, kao što je opisano u odeljku 3.5.2. Takođe, obratite pažnju na to da TCP pokreće tajmer kada preda segment protokolu IP, ako tajmer nije već uključen, zbog nekog drugog segmenta. (Može se zamisliti da je tajmer pridružen najstarijem nepotvrđenom segmentu.) Istek vremena za ovaj tajmer je TimeoutInterval, koji se izračunava na osnovu vrednosti EstimatedRTT i DevRTT, kao što je opisano u odeljku 3.5.3.


```
/* Pretpostavka je da pošiljalac nije ograničen kontrolom toka ili zagušenja
protokola TCP, da je veličina podataka odozgo manja od MSS i da se prenos
podataka vrši samo u jednom smeru. */

NextSeqNum=InitialSeqNumber
SendBase=InitialSeqNumber

loop (forever) { switch(event)
   event: data received from application above
      create TCP segment with sequence number NextSeqNum
      if (timer currently not running)
         start timer
      pass segment to IP
      NextSeqNum=NextSeqNum+length(data)
      break;

   event: timer timeout
      retransmit not-yet-acknowledged segment with
         smallest sequence number
      start timer
      break;

   event: ACK received, with ACK field value of y
      if (y > SendBase) {
          SendBase=y
          if (there are currently any not-yet-acknowledged segments)
               start timer
          }
      break;

} /* kraj beskonačne petlje */
```

Slika 3.33 ◆ Jednostavan TCP pošiljalac

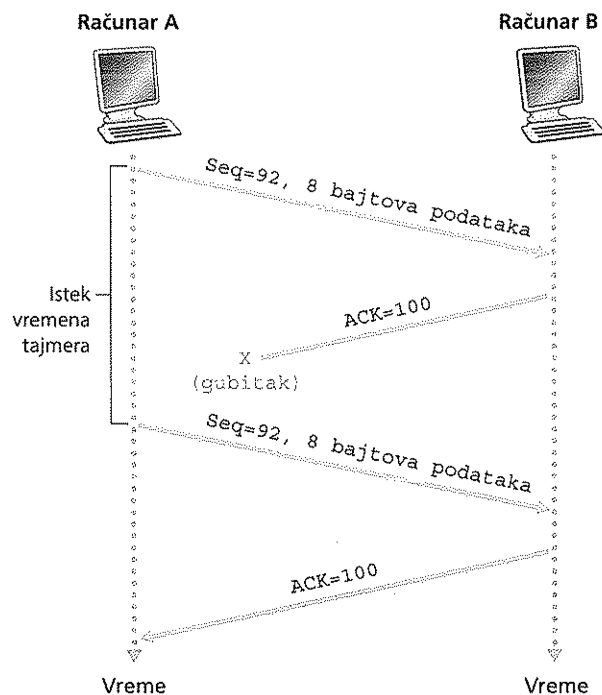
Drugi, glavni događaj je istek vremena tajmera. TCP se po isteku vremena tajmera ponaša tako što ponovo šalje segment, koji je izazvao ovaj događaj. TCP zatim ponovo pokreće tajmer.

Treći važan događaj koji TCP pošiljalac mora da obradi je pristizanje segmenta potvrde (ACK) od primaoca (tačnije, segment koji sadrži važeću vrednost u polju ACK). Kada nastupi ovaj događaj, TCP poredi ACK vrednost y sa svojom promenljivom SendBase. Promenljiva TCP stanja SendBase je redni broj najstarijeg nepotvrđenog bajta. (Prema tome, SendBase–1 je redni broj poslednjeg bajta za koji se zna da ga je primalac primio pravilno i po redu.) Kao što je ranije napomenuto, TCP koristi kumulativne potvrde, pa tako y potvrđuje prijem svih bajtova pre bajta broj y. Ako je y > SendBase, onda je taj ACK potvrda prijema za jedan ili više prethodno

nepotvrđenih segmenata. Na taj način pošiljalac ažurira promenljivu SendBase; on takođe ponovo pokreće tajmer, ako ima nekih još nepotvrđenih segmenata.

Nekoliko zanimljivih slučajeva

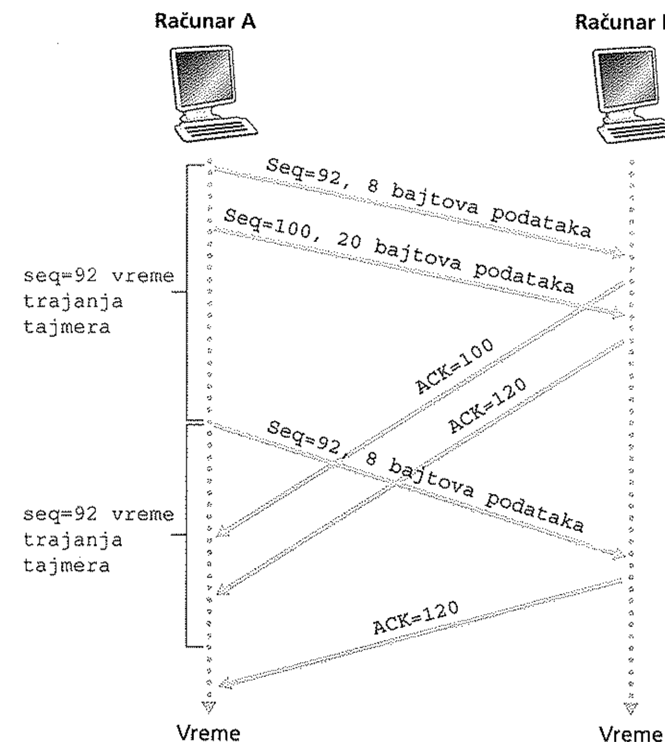
Upravo smo opisali krajnje pojednostavljenu verziju kako TCP obezbeđuje pouzdan prenos podataka. Ali, čak i ta, krajnje pojednostavljena verzija, krije brojne, teško uočljive osobine. Da biste bolje uvideli kako ovaj protokol radi, pogledajmo nekoliko jednostavnih slučajeva. Na slici 3.34 prikazan je prvi slučaj u kojem računar A šalje jedan segment računaru B. Uzmimo da segment ima redni broj 92 i da sadrži osam bajtova podataka. Kada pošalje ovaj segment, računar A čeka od računara B segment sa brojem potvrde 100. Iako je segment iz A stigao do računara B, potvrda od B prema A se gubi. U ovom slučaju, nastupa događaj isteka vremena tajmera i računar A ponovo šalje isti segment. Naravno, kada računar B ponovo primi isti segment, on će na osnovu rednog broja zaključiti da taj segment sadrži podatke koji su već primljeni. Stoga, TCP na računaru B odbacuje bajtove iz ponovo poslatog segmenta.



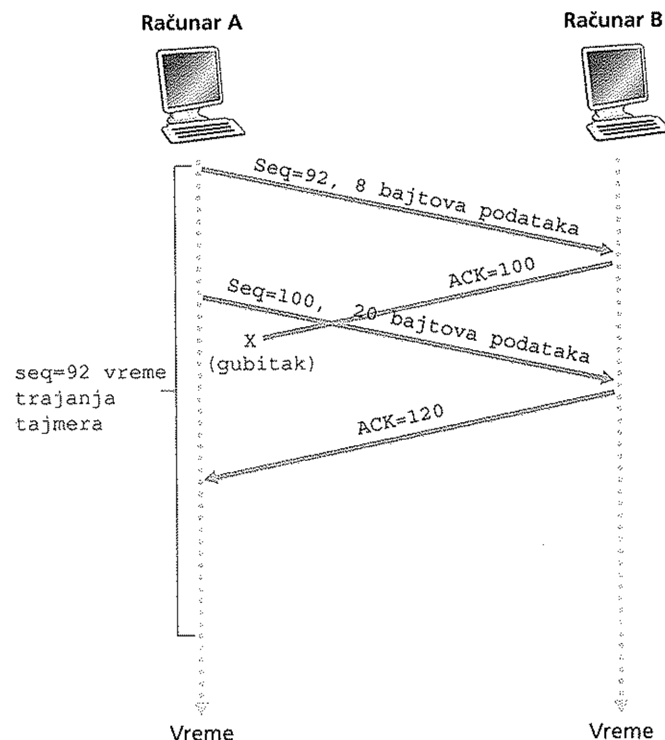
Slika 3.34 ◈ Ponovno slanje zbog izgubljene potvrde prijema

U drugom slučaju, prikazanom na slici 3.35, računar A šalje dva segmenta, jedan za drugim. Prvi segment ima redni broj 92 i 8 bajtova podataka, a drugi segment ima redni broj 100 i 20 bajtova podataka. Pretpostavimo da oba segmenta stižu neoštećena do računara B i da računar B šalje dve zasebne potvrde za oba ta segmenata. Prvi broj potvrde je 100, a drugi 120. Uzmimo sada da nijedna od ovih potvrda do računara A ne stigne pre isteka vremena tajmera. Kada nastupi događaj isteka vremena tajmera, računar A ponovo šalje prvi segment sa rednim brojem 92 i ponovo pokreće tajmer. Ako ACK za drugi segment stigne pre isteka novog vremena, drugi segment neće biti ponovo poslat.

U trećem i poslednjem slučaju, uzmimo da računar A šalje dva segmenta, isto kao u drugom primeru. Potvrda prvog segmenta se gubi u mreži, ali neposredno pre događaja isteka vremena tajmera, računar A prima potvrdu sa brojem 120. Prema tome, računar A zna da je računar B primio *sve*, zaključno sa bajtom 119, pa ne šalje ponovo nijedan od ova dva segmenta. Ovaj slučaj prikazan je na slici 3.36.



Slika 3.35 ◈ Segment 100 se ne šalje ponovo



Slika 3.36 ◈ Kumulativnim potvrđivanjem se izbegava ponovno slanje prvog segmenta

Udvostručavanje vremena trajanja tajmera

Sada ćemo razmotriti nekoliko izmena koje se koriste u većini realizacija protokola TCP. Prva se tiče vremena trajanja tajmera, pošto na tajmeru istekne vreme. U ovoj izmeni, kad god nastupi događaj isteka vremena, TCP ponovo šalje nepotvrđeni segment sa najmanjim rednim brojem, kao što je već opisano. Ali, svaki put, kada TCP ponovo šalje neki segment, on udvostručava vreme trajanja tajmera u odnosu na prethodnu vrednost, umesto da ga ponovo izračuna na osnovu poslednjih vrednosti `EstimatedRTT` i `DevRTT` (onako kao što je opisano u odeljku 3.5.3). Na primer, uzmimo da `TimeoutInterval`, pridružen najstarijem još nepotvrđenom segmentu, kada prvi put istekne vreme, iznosi 0,75 sekunde. TCP će tada ponovo poslati ovaj segment i postaviti novo vreme tajmera na 1,5 sekundu. Ako ovo vreme od 1,5 sekunde još jednom istekne, TCP će opet ponoviti slanje tog segmenta, ali ovog puta postavlja 3,0 sekunde, kao vreme koje je potrebno da istekne pre ponovnog slanja. Stoga, vreme koje je potrebno da istekne pre svakog ponovnog slanja raste eksponencijalno. Međutim, ako se tajmer ponovo pokreće, nakon što se dogodi neki od druga dva događaja (tj. ako se prime podaci od aplikacije odozgo, ili se primi ACK potvrda), vrednost `TimeoutInterval` izračunava se od najnovijih vrednosti `EstimatedRTT` i `DevRTT`.

Ovom izmenom postiže se kontrola zagušenja u ograničenoj meri. (Potpunije oblike TCP kontrole zagušenja proučićemo u odeljku 3.7.) Do isteka vremena došlo je najverovatnije zbog zagušenja na mreži, odnosno prevelikog broja paketa koji stižu na jedan (ili više) redova za čekanje u ruterima na putanji od izvora do odredišta, zbog čega se paketi odbacuju i/ili dugo čekaju u redovima. Ako tokom zagušenja izvori uporno nastave da ponavljaju slanje paketa, zagušenje može da postane još gore. Umesto toga, TCP postupa uviđavnije, tako što svaki pošiljalac ponavlja slanje u sve dužim i dužim vremenskim razmacima. Kada budemo razmatrali CSMA/CD u poglavlju 5, videćemo da Eternet koristi sličan pristup.

Brzo ponovno slanje

Jedan od problema ponovnog slanja, koje se pokreće istekom vremena tajmera, jeste u tome što to vreme može da bude relativno dugo. Kada se segment izgubi, pošiljalac mora dugo da čeka pre nego što ponovo pošalje izgubljeni paket, pa tako povećava kašnjenje s kraja na kraj. Srećom, pošiljalac često može da otkrije gubitke paketâ mnogo pre isteka ovog vremena, kada primi takozvane duplirane ACK potvrde. **Duplirani ACK** je ACK kojim se ponovo potvrđuje prijem segmenta, za koji je pošiljalac već ranije primio takvu potvrdu. Da bismo shvatili šta pošiljalac radi u slučaju dupliranja ACK potvrde, moramo pre toga da pogledamo zašto primalac uopšte šalje ponovljeni ACK. U tabeli 3.2 ukratko je navedeno kako TCP primalac potvrđuje prijem segmenata [RFC 5681]. Kada TCP primalac primi segment sa rednim brojem većim od sledećeg očekivanog rednog broja, on primećuje prazninu u toku podataka – tj. primećuje da neki segment nedostaje. Ova praznina može da bude posledica gubitka ili promene redosleda segmenta unutar mreže. Pošto TCP ne koristi negativne potvrde prijema, primalac ne može da vrati jasnu negativnu potvrdu pošiljaocu. Umesto toga, on jednostavno ponovo potvrđuje (odnosno, pravi još jednu ACK potvrdu) za poslednji bajt podataka, koji je primio u pravilnom redosledu. (Obratite pažnju na to da je u tabeli 3.2 prikazan slučaj kada primalac ne odbacuje segmente van redosleda.)

Događaj	Postupak TCP primaoca
Pristizanje segmenta po redosledu sa očekivanim rednim brojem. Već je potvrđen prijem svih podataka do očekivanog rednog broja.	Odlaže slanje ACK potvrde. Čeka 500 msec na pristizanje sledećeg segmenta po redosledu. Ukoliko sledeći segment po redu ne stigne za to vreme, šalje ACK potvrdu.
Pristizanje segmenta po redosledu sa očekivanim rednim brojem. Ranije pristigli segment po redosledu čeka slanje ACK potvrde.	Odmah šalje kumulativni ACK, potvrđujući prijem oba segmenta, pristigla po redosledu.
Pristizanje segmenta van redosleda sa rednim brojem većim od očekivanog. Otkriva se praznina.	Odmah šalje duplirani ACK, u kome navodi redni broj sledećeg bajta, koji očekuje (koji je jednak donjem kraju u praznini).
Pristizanje segmenta koji delimično ili potpuno ispunjava prazninu u primljenim podacima.	Odmah šalje ACK, pod uslovom da taj segment počinje od donjeg kraja praznine.

Tabela 3.2 ◈ Preporuke za pravljenje ACK potvrda za protokol TCP [5681]

Pošto pošiljalac često šalje više segmenata jedan za drugim, kada se jedan segment izgubi, verovatno će biti mnogo uzastopnih dvostrukih ACK potvrda. Ako TCP pošiljalac primi tri ACK potvrde za iste podatke, on to uzima kao znak da se izgubio segment iza onog koji je tri puta potvrđen. (U problemima za domaći zadatak razmatramo pitanje zašto pošiljalac čeka da dobije trostruke ACK potvrde, umesto samo jedne ponovljene ACK potvrde.) U slučaju kada primi tri ponovljene ACK potvrde, TCP pošiljalac preduzima **brzo ponovno slanje** [RFC 5681] – *pre* isteka vremena tajmera, ponovo šalje segment koji nedostaje. Ovo je prikazano na slici 3.37, na kojoj je drugi segment izgubljen, zatim ponovo poslat pre isteka tajmera. U protokolu TCP sa brzim ponovnim slanjem, sledeći deo koda zamenjuje događaj primanja ACK potvrde sa slike 3.33:

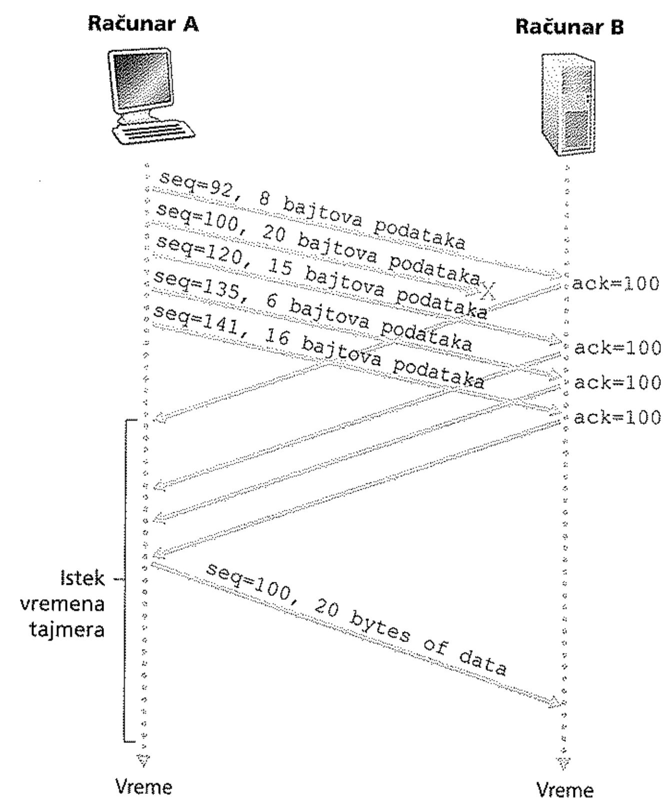
```
event: ACK received, with ACK field value of y
        if (y > SendBase) {
                SendBase=y
                if (there are currently any not yet
                          acknowledged segments) start timer
                }

        else { /* duplirana ACK potvrda za već potvrđen
                  segment */
            increment number of duplicate ACKs received for y
            if (number of duplicate ACKS received
                for y==3)
                /* TCP brzo ponovno slanje */
                resend segment with sequence number y
            }
        break;
```

Već smo primetili da se javlja mnogo skrivenih pitanja, kada se u stvarnom protokolu kakav je TCP primenjuje mehanizam sa istekom vremena i ponovnim slanjem. Gore pomenuti postupci, koji su nastali kao rezultat više od 20 godina iskustva sa tajmerima u protokolu TCP, trebalo bi da vas ubede da je zaista tako!

Vrati-se-za-N ili selektivno ponavljanje?

Naše proučavanje mehanizma za oporavak od grešaka protokola TCP zaključujemo sledećim pitanjem: da li je TCP protokol GBN ili protokol SR? Sećate se da su potvrde prijema protokola TCP kumulativne i da primalac segmente primljene pravilno, ali izvan redosleda, ne potvrđuje zasebno. Zato, kao što je prikazano na slici 3.33 (pogledajte takođe sliku 3.19), TCP pošiljalac mora da čuva samo najmanji redni broj poslatog i nepotvrđenog bajta (SendBase) i redni broj sledećeg bajta koji bi trebalo da se pošalje (NextSeqNum). U tom pogledu, TCP dosta liči na GBN

protokol, ali ipak postoje i velike razlike između protokola TCP i vrati-se-za-N. U većini praktično ostvarenih protokola TCP privremeno se čuvaju segmenti koji su primljeni pravilno, mada izvan redosleda [Stevens 1994]. Razmotrimo takođe šta se događa kada pošiljalac pošalje niz segmenata 1, 2, ..., N i svi oni stignu do primaoca ispravno i u pravilnom redosledu. Pretpostavimo da se izgubi potvrda za paket $n < N$, a da preostalih $N - 1$ potvrda stigne do pošiljaoca pre isteka njihovih vremena tajmera. U takvom slučaju, protokol GBN bi ponovo poslao ne samo paket n, već i sve naredne pakete $n + 1, n + 2, ..., N$. S druge strane, TCP bi ponovo poslao najviše jedan segment, tačnije segment n. Štaviše, TCP ne bi poslao čak ni segment n, ako bi potvrda za segment $n + 1$ stigla pre isteka vremena za segment n.



Slika 3.37 ◇ Brzo ponovno slanje: ponovno slanje segmenta koji nedostaje pre nego što istekne vreme za taj segment

Predložena izmena protokola TCP, takozvano **selektivno potvrđivanje** [RFC 2018], omogućava TCP primaocu da selektivno potvrđuje segmente, primljene izvan redosleda, umesto da kumulativno potvrđuje poslednji pravilano primljen se-

gment u ispravnom redosledu. Kada se kombinuje sa selektivnim ponovnim slanjem – gde se izostavlja ponovno slanje segmenata, koje je primalac selektivno potvrdio – TCP veoma liči na naš opšti protokol tipa SR. Prema tome, mehanizam za oporavak od grešaka protokola TCP bi najbolje bilo razvrstati kao mešavinu protokola GBN i protokola sa selektivnim ponavljanjem.

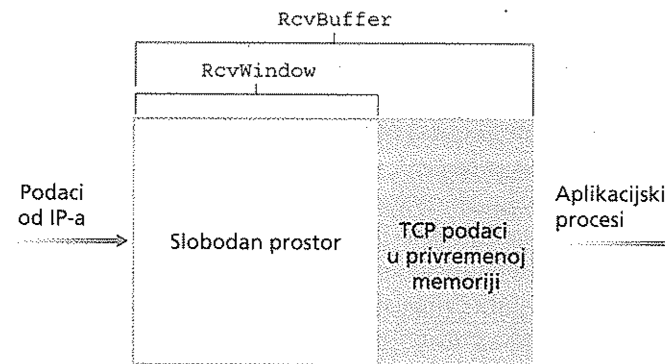
3.5.5 Kontrola toka

Verovatno se sećate da računari na obe strane TCP veze izdvajaju deo privremene memorije za prijemni bafer za tu vezu. Kada TCP veza primi bajtove koji su ispravni i u pravilnom redosledu, ona ih stavlja u prijemni bafer. Odgovarajući proces aplikacije čita podatke iz ovog bafera, ali ne obavezno odmah, čim ti podaci stignu. U stvari, prijemna aplikacija je možda zauzeta nekim drugim zadatkom i čak možda neće uspeti da učita podatke, dugo vremena nakon što su pristigli. Ako aplikacija relativno sporo učitava podatke, pošiljalac može vrlo lako da preplavi prijemni bafer te veze prebrzim slanjem prevelike količine podataka.

TCP svojim aplikacijama nudi **uslugu kontrole toka**, kojom se otklanja mogućnost da pošiljalac preplavi bafer primaoca. Kontrola toka je u stvari usluga usklađivanja brzine – usklađivanja brzine kojom pošiljalac šalje podatke sa brzinom kojom prijemna aplikacija učitava podatke. Kao što je već napomenuto, TCP pošiljalac može da uspori i zbog zagušenja unutar IP mreže; ta vrsta kontrole naziva se **kontrola zagušenja**, tema koju podrobnije obrađujemo u odeljcima 3.6 i 3.7. Iako su to mere koje se preduzimaju kontrolom toka i kontrolom zagušenja slične (usporavanje pošiljaoca), one se očigledno preduzimaju iz veoma različitih razloga. Nažalost, mnogi autori mešaju ove izraze, pa bi mudar čitalac trebalo pažljivo da ih razluči. Sada ćemo opisati kako TCP obezbeđuje uslugu kontrole toka. Da ne bi drveće zaklonilo šumu, u celom ovom odeljku polazimo od pretpostavke da je u pitanju protokol TCP u kome TCP primalac odbacuje segmente koji ne stižu u pravilnom redosledu.

TCP obezbeđuje kontrolu toka tako što *pošiljalac* održava promenljivu nazvanu **prijemni prozor**. Slobodnije rečeno, prijemni prozor se koristida bi pošiljalac mogao da nasluti koliko ima mesta u baferu primaoca. Pošto TCP radi u punom dupleksu, pošiljaoci na obe strane veze održavaju zaseban prijemni prozor. Razmotrimo kako se prijemni prozor koristi prilikom prenosa fajlova. Pretpostavimo da računar A preko TCP veze šalje računaru B veliku datoteku. Računar B dodeljuje toj vezi prijemni bafer; označimo njegovu veličinu sa RcvBuffer. S vremena na vreme, proces aplikacije na računaru B učitava podatke iz ovog bafera. Definisaćemo sledeće promenljive:

- `LastByteRead`: broj poslednjeg bajta u toku podataka koji je proces aplikacije na računaru B pročitao iz bafera.
- `LastByteRcvd`: broj poslednjeg bajta u toku podataka koji je stigao sa mreže i koji je smešten u prijemni bafer na računaru B.



Slika 3.38 ◈ Prijemni prozor (rwnd) i prijemni bafer (RcvBuffer)

Pošto TCP ne sme da prepuni dodeljeni bafer, uvek mora da bude:

`LastByteRcvd - LastByteRead ≤ RcvBuffer`

Prijemni prozor, označen sa rcwd, postavljen je na veličinu slobodnog prostora u baferu:

`rwnd = RcvBuffer - [LastByteRcvd - LastByteRead]`

Pošto se slobodan prostor vremenom menja, rwnd je dinamička promenljiva. Promenljiva rwnd prikazana je na slici 3.38.

Kako veza koristi promenljivu rwnd, da bi obezbedila uslugu kontrole toka? Računar B obaveštava računar A o tome koliko slobodnog prostora ima u baferu veze, tako što trenutnu vrednost rwnd stavlja u polje prijemnog prozora svih segmenata koje šalje računaru A. Na početku, računar B postavlja rwnd = RcvBuffer. Obratite pažnju:da bi se ovo postiglo, računar B mora da prati nekoliko promenljivih koje se odnose na tu vezu.

Računar A održava dve promenljive: LastByteSent (poslednji poslati bajt) i LastByteAcked (poslednji potvrđen bajt). Obratite pažnju na to da je razlika između te dve promenljive, LastByteSent – LastByteAcked, jednaka količini nepotvrđenih podataka, koje je A poslao tom vezom. Ako vodi računa o tome da količina nepotvrđenih podataka ostane manja od vrednosti rwnd, računar A može biti siguran da ne preplavljuje prijemni bafer računara B. Prema tome, računar A tokom čitavog trajanja veze obezbeđuje da je:

`LastByteSent - LastByteAcked ≤ rwnd`

Kod ovog rešenja postoji jedan manji tehnički problem. Da biste ga sagledali, pretpostavimo da se prijemni bafer računara B popuni tako da je rwnd = 0. Pošto

se računaru A objavi da je rwnd = 0, pretpostavimo takođe da računar B nema više *ništa* da pošalje računaru A. Pogledajmo sada šta se događa. Pošto proces aplikacije na računaru B isprazni bafer, TCP ne šalje računaru A novi segment sa novom vrednošću rwnd, jer on to radi samo ako ima podatke za slanje, ili ako bi trebalo da pošalje neku potvrdu. Prema tome, računar A nikada neće saznati da se oslobodio prostor u prijemnom baferu računara B – računar A je blokiran i ne može više da šalje podatke! Da bi se rešio ovaj problem, TCP specifikacija zahteva da računar A nastavi da šalje segmente sa po jednim bajtom podataka, kada veličina prijemnog prozora računara B bude jednaka nuli. Primalac potvrđuje prijem tih segmenata. U jednom trenutku bafer počinje da se prazni i potvrde prijema prenose računaru A vrednost rwnd različitu od nule.

Na internet adresi http://www.awl.com/kuroseross za ovu knjigu nalazi se interaktivni *Java* aplet, koji slikovito prikazuje rad prijemnog prozora protokola TCP.

Pošto smo opisali uslugu kontrole toka protokola TCP, ovde ukratko spominjemo i to da protokol UDP ne nudi kontrolu toka. Da biste shvatili o čemu se tačno radi, razmotrite slanje niza UDP segmenata, od jednog procesa na računaru A do drugog procesa naračunaru B. U uobičajenoj implementaciji protokola UDP, protokol UDP dodaje segmente u bafer konačne veličine, koji „prethodi" odgovarajućem soketu (odnosno, vratima prema procesu). Proces iz bafera odjednom učitava čitav segment. Ako proces ne učitava segmente iz bafera dovoljno brzo, bafer se prepunjava i pojedini segmenti se gube.

3.5.6 Upravljanje TCP vezom

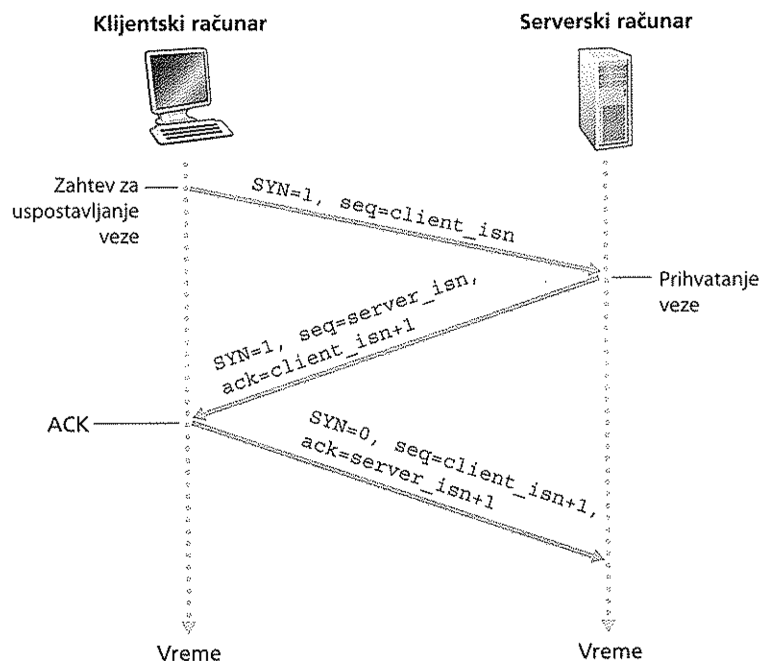
U ovom pododeljku podrobnije razmatramo kako se TCP veza uspostavlja i raskida. Mada ova tema ne izgleda posebno uzbudljiva, veoma je važna, jer uspostavljanje TCP veze može značajno da utiče na kašnjenje koje korisnik opaža (na primer, prilikom krstarenja po vebu). Štaviše, većina najčešćih napada na mrežu – uključujući neverovatno popularan napad plavljenje SYN segmenataima – zloupotrebljava ranjivost upravljanja TCP vezom. Pogledajmo prvo kako se uspostavlja TCP veza. Pretpostavimo da proces koji se izvršava na jednom računaru (klijent) želi da uspostavi vezu sa drugim procesom na drugom računaru (server). Proces klijentske aplikacije prvo obaveštava TCP klijenta da želi da uspostavi vezu sa procesom na serveru. TCP na klijentu zatim kreće u uspostavljanje TCP veze sa protokolom TCP na serveru na, sledeći način:

- *Korak 1.* Klijentska strana protokola TCP prvo šalje poseban TCP segment serverskoj strani protokola TCP. Ovaj poseban segment ne sadrži podatke aplikativnog sloja. Međutim, jedan od bitova oznaka u zaglavlju segmenta (slika 3.29), bit SYN, ima vrednost 1. Zbog toga se ovaj poseban segment naziva SYN segmentom. Osim toga, klijent nasumice bira početni redni broj (client_isn) i stavlja taj broj u polje rednog broja početnog TCP SYN segmenta. Ovaj segment se enkapsulira u IP datagram i šalje serveru. Posebno je važno da se pravilnim, slučajnim izborom vrednosti client_isn izbegnu izvesni bezbednosni rizici [CERT 2001-09].
- *Korak 2.*Kada IP datagram, koji sadrži TCP SYN segment, stigne do računara servera (ako uopšte stigne!), server izvlači TCP SYN segment iz datagrama, toj vezi dodeljuje TCP bafer i promenljive veze i klijentu protokola TCP šalje segment, kojim se odobrava uspostavljanje veze. (Videćemo u poglavlju 8 da pri ovom dodeljivanju bafera i promenljivih, pre nego što se završi treći korak u ovom trostrukom usaglašavanju, protokol TCP može biti izložen napadu odbijanja usluge, poznatom kao plavljenje SYN segmentima.) Ovaj segment odobrenja veze takođe ne sadrži podatke aplikativnog sloja. Međutim, on u zaglavlju segmenta sadrži tri značajne informacije. Prvo, bit SYN postavljen je na 1. Drugo, polje potvrde u zaglavlju TCP segmenta postavljeno je na vrednost client_isn+1. Konačno, server bira i vlastiti početni redni broj (server_isn) i stavlja tu vrednost u polje rednog broja u zaglavlju TCP segmenta. Ovaj segment odobrenja veze kao da kaže: „Primio sam vaš SYN paket za pokretanje veze sa vašim početnim rednim brojem client_isn. Pristajem na uspostavljanje ove veze. Moj početni redni broj je server_isn." Segment odobrenja veze naziva se **SYNACK segment**.
- *Korak 3.* Pošto primi SYNACK segment, klijent takođe ovoj vezi dodeljuje privremene bafere promenljive. Klijentski računar zatim šalje serveru još jedan segment; ovaj poslednji segment potvrđuje prijem serverovog segmenta odobrenja veze (klijent to postiže stavljanjem vrednosti server_isn+1 u polje potvrde u zaglavlju TCP segmenta). Bit SYN postavljen je na 0, pošto je veza uspostavljena. U ovoj trećoj fazi trostrukog usaglašavanja moguće je da se u telu segmenta nalaze podaci koje klijent šalje serveru.

Kada se završe prethodna tri koraka, klijentski i serverski računar mogu jedan drugom da šalju segmente, koji sadrže podatke. U svakom od ovih, budućih segmenata, bit SYN imaće vrednost nula. Obratite pažnju na to da se za uspostavljanje veze između dva računara šalju tri paketa, kao što je prikazano na slici 3.39. Zbog toga se postupak uspostavljanja veze često naziva i **trostruko usaglašavanje**. U problemima za domaći zadatak se trostruko usaglašavanje protokola TCP istražuje sa više stanovišta (Zašto su potrebni početni redni brojevi? Zašto je neophodno trostruko usaglašavanje, a ne dvostruko?). Zanimljivo je da alpinista i stožer (koji stoji ispod alpiniste i čiji je posao da rukuje bezbednosnim užetom alpiniste) za međusobnu komunikaciju koriste trostruko usaglašavanje, koje je istovetno kao ovo iz protokola TCP, kako bi, pre nego što alpinista počne da se penje, bili sigurni da su obojica spremni.

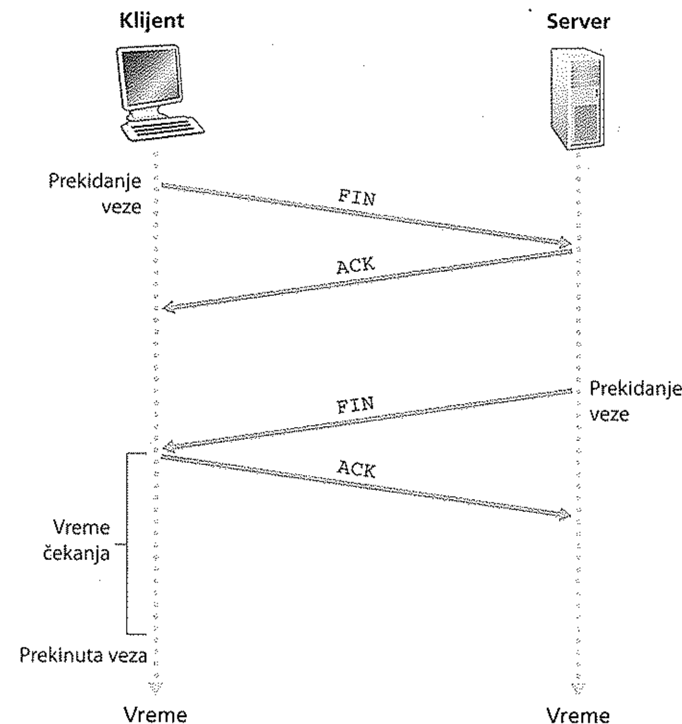
Sve što je dobro ima kraj, pa isto važi i za TCP vezu. Bilo koji od dva procesa koji učestvuju u TCP vezi može da je prekine. Kada se veza okonča, „resursi" (odnosno, bafer i promenljive) u računarima se oslobađaju. Na primer, pretposta-

vimo da klijent odluči da prekine vezu, kao što je prikazano na slici 3.40. Proces klijentske aplikacije izdaje komandu za prekidanje veze. TCP klijent zbog toga šalje serverskom procesu poseban TCP segment. Bit oznake FIN (videti sliku 3.29) u zaglavlju ovog posebnog segmenta postavljen je na 1. Server zatim šalje vlastiti segment prekida, u kome je bit FIN postavljen na 1. Na kraju, klijent potvrđuje prijem serverovog segmenta prekida. U tom trenutku oslobađaju se svi resursi na oba računara.



Slika 3.39 ◈ Trostruko usaglašavanje protokola TCP: razmena segmenata

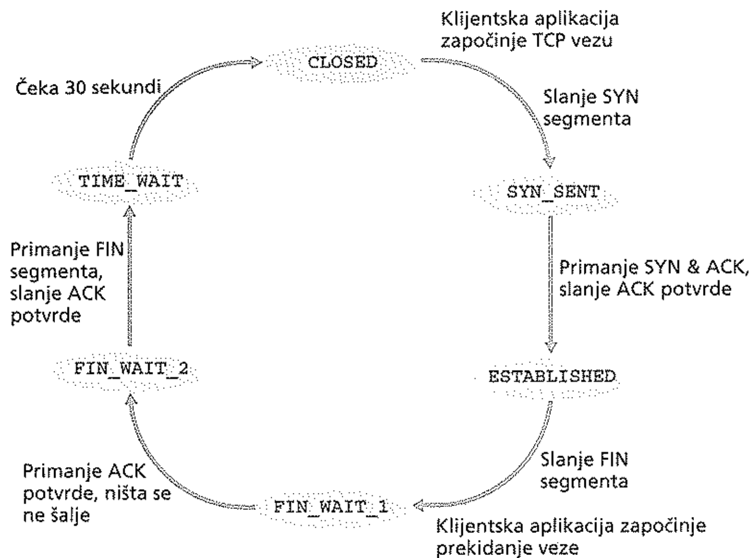
Tokom trajanja TCP veze, protokol TCP koji se izvršava na oba računara prolazi kroz različita **TCP stanja**. Na slici 3.41 prikazan je uobičajen niz TCP stanja kroz koja prolazi TCP *klijent*. TCP klijent je na početku zatvoren (stanje CLOSED). Aplikacija na klijentskoj strani otpočinje novu TCP vezu (pravljenjem objekta Socket u našim Python primerima iz poglavlja 2). Zbog toga TCP klijent šalje SYN segment do TCP servera. Pošto pošalje SYN segment, TCP klijenta prelazi u stanje SYN_SENT. Dok je u ovom stanju, TCP klijent očekuje segment od TCP servera koji sadrži potvrdu za klijentov prethodni segment i čiji je bit SYN postavljen na 1. Kada primi takav segment, TCP klijent prelazi u stanje ESTABLISHED, koji znači uspostavljenu vezu. Dok je u ovom stanju, TCP klijent može da šalje i prima TCP segmente, koji sadrže korisne podatke (odnosno, podatke iz aplikacije).



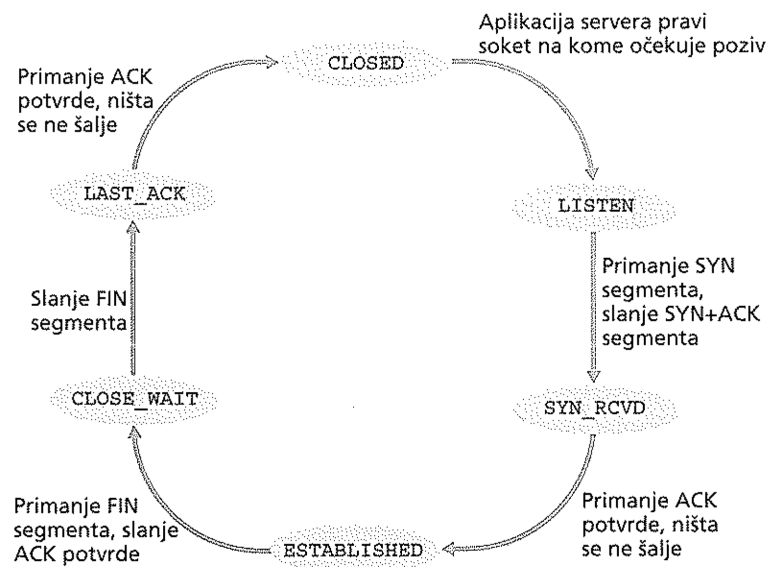
Slika 3.40 ◈ Prekidanje TCP veze

Uzmimo da klijentska aplikacija odluči da prekine vezu (mada i server može da odluči da prekine vezu). Zbog toga TCP klijent šalje TCP segment sa bitom FIN postavljenim na 1 i prelazi u stanje FIN_WAIT_1. Dok je u ovom stanju, TCP klijent očekuje od TCP servera segment sa potvrdom. Kada primi taj segment, TCP klijent prelazi u stanje FIN_WAIT_2. Dok je u stanju FIN_WAIT_2, klijent čeka na drugi segment sa servera, gde je bit FIN jednak 1; kada primi taj segment, TCP klijent potvrđuje serverov segment i prelazi u stanje TIME_WAIT. Stanje TIME_WAIT omogućava TCP klijentu da ponovo pošalje konačnu potvrdu u slučaju da se ACK potvrda izgubi. Vreme provedeno u stanju TIME_WAIT zavisi od načina implementacije protokola, ali uobičajene vrednosti su 30 sekundi, 1 minut ili 2 minuta. Nakon čekanja, veza se zvanično zatvara i svi resursi na strani klijenta (uključujući i brojeve portova) se oslobađaju.

Na slici 3.42 prikazan je niz stanja kroz koja prolazi TCP na strani servera, pod pretpostavkom da klijent pokreće postupak za prekidanje veze. Prelasci iz stanja u stanje su očigledni. Na ova dva dijagrama promene stanja prikazali smo samo kako se TCP veza uobičajeno uspostavlja i prekida. Nismo opisali šta se događa u nekim neuobičajenim slučajevima, na primer, kada obe strane veze istovremeno žele da pokrenu ili prekinu vezu. Ako vas zanima da proučite to i neka druga napredna pitanja koja se tiču protokola TCP pogledajte vrlo razumljivu knjigu [Stevens 1994].



Slika 3.41 ◈ Uobičajeni niz TCP stanja kroz koja prolazi TCP klijent



Slika 3.42 ◈ Uobičajeni niz TCP stanja kroz koja prolazi TCP na strani servera

BEZBEDNOST PRE SVEGA

NAPAD PLAVLJENJE SYN SEGMENTIMA

Videli smo u našoj priči o trostrukom usaglašavanju protokola TCP da server izdvaja i inicijalizuje promenljive i bafer veze, kao odgovor na primljeni SYN segment. Posle toga, server šalje SYNACK segment u svom odgovoru i očekuje ACK segment od klijenta, što je treći i poslednji korak u trostrukom usaglašavanju, pre nego što se veza potpuno uspostavi. Ukoliko klijent ne pošalje ACK, kako bi okončao treći korak u trostrukom usaglašavanju, server obično (posle jednog minuta ili nešto duže) prekida ovu poluotvorenu vezu i oslobađa izdvojene resurse.

Ovakav način upravljanja TCP vezom ostavlja mesto za klasičan napad odbijanja usluge (DoS), tačnije, **napad plavljenje SYN segmentima (SYN flood attack)**. U ovom napadu, loši momci šalju veći broj TCP SYN segmenata, pri čemu ne sprovode do kraja treći korak za uspostavljanje veze. Zahvaljujući tom mnoštvu segmenata SYN, resursi servera se brzo potroše, jer se izdvoje (ali se nikad ne iskoriste!) za poluotvorene veze. Pošto su resursi servera potrošeni, odbija se pružanje usluge pravim klijentima. Takvi napadi, plavljenjem SYN segmenata, bili su među prvim napadima odbijanja usluge - DoS koji se pominju u dokumentima udruženja CERT [CERT SYN 1996]. Srećom, postoji uspešna odbrana, nazvana **SYN kolačići (SYN cookies)** [RFC 4987], koji su sada razvijeni u većini operativinih sistema. SYN kolačići rade na sledeći način:

- Kada server primi SYN segment, on ne zna da li segment dolazi od legitimnog korisnika, ili je deo napada plavljenja SYN segmentima. Stoga, umesto da kreira poluotvorenu TCP vezu za ovaj SYN segment, server kreira neki početni TCP redni broj koji predstavlja složenu funkciju (heš funkcija), izvorišne i odredišne IP adrese i brojeva portovaSYN segmenta, kao i tajni broj koji zna samo server. Ovaj pažljivo napravljen početni redni broj je tzv. „kolačić". Server tada šalje klijentu SYNACK paket sa ovim posebnim početnim rednim brojem. *Važno, server se ne seća kolačića ili bilo koje informacije stanja, koja odgovara SYN segmentu.*
- Legitimni klijent će vratiti ACK segment. Kada server primi ovu potvrdu prijema, mora da proveri da li ACK odgovara istom SYN segmentu koji je poslat ranije. Ali, koko ovo izvršiti kada server ne održava memoriju o SYN segmentima? Kao što možete i da pretpostavite, to se radi pomoću kolačića. Setite se da je za legitimnu potvrdu prijema vrednost polja za potvrdu jednaka početnom rednom broju u segmentu SYNACK (vrednost kolačića u ovom slučaju) plus jedan (videti sliku 3.39). Server može tada da izvrši istu heš funkciju, koristeći IP adresu izvora i odredišta i brojeve portova u segmentu SYNACK (koji su isti kao i u originalnom SYN segmentu) i tajni broj. Ukoliko je rezultat funkcije plus jedan istikao i vrednost potvrde prijema (kolačić) u SYNACK segmentu poslat klijentu, server zaključuje da ACK odgovara ranijem segmentu SYN i iz tog razloga je važeći. Server kreira potpuno otvorenu vezu preko soketa.
- S druge strane, ukoliko klijent ne vrati neki ACK segment, originalan SYN segment nije načinio štetu serveru, jer serveri još uvek nisu raspodelili nijedan resurs kao odgovor na originalan, izmišljeni segment SYN.

U prethodnom razmatranju pretpostavili smo da su i klijent i server spremni za komunikaciju, odnosno da server očekuje poziv na portu, na koji klijent šalje svoj segment SYN. Razmotrimo šta se događa kada računar primi TCP segment čiji se broj porta ili izvorna IP adresa ne poklapaju ni sa jednim postojećim soketom na računaru. Na primer, pretpostavimo da računar primi TCP SYN paket sa odredišnim portom 80, ali da računar ne prihvata veze na portu 80 (tj. na njegovom portu 80 nije pokrenut veb server). Taj računar će tada poslati izvornom računaru poseban segment za poništavanje zahteva (reset segment). Ovaj TCP segment sadrži bit oznake RST (pogledajte odeljak 3.5.2), postavljen na 1. Kada računar pošalje takav segment, on poručuje izvoru: „Nemam soket za taj segment. Molim nemojte ponovo da šaljete ovaj segment". Kada računar primi UDP paket čiji se broj odredišnog porta ne poklapa ni sa jednim od pokrenutih UDP soketa, on šalje poseban ICMP datagram, kao što je opisano u poglavlju 4.

Pošto smo dobro upoznali kako se upravlja TCP vezama, razmotrimo još jednom alat za skeniranje portova nmap i pobliže razmotrimo kako radi. Da bi istražio određeni TCP port, recimo port 6789, na ciljanom računaru, nmap šalje TCP segment SYN sa odredišnim portom 6789 tom računaru. Moguće je da se dogodi nešto od sledećeg:

- *Izvorni računar dobija TCP segment SYNACK od ciljanog računara.* Pošto to znači da se neka aplikacija izvršava sa TCP portom 6789 na ciljanom računaru, nmap vraća odgovor „otvoren".
- *Izvorni računar dobija TCP segment RST od ciljanog računara.* To znači da je segment SYN stigao do ciljanog računara, ali da se na tom računaru ne izvršava aplikacija sa TCP portom 6789. Ipak, napadač bar zna da segmenti, upućeni do računara na portu 6789, nisu zaustavljeni zaštitnim barijerama na putanji između izvornog i ciljanog računara. (Zaštitne barijere razmatramo u odeljku 8.)
- *Izvorni računar ne dobija ništa.* Ovo verovatno znači da je taj segment SYN zaustavljen nekom zaštitnom barijerom i da nikada nije stigao do ciljanog računara.

Nmap je moćan alat, koji može da se „zloupotrebi", ne samo za otvorene TCP portove, već i za otvorene UDP portove, za zaštitne barijere i njihove konfiguracije, pa čak i za verzije aplikacija i operativnih sistema. Većina toga se obavlja veštim rukovanjem segmentima za upravljanje TCP vezama [Skoudis 2006]. Alatku nmap možete preuzeti sa adrese www.nmap.org.

Ovim zaključujemo uvod u kontrolu grešaka i kontrolu toka u protokolu TCP. U odeljku 3.7 vratićemo se na TCP i nešto podrobnije razmotriti njegovu kontrolu zagušenja. Pre toga se, međutim, vraćamo na razmatranje pitanja koja se tiču kontrole zagušenja u širem smislu.

3.6 Principi kontrole zagušenja

U prethodnim odeljcima ispitali smo opšte principe i određene mehanizme koje TCP, suočen sa gubitkom paketa, koristi da bi obezbedio uslugu pouzdanog prenosa podataka. Već smo pomenuli da, u praksi, do tih gubitaka obično dolazi zbog prepunjavanja privremene memorije, to jest bafera u ruterima, kada je mreža zagušena. Ponovno slanje paketâ leči posledicu zagušene mreže (gubitak određenog segmenta u transportnom sloju), ali ne leči uzrok zagušenja mreže – previše izvora koji pokušavaju da pošalju podatke prevelikom brzinom. Da bi se izlečio uzrok zagušenja, potrebni su mehanizmi koji će da uspore pošiljaoce, kada preti zagušenje mreže.

U ovom odeljku razmotrićemo problem kontrole zagušenja u opštem smislu, pokušavajući da shvatimo zašto je zagušenje loša stvar, kako se zagušenje mreže odražava na performanse aplikacija gornjeg sloja, kao i da proučimo različite mere koje se mogu preduzeti, da bi se izbeglo zagušenje mreže, ili delovalo na njega. Ovakvo, široko proučavanje kontrole zagušenja, u slučaju pouzdanog prenosa podataka, ima opravdanje jer se zagušenje nalazi visoko na našoj listi „prvih deset" najznačajnijih problema umrežavanja. Ovaj odeljak zaključujemo opisom kontrole zagušenja u usluzi **raspoložive bitske brzine ABR** (available bit-rate) u **ATM** (asynchronous transfer mode) **mrežama** tj. **asinhronom režimu prenosa**. Sledeći odeljak sadrži podroban opis algoritma za kontrolu zagušenja protokola TCP.

3.6.1 Uzroci i posledice zagušenja

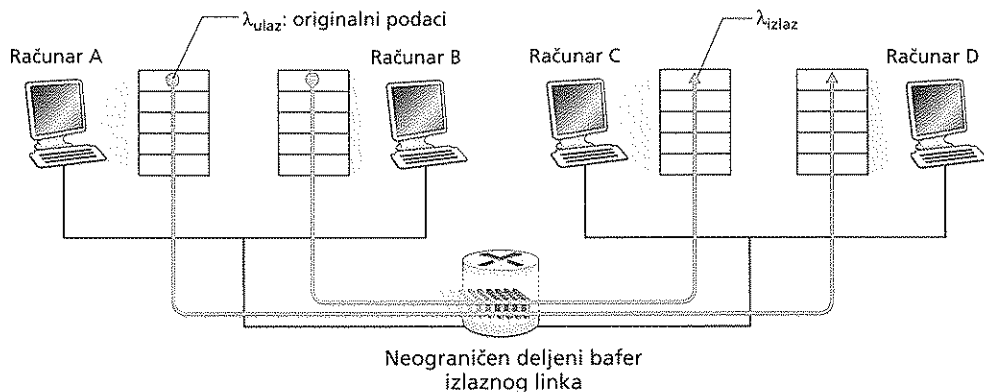
Proučavanje kontrole zagušenja započinjemo ispitivanjem tri, sve složenija slučaja, u kojima dolazi do zagušenja. U svakom od tih slučajeva videćemo zašto uopšte dolazi do zagušenja i koja je cena (u smislu resursa koji se nedovoljno koriste i loših performansi na krajnjim sistemima) koja se plaća zbog zagušenja. Nećemo (za sada) obraćati veću pažnju na to kako se postupa u slučaju pojave zagušenja ili sprečavanja zagušenja, već se usredsređujemo na jednostavnija pitanja, kako bismo razumeli šta se događa kada računari povećavaju brzine prenosa i mreža postaje zagušena.

Slučaj 1: Dva pošiljaoca, jedan ruter sa beskonačnim baferom

Počinjemo sa možda najjednostavnijim slučajem zagušenja: dva računara (A i B) dele vezu sa jednim skokom između izvora i odredišta, kao što je prikazano na slici 3.43.

Pretpostavimo da aplikacija na računaru A šalje podatke vezom (na primer, predavanjem podataka protokolu transportnog sloja preko soketa) prosečnom brzinom od λ_{ulaz} bajtova/sekundi. Ovi podaci su izvorni u smislu: svaka celina podataka samo jednom šalje u soket.. Protokol transportnog sloja ispod aplikacije je jednosta-

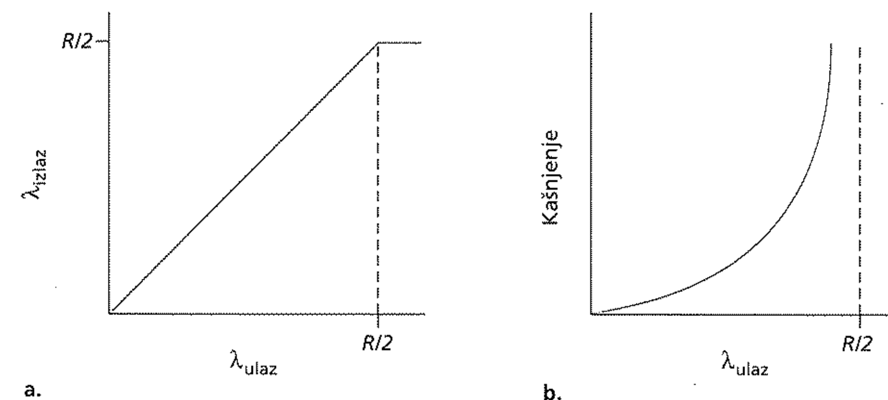
van. Podaci se enkapsuliraju i šalju; nema ispravljanja grešaka (na primer, ponovnim slanjem), kontrole toka, niti kontrole zagušenja. Zanemarujući dodatno opterećenje koje potiče od dodavanja informacija u zaglavlja transportnog sloja i nižih slojeva, brzina saobraćaja koju računar A prosleđuje ruteru iznosi λulaz bajtova/sekundi. Računar B radi na sličan način, pa zbog jednostavnosti pretpostavljamo da i on šalje brzinom od λulaz bajtova/sekundi. Paketi iz računara A i B prolaze kroz ruter i preko deljenog izlaznog linka kapaciteta R. Ruter ima bafer koji mu omogućava čuvanje pristiglih paketa, kada je brzina pristizanja paketa veća od kapaciteta izlaznog linka. U ovom prvom slučaju pretpostavljamo da ruter ima beskonačan bafer.



Slika 3.43 ◈ Slučaj zagušenja 1: dve veze dele link sa jednim skokom sa beskonačnim baferom

Na slici 3.44 prikazane su performanse veze računara A u ovom prvom slučaju. Levi grafikon prikazuje **propusnu moć po vezi** (broj bajtova u sekundi kod primaoca) u zavisnosti od brzine slanja veze. Za brzinu slanja između 0 i $R/2$, propusna moć kod primaoca jednaka je brzini slanja kod pošiljaoca – sve što pošiljalac pošalje, primalac prima sa konačnim kašnjenjem. Međutim, ako je brzina slanja veća od $R/2$, propusna moć iznosi samo $R/2$. Ova gornja granica propusne moći potiče od toga što kapacitet linka dele dve veze. Link jednostavno ne može primaocu da isporuči pakete stalnom brzinom većom od $R/2$. Bez obzira na to koliko računari A i B povećavaju brzine slanja, nijedan od njih nikada neće postići propusnu moć veću od $R/2$.

Postizanje propusne moći od $R/2$ po vezi izgleda kao izuzetan uspeh, jer je link potpuno iskorišćen isporukom paketâ na njihova odredišta. Međutim, desni grafikon na slici 3.44 prikazuje šta se događa kada link radi blizu maksimalnog kapaciteta. Kako se brzina slanja približava vrednosti $R/2$ (sa leve strane), prosečno kašnjenje postaje sve veće i veće. Kada brzina slanja pređe $R/2$, prosečan broj paketa u redu za čekanje na ruteru je neograničen, a prosečno kašnjenje od izvora do odredišta postaje beskonačno (pod pretpostavkom da veze rade ovim brzinama slanja beskonačno dugo i da postoji beskonačan prostor u baferu). Prema tome, iako rad sa ukupnom propusnom moći blizu R može da bude idealan, što se tiče same propusne moći, daleko je od idealnog, što se tiče kašnjenja. *Čak i u ovom (krajnje) idealizovanom slučaju, već smo pronašli jednu od posledica zagušene mreže – veliko kašnjenje u redovima za čekanje, koje nastaje kada se brzina pristizanja paketâ približava kapacitetu linka.*



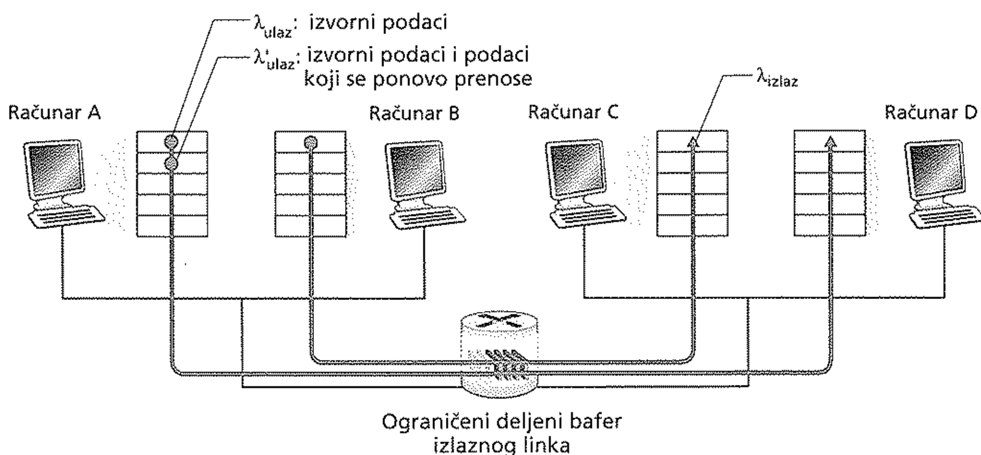
Slika 3.44 ◈ Slučaj zagušenja 1: propusna moć i kašnjenje u zavisnosti od brzine slanja

Slučaj 2: Dva pošiljaoca, jedan ruter sa konačnim baferom

Sada ćemo neznatno prilagoditi prvi slučaj sledećim dvema izmenama (slika 3.45). Prvo, pretpostavljamo da je veličina bafera na ruteru konačna. Posledica ove, sasvim realne pretpostavke je da se paketi odbacuju, ako stignu u već popunjen bafer. Drugo, pretpostavićemo da su obe veze pouzdane. Ako ruter odbaci paket koji sadrži segment transportnog sloja, pošiljalac ga pre ili kasnije ponovo šalje. Pošto se neki paketi šalju ponovo, ovog puta moramo pažljivije da koristimo izraz *brzina slanja*. Drugim rečima, brzinu kojom aplikacija šalje izvorne podatke u logički priključak ponovo ćemo označiti sa λ_{ulaz} bajtova/sekundi. Brzinu kojom transportni sloj šalje segmente (koji sadrže izvorne podatke *kao i* podatke koji se ponovo šalju) u mrežu ćemo označiti sa λ'_{ulaz} bajtova/sekundi. λ'_{ulaz} se ponekad naziva **ponuđeno opterećenje** mreže.

Performanse koje se ostvaruju u slučaju 2 u velikoj meri zavise od toga kako se vrši ponovno slanje. Prvo, uzmimo nerealnu okolnost da je računar A u stanju da nekako (volšebno!) utvrdi da li je bafer u ruteru slobodan i da šalje pakete samo kad je bafer slobodan. U takvim okolnostima ne bi bilo gubitaka paketa, λ_{ulaz} bilo bi jed-

nako λ'_{ulaz}, a propusna moć veze bila bi jednaka λ_{ulaz}. Ovaj slučaj prikazan je na slici 3.46(a). Što se tiče propusne moći, performanse su idealne – sve što se pošalje bude i primljeno. Obratite pažnju na to u ovom slučaju prosečna brzina slanja ne može preći $R/2$, jer smo pretpostavili da nikad ne dolazi do gubitka paketa.

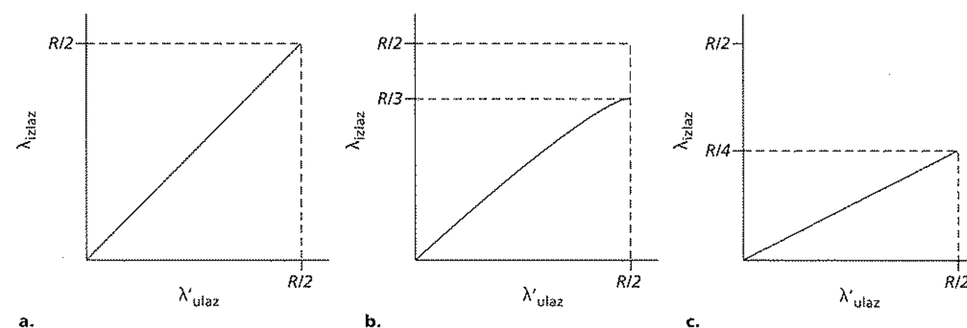


Slika 3.45 ◆ Slučaj 2: dva računara (koji ponovo šalju izgubljene pakete) i ruter sa konačnim baferom

Razmotrimo sledeći malo realniji slučaj, kada pošiljalac ponovo šalje samo one pakete za koje se pouzdano zna da su izgubljeni. (I ova je pretpostavka malo nategnuta. Međutim, predajni računar može da se postavi, da čeka dovoljno dugo pre ponovnog slanja, tako da bude praktično siguran da se paket koji za to vreme nije potvrđen izgubio.) U ovom slučaju, performanse bi izgledale približno kao na slici 3.46(b). Da bismo shvatili šta se ovde događa, razmotrimo slučaj kada ponuđeno opterećenje λ'_{ulaz} (brzina slanja prvobitnih i ponovljenih podataka) iznosi $R/2$. Prema slici 3.46(b), za tu vrednost ponuđenog opterećenja, brzina kojom se podaci isporučuju prijemnoj aplikaciji iznosi $R/3$. Prema tome, od prenetih 0,5 R jedinica podataka, 0,333 R bajtova/sekundi (prosečno) predstavlja izvorne podatke, a 0,166 R bajtova/sekundi (prosečno) predstavlja ponovo poslate podatke. *Ovde vidimo još jednu posledicu zagušene mreže – pošiljalac mora ponovo da šalje podatke, da bi nadoknadio pakete koji su odbačeni (izgubljeni) zbog prepunjenog bafera.*

Na kraju, razmotrimo slučaj, kada kod pošiljaoca prerano istekne vreme i on ponovo pošalje paket koji kasni zbog čekanja u redu, ali nije stvarno izgubljen. U tom slučaju, može se dogoditi da do primaoca stignu i izvorni paket i paket koji je ponovo poslat. Naravno, primaocu je potreban samo jedan primerak ovog paketa i on odbacuje paket koji je ponovo poslat. U ovom slučaju, rad koji je ruter uložio u prosleđivanje ponovno poslate kopije prvobitnog paketa je uzaludan, jer je primalac

već dobio prvobitni primerak ovog paketa. Ruter bi bolje iskoristio prenosni kapacitet linka da je umesto toga poslao neki drugi paket. *Ovde vidimo još jednu posledicu zagušene mreže – nepotrebno ponovno slanje izazvano velikim kašnjenjem dovodi do toga da ruter troši propusni opseg linka za prosleđivanje nepotrebnih kopija paketa.* Slika 3.46(c) prikazuje propusnu moć u odnosu na ponuđeno opterećenje, ako se pretpostavi da ruter svaki paket prosleđuje (u proseku) dva puta. Pošto se svaki paket prosleđuje dva puta, ostvarena propusna moć imaće približnu vrednost $R/4$, kada ponuđeno opterećenje dostigne vrednost $R/2$.

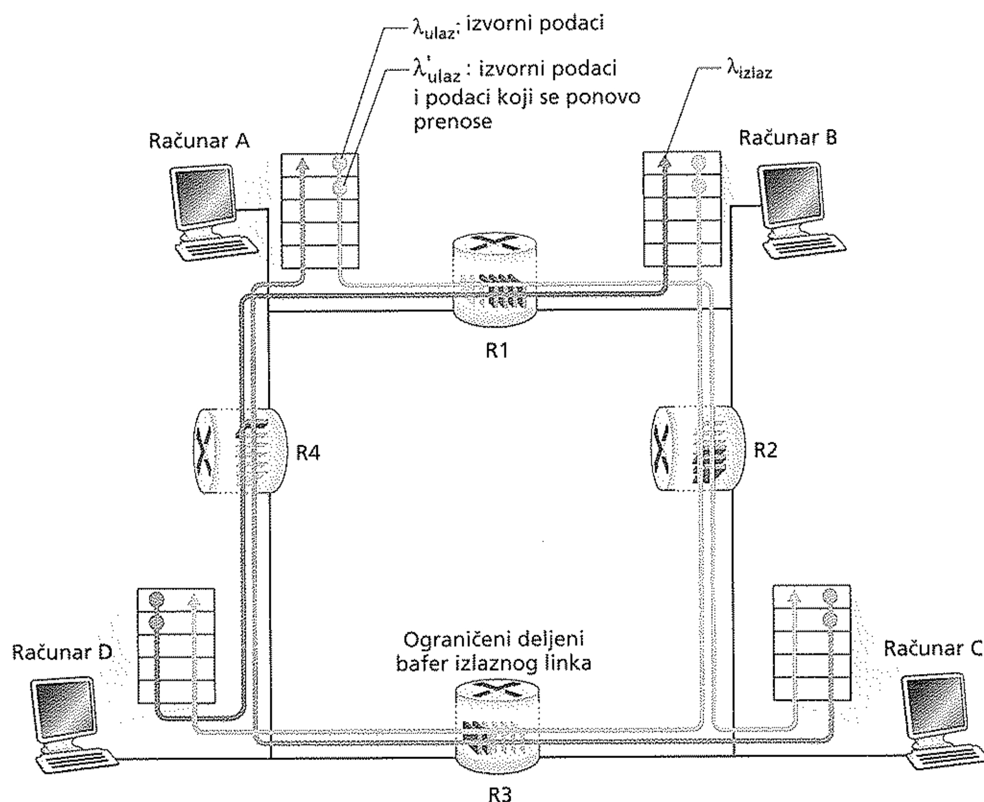


Slika 3.46 ◆ Performanse u slučaju 2 sa konačnim baferom

Slučaj 3: Četiri pošiljaoca, ruteri sa konačnim baferima i putanja sa više skokova

U poslednjem slučaju zagušenja, četiri računara šalju pakete putanjama koje se preklapaju i od kojih svaka ima po dva skoka, kao što je prikazano na slici 3.47. Ponovo pretpostavljamo da svi računari koriste mehanizam tajmera i ponovnog slanja, kojim ostvaruju uslugu pouzdanog prenosa podataka, da svi računari imaju istu vrednost λulaz, a da linkovi svih rutera imaju kapacitet od R bajtova/sekundi.

Razmotrimo vezu od računara A do računara C koja prolazi kroz rutere R1 i R2. Veza A–C deli ruter R1 sa vezom DB, a ruter R2 sa vezom BD. Za ekstremno male vrednosti λulaz, prepunjavanje bafera se retko događa (kao u slučajevima zagušenja 1 i 2), a propusna moć je približno jednaka ponuđenom opterećenju. Za nešto veće vrednosti λulaz, odgovarajuća propusna moć je takođe veća, zato što se više izvornih podataka prenosi u mrežu i isporučuje na odredište, a prepunjavanje bafera se još uvek retko događa. Prema tome, za manje vrednosti λulaz povećanje vrednosti λulaz dovodi do povećanja vrednosti λizlaz.



Slika 3.47 ◈ Četiri pošiljaoca, ruteri sa konačnim baferima i putanje sa više skokova

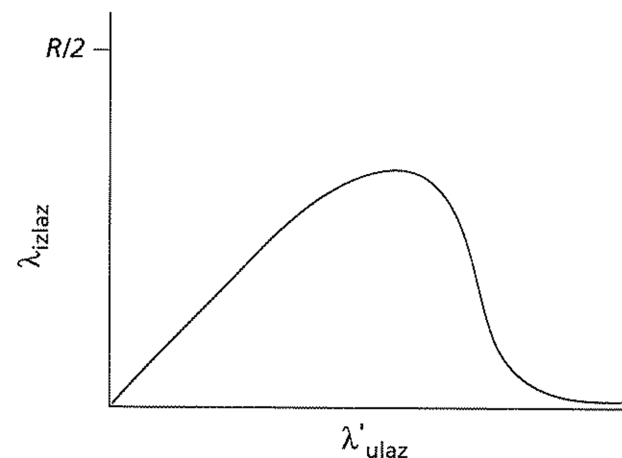
Pošto smo razmotrili slučaj izuzetno malog saobraćaja, pogledajmo sada slučaj kada je vrednost λulaz (pa tako i λ'ulaz) izuzetno velika. Pogledajmo ruter R2. Saobraćaj A–C koji stiže do rutera R2 (a do rutera R2 dolazi tako što je prosleđen sa rutera R1) može da pristiže brzinom od najviše R, koliko iznosi kapacitet linka od R1 do R2, bez obzira na vrednost λulaz. Ako je vrednost λ'ulaz izuzetno velika za sve veze (uključujući i vezu B–D), onda brzina pristizanja saobraćaja B–D na ruteru R2 može da bude mnogo veća od brzine saobraćaja A–C. Pošto saobraćaj A–C i saobraćaj B–D moraju da se nadmeću na ruteru R2 za ograničeni prostor bafera, količina saobraćaja A–C, koja uspešno prođe kroz R2 (odnosno, koja se ne izgubi zbog prepunjenog bafera) postaje sve manja i manja, kako ponuđeno opterećenje od B–D postaje sve veće i veće. U krajnjem slučaju, kada se ponuđeno opterećenje približi beskonačnosti, prazan bafer na ruteru R2 odmah se puni paketima B–D, a propusna moć veze A–C na ruteru R2 teži nuli. Ovo, sa jedne strane, *dovodi do toga da propusna moć A–C od kraja na kraj teži nuli* u slučaju krajnje velikog saobraćaja.

Ova razmatranja pokazuju značajnu zavisnost propusne moći u odnosu na ponuđeno opterećenje, prikazanu na slici 3.48.

Razlog za moguće opadanje propusne moći sa povećanjem ponuđenog opterećenja očigledan je kada se razmotri sav uzaludni rad koji se obavi u mreži. U slučaju sa velikim saobraćajem, koji smo upravo opisali, kad god se na ruteru drugog skoka ispusti paket, to znači da je rad koji je ruter prvog skoka uložio u prosleđivanje tog paketa drugom bio „uzaludan“. Mreža bi isto tako dobro prošla (tačnije, isto bi tako loše prošla) da je prvi ruter jednostavno odbacio paket i ostao besposlen. U stvari, prenosni kapacitet koji je prvi ruter upotrebio za prosleđivanje paketa drugom ruteru mogao bi da bude mnogo korisnije upotrebljen, da je prenet neki drugi paket. (Na primer, kada se bira paket za slanje, bilo bi bolje da ruter dâ prednost paketima koji su već prešli određen broj uzvodnih rutera.) *Ovde vidimo još jednu posledicu odbacivanja paketa zbog zagušenja – kada se paket odbaci negde na putanji, to znači da je prenosni kapacitet svih linkova, upotrebljenih za prosleđivanje paketâ, do trenutka odbacivanja paketa uzaludno utrošen.*

3.6.2 Rešenja koja se koriste za kontrolu zagušenja

U odeljku 3.7 podrobno ćemo razmotriti rešenja koja se koriste za kontrolu zagušenja kod protokola TCP. Ovde ukazujemo na dva pristupa za kontrolu zagušenja, koji se praktično sprovode i razmatramo određene arhitekture mreže i protokole za kontrolu zagušenja koji oličavaju te pristupe.



Slika 3.48 ◈ Performanse u trećem slučaju sa konačnim baferima i putanjama sa više skokova

U najširem smislu, rešenja koja se koriste za kontrolu zagušenja možemo da razlikujemo po tome da li mrežni sloj na bilo koji način neposredno pomaže transportnom sloju, u cilju kontrole zagušenja.

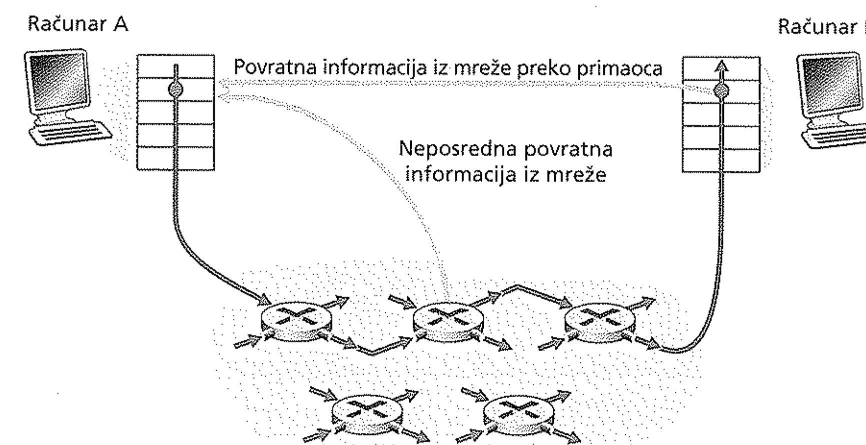
- *Kontrola zagušenja sa kraja na kraj.* U kontroli zagušenja sa kraja na kraj, mrežni sloj, što se tiče kontrole zagušenja, ni na koji način *ne pomaže transportnom* sloju. Čak i samo prisustvo zagušenja na mreži, krajnji sistemi mogu da utvrde jedino na osnovu ponašanja posmatrane mreže (na primer, gubici paketa i kašnjenje). U odeljku 3.7 videćemo da TCP mora da prihvati ovu kontrolu zagušenja sa kraja na kraj, jer sloj IP ne obezbeđuje krajnjim sistemima nikakvu povratnu informaciju o zagušenju mreže. Gubici TCP segmenta (na koje ukazuju istek vremena ili trostruke potvrde prijema) uzimaju se kao pokazatelj da je došlo do zagušenja mreže i TCP saglasno tome smanjuje veličinu svog prozora. Takođe ćemo videti novije predloge za kontrolu zagušenja protokola TCP u kojima se povećanje vrednosti kašnjenja povratnog puta koristi kao pokazatelj povećanog zagušenja mreže.
- *Kontrola zagušenja uz pomoć mreže.* Kod kontrole zagušenja uz pomoć mreže, komponente mreže (tj. ruteri) daju pošiljaocu jasne povratne informacije o stanju zagušenja u mreži. Ovu povratnu informaciju može da sačinjava samo jedan bit koji ukazuje na zagušenje linka. Takav pristup koji je bio prihvaćen u prvobitnim IBM SNA [Schwartz 1982] i DEC DECnet [Jain 1989; Ramakrishnan 1990] mrežama, nedavno je predložen za TCP/IP mreže [Floyd TCP 1994; RFC 3168], a koristi se i za ABR (raspoloživa bitska brzina, engl. available bit rate) kontrolu zagušenja u ATM mrežama, o čemu kasnije govorimo. Moguće su i složenije povratne informacije o mreži. Na primer, jedan oblik ABR kontrole zagušenja u ATM mrežama, koji ćemo uskoro proučiti, omogućava da ruter izričito obavesti pošiljaoca o brzini prenosa, koju on (taj ruter) može da podrži na izlaznom linku. Protokol XCP [Katabi 2002] svim pošiljaocima obezbeđuje povratnu informaciju koju ruter izračunava i koja se nalazi u zaglavlju paketa, a u kojoj se navodi da bi pošiljalac trebalo da poveća, ili da smanji svoju brzinu prenosa.

Kod kontrole zagušenja uz pomoć mreže, informacija o zagušenju obično se prosleđuje od mreže ka pošiljaocu, na jedan od dva načina prikazana na slici 3.49. Neposredna povratna informacija se sa mrežnog rutera šalje pošiljaocu. Ova vrsta obaveštenja obično je u obliku **paketa zagušenja** (koji u suštini kaže: „Zagušen sam!"). Drugi oblik obaveštavanja obavlja se tako što ruter označava ili ažurira odgovarajuće polje u paketu, koji putuje od pošiljaoca ka primaocu, kako bi ukazao na zagušenje. Kada primi tako označeni paket, primalac obaveštava pošiljaoca da je dobio upozorenje o zagušenju. Obratite pažnju na to da ovaj poslednji oblik obaveštavanja traje najmanje koliko i vreme punog povratnog puta.

3.6.3 Primer kontrole zagušenja uz pomoć mreže: ABR kontrola zagušenja ATM mreža

Ovaj odeljak zaključujemo kratkim opisom algoritma za kontrolu zagušenja u protokolu ABR u ATM mrežama – protokolu koji za kontrolu zagušenja koristi pomoć mreže. Naglašavamo da nam ovde nije cilj da podrobno opišemo arhitekturu ATM

mreža, već da prikažemo protokol koji koristi značajno drugačiji pristup za kontrolu zagušenja od onog koji se koristi u internetovom protokolu TCP. Umesto toga, prikazujemo samo nekoliko stavki ATM arhitekture, koje su neophodne za shvatanje ABR kontrole zagušenja.



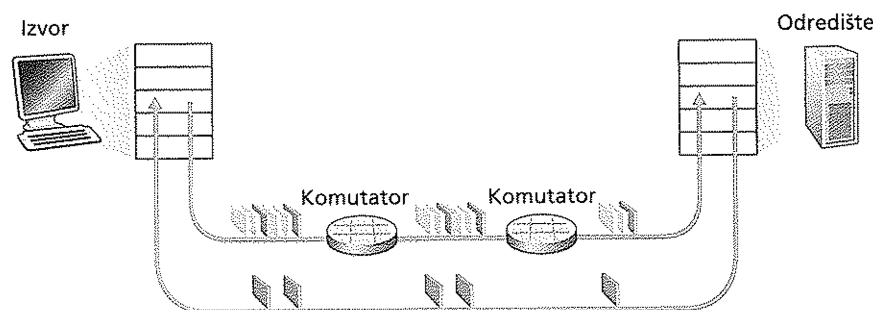
Slika 3.49 ◈ Dve putanje povratnih informacija o zagušenju koje daje mreža

U osnovi, ATM mreže za komutiranje paketa koriste rešenje sa virtuelnim kolima (VC). Sigurno se sećate iz poglavlja 1 da to znači da svaki komutator na putanji od izvora do odredišta vodi računa o stanju određenog virtuelnog kola od izvora do odredišta. Ovakvo vođenje računa o stanju svih virtuelnih kola omogućava komutatoru da prati ponašanje pojedinačnih pošiljalaca (npr. da prati njihovu prosečnu brzinu slanja) i da preduzima odgovarajuće mere za kontrolu zagušenja, u zavisnosti od izvora (npr. da jasno signalizira pošiljaocu da smanji brzinu, kada komutator postane zagušen). ATM mreža je, zbog takvog vođenja računa o stanju svih virtuelnih kola u mrežnim komutatorima, idealno prilagođena za kontrolu zagušenja uz pomoć mreže.

Protokol ABR je projektovan kao prilagodljiva usluga prenosa podataka, koja podseća na TCP. Kada je mreža manje opterećena, ABR usluga može slobodno da koristi dostupan višak propusnog opsega; kada je mreža zagušena, ABR usluga bi trebalo da smanji brzinu prenosa, na neku unapred postavljenu najmanju brzinu. Iscrpno uputstvo za ABR kontrolu zagušenja i upravljanje saobraćajem u ATM mrežama može se naći u dokumentu [Jain 1996].

Na slici 3.50 okvirno je prikazano okruženje ABR kontrole zagušenja u ATM mreži. U našem razmatranju koristimo terminologiju ATM mreža (na primer, koristimo izraz *komutator* umesto izraza *ruter* i izraz *ćelija* umesto izraza *paket*). U ABR usluzi ATM mreže, ćelije podataka se kroz niz usputnih komutatora prenose se od izvora do odredišta. Između ćelija podataka umetnute su **ćelije za upravljanje resursima, ćelije RM** (Resource Management cells); ove RM ćelije se mogu upo-

trebiti za razmenu informacija o zagušenju između računara i komutatora. Kada RM ćelija stigne na odredište, ona se okreće i vraća pošiljaocu (nakon što se na odredištu, po potrebi, promeni njen sadržaj). Komutator može i sâm da napravi RM ćeliju i neposredno je pošalje izvoru. Tako se RM ćelije koriste i za neposredne povratne informacije od mreže i za povratne informacije o mreži preko primaoca, kao što je prikazano na slici 3.50.



Slika 3.50 ◇ Okvirni prikaz kontrole zagušenja ABR usluge ATM mreže

ABR kontrola zagušenja u ATM mreži koristi pristup koji se zasniva na brzini. Drugim rečima, pošiljalac tačno izračunava najveću brzinu kojom može da šalje i tome se prilagođava. ABR predviđa tri mehanizma kojima komutatori upozoravaju primaoce o zagušenju u mreži:

- *EFCI bit. Svaka ćelija podataka* sadrži **bit za nedvosmisleno upozorenje o zagušenju unapred, bit EFCI** (eng. explicit forward congestion indication). Zagušeni mrežni komutator može da postavi EFCI bit u ćeliji podataka na 1 i tako upozori odredišni računar na zagušenje. Odredišni računar mora da proverava EFCI bit u svim primljenim ćelijama podataka. Kada RM ćelija stigne na odredište, a u prethodno primljenoj ćeliji podataka EFCI bit je bio postavljen na 1, tada odredišni računar postavlja bit za upozorenje o zagušenju, bit CI (eng. congestion indication) u RM ćeliji na 1 i RM ćeliju šalje nazad pošiljaocu. Pomoću EFCI bita u ćelijama podataka i CI bita u RM ćelijama pošiljalac može da se upozori na zagušenje u mrežnom komutatoru.
- *Bitovi CI i NI.* Kao što je već rečeno, između ćelija podataka od pošiljaoca ka primaocu su umetnute RM ćelije. Broj umetnutih RM ćelija može da se podešava parametrom, čija je podrazumevana vrednost jedna RM ćelija na svakih 32 ćelije podataka. Te RM ćelije sadrže **bit za upozorenje o zagušenju (CI bit)** i **bit „ne povećavaj", bit NI** (eng. no increase), koje postavlja komutator zagušene mreže. Tačnije, komutator može unutar RM ćelije, koja prolazi kroz njega, u slučaju blagog zagušenja da postavi bit NI na 1, a u slučaju velikog zagušenja da postavi bit CI na 1. Kada odredišni računar primi RM ćeliju, on je vraća pošiljaocu, ne dirajući njene bitove CI i NI (osim što odredišni računar može da postavi bit CI na 1, zbog gore opisanog mehanizma EFCI).
- *Postavljanje vrednosti ER.* Svaka RM ćelija takođe sadrži dvobajtno **polje eksplicitno zadate brzine, polje ER** (eng. explicit rate). Zagušeni komutator može da smanji vrednost u polju ER unutar RM ćelije, koja prolazi kroz njega. Na taj način, vrednost u polju ER biće postavljena na najmanju brzinu, koju mogu da podrže svi komutatori na putanji od izvora do odredišta.

ABR izvor ATM mreže prilagođava brzinu kojom šalje ćelije, u zavisnosti od vrednosti u poljima CI, NI i ER u vraćenoj RM ćeliji. Pravila za to podešavanje brzine su prilično složena i pomalo dosadna za objašnjavanje. Čitaoce koje ovo zanima, više mogu naći u knjizi [Jain 1996].

3.7 TCP kontrola zagušenja

U ovom odeljku vraćamo se proučavanju protokola TCP. Kao što smo naučili u odeljku 3.5, TCP obezbeđuje uslugu pouzdanog transporta između dva procesa, koji se izvršavaju na različitim računarima. Još jedna ključna stavka protokola TCP je njegov mehanizam kontrole zagušenja. Kao što je napomenuto u prethodnom odeljku, TCP mora da koristi kontrolu zagušenja sa kraja na kraj, a ne kontrolu zagušenja uz pomoć mreže, pošto sloj IP krajnjim sistemima ne nudi jasne povratne informacije o zagušenju na mreži.

Pristup koji prihvata TCP je da se svaki pošiljalac primora da ograniči brzinu kojom šalje saobraćaj u svoju vezu, u zavisnosti od uočenog zagušenja mreže. Ako TCP pošiljalac smatra da je na putanji između njega i odredišta zagušenje malo, on povećava brzinu slanja; ako uoči da postoji zagušenje, on je smanjuje. Ovaj pristup, međutim, otvara tri pitanja. Prvo: kako da TCP pošiljalac ograniči brzinu kojom šalje saobraćaj u svoju vezu? Drugo: kako da TCP pošiljalac prepozna da postoji zagušenje na putanji između njega i odredišta? I treće: koji bi algoritam pošiljalac trebalo da koristi za menjanje brzine slanja, u zavisnosti od uočenog zagušenja od jednog kraja do drugog?

Pogledajmo prvo kako TCP pošiljalac ograničava brzinu kojom šalje saobraćaj u svoju vezu. U odeljku 3.5 videli smo da se svaka strana TCP veze sastoji od prijemnog bafera, predajnog bafera i nekoliko promenljivih (`LastByteRead`, `RcvWindow` itd). TCP mehanizam za kontrolu zagušenja zahteva da se sa obe strane veze održava još jedna promenljiva, **prozor zagušenja**. Prozor zagušenja, označen sa `cwnd` (congestion window), nameće ograničenje brzine kojom TCP pošilja-

lac može da šalje saobraćaj u mrežu. Tačnije, količina nepotvrđenih podataka kod pošiljaoca ne sme da pređe manju od vrednosti cwnd i rwnd, to jest:

```
LastByteSent - LastByteAcked < min{cwnd, rwnd}
```

Da bismo se usredsredili na kontrolu zagušenja (za razliku od kontrole toka), ubuduće ćemo pretpostaviti da je prijemni TCP bafer toliko veliki da se ograničenje prijemnog prozora može zanemariti; pa se tako količina nepotvrđenih podataka kod pošiljaoca ograničava jedino pomoću vrednosti cwnd. Pretpostavićemo takođe da pošiljalac uvek ima podatke za slanje, tj. da su svi segmenti iz prozora zagušenja poslati.

Navedeno ograničenje određuje količinu nepotvrđenih podataka kod pošiljaoca i tako posredno ograničava njegovu brzinu slanja. Da biste to razumeli, razmotrite vezu sa zanemarljivim gubicima i kašnjenjem u prenosu paketa. U ovakvom slučaju, ovo ograničenje dozvoljava pošiljaocu da na početku svakog povratnog puta (RTT) kroz vezu pošalje približno cwnd bajtova podataka, a na kraju povratnog puta pošiljalac prima potvrde za te podatke. *Na taj način brzina slanja pošiljaoca iznosi približno cwnd/RTT bajtova u sekundi. Prema tome, pošiljalac može da podesi brzinu slanja podataka kroz vezu podešavanjem vrednosti cwnd.*

Razmotrimo sada kako TCP pošiljalac opaža da li postoji zagušenje na putanji između njega i odredišta. Definišimo „događaj gubitka" kod TCP pošiljaoca, bilo kao istek vremena tajmera, ili primanje trostruke ACK potvrde od primaoca. (Sećate se razmatranja događaja isteka vremena u odeljku 3.5.4 na slici 3.33 i kasnije dopune koja je obuhvatila brzo ponovno slanje po prijemu trostruke ACK potvrde.) Kada postoji izuzetno zagušenje, dolazi do prepunjavanja jednog ili više bafera u ruterima duž putanje, što dovodi do odbacivanja datagrama (koji obuhvataju TCP segmente). Odbačeni datagram predstavlja nastanak događaja gubitka kod pošiljaoca – bilo da dođe do isteka vremena, ili stignu trostruke ACK potvrde – što pošiljalac uzima kao znak da postoji zagušenje na putanji od njega do primaoca.

Pošto smo razmotrili kako se zagušenje otkriva, razmotrimo sada povoljniji slučaj, kada u mreži nema zagušenja, tj. kada nema događaja gubitaka. U tom slučaju, TCP pošiljalac prima potvrde za prethodno nepotvrđene segmente. Kao što ćemo videti, TCP uzima dolazak ovih potvrda kao znak da je sve u redu – da se segmenti poslati u mrežu uspešno isporučuju na odredište – pa će to iskoristiti da poveća svoj prozor zagušenja (a samim tim i brzinu slanja). Obratite pažnju na to da, ako potvrde stižu relativno sporo (npr. ako putanja sa kraja na kraj ima veliko kašnjenje, ili sadrži link malog propusnog opsega) onda se prozor zagušenja relativno sporo povećava. S druge strane, ako potvrde stižu brzo, prozor zagušenja se povećava mnogo brže. Pošto TCP koristi potvrde prijema, kako bi pokrenuo (ili dao ritam) postupak povećanja prozora zagušenja, za TCP protokol se kaže da ima **vlastiti ritam** (eng. self-clocking).

Imajući u vidu *mahanizam* koji podešava vrednost cwnd, kako bi se kontrolisala brzina slanja, ključno pitanje ostaje: *kako* će TCP pošiljalac utvrditi brzinu slanja? Ukoliko TCP pošiljaoci generalno šalju previše brzo, mogu da zaguše mrežu, što dovodi do kolapsa, usled zagušenja koje smo videli na slici 3.48. Zaista, verzija protokola TCP, kojom ćemo se uskoro baviti, razvijena je kao odgovor na posmatran kolaps usled zagušenja interneta [Jacobson 1988] u ranijim verzijama protokola TCP. Međutim, ako su TCP pošiljaoci previše obazrivi i šalju previše sporo, može se desiti da nedovoljno iskoriste propusni opseg u mreži; odnosno, TCP pošiljaoci mogu da šalju i većom brzinom bez zagušenja mreže. Kako onda TCP pošiljaoci mogu da utvrde svoje brzine slanja pri kojima neće zagušiti mrežu, a u isto vreme će iskoristiti sav raspoloživi propusni opseg? Da li su TCP pošiljaoci nedvosmisleno usklađeni, ili postoji distribuirani pristup u kome TCP pošiljaoci mogu da podese svoje brzine slanja samo na osnovu lokalnih informacija? Protokol TCP odgovara na ova pitanja, koristeći sledeće vodeće principe:

- *Izgubljeni segment ukazuje na zagušenje, pa bi iz tog razloga brzina pošiljaoca trebalo da se smanji, kada se segment izgubi.* Setite se naše rasprave iz odeljka 3.5.4, kada smo događaj isteka vremena ili prijem četiri potvrde za dati segment (jedna originalna ACK potvrda i tri kopije ACK potvrde) tumačili kao jedan implicitan pokazatelj „događaja gubitka" segmenta koji sledi četvorostruko potvrđeni segment. Ovo će prouzrokovati ponovno slanje izgubljenog segmenta. Sa stanovišta kontrole zagušenja, pitanje je kako bi TCP pošiljalac trebalo da smanji veličinu svog prozora zagušenja, i stoga svoju brzinu slanja, kao odgovor na ovaj izvedeni događaj gubitka.
- *Potvrđeni segment ukazuje da mreža isporučuje segmente pošiljaoca primaocu, pa brzina pošiljaoca može da se poveća, kada stigne ACK potvrda za nepotvrđeni segment.* Dolazak potvrda se uzima kao implicitni pokazatelj da je sve u redu – segmenti su uspešno isporučeni od pošiljaoca primaocu, a mreža zbog toga nije zagušena. Stoga, veličina prozora zagušenja može da se poveća.
- *Isprobavanje propusnog opsega.* S obzirom da ACK potvrde ukazuju da nema zagušenja na putanji od izvora do odredišta, a događaji gubitka ukazuju na zagušenu putanju, strategija protokola TCP za prilagođanje brzine je povećanje brzine, kao odgovor na pristigle ACK potvrde, sve dok se ne pojavi događaj gubitka, kada bi brzina prenosa trebalo da se smanji. TCP pošiljalac zato povećava svoju brzinu prenosa, da bi isprobao pri kojoj brzini počinje zagušenje, odustaje od te brzine i zatim ponovo počinje sa isprobavanjem, kako bi video da li se brzina koja vodi do zagušenja promenila. Ponašanje TCP pošiljaoca moglo bi se uporediti sa detetom koje stalno traži (i dobija) sve više i više poslastica, dok mu se konačno ne kaže: „Ne!", ono malo odstupi, ali nedugo zatim počinje ponovo da traži poslastice. Primetite da ne postoji eksplicitna naznaka stanja zagušenja od strane mreže – ACK potvrde i događaji gubitka služe kao impli-

citni signali – svaki TCP pošiljalac radi na osnovu lokalnih informacija, asinhrono u odnosu na ostale TCP pošiljaoce.

Uzimajući u obzir ovaj pregled TCP kontrole zagušenja, u poziciji smo da razmotrimo detalje proslavljenog **TCP algoritma za kontrolu zagušenja**, koji je prvi put opisan u [Jacobson 1988] a standardizovan u dokumentu [RFC 5681]. Algoritam ima tri glavne komponente: (1) spori start, (2) izbegavanje zagušenja i (3) brzi oporavak. Spori start i izbegavanje zagušenja su obavezne komponenete protokola TCP, koje se razlikuju po tome kako povećavaju veličinu prozora cwnd, odgovarajući na primljene ACK potvrde. Ubrzo ćemo videti da spori start povećava veličinu cwnd brže (uprkos imenu!) u odnosu na izbegavanje zagušenja. Brzi oporavak se preporučuje, ali ne zahteva od TCP pošiljaoca.

Spori start

Na početku TCP veze, vrednost cwnd obično se postavlja na malu vrednost od jedan MSS [RFC 3390], što daje početnu brzinu slanja od približno MSS/RTT. Na primer, ako je MSS = 500 bajtova, a RTT = 200 ms, imaćemo početnu brzinu slanja od približno svega 20 kb/s. Pošto propusni opseg, dostupan TCP pošiljaocu, može da bude mnogo veći od MSS/RTT TCP, pošiljalac bi želeo da brzo pronađe raspoloživi propusni opseg. Prema tome, u stanju **spori start** vrednost cwnd počinje od 1 MSS i povećava se za po 1 MSS, svaki put kada se preneseni segment prvi put potvrdi. U prmeru na slici 3.51, TCP šalje prvi segment u mrežu i čeka na potrdu prijema. Kada ova potvrda prijema stigne, TCP pošiljalac povećava prozor zagušenja za po jedan MSS i šalje 2 segmenta maksimalne veličine. Ovi segmenti se tada potvrđuju, a pošiljalac povećava prozor zagušenja za po 1 MSS za svaki potvrđeni segment, čime se dobija prozor zagušenja od 4 MSS i tako dalje. Ovaj proces rezultuje dupliranjem brzine prenosa za svakih RTT vremena. Prema tome, brzina slanja protokola TCP na početku me mala, ali raste eksponencijalno tokom faze sporog starta.

Ali, kada bi ovaj eksponencijalni rast trebalo da se završi? Spori start pruža nekoliko odgovora na ovo pitanje. Prvo, ukoliko postoji događaj gubitka (tj. zagušenje) na koje ukazuje istek vremena, TCP pošiljalac podešava vrednost cwnd na 1 i počinje iznova proces sporog starta. On takođe podešava drugu promenljivu stanja ssthresh (skraćenica za „prag algoritma sporog starta“, engl. slow start threshold) na cwnd/2 – polovina od vrednosti prozora zagušenja, kada se otkrije zagušenje. Drugi način na koji se spori start može okončati je direktno povezan sa vrednošću ssthresh. Pošto vrednost ssthresh predstavlja polovinu vrednosti cwnd, kada je zagušenje poslednji put otkriveno, možda će delovati nepažljivo dalje dupliranje vrednosti cwnd, kada dostigne ili prevaziđe vrednost ssthresh. Na taj način, kada se vrednost cwnd izjednači sa vrednošću ssthresh, spori start se završava, a TCP prelazi u režim rada izbegavanja zagušenja. Kao što ćemo videti, protokol TCP povećava vrednost cwnd mnogo opreznije, kada je u režimu rada izbegavanja zagušenja. Poslednji način na koji spori start može da se završi je ako se otkriju tri kopije ACK potvrda, i tada protokol TCP izvršava brzo ponovno

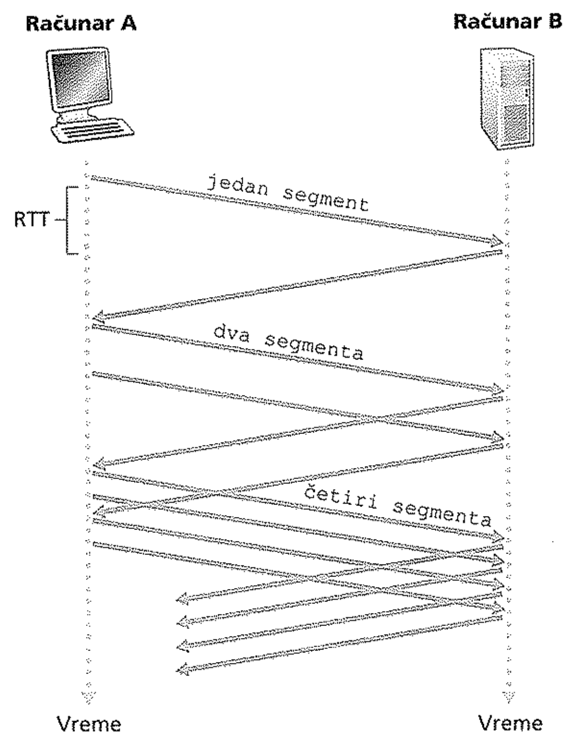
slanje (videti odeljak 3.5.4) i ulazi u stanje brzog oporavka, o čemu ćemo biti reči u nastavku. Ponašanje protokola TCP u sporom startu je sažeto prikazano u FSM opisu TCP kontrole zagušenja, na slici 3.52. Algoritam sporog starta ima korene u [Jacobson 1988]; jedan sličan pristup sporog starta takođe je predložen nezavisno u delu [Jain 1986].

DELJENJE PROTOKOLA TCP: OPTIMIZOVANJE PERFORMANSI U USLUGAMA OBLAKA

Poželjno je da usluge oblaka, kao što su: pretraživanje, e-pošta i društvene mreže, mogu brzo da odgovore, dajući korisnicima savršenu iluziju da se usluge izvršavaju unutar njihovih krajnjih sistema (uključujući i njihove pametne telefone). Ovo može biti veliki izazov, jer se korisnici često nalaze veoma daleko od centara podataka koji su odgovorni za opsluživanje dinamičkog sadržaja, povezanog sa uslugama oblaka. Zaista, ako je krajnji sistem udaljen od centra podataka, onda će RTT vreme biti veliko, što će potencijalno voditi do slabog vremena odziva preformansi, usled sporog starta protokola TCP.

Uzmite u obzir kao studiju slučaja kašnjenje u odgovoru na upit pretraživanja. Tipično, server zahteva tri TCP prozora tokom sporog starta, da bi isporučio odgovor [Pathak 2010]. Prema tome, vreme od trenutka kada krajnji sistem uspostavlja TCP vezu do trenutka kada primi poslednji paket odgovora je okvirno 4 · RTT (jedan RTT da uspostavi TCP vezu plus tri RTT za tri prozora sa podacima) plus vreme obrade u centru podataka. Ova RTT kašnjenja mogu da dovedu do značajnog kašnjenja prilikom davanja rezultata pretrage, za značajan deo upita. Osim toga, u pristupnim mrežama može doći do značajnog gubitka paketa, što dovodi do TCP ponovnog slanja i čak većih kašnjenja.

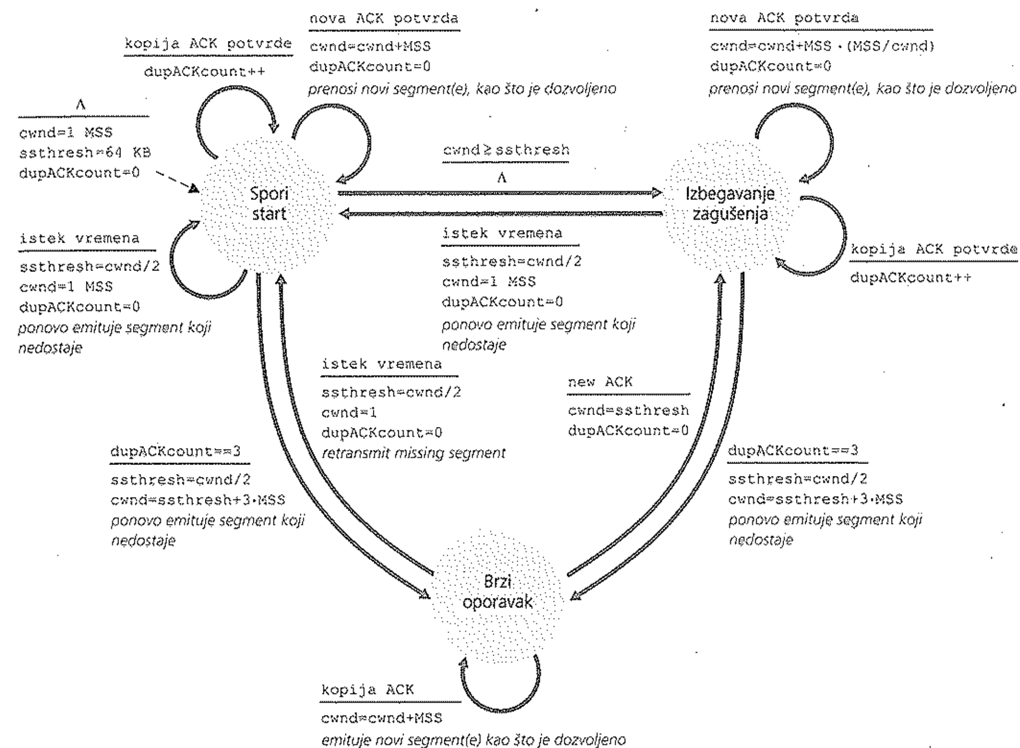
Jedan od načina za rešavanje ovog problema i poboljšanje preformansi za korisnika je: (1) raspoređivanje pristupnih servera bliže korisniku i (2) korišćenje **TCP deljenja** prekidanjem TCP veze na pristupnom serveru. Pomoću TCP deljenja klijent uspostavlja TCP vezu sa obližnjim pristupnim serverom, a pristupni server održava postojanu TCP vezu sa centrom podataka sa veoma velikim TCP prozorom zagušenja [Tariq 2008, Pathak 2010, Chen 2011]. Ovim pristupom vreme odziva postaje približno 4 RTT_{FE} + RTT_{BE} + vreme obrade, gde je RTT_{FE} vreme povratnog puta između klijenta i pristupnog servera, a RTT_{BE} vreme povratnog puta između pristupnog servera i centra podataka (pozadinski server). Ako je pristupni server blizu klijenta, onda je ovo vreme odziva približno RTT plus vreme obrade, pošto je RTT_{FE} zanemarljivo malo, a RTT_{BE} je približno jednako RTT. U suštini, TCP deljenje može da smanji kašnjenje mreže približno sa 4 RTT na RTT, što značajno poboljšava perfomanse korisnika, posebno za korisnike koji su udaljeni od najbližeg centra podataka. TCP deljenje takođe pomaže u smanjenju TCP kašnjenja usled ponovnog slanja, koje je prouzrokovano gubicima u pristupnim mrežama. Danas *Google* i *Akamai* intenzivno koriste svoje CDN servere u pristupnim mrežama (videti odeljak 7.2) za izvršavanje TCP deljenja za usluge oblaka koje podržavaju [Chen 2011].



Slika 3.51 ◆ Spori start protokola TCP

Izbegavanje zagušenja

Prilikom ulaza u stanje izbegavanja zagušenja vrednost `cwnd` je približno pola svoje vrednosti, u odnosu na trenutak kada se zagušenje poslednji put pojavilo – zagušenje bi trebalo da bude tu iza ugla! Prema tome, umesto da duplirate vrednost `cwnd` za svaki RTT, TCP usvaja malo konzervativniji prisup i povećava vrednost `cwnd` jednim MSS za svaki RTT [RFC 5681]. Ovo može da se postigne na nekoliko načina. Uobičajeni prisup je da TCP pošiljalac poveća vrednost `cwnd` za MSS bajtova (MSS/cwnd), svaki put kada pristigne nova potvrda prijema. Na primer, ako je MSS 1460 bajtova, a `cwnd` 14600 bajtova, tada će se 10 segmenata poslati unutar jednog RTT. Svaka potvrda prijema koja pristigne (pretpostavljamo jedna ACK potvrda po segmentu), povećava veličinu prozora zagušenja za 1/10 MSS, pa će vrednost prozora zagušenja biti povećana za jedan MSS posle ACK potvrda posle prijema svih 10 segmenata.



Slika 3.52 ◆ FSM opis TCP kontrole zagušenja

Međutim, kada bi trebalo da se završi linearno povećavanje izbegavanja zagušenja (od 1MSS po RTT)? Algoritam izbegavanja zagušenja protokola TCP se ponaša isto kada se pojavi istek vremena. Kao i u slučaju sporog starta: vrednost `cwnd` se podešava na 1 MSS, a vrednost `ssthresh` se ažurira na polovinu vrednosti `cwnd`, kada se pojavi događaj gubitka. Setite se, međutim, da događaj gubitka može takođe biti izazvan događajem trostrukih kopija ACK potvrda. U ovom slučaju, mreža nastavlja da isporučuje segmente od pošiljaoca do primaoca (kao što pokazuje prijem kopije ACK potvrde). Prema tome ponašanje protokola TCP za ovu vrstu događaja gubitka bi trebalo da bude manje drastično od događaja na koji ukazuje istek vremena: TCP deli na pola vrednost `cwnd` (dodavanjem 3 MSS kao dobru meru, da bi se uračunao prijem trostruke kopije ACK potvrde) i postavlja se vrednost `sstresh` da bude polovina vrednosti `cwnd`, kada se primi trostruka kopija ACK potvrde. Tada nastupa stanje brzog oporavka.

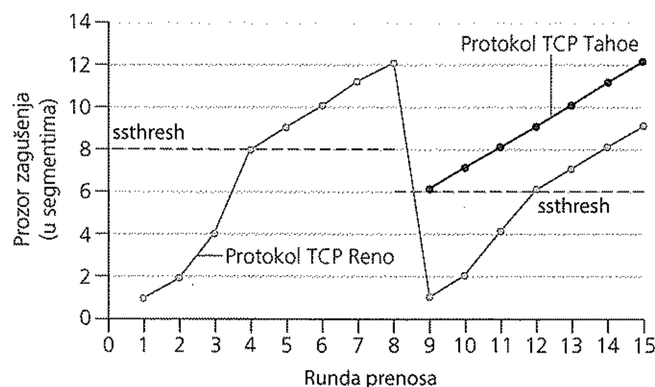
Brzi oporavak

U brzom oporavku vrednost cwnd se povećava za 1 MSS za svaku kopiju ACK potvrde koja je primljena za segment koji nedostaje, koji je prouzrokovao da protokol TCP uđe u stanje brzog oporavka. Na kraju, kad stigne ACK potvrda za segment koji nedostaje, protokol TCP ulazi u stanje izbegavanja zagušenja, nakon što se vrednost cwnd smanji. Ako se pojavi događaj isteka vremena, brzi oporavak prelazi u stanje sporog starta, nakon što izvrši istu radnju kao i u sporom startu i izbegavanju zagušenja: vrednost cwnd se podešava na 1 MSS, a vrednost ssthresh se podešava na polovinu vrednosti cwnd, kada se pojavi događaj gubitka.

Brzi oporavak je komponenta protokola TCP, koja se preporučuje, ali ne i zahteva, [RFC 5681]. Interesantno je da je ranija verzija protokola TCP, poznata kao protokol **TCP Tahoe**, bezuslovno seče svoj prozor zagušenja na 1 MSS i ulazila je u fazu sporog starta, nakon događaja gubitka na koji ukazuje istek vremena ili trostruka kopija ACK potvrde. Najnovija verzija protokola TCP, **TCP Reno** sadrži brzi oporavak.

Video napomena — Ispitivanje ponašanja protokola TCP

Slika 3.53 pokazuje razvoj prozora zagušenja protokola TCP za protokole Reno i Tahoe. Na ovoj slici, vrednost praga je na početku jednaka 8 MSS. Za prvih osam rundi prenosa, Tahoe i Reno preduzimaju identične akcije. Prozor zagušenja eksponencijalno raste tokom sporog starta i dostiže vrednost praga u četvrtoj rundi prenosa. Prozor zagušenja zatim linearno raste, dok se ne pojavi događaj trostruke kopije ACK potvrde, odmah posle runde 8. Primetite da je prozor zagušenja 12 · *MSS*, kada se pojavi ovaj događaj gubitka. Vrednost ssthresh se tada podešava na 0,5·cwnd = 6·MSS. U protokolu TCP Reno prozor zagušenja je podešen na cwnd = 6·MSS i zatim linearno raste. U protokolu Tahoe prozor zagušenja je podešen na 1 MSS i raste eksponencijalno, dok ne dostigne vrednost ssthresh, odakle raste linearno.

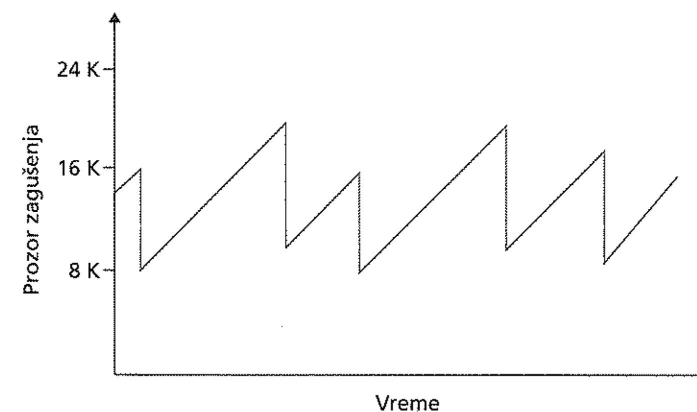


Slika 3.53 ◈ Razvijanje prozora zagušenja protokola TCP (Tahoe i Reno)

Slika 3.52 prikazuje kompletan FSM opis algoritma kontrole zagušenja protokola TCP – spori start, izbegavanje zagušenja i brzi oporavak. Slika takođe pokazuje gde može da se pojavi prenos novih segmenata ili ponovni prenos segmenata. Iako je važno napraviti razliku između TCP kontrole greške/ponovnog prenosa i TCP kontrole zagušenja, takođe je važno da procenite kako su ova dva aspeka protokola neraskidivo povezani.

Kontrola zagušenja protokola TCP: osvrt

S obzirom da smo detaljno objasnili spori start, izbegavanje zagušenja i brzi oporavak, trebalo bi da se povučemo, kako bi sagledali celokupnu sliku. Ako ignorišemo početni spori start kada se veza uspostavi i ako pretpostavimo da na gubitke ukazuju trostruke kopije ACK potvrde, a ne istek vremena, kontrola zagušenja protokola TCP se sastoji iz linearnog (aditivno) povećanja vrednosti cwnd od 1 MSS po RTT, a zatim se vrednost cwnd deli na pola (multiplikativno smanjenje) u događaju trostruke kopije ACK potvrde. Iz ovog razloga, TCP kontrola zagušenja se često naziva **aditivno povećanje/multiplikativno smanjenje (AIMD)**, što predstavlja oblik kontrole zagušenja. AIMD kontrola zagušenja izaziva „testerasto" ponašanje, prikazano na slici 3.54, koje na lep način prikazuje naše ranije opažanje „probanja vrednosti" propusnog opsega protokola TCP.– Protokol TCP linearno povećava veličinu svog prozora zagušenja (i stoga svoju brzinu prenosa), sve dok se ne pojavi događaj trostruke kopije ACK potvrde. On tada samnjuje veličinu svog prozora zagušenja za faktor dva, ali onda ponovo počinje linerano da povećava vrednost, isprobavajući da li postoji dodatni raspoloživi propusni opseg.



Slika 3.54 ◈ Kontrola zagušenja aditivno povećanje/multiplikativno smanjenje

Kao što smo ranije napomenuli, mnoge TCP implementacije koriste algoritam Reno [Padhye 2001]. Predložene su mnoge varijante algoritma Reno [RFC 3782;

RFC 2018]. Algoritam Vegas protokola TCP [Brakmo 1995; Ahn 1995] pokušava da izbegne zagušenje prilikom održavanja dobre propusne moći. Osnovna ideja algoritma Vegas je da: (1) otkrije zagušenje u ruterima između izvora i odredišta, *pre* nego što se pojavi gubitak paketa i (2) snizi brzinu linearno, kada se otkrije predstojeći gubitak paketa. Predstojeći gubitak paketa može da se predvidi posmatranjem RTT vremena. Duže RTT vreme paketa znači veće zagušenje na ruterima. *Linux* podržava brojne algoritme za kontrolu zagušenja (uključujući TCP Reno i TCP Vegas) i dozvoljava administratoru sistema da konfiguriše koja će se verzija protokola TCP koristiti. Podrazumevana verzija protokola TCP u verziji *Linuxa* 2.6.18 je podešena na CUBIC [Ha 2008], verziju protokola TCP koja je razvijena za aplikacije visokog propusnog opsega. Za nedavna ispitivanja mnogih vrsta TCP protokola videti [Afanasyev 2010].

AIMD algoritam protokola TCP je razvijen na osnovu ogromnih inžinjerskih istraživanja i eksperimentisanja sa kontrolom zagušenja u operacionoj mreži. Deset godina nakon razvoja protokola TCP, teorijske analize su prikazale da algoritam kontrole zagušenja protokola TCP služi kao distribuirani asinhrono optimizovan algoritam koji deluje simultano, optimizujući nekoliko važnih aspekata korisnika i performansi mreže [Kelly 1998]. Od tada je razvijena bogata teorija kontrole zagušenja [Srikant 2004].

Makroskopski opis propusne moći protokola TCP

Imajući u vidu testerasto ponašanje protokola TCP, prirodno je da razmotrimo kolika može biti prosečna propusna moć (odnosno prosečna brzina) dugovečne TCP veze. U ovoj analizi ignorisaćemo fazu sporog starta, koja se javlja posle događaja isteka vremena. (Ove faze su obično veoma kratke, jer je pošiljalac prerastao fazu eksponencijalno brzo). Tokom određenog intervala povratnog puta, brzina pri kojoj protokol TCP šalje podatke je funkcija prozora zagušenja i trenutnog *RTT*. Kada je veličina prozora *w* bajtova, a trenutno vreme povratnog puta *RTT* sekundi, tada je brzina prenosa protokola TCP otprilike *w/RTT*. Protokol TCP tada isprobava dodatni propusni opseg, povećavanjem *w* za 1 MSS za svaki *RTT*, sve dok se ne pojavi događaj gubitka. Označimo sa *W* vrednost *w*, kada se pojavi događaj gubitka. Ako pretpostavimo da su *RTT* i *W* približno konstantni tokom trajanja veze, onda se brzina prenosa protokola TCP kreće od $W/(2 \cdot RTT)$ do W/RTT.

Ova pretpostavka dovodi do veoma pojednostavljenog makroskopskog modela ponašanja za stabilno stanje protokola TCP. Mreža ispušta paket iz veze kada se brzina poveća na *W/RTT*; brzina se tada deli na pola i zatim povećava za *MSS/RTT* za svaki *RTT* sve dok ne dostigne *W/RTT*. Ovaj proces se ponavlja iznova i iznova. S obzirom da se propusna moć protokola TCP (odnosno brzina) linearno povećava između dve ekstremne vrednosti, imamo

$$\text{prosečnu propusnu moć veze} = \frac{0{,}75 \cdot W}{RTT}$$

Koristeći ovaj visoko idealizovani model za dinamiku stabilnog stanja protokola TCP, možemo takođe da izvedemo interesantan zaključak koji povezuje verovatnoće gubitka veze sa njenim raspoloživim propusnim opsegom [Madhavi 1997]. Ovo izvođenje je opisano u problemima iz domaćeg zadatka. Mnogo sofisticiraniji model, zasnovan na praksi, koji se slaže sa izmerenim podacima je [Padhye 2000].

Protokol TCP preko putanja velikog propusnog opsega

Važno shvatiti da se TCP kontrola zagušenja razvijala tokom godina i da je i dalje nastavila da se razvija. Kratak pregled trenutnih varijanti TCP protokola i diskusiju o TCP razvoju videti [Floyd 2001, RFC 5681, Afanasyev 2010]. Ono što je bilo dobro za internet, kada je većina TCP veza nosila SMTP, FTP i Telnet saobraćaj, ne mora obavezno da bude dobro za današnji internet kojim dominira HTTP, ili za budući internet sa uslugama o kojima još uvek možemo da maštamo.

Potrba za kontinuiranim razvojem TCP protokola može da se prikaže uzimanjem u obzir TCP veza velike brzine koje su potrebne za mrežne i aplikacije računarstva u oblaku. Na primer, uzmimo TCP vezu sa segmentima od 1 500 bajtova i *RTT* vremenom od 100 ms, i pretpostavimo da brzinom 10 Gbps želimo da pošaljemo podtke preko ove veze. Prateći dokument [RFC 3649], primećujemo da bi korišćenjem gornje formule za propusnu moć protokola TCP, u cilju dobijanja propusne moći od 10 Gbps, prosečna veličina prozora zagušenja trebalo da bude 83 333 segmenta. To je *dosta* segmenata, pa se možemo zabrinuti da će se jedan od tih 83 333 segmenata u prozoru zagubiti? Ili, da kažemo na drugi način, koji deo prenešenih segmenata može da se izgubi, a da se dozvoli da algoritam TCP kontrole zagušenja, naveden na slici 3.52, dostigne željenu brzinu od 10 Gbps? U pitanjima iz domaćeg zadatka za ovo poglavlje vodimo vas kroz izvođenje formule koja se odnosi na propusnu moć TCP veze kao funkcije verovatniće gubitka (*L*), vremena povratnog puta (*RTT*) i maksimalne veličine segmenta (MSS):

$$\text{prosečna propusna moć veze} = \frac{1{,}22 \cdot MSS}{RTT\sqrt{L}}$$

Koristeći ovu formulu, da bismo postigli propusnu moć od 10 Gbps, algoritam kontrole zagušenja današnjeg protokola TCP može samo da toleriše verovatnoću gubitka segmenta od $2 \cdot 10^{10}$ (ili to je jednako jednom događaju gubitka za svakih 5 000 000 000 segmenata) – veoma mala verovatnoća. Ovo opažanje je navelo brojne istraživače da ispitaju nove verzije protokola TCP, koje su posebno dizajnirane za ovo okruženje visoke brzine; videti diskusije o ovim pokušajima [Jin 2004; RFC 3649; Kelly 2003; Ha 2008].

3.7.1 Fer ponašanje

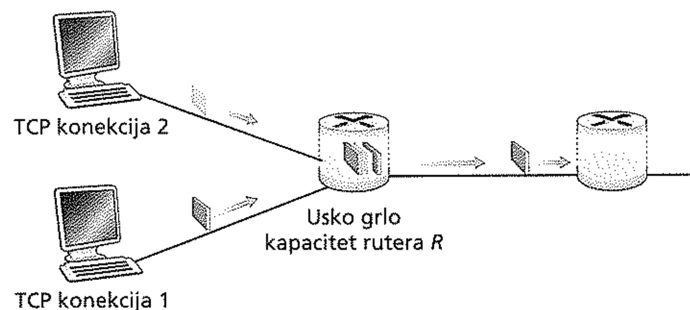
Uzmite u obzir *K* TCP veza, svaka sa drugačijom putanjom s jednog na drugi kraj, ali sve prosleđuju kroz link uskog grla, brzinom prenosa od *R* bps. (Pod *linkom*

uskog grla podrazumevamo da za svaku vezu, svi ostali linkovi duž putanje veze nisu zagušeni i da imaju obilan kapacitet prenosa u poređenju sa kapacitetom prenosa linka uskog grla). Pretpostavimo da svaka veza prenosi veliku datoteku i da se kroz link uskog grla ne prenosi saobraćaj protokola UDP. Mehanizam kontrole zagušenja kaže da bi bilo *pravedno*, kada bi prosečna brzina prenosa svake veze bila približno *R/K*; odnosno, svaka veza dobija jednaki udeo u propusnom opsegu linka.

Da li je algoritam AIMD protokola TCP pravedan, posebno s obzirom na različite TCP veze koje mogu da se uspostave u različita vremena i stoga mogu da imaju različite veličine prozora u datom trenutku vremena? [Chiu 1989] pruža elegantno i intuitivno objašnjenje zašto TCP kontrola zagušenja teži da pruži jednak udeo propusnog opsega linka uskog grla svakoj od konkurentnih TCP veza.

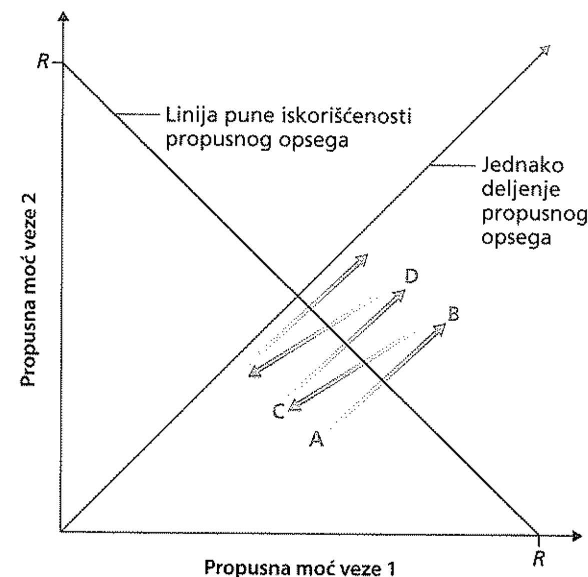
Razmotrimo sada jednostavan slučaj dve TCP veze koje dele link brzine prenosa *R*, kao što je prikazano na slici 3.55. Pretpostavimo da dve veze imaju isti MSS i RTT (tako da, ako imaju istu veličinu prozora zagušenja, onda one imaju istu propusnu moć) dalje, da imaju istu količinu podataka za slanje i da nijedna druga TCP veza ili UDP datagrami ne prolaze kroz ovaj deljeni link. Takođe, ignorišite fazu sporog starta protokola TCP i pretpostavite da TCP veze funkcionišu u režimu rada CA (AIMD) u svakom trenutku.

Slika 3.56 prikazuje propusnu moć koja je realizovana pomoću dve TCP veze. Ako bi protokol TCP trebalo podjednako da deli link propusnog opsega ovim dvema vezama, onda bi realizovana propusna moć trebalo da padne do strelice od 45 stepeni (jednako deljenje propusnog opsega) koja polazi iz koordinantnog početka. Idealno, suma dve propusne moći bi trebalo da bude jednaka *R* (da sve veze koje primaju jednak, ali nulti udeo kapaciteta linka, nije poželjna situacija!). Stoga, trebalo bi da se postigne da propusna moć opadne negde između preseka linije deljenja jednakog propusnog opsega i linije iskorišćenja punog propusnog opsega na slici 3.56.



Slika 3.55 ◈ Dve TCP veze koje dele link uskog grla

Pretpostavimo da su veličine TCP prozora takve da u datom vremenskom trenutku veze 1 i 2 realizuju propusne moći na koje ukazuje tačka A na slici 3.56. S obzirom da je količina propusnog opsega linka koje zajedno koriste obe veze manja od *R*, neće se pojaviti gubitak, a obe veze će povećati svoj prozor za 1 MSS po RTT-u, kao rezultat algoritma izbegavanja zagušenja protokola TCP. Stoga, zajednička propusna moć dve veze se prostire duž linije od 45 stepeni (jednako povećanju za ove veze), počevši od tačke A. Eventualno, propusni opseg, koji zajedno koriste dve veze će biti veći od *R*, a možda će se pojaviti gubitak paketa. Pretpostavimo da se kod veza 1 i 2 desio gubitak paketa kada su realizovale propusne moći, na šta ukazuje tačka *B*. Veze 1 i 2 će tada smanjiti svoje prozore za faktor dva. Dobijene propusne moći su tako u tački *C*, na pola puta duž vektora koji počinje u tački *B* i završava se u koordinantnom početku. Pošto je korišćenje zajedničkog propusnog opsega manje od *R* u tački *C*, dve veze ponovo povećavaju svoje propusne moći duž linije od 45 stepeni od tačke *C*. Najzad, gubitak će se ponovo pojaviti, na primer, u tački *D*, i dve veze ponovo smanju veličinu svog prozora za faktor dva, i tako dalje. Trebalo bi da shvatite da propusni opseg, koji su dve veze ostvarile, može da flukturia duž linije jednakog deljenja propusnog opsega. Takođe, trebalo bi da se uverite da će dve veze konvergirati ka ovom ponašanju, bez obzira na mesto u dvodimenzionalnom prostoru! Iako je u ovom scenariju veliki broj idealizovanih pretpostavki, on i dalje pruža intuitivan osećaj, zbog čega protokol TCP rezultuje u jednakom deljenju propusnog opsega između veza.



Slika 3.56 ◈ Propusna moć realizovana pomoću TCP veza 1 i 2

U našem idealizovanom scenariju pretpostavili smo da samo TCP veze prolaze kroz link uskog grla, da veze imaju istu RTT vrednost, i da se samo pojedinačna TCP veza povezuje sa parom računar-odredište. U stvarnosti se ovi uslovi obično ne susreću, a klijentsko-serverske aplikacije mogu stoga da dobiju veoma nejednake

delove propusnog opsega linka. Prikazano je da, kada višestruke veze dele zajedničko usko grlo, sesije sa manjim RTT mogu da uhvate raspoloživi propusni opseg na linku, mnogo brže nego što se on oslobodi (odnosno, otvaraju svoje prozore zagušenja brže) i stoga imaju veću propusnu moć od onih veza sa dužim vremenima povratnog puta [Lakshman 1997].

Fer ponašanje i UDP

Upravo smo videli kako kontrola zagušenja protokola TCP uređuje brzinu prenosa putem mehanizma sa prozorom zagušenja. Mnoge multimedijalne aplikacije, kao što je internet telefoniranje ili video konferencije, upravo se iz tog razloga ne izvršavaju preko protokola TCP – one ne žele da se njihova brzina prenosa guši, čak i kada je mreža veoma zagušena. Umesto toga, ove aplikacije više vole da se izvršavaju preko protokola UDP, koji nema ugrađenu kontrolu zagušenja. Kada se izvršavaju preko protokola UDP, aplikacije ubacuju audio i video zapise stalnom brzinom u mrežu i povremeno gube pakete, umesto da smanjuju brzinu na „fer" nivo u trenucima zagušenja i da ne gube pakete. Sa stanovišta protokola TCP, multimedijalne aplikacije koje se izvršavaju preko protokola UDP nisu fer – ne sarađuju sa drugim vezama, niti podešavaju svoju brzinu prenosa na odgovarajući način. Pošto kontrola zagušenja protokola TCP smanjuje brzinu prenosa, kada se suoči sa povećanim zagušenjem (gubljenjem podataka), a UDP to ne radi, moguće je da UDP saobraćaj nadvlada TCP saobraćaj. Danas, veliko područje istraživanja predstavlja razvoj mehanizama za kontrolu zagušenja na internetu, kojim bi se sprečilo da UDP saobraćaj potpuno blokira propusnu moć interneta [Floyd 1999; Floyd 2000; Kohler 2006].

Fer ponašanje i paralelne TCP veze

Čak i kada bismo mogli da primoramo UDP saobraćaj na fer ponašanje, problem fer ponašanja još ne bi bio potpuno rešen, zato što ništa ne sprečava TCP aplikacije da koriste više paralelnih veza. Na primer, veb pretraživači često koriste više paralelnih TCP veza za prenos više objekata sa iste veb stranice. (U većini veb pretraživača moguće je podesiti tačan broj višestrukih veza.) Kada jedna aplikacija koristi više paralelnih veza, ona na zagušenom linku dobija veći deo propusnog opsega. Na primer, uzmimo link brzine R, koji podržava devet pokrenutih klijentsko-serverskih aplikacija, tako da svaka aplikacija koristi jednu TCP vezu. Ako se priključi jedna nova aplikacija i takođe koristi jednu TCP vezu, svaka aplikacija dobija približno istu brzinu prenosa od $R/10$. Ali, ako ova nova aplikacija koristi 11 paralelnih TCP veza, njoj će biti dodeljeno više od $R/2$, što nije fer alokacija. S obzirom da veb saobraćaj preovladava na internetu, višestruke paralelne veze nisu neuobičajene.

3.8 Rezime

Ovo poglavlje počeli smo razmatranjem usluga koje protokoli transportnog sloja mogu da obezbede mrežnim aplikacijama. Jedna krajnost je to što protokol transportnog sloja može biti veoma jednostavan i nuditi aplikacijama samo osnovne

usluge, tj. multipleksiranje/demultipleksiranje za procese koji razmenjuju podatke. Protokol UDP na internetu primer je takvog krajnje pojednostavljenog protokola transportnog sloja. Druga krajnost je to što protokol transportnog sloja aplikacijama nudi razne garancije, kao što su pouzdana isporuka podataka, garancije u vezi sa kašnjenjem i propusnim opsegom. I pored toga, usluge koje transportni protokol može da ponudi često su ograničene modelom usluga protokola mrežnog sloja koji se nalazi ispod njega. Ako protokol mrežnog sloja ne može segmentima transportnog sloja da obezbedi garancije vezane za kašnjenje ili propusni opseg, onda ni protokol transportnog sloja ne može da obezbedi isporuku u skladu sa garancijama vezanim za kašnjenje ili propusni opseg za poruke koje razmenjuju procesi.

U odeljku 3.4 naučili smo da protokol transportnog sloja može da obezbedi pouzdan prenos podataka, čak i kada mrežni sloj ispod njega nije pouzdan. Videli smo da obezbeđivanje pouzdanog prenosa podataka znači da se mora voditi računa o mnogim pojedinostima, koje se ne vide na prvi pogled, ali da se može obaviti pažljivim kombinovanjem potvrda prijema, tajmera, ponovnih slanja i rednih brojeva.

Iako smo u ovom poglavlju obrađivali pouzdan prenos podataka, trebalo bi imati na umu da pouzdan prenos podataka može da obezbede i protokoli sloja veze, kao i protokoli mrežnog, transportnog ili aplikativnog sloja. U bilo kom od četiri navedena sloja u skupu protokola moguće je primeniti potvrde prijema, tajmere, ponovna slanja i redne brojeve i tako obezbediti pouzdan prenos podataka za sloj iznad. U stvari, tokom godina naučnici i inženjeri računarstva su nezavisno projektovali i primenjivali protokole sloja veze, mrežnog, transportnog i aplikativnog sloja, koji obezbeđuju pouzdan prenos podataka (mada su mnogi od tih protokola neprimetno nestali).

U odeljku 3.5 detaljnije smo razmotrili protokol TCP, pouzdan protokol transportnog sloja interneta sa uspostavljanjem veze. Naučili smo da je TCP složen protokol koji obuhvata upravljanje vezom, kontrolu toka i procenu vremena povratnog puta, kao i pouzdan prenos podataka. TCP je u suštini mnogo složeniji od našeg opisa – namerno nismo opisivali niz dopuna, prepravki i poboljšanja protokola TCP, koji se naširoko primenjuju u različitim verzijama protokola TCP. Sva ta složenost, međutim, nevidljiva je za mrežnu aplikaciju. Ako klijent želi da sa jednog računara pouzdano pošalje podatke serveru na drugom računaru, on jednostavno otvara TCP soket prema serveru i ubacuje podatke u taj soket. Ta klijentsko-serverska aplikacija živi u blaženom neznanju o složenosti protokola TCP.

U odeljku 3.6 ispitali smo kontrolu zagušenja u opštem slučaju, a u odeljku 3.7 smo pokazali kako se u protokolu TCP implementira kontrola zagušenja. Naučili smo da je kontrola zagušenja obavezna za dobrobit mreže. Kad ne bi bilo kontrole zagušenja, na mreži bi lako nastao zastoj saobraćaja, tako da bi malo ili nimalo podataka moglo da se prenese sa kraja na kraj. U odeljku 3.7 naučili smo da TCP primenjuje mehanizam kontrole zagušenja s kraja na kraj, koji aditivno povećava brzinu prenosa, kada proceni da je putanja za određenu TCP vezu prohodna, a multiplikativno smanjuje brzinu prenosa kada dođe do gubitaka. Taj mehanizam takođe teži da svakoj TCP vezi, koja prolazi kroz zagušeni link dodeli podjednak deo njegovog propusnog opsega. Takođe smo u izvesnoj meri proučili uticaj uspostavljanja

TCP veze i sporog starta na ukupno vreme čekanja. Zapazili smo da uspostavljanje veze i spori start u mnogim slučajevima značajno doprinose kašnjenju s kraja na kraj. Ponovo naglašavamo da, iako je kontrola zagušenja protokola TCP tokom vremena značajno napredovala, i dalje ostaje područje intenzivnog istraživanja i verovatno će se i dalje razvijati tokom sledećih godina.

Naše razmatranje protokola interneta u ovom poglavlju posebno smo usmerili na protokole UDP i TCP – dve „tegleće mazge" transportnog sloja interneta. Međutim, dve decenije iskustva sa ovim protokolima pokazalo je da postoje mnoge okolnosti u kojima nijedan od njih nije savršen. Zbog toga, istraživači već dugo rade na razvoju novih protokola transportnog sloja, od kojih je nekoliko njih predloženo za primenu od strane udruženja IETF.

Protokol DCCP (Datagram Congestion Control Protocol – protokol za kontrolu zagušenja datagrama) [RFC 4340] nudi nepouzdanu uslugu, nalik protokolu UDP, sa manjom količinom dodatnih podataka, zasnovan na porukama, ali sa kontrolom zagušenja na nivou aplikacija koji je usaglašen sa protokolom TCP. Ukoliko je nekoj aplikaciji neophodna pouzdana ili delimično pouzdana usluga prenosa podataka, onda se ona sprovodi unutar same aplikacije, mehanizmima sličnim onima koje smo proučavali u odeljku 3.4. Protokol DCCP je prvenstveno zamišljen da se koristi u aplikacijama koje se koriste za protok multimedijalnog sadržaja (pogledajte poglavlje 7) u kojima je moguće uravnotežiti pravovremenu i pouzdanu isporuku podataka, ali uz vođenje računa o zagušenju mreže.

Protokol SCTP (Stream Control Transmission Protocol – protokol za kontrolisani prenos protoka podataka) [RFC 2960, RFC 3286] je pouzdan protokol koji se zasniva na porukama i koji omogućava da nekoliko različitih „tokova" na nivou aplikacija bude multipleksirano kroz samo jednu SCTP vezu (rešenje poznato pod nazivom „višestruki protok podataka"). Sa stanovišta pouzdanosti, različitim tokovima unutar mreže upravlja se odvojeno, tako da gubitak paketa u jednom toku ne utiče na isporuku podataka u drugom toku. SCTP takođe omogućava da se podaci prenose preko dve izlazne putanje, kada je računar povezan sa dve ili više mreža, opcionu isporuku bez redosleda i brojne druge mogućnosti. Algoritmi za kontrolu toka i kontrolu zagušenja protokola SCTP su u suštini istovetni onima iz protokola TCP.

Protokol TFRC (TCP-Friendly Rate Control – protokol TCP sa lakom kontrolom brzine) [RFC 2448] je prvenstveno protokol za kontrolu zagušenja, a ne baš pravi protokol transportnog sloja. U njemu su utvrđeni mehanizmi za kontrolu zagušenja, koji se mogu koristiti u drugim transportnim protokolima, kao što je DCCP (i zaista, jedan od dva protokola koji aplikacija može da izabere u protokolu DCCP je protokol TFRC). Cilj protokola TFRC je da ublaži ponašanje koje liči na „zupce testere" (pogledajte sliku 3.54) u kontroli zagušenja protokola TCP, pri čemu, dugoročno posmatrano, zadržava brzinu prenosa koja je „razumno" bliska onoj iz protokola TCP. Ravnomernijom brzinom prenosa u odnosu na TCP, protokol TFRC je bolje prilagođen za multimedijalne aplikacije, kao što su one za internet telefoniranje i protok multimedijalnih zapisa, kojima je važna takva ravnomernija brzina prenosa. TFRC je protokol čiji se rad zasniva na „proračunu", koji koristi merenje verovatnoće gubitaka paketa kao ulaznu vrednost u jednačinu [Padhye 2000], kojom se procenjuje kolika bi propusna moć protokola TCP bila ukoliko bi došlo do takvog gubitka paketa. Ova vrednost se uzima kao brzina slanja koju protokol TFRC teži da postigne.

Samo će budućnost pokazati da li će se protokoli DCCP, SCTP ili TFRC naširoko razviti. Mada ovi protokoli nude očigledna poboljšanja u odnosu na protokole TCP i UDP, protokoli TCP i UDP godinama unazad dokazali su se kao „sasvim dobri". Da li će „bolje" pobediti „sasvim dobro" zavisiće od složene mešavine tehničkih, društvenih i ekonomskih uticaja.

U poglavlju 1 smo rekli da se računarska mreža može podeliti na „obod mreže" i na „jezgro mreže". Obod mreže obuhvata sve što se događa na krajnjim sistemima. Pošto smo obradili aplikativni sloj i transportni sloj, naš opis oboda mreže je završen. Vreme je za istraživanje jezgra mreže! To putovanje počinje u sledećem poglavlju, u kome proučavamo mrežni sloj, a nastavlja se u poglavlju 5 u kome proučavamo sloj veze.

Domaći zadatak: problemi i pitanja

Poglavlje 3 Kontrolna pitanja

ODELJCI 3.1–3.3

R1. Posmatrajmo mrežni sloj koji nudi sledeću uslugu. Mrežni sloj na izvornom računaru preuzima segment najveće veličine 1 200 bajtova i adresu odredišnog računara od transportnog sloja. Mrežni sloj garantuje isporuku tog segmenta do transportnog sloja na odredišnom računaru. Pretpostavimo da se na odredišnom računaru izvršava veći broj procesa mrežnih aplikacija.

a. Napravite najjednostavniji mogući protokol transportnog sloja koji će podatke aplikacije preneti do željenog procesa na odredišnom računaru. Pretpostavka je da je operativni sistem na odredišnom računaru dodelio 4-bajtne brojeve portova svim procesima aplikacija koje se izvršavaju.

b. Promenite svoj protokol, tako da odredišnom procesu obezbeđuje „povratnu adresu".

c. Da li u Vašem protokolu transportni sloj „mora da radi nešto" u jezgru računarske mreže?

R2. Imamo planetu gde svi žive u porodicama od šest članova, svaka porodica živi u sopstvenoj kući, svaka kuća ima jedinstvenu adresu i svaka osoba u određenoj kući ima jedinstveno ime. Pretpostavimo da ta planeta ima poštansku službu koja prenosi pisma između kuća odakle se šalju do kuća na koja su upućena. Poštanska služba zahteva da: (1) pismo bude u koverti i da (2) na koverti bude ispisana samo adresa kuće kojoj je pismo upućeno (i ništa više). Pretpostavimo da sve porodice imaju predstavnika porodice koji skuplja i deli pisma ostalim članovima porodice. Na pismu nema oznake kom primaocu je određeno pismo tačno upućeno.

a. Koristeći rešenje problema R1 gore kao inspiraciju, opišite protokol koji bi ti predstavnici mogli da koriste za isporuku pisama koje članovi, pošiljaoci jedne porodice, šalju članovima, primaocima druge porodice pisama.

b. Da li u Vašem protokolu poštanska služba mora da otvara koverte i pregleda sadržaj pisama kako bi obavila svoju uslugu?

R3. Uzmimo TCP vezu između računara A i računara B. Pretpostavimo da TCP segmenti koji putuju od računara A prema računaru B imaju broj izvornog porta x, a broj odredišnog porta y. Koji su brojevi izvornog i odredišnog porta za segmente koji putuju od računara B prema računaru A?

R4. Opišite zašto bi se programer odlučio da se aplikacija izvršava preko protokola UDP, a ne preko protokola TCP.

R5. Zbog čega se za prenos glasa i video zapisa u savremenom internetu sve više koristi protokol TCP umesto protokola UDP. (*Savet:* odgovor koji tražimo nema nikakve veze sa mehanizmom kontrole zagušenja protokola TCP.)

R6. Da li je moguće da aplikacija koristi prednosti pouzdanog prenosa podataka, iako se izvršava protokolom UDP? Ako jeste, na koji način?

R7. Pretpostavimo da neki proces na računaru C ima UDP soket sa brojem porta 6789. Pretpostavimo da računari A i B pošalju po jedan UDP segment računaru C sa brojem odredišnog porta 6789. Da li će oba ta segmenta biti upućena na isti soket na računaru C? Ako je tako, kako će proces na računaru C znati koji od tih segmenata potiče od kog računara?

R8. Pretpostavimo da se neki veb server izvršava na računaru C na portu 80. Pretpostavimo da taj veb server koristi trajne veze i da trenutno prima zahteve od dva različita računara, A i B. Da li se svi zahtevi upućuju kroz isti soket na računaru C? Ukoliko bi se prenosili kroz različite sokete, da li bi oba ta soketa imala broj porta 80? Objasnite i obrazložite svoj odgovor.

ODELJAK 3.4

R9. Zašto je bilo neophodno da u protokol `rdt` uvedemo redne brojeve?

R10. Zašto je bilo neophodno da u protokol `rdt` uvedemo tajmere?

R11. Pretpostavimo da je kašnjenje povratnog puta između pošiljaoca i primaoca stalno i da pošiljalac zna koliko iznosi. Da li nam je i dalje potreban tajmer u protokolu `rdt 3.0`, pod pretpostavkom da može doći do gubitka paketa? Obrazložite svoj odgovor.

R12. Pogledajte *Java* aplet Go-Back-N na veb stranici ove knjige.

a. Postavite da izvorište pošalje pet paketa, a zatim zaustavite animaciju, pre nego što bilo koji od tih paketa stigne do odredišta. Tada poništite prvi paket i nastavite prikazivanje animacije. Opišite šta se događa.

b. Ponovite ovo, ali sada sačekajte da prvi paket stigne do odredišta i poništite prvu potvrdu prijema. Ponovo opišite šta se događa.

c. Na kraju probajte sa slanjem šest paketa. Šta se dešava?

R13. Ponovite zadatak R12, ali ovog puta sa *Java* apletom Selective Repeat. Po čemu se ova dva protokola razlikuju?

ODELJAK 3.5

R14. Tačno ili netačno?

a. Računar A šalje veliku datoteku računaru B preko TCP veze. Pretpostavimo da računar B nema podataka koje bi slao računaru A. Računar B neće računaru A slati potvrde prijema, jer ne može da ih šlepuje uz podatke.

b. Veličina TCP prijemnog prozora `rwnd`se senikada ne menja tokom trajanja veze.

c. Pretpostavimo da računar A šalje veliku datoteku računaru B preko TCP veze. Broj nepotvrđenih bajtova koje A šalje ne može da premaši veličinu prijemnog bafera.

d. Pretpostavimo da računar A šalje veliku datoteku računaru B preko TCP veze. Ako je redni broj nekog segmenta u ovoj vezi m, onda je redni broj sledećeg segmenta obavezno $m + 1$.

e. TCP segment ima u svom zaglavlju polje za *rwnd*.

f. Pretpostavimo da je poslednja vrednost `SampleRTT` u nekoj TCP vezi jednaka 1 sekund. Trenutna vrednost `TimeoutInterval` za tu vezu svakako je ≥ 1 sekund.

g. Pretpostavimo da računar A šalje jedan segment sa rednim brojem 38 i 4 bajta podataka, preko TCP veze računaru B. Broj potvrde prijema koji se nalazi u tom segmentu mora da bude 42.

R15. Pretpostavimo da računar A šalje dva TCP segmenta, jedan za drugim, računaru B preko TCP veze. Prvi segment ima redni broj 90; drugi ima redni broj 110.

a. Koliko podataka sadrži prvi segment?

b. Pretpostavimo da se prvi segment izgubi, ali da drugi segment stigne do B. Koji je broj potvrde koju računar B šalje računaru A?

R16. Uzmimo primer sa *Telnet*-om, opisan u odeljku 3.5. Nekoliko sekundi nakon slova „C" korisnik upiše slovo „R". Koliko se segmenata šalje posle upisivanja slova „R" i šta se stavlja u polja rednog broja i broja potvrde prijema ovih segmenata?

ODELJAK 3.7

R17. Pretpostavimo da na nekom linku, koji je usko grlo brzine R b/s, postoje dve TCP veze. Svaka od njih šalje veliku datoteku (u istom smeru, preko linka koji je usko grlo). Prenos datoteka počinje istovremeno. Koju brzinu prenosa bi TCP pokušao da dodeli svakoj od ovih veza?

R18. Tačno ili netačno? Posmatramo kontrolu zagušenja protokola TCP. Kada kod pošiljaoca istekne tajmer, vrednost `ssthresh` postavlja se na jednu polovinu njegove prethodne vrednosti.

R19. U objašnjenju TCP deljenja u bočnom stupcu odeljka 7.2, tvrdi se da je vreme odziva TCP koji se deli približno 4 RTT_{FE} + RTT_{BE} + vreme obrade. Obrazložite ovu tvrdnju.

Problemi

P1. Pretpostavimo da klijent A pokreće *Telnet* sesiju sa serverom S. Približno u isto vreme klijent B takođe pokreće *Telnet* sesiju sa serverom S. Obezbedite moguće brojeve izvornih i odredišnih portova za:

a. segmente koji se šalju iz A prema S;
b. segmente koji se šalju od B prema S;
c. segmente koji se šalju od S prema A;
d. segmente koji se šalju od S prema B;
e. Ako su A i B različiti računari, da li je moguće da broj izvornog porta u segmentima iz A prema S bude isti kao u segmentima iz B prema S?
f. Kako je u slučaju da su na istom računaru?

P2. Pogledajte sliku 3.5. Koje su vrednosti brojeva izvornog i odredišnog porta u segmentima, koji teku od servera prema klijentskim procesima? Koje su IP adrese u datagramima mrežnog sloja, koji prenose segmente transportnog sloja?

P3. UDP i TCP za svoje kontrolne zbirove koriste prvi komplement. Pretpostavimo da imate sledeća tri 8-bitna bajta: 01010011, 01100110, 01110100. Koji je prvi komplement zbira ovih 8-bitnih bajtova? (Obratite pažnju na to da, iako UDP i TCP koriste 16-bitne reči kada izračunavaju kontrolni zbir, za ovaj problem se zahteva da radite sa sabircima od 8 bitova.) Prikažite šta ste radili. Zašto UDP koristi prvi komplement za zbir; tj. zašto jednostavno ne koristi sâm zbir? Kako primalac korišćenjem prvog komplementa otkriva greške? Da li je moguće da greška u jednom bitu ostane neprimećena? A greška u dva bita?

P4. a. Pretpostavimo da imate sledeća dva bajta: 01011100 i 01100101. Koji je prvi komplement zbira ova dva bajta?

b. Pretpostavimo da imate sledeća dva bajta: 11011010 i 01100101. Koji je prvi komplement zbira ova dva bajta?

c. Za bajtove u delu (a) navedite primer u kome se jedan bit promeni u svakom od ova dva bajtova, a da se prvi komplement ne promeni.

P5. Pretpostavimo da UDP primalac izračunava internet kontrolni zbir za primljeni UDP segment i da pronađe da se poklapa sa vrednošću koja se nalazi u polju kontrolnog zbira. Može li primalac da bude potpuno siguran da nije došlo do greške u bitovima? Objasnite.

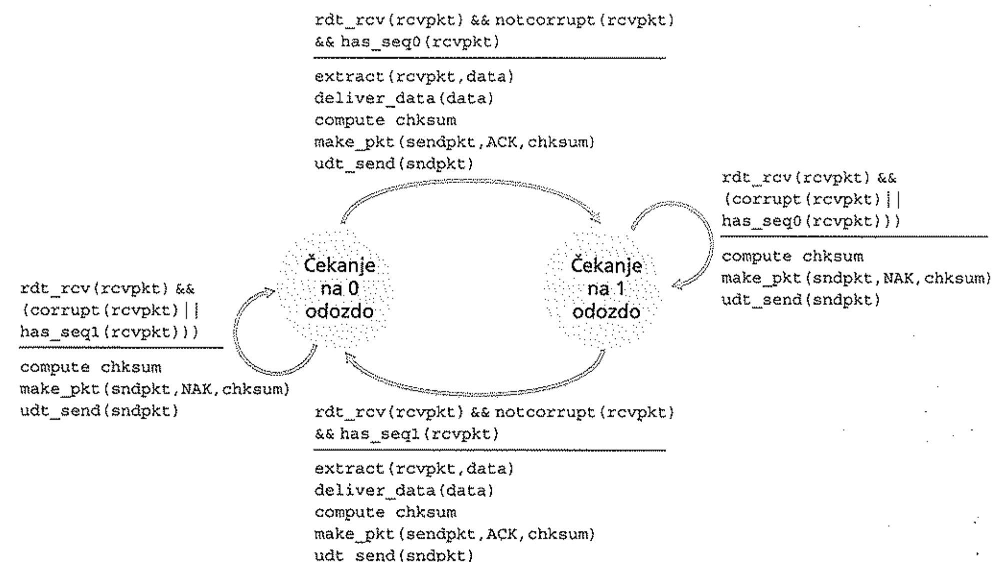
P6. Pogledajte razloge za izmenu protokola `rtd2.1`. Pokažite da primalac, prikazan na slici 3.57, kada radi sa pošiljaocem, prikazanim na slici 3.11, može dovesti pošiljaoca i primaoca u stanje blokade, tako da svako od njih čeka na događaj koji nikada neće nastupiti.

P7. U protokolu `rdt3.0`, ACK paketi koji idu od primaoca ka pošiljaocu nemaju redne brojeve (mada imaju ACK polje koje sadrži redni broj paketa čiji prijem potvrđuju). Zbog čega u ACK paketima nisu potrebni redni brojevi?

P8. Skicirajte FSM za prijemnu stranu protokola `rdt3.0`.

P9. Opišite rad protokola `rdt3.0` kada su paketi podataka i paketi potvrda prijema oštećeni. Opis bi trebalo da bude sličan onome koji se koristi na slici 3.16.

P10. Posmatramo kanal u kome može doći do gubitka paketa, ali je poznato najveće kašnjenje. Dopunite protokol `rdt2.1`, tako da pošiljalac koristi istek vremena tajmera i ponovno slanje. Svojim rečima objasnite zbog čega Vaš protokol može pravilno da komunicira preko ovog kanala.



Slika 3.57 ◈ Netačan primalac za protokol rdt 2.1

P11. Uzmite u obzir protokol `rdt2.2` na slici 3.14 i kreiranje novog paketa u povratnom prelazu (tj. prelaz iz stanja nazad u početno stanje) u stanjima Čekaj-na-0-odozdo i Čekaj-na-1-odozdo: `sndpkt=make_pkt(ACK, 0, checksum)` i `sndpkt=make_pkt(ACK, 0, checksum)`. Da li će protokol raditi pravilno, ako se ova radnja ukloni iz povratnog prelaza u stanje Čekaj-na-1-odozdo? Obrazložite svoj odgovor. Šta će se desiti, ako se ovaj događaj ukloni iz povratnog prelaza u stanje Čekaj-na-0-odozdo? [*Savet*: u ovom poslednjem slučaju, uzmite u obzir šta bi se desilo, ako bi prvi paket od pošiljaoca do primaoca bio oštećen.]

P12. Strana pošiljaoca u protokolu `rdt3.0` jednostavno zanemaruje (odnosno, ništa ne preduzima) one primljene pakete koji su pogrešni, ili ako je u paketu potvrde pogrešna vrednost u polju `za broj potvrde` prijema. Pretpostavimo da u takvom slučaju `rdt3.0` ponavlja slanje trenutnog paketa podataka. Da li bi protokol i dalje radio? (*Savet*: ispitajte šta bi se dogodilo u slučaju da postoje samo greške u bitovima; da nema gubitaka paketa, ali da može doći do prevremenog isticanja vremena tajmera. Razmotrite koliko puta bi se slao *n*-ti paket u graničnom slučaju kada *n* teži beskonačnosti.)

P13. Posmatramo protokol rdt 3.0. Nacrtajte dijagram koji prikazuje da, ukoliko mrežna veza između pošiljaoca i primaoca može da promeni redosled poruka (odnosno, da dve poruku koje se prostiru u medijumu između pošiljaoca i primaoca mogu da promene redosled), naizmenični bitski protokol neće pravilno raditi (proverite da li ste na pravi način identifikovali smisao u kojem on neće pravilno raditi). Vaš dijagram bi trebalo da sadrži pošiljaoca sa leve strane i primaoca sa desne, pri čemu će osa vremena ići ka dnu strane, dok dijagram prikazuje podatke (D) i potvrde prijema (A) razmenjenih poruka. Proverite da li pokazujete na redni broj koji je povezan sa bilo kojim podatkom ili segmentom potvrde.

P14. Posmatramo protokol za pouzdan prenos podataka koji koristi samo negativne potvrde. Pretpostavite da pošiljalac retko šalje podatke. Da li je protokol samo sa negativnim potvrdama bolji od protokola koji koristi samo pozitivne potvrde? Zašto? Sada pretpostavite da pošiljalac ima mnogo podataka za slanje, a da u vezi sa kraja na kraj retko dolazi do gubitaka. Da li bi u ovom drugom slučaju bolji bio protokol samo sa negativnim potvrdama od protokola koji koristi pozitivne potvrde? Zašto?

P15. Pogledajte primer sa velikom razdaljinom, prikazan na slici 3.17. Kolika bi morala da bude veličina prozora, da bi iskorišćenost kanala bila veća od 98%? Pretpostavite da je veličina paketa 1 500 bajtova, uključujući oba polja zaglavlja i podatke.

P16. Pretpostavimo da aplikacija koristi protokol rdt 3.0, kao svoj protokol transportnog sloja. Pošto protokol stani i čekaj ima vaoma malu iskorišćenost kanala (prikazan u primeru sa velikom razdaljinom), programeri ove aplikacije su dozvolili primaocu da nastavi da vraća određeni broj (više od dva) naizmeničnih ACK 0 i ACK 1 potvrda, čak i kada odgovarajući podaci ne stignu do primaoca. Da li će programeri ove aplikacije povećati iskorišćenost kanala? Zašto? Da li ima nekih potencijalnih problema u vezi sa ovim pristupom? Objasnite.

P17. Posmatrajte dva mrežna entiteta A i B, koji su povezani perfektnim dvosmernim kanalom (naime, svaka poruka biće ispravno primljena; kanal neće oštetiti, izgubiti ili promeniti redosled paketa). A i B bi trebalo da naizmenično isporuče poruke sa podacima jedan drugome: prvo, A mora da isporuči poruku entitetu B, zatim entitet B bi trebalo da isporuči poruku entitetu A, a A će isporučiti poruku entitetu B i tako dalje. Ako se neki entitet nalazi u stanju kada ne bi trebalo da pokušava da isporuči poruku drugoj strani, a postoji događaj kao što je `rdt_send(data)` poziv odozgo, koji pokušava da prosledi podatke nadole, da bi se preneli do druge strane, ovaj poziv odogzgo može jednostavno da bude ignorisan pozivom `rdt_unable_to_send(data)`, koji informiše gornji sloj da trenutno ne može da šalje podatke. [Napomena: ova pojednostavljena pretpostavka je učinjena kako ne biste morali da brinete o smeštanju podataka u bafer.]

Nacrtajte FSM specifikaciju za ovaj protokol (jedan FSM za A, i jedan FSM za B!). Primetite da ovde ne morate da brinete o mehanizmu pouzdanosti; ključna tačka ovog pitanja je da se kreira FSM specifikacija koja odražava sinhronizovano ponašanje dva entiteta. Trebalo bi da koristite sledeće događaje i radnje koje imaju isto značenje kao kod protokola rdt 1.0 na slici 3.9: `rdt_send(data), packet = make_pkt(data), udt_send(packet), rdt_rcv(packet), extract (packet,data), deliver_data(data)`. Proverite da li Vaš protokol odražava tačnu izmenu slanja između entiteta A i B. Takođe, proverite da li se ukazuje na stanja za entitete A i B u FSM opisima.

P18. U opštem protokolu SR, koji smo proučavali u odeljku 3.4.4, pošiljalac prenosi poruku čim bude dostupna (ukoliko je u prozoru), bez čekanja na potvrdu prijema. Sada pretpostavimo da želimo da protokol SR istovremeno šalje dve poruke. Odnosno, pošiljalac šalje par poruka, a zatim par sledećih poruka šalje tek kada sazna da su obe poruke iz prvog para ispravno primljene.

Pretpostavimo da poruke mogu da se izgube u kanalu, ali da se ne oštećuju i da im se redosled ne menja. Napravite protokol sa kontrolom grešaka za jednosmeran pouzdan prenos poruka. Napravite FSM opis pošiljaoca i primaoca. Opišite format paketa koji se šalju od pošiljaoca ka primaocu i obratno. Ako koristite drugačije pozive procedura od onih iz odeljka 3.4 (na primer, `udt_send(), start_timer(), rdt_rcv()` itd), jasno opišite šta tačno rade. Prikažite jedan primer (vremenski prikaz događaja pošiljaoca i primaoca) u kome će se videti na koji način se Vaš protokol oporavlja od gubitka paketa.

P19. Uzmimo slučaj u kome računar A želi istovremeno da šalje poruke računarima B i C. A je povezan sa B i C kanalom za difuzno emitovanje – paket poslat iz računara A prenosi se tim kanalom istovremeno računarima B i C.

Pretpostavite da kanal za difuzno emitovanje koji povezuje A, B i C može nezavisno da gubi i oštećuje poruke (na primer, da poruka poslata iz A bude pravilno primljena u B, ali ne i u C). Napravite protokol sa kontrolom grešaka po principu stani i čekaj za pouzdan prenos paketa od A do B i C, takav da A neće prihvatati nove podatke od gornjeg sloja, sve dok ne utvrdi da su i B i C pravilno primili trenutni paket. Napravite FSM opis računara A i C. (*Savet*: FSM opis računara B bi u suštinitrebalo da bude isti kao za računar C.) Osim toga, opišite format(e) paketa koji se koriste.

P20. Uzmimo slučaj u kome računar A i računar B žele da pošalju poruke računaru C. Računari A i C su povezani kanalom u kome poruke mogu da se izgube, ili mogu da se oštete (ali da im se redosled ne menja). Računari B i C su povezani drugim kanalom (nezavisno od kanala kojim su povezani A i C) istih osobina. Transportni sloj na računaru C može da naizmenično isporučuje poruke od računara A i B sloju iznad (odnosno, može prvo da isporuči podatke paketa iz računara A, a zatim podatke iz B i tako redom). Napravite protokol sa kontrolom grešaka po principu stani i čekaj za pouzdan prenos paketa od računara A i B do računara C, sa naizmeničnom isporukom u računaru C, kao što je prethodno opisano. Napravite FSM opis računara A i C. (*Savet*: FSM opis računara B bi u suštini trebalo da bude isti kao i za računar A.) Osim toga, opišite format(e) paketa koji se koriste.

P21. Pretpostavimo da imamo dva mrežna entiteta, A i B. B bi trebalo da isporuči poruke sa podacima koje šalje prema A u skladu sa sledećim pravilima. Kada A primi zahtev od gornjeg sloja da pribavi sledeću poruku sa podacima (D) od B, A mora da pošalje poruku zahteva (R) prema B, kroz kanal od A do B. B može da pošalje poruku sa podacima (D) prema A kroz kanal od B do A, tek kada primi poruku R. A bi trebalo da isporuči tačno jedan primerak svake poruke D gornjem sloju. Poruke R mogu da se izgube (ali ne i da se oštete) u kanalu od A do B; jednom poslate poruke D, uvek se pravilno isporučuju. Kašnjenje u oba kanala je nepoznato i promenljivo.

Napravite protokol (dajte njegov FSM opis) koji obuhvata odgovarajuće mehanizme kojima će se nadoknaditi mogući gubici paketa u kanalu od A do B i koji ostvaruje predavanje poruka gornjem sloju u entitetu A, kao što je gore opisano. Upotrebite samo mehanizme koji su apsolutno neophodni.

P22. Uzmimo GBN protokol sa veličinom prozora pošiljaoca od 4 i rasponom rednih brojeva 1024. Pretpostavimo da u trenutku t sledeći paket po redu koji primalac očekuje ima redni broj k. Pretpostavite da medijum ne menja redosled porukâ. Odgovorite na sledeća pitanja:

a. Koji su mogući skupovi rednih brojeva u prozoru pošiljaoca u trenutku t? Obrazložite odgovor.

b. Koje su sve moguće vrednosti polja ACK u svim mogućim porukama koje se u trenutku t vraćaju pošiljaocu? Obrazložite svoj odgovor.

P23. Posmatramo protokole GBN i SR. Pretpostavimo da je prostor rednih brojeva veličine k. Odredite za svaki od ovih protokola najveći dozvoljeni prozor pošiljaoca kojim će izbeći pojavljivanje problema kakvi se javljaju na slici 3.27.

P24. Odgovorite na sledeća pitanja sa tačno ili netačno, i ukratko obrazložite odgovor:

a. Da li je u protokolu SR moguće da pošiljalac primi ACK potvrdu za paket van njegovog trenutnog prozora?

b. Da li je u protokolu GBN moguće da pošiljalac primi ACK potvrdu za paket van njegovog trenutnog prozora?

c. Protokol naizmeničnih bitova je isto što i protokol SR sa veličinama prozora pošiljaoca i primaoca jednakom 1.

d. Protokol naizmeničnih bitova je isto što i protokol GBN sa veličinama prozora pošiljaoca i primaoca jednakom 1.

P25. Rekli smo da aplikacija može da izabere UDP za transportni protokol zato što UDP nudi bolju kontrolu aplikacije (u odnosu na TCP) toga koji podaci se šalju u segmentima i kada se šalju.

a. Zašto aplikacija ima veću kontrolu toga koji podaci se šalju u segmentima?

b. Zašto aplikacija ima veću kontrolu toga kada se šalju segmenti?

P26. Uzmimo u obzir prenošenje ogromne datoteke od L bajtova od računara A do računara B. Pretpostavimo da imamo MSS od 536 bajtova.

a. Kolika maksimalna vrednost L može da bude da TCP redni brojevi ne budu iscrpljeni? Setite se da polje rednog broja protokola TCP ima 4 bajta.

b. Za L koji dobijamo u (a), pronađite koliko je potrebno vremena da se prenese datoteka. Pretpostavimo da je svakom segmentu dodato ukupno 66 bajtova za zaglavlja transportnog, mrežnog i sloja veze podataka pre slanja dobijenog paketa preko linka od 155 Mb/s. Ignorišite kontrolu toka i kontrolu zagušenja tako da A može da šalje segmente jadan za drugim i kontinuirano.

P27. Računari A i B komuniciraju preko TCP veze, a računar B je već primio od računara A sve bajtove do bajta 126. Pretpostavimo da računar A posle toga šalje dva segmenta jedan za drugim računaru B. Prvi segment sadrži 80 bajtova, a drugi segment sadrži 40 bajtova. U prvom segmentu, redni broj je 127, broj izvorišnog porta je 302, a broj odredišnog porta je 80. Računar B uvek kada primi segment od računara A šalje potvrdu prijema.

a. Koji je redni broj, broj izvorišnog porta i broj odredišnog porta u drugom segmentu, koji se šalje od računara A do računara B?

b. Ukoliko prvi segment stigne pre drugog segmenta, koji je broj potvrde prijema, broj izvorišnog porta i broj odredišnog porta u potvrdi prijema prve poruke?

c. Ukoliko drugi segment stigne pre prvog segmenta, koji je broj potvrde prijema u potvrdi prijema segmenta koji je prvi pristigao?

d. Pretpostavimo da dva segmenta koje šalje računar A stignu po redosledu do računara B. Prva potvrda prijema se gubi, a druga potvrda stiže posle prvog isteka vremena. Nacrtajte vremenski dijagram, na kome su prikazani ovi segmenti i svi ostali poslati segmenti i potvrde prijema. (Pretpostavka je da nije bilo drugih gubitaka paketa.) Za svaki segment na svojoj slici, navedite redni broj i broj bajtova podataka; za svaku potvrdu prijema koju dodajete, navedite broj potvrde.

P28. Računari A i B su neposredno povezani linkom brzine 100 MB/s. Između ova dva računara uspostavljena je jedna TCP veza, a računar A preko te veze šalje računaru B ogromnu datoteku. Računar A može da šalje podatke aplikacije u svoj TCP soket brzinom većom od 120 Mb/s, ali računar B može da ih učitava iz svog bafera najvećom brzinom od 50 Mb/s. Opišite efekat TCP kontrole toka.

P29. O SYN kolačićima govorili smo u odeljku 3.5.6.

a. Zašto server mora da koristi poseban početni redni broj u polju SYNACK?

b. Pretpostavimo da napadač zna da ciljani računar koristi SYN kolačiće. Može li napadač jednostavnim slanjem ACK paketa do ciljanog računara da napravi poluotvorene ili potpuno otvorene veze? Ako ne može, zašto?

c. Pretpostavimo da napadač prikuplja veliku količinu početnih rednih brojeva koje je server poslao. Da li napadač slanjem ACK potvrda sa ovim početnim rednim brojevima može da uzrokuje da server kreira mnogo potpuno otvorenih veza? Zašto?

P30. Posmatrajte mrežu prikazanu u slučaju 2 u odeljku 3.6.1. Pretpostavimo da računari A i B sa kojih se šalje imaju stalne vrednosti isteka vremena.

a. Prodiskutujte da li je moguće da se povećanjem brzine konačnog bafera rutera smanji propusna moć (λ).

b. Sada pretpostavimo da oba računara dinamički podešavaju svoje vrednosti isteka vremena (kao što to radi protokol TCP), na osnovu odlaganja smeštanja u bafer na ruteru. Da li će povećanje veličine bafera dovesti do povećanja propusne moći? Zašto?

P31. Pretpostavimo da je pet izmerenih vrednosti `SampleRTT` (videti odeljak 3.5.3) 106 ms, 120 ms, 140 ms, 90 ms i 115 ms. Izračunajte `EstimatedRTT` nakon što dobijete svaku od ovih `SampleRTT` vrednosti, pomoću vrednosti $\alpha = 0{,}125$ i pretpostavljajući da je vrednost `EstimaterRTT` bila 100 ms neposredno pre nego što je prvi od pet uzoraka ovih vrednosti dobijen. Izračunajte takođe `DevRTT` nakon što se dobije svaki od ovih uzoraka, pretpostavljajući da je vrednost $\beta = 0{,}25$ i da je vrednost `DevRTT` bila 5 ms neposredno pre nego što je dobijen prvi od ovih pet uzoraka. Konačno, izračunajte TCP `TimeoutInterval` nakon što se dobije svaki od ovih uzoraka.

P32. Razmotrite TCP postupak za procenu RTT vremena. Pretpostavite da je $\alpha = 0{,}1$. Neka `SampleRTT1` bude najnoviji uzorak RTT vremena, `SampleRTT2` prvi prethodni uzorak RTT vremena i tako redom.

a. Za datu TCP vezu, pretpostavite da su vraćene četiri potvrde prijema sa odgovarajućim uzorcima RTT vremena: `SampleRTT4`, `SampleRTT3`, `SampleRTT2` i `SampleRTT1`. Izrazite `EstimatedRTT` pomoću ova četiri uzorka RTT vremena.

b. Napravite obrazac za opšti slučaj sa n uzoraka RTT vremena.

c. Neka n u obrascu (b) teži beskonačnosti. Objasnite zašto se taj postupak za pronalaženje proseka naziva eksponencijalni klizni prosek.

P33. U odeljku 3.5.3 smo opisali procenjivanje vremena povratnog puta, koje se koristi u protokolu TCP. Šta mislite zbog čega TCP ne meri `SampleRTT` za ponovo poslate segmente?

P34. Koji je odnos promenljive `SendBase` u odeljku 3.5.4 i promenljive `LastByteRcvd` u odeljku 3.5.5?

P35. Koji je odnos promenljive `LastByteRcvd` u odeljku 3.5.5 i promenljive y u odeljku 3.5.4?

P36. U odeljku 3.5.4 smo videli da TCP čeka da dobije tri istovetne ACK potvrde, kako bi pokrenuo brzo ponovno slanje. Šta mislite zbog čega su projektanti TCP-a odlučili da se brzo prenošenje ne pokreće odmah pošto se primi prva ponovljena ACK potvrda za neki segment?

P37. Uporedite protokole GBN, SR i TCP (nema zakasnelih ACK potvrda). Pretpostavimo da su vrednosti isteka vremena za sva tri protokola dovoljno dugačke da prijemni računar (računar B) i računar koji šalje (računar A) mogu da prime 5 uzastopnih segmenata podataka i njihove odgovarajuće ACK potvrde. Pretpostavimo da računar A šalje 5 segmenata podataka računaru B, i da se drugi segment (poslat sa računara A izgubi). Na kraju, računar B je ispravno primio svih pet segmenata podataka.

a. Koliko je ukupno segmenata računar A poslao, a koliko ACK potvrda je ukupno poslao računar B? Koji su njihovi redni brojevi? Odgovorite na ovo pitanje za sva tri protokola.

b. Ako su vrednosti isteka vremena mnogo duže od 5 RTT, koji će tada protokol uspešno da isporuči svih pet segmenata podataka u najkraćem vremenskom intervalu?

P38. U opisu protokola TCP na slici 3.53 vrednost praga, `ssthresh` je podešena na `ssthresh=cwnd/2` na nekoliko mesta, a vrednost `ssthresh` je podešena na polovinu veličine prozora kada se pojavi događaj gubitka. Da li brzina po kojoj pošiljalac šalje kada se pojavi događaj gubitka, mora da bude približno jednaka `cwnd`segmenata po RTT? Obrazložite Vaš odgovor. Ako je odgovor ne, da li možete da predložite drugi način na koji vrednost `ssthresh` može da se podesi?

P39. Posmatrajte sliku 3.46(b). Ako λ'ulaz pređe $R/2$, može li λizlaz preći $R/3$? Objasnite. Posmatrajte sada sliku 3.46(c). Ako λ'ulaz pređe $R/2$, može li λizlaz da pređe $R/4$, pod pretpostavkom da se od rutera ka primaocu paket u proseku dva puta prosleđuje? Objasnite.

spiti anje p našanja pr t k la

Video napomena

P40. Pogledajte sliku 3.58. Pod pretpostavkom da se prikazano ponašanje događa u protokolu TCP Reno, odgovorite na sledeća pitanja. U svim slučajevima, trebalo bi za svaki odgovor da dodate kratko obrazloženje.

a. Označite vremenske intervale u kojima TCP obavlja spori start.

b. Označite vremenske intervale u kojima TCP obavlja izbegavanje zagušenja.

c. Da li je gubitak segmenta nakon 16. povratnog puta otkriven na osnovu tri istovetne ACK potvrde ili posle isteka određenog vremena?

d. Da li je gubitak segmenta nakon 22. runde prenosa otkriven na osnovu tri istovetne ACK potvrde ili posle isteka određenog vremena?

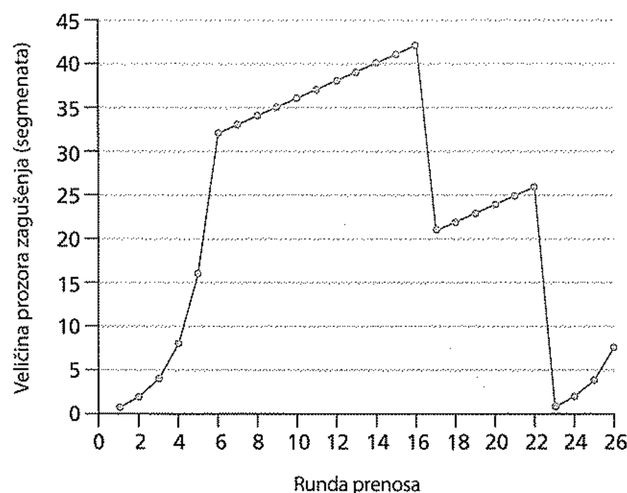
e. Kolika je početna vrednost `ssthresh` tokom prve runde prenosa?

f. Kolika je vrednost `ssthresh` tokom 18. runde prenosa?

g. Kolika je vrednost `ssthresh` tokom 24. runde prenosa?

h. Tokom koje runde prenosa se šalje 70. segment?

i. Pod pretpostavkom da se gubitak paketa otkrije nakon 26. runde na osnovu tri istovetne ACK potvrde, koje će biti vrednosti prozora zagušenja i `ssthresh`?



Slika 3.58 ◈ Veličina prozora protokola TCP u funkciji vremena

j. Pretpostavimo da se koristi protokol TCP Tahoe (umesto protokola TCP Reno) i da je trostruka kopija ACK potvrde primljena u šesnaestoj rundi. Koje je veličine vrednost `ssthresh` i prozor zagušenja u devetnaestoj rundi?

k. Pretpostavimo opet da se koristi protokol TCP Tahoe i da se događaj isteka vremena desio u dvadeset drugoj rundi. Koliko paketa je poslato od sedamnaeste do dvadeset druge runde, uključujući i nju?

P41. Pogledajte sliku 3.56 na kojoj je prikazano konvergentno ponašanje AIMD algoritma protokola TCP. Pretpostavite da umesto multiplikativnog smanjivanja, TCP umanjuje veličinu prozora za stalan iznos. Da li bi takav AIMD algoritam konvergirao ka algoritmu sa jednakim delovima? Obrazložite svoj odgovor dijagramom, sličnim onom sa slike 3.56.

P42. U odeljku 3.5.4 opisali smo udvostručavanje vremena tajmera nakon događaja isteka vremena tajmera. Ovaj mehanizam predstavlja jednu vrstu kontrole zagušenja. Zašto je protokolu TCP potreban mehanizam kontrole zagušenja koji se zasniva na veličini prozora (koji smo proučavali u odeljku 3.7) pored ovog mehanizma udvostručavanja zadatog vremena?

P43. Računar A šalje ogromnu datoteku računaru B preko TCP veze. U toj vezi nikad ne dolazi do gubitaka paketâ i tajmeri nikad ne isteknu. Označite brzinu prenosa linka koji povezuje računar A sa internetom sa R b/s. Pretpostavimo da proces u računaru A može da šalje podatke u svoj TCP soket brzinom od S b/s, gde je $S = 10 \cdot R$. Dalje, pretpostavimo da je prijemni bafer protokola TCP dovoljan da primi celu datoteku, a da predajni bafer može da primi samo jedan procenat datoteke. Šta sprečava proces u računaru A od toga da stalno predaje podatke u svoj TCP soket brzinom od S b/s? TCP kontrola toka? TCP kontrola zagušenja? Nešto drugo? Objasnite.

P44. Posmatramo slanje velike datoteke od jednog do drugog računara preko TCP veze bez gubitaka.

a. Pretpostavimo da TCP za kontrolu zagušenja koristi AIMD algoritam bez sporog starta. Pretpostavimo da `cwnd` poraste za 1 MSS, uvek kada se primi više ACK potvrda i pretpostavimo da su vremena povratnog puta približno jednaka, koliko je potrebno da `cwnd` poraste od 6 MSS na 12 MSS (pod pretpostavkom da se ne događaju gubici paketa)?

b. Kolika je prosečna propusna moć (izražena vrednostima MSS i RTT) za ovu vezu za vreme = 6 RTT?

P45. Podsetite se najopštijeg opisa propusne moći TCP veze. Za vreme kada se brzina veze menja od $W/(2 \cdot RTT)$ do W/RTT, izgubi se samo jedan paket (pri samom isteku tog vremena).

a. Pokažite da je verovatnoća gubitaka (udeo izgubljenih paketa) jednak

$$L = \text{verovatnoća gubitaka} = \frac{1}{\frac{3}{8}W^2 + \frac{3}{4}W}$$

b. Iskoristite prethodni rezultat da biste pokazali da, ako neka veza ima verovatnoću gubitaka L, onda je njena prosečna brzina približno jednaka:

$$\approx \frac{1{,}22 \cdot MSS}{RTT\sqrt{L}}$$

P46. Uzmimo u obzir da samo jedna veza protokola TCP (Reno) koristi jedan link od 10 Mb/s koji ne smešta podatke u bafer. Pretpostavimo da je ovaj link jedini zagušen link između računara koji šalju i primaju. Pretpostavimo da TCP pošiljalac ima veliku datoteku koju šalje primaocu, a primaočev prijemni bafer je mnogo veći od prozora zagušenja. Takođe smo pretpostavili i sledeće: svaki TCP segment je veličine 1 500 bajtova; dvosmerno kašnjenje usled prostiranja ove veze je 150 msec; i da je TCP veza uvek u fazi izbegavanja zagušenja, odnosno, ignorišite spori start.

a. Koja je maksimalna veličina prozora (u segmentima) koju ova TCP veza može da dostigne?

b. Koja je prosečna veličina prozora (u segmentima) i prosečna propusna moć (u b/s) ove TCP veze?

c. Koliko je vremena potrebno za ovu TCP vezu, da bi ponovo dostigla maksimalnu veličinu prozora posle oporavka od gubitka paketa?

P47. Uzmimo u obzir scenario opisan u prethodnom problemu. Pretpostavimo da link brzine 10 Mb/s može da smesti u bafer ograničen broj segmenata. Prodiskutujte zašto bi trebalo da izaberete veličinu bafera koja je barem proizvod brzine linka C i dvosmernog kašnjenja usled prostiranja između pošiljaoca i primaoca, kako bi link uvek bio zauzet slanjem podataka.

P48. Ponovite problem 43, ali zamenite link brzine 10 Mb/s linkom brzine 10 Gb/s. Primetite da ste u odgovoru na deo c, shvatili da bi trebalo dosta vremena da veličina prozora zagušenja dostigne maksimalnu veličinu prozora posle oporavka od gubitka paketa. Skicirajte rešenje za objašnjenje ovog problema.

P49. Označite sa T (koji se meri u broju RTT vremena) vremenski interval koji je potreban TCP vezi da poveća veličinu svog prozora zagušenja sa $W/2$ na W, gde W predstavalja maksimalnu veličinu prozora zagušenja. Objasnite da je T funkcija prosečne propusne moći protokola TCP.

P50. Uzmite u obzir pojednostavljeni algoritam AIMD protokola TCP gde se veličina przora zagušenja meri brojem segmenata, a ne bajtovima. U aditivnom povećanju, veličina prozora zagušenja se povećava za samo jedan segment u svakom RTT. U multiplikativnom smanjenju, veličina prozora zagušenja se smanjuje za pola (ako rezultat nije ceo broj, zaokružuje se nadole do najbližeg celog broja). Pretpostavimo da dve TCP veze, C_1 i C_2 dele jedan zagušeni link brzine 30 segmenata u sekundi. Pretpostavimo da se obe ove veze nalaze u fazi izbegavanja zagušenja. RTT veze C_1 je 50 msec, a veze C_2 100 msec. Pretpostavimo da brzina podataka na linku premaši kapacitet linka, obe TCP veze će imati gibitke segmenata.

a. Ako obe veze C_1 i C_2, u trenutku t_0 imaju prozor zagušenja veličine 10 segmenata, koliki su njihovi prozori zagušenja posle 1000ms?

b. Posmatrano dugoročno, da li će ove dve veze dobiti isti deo propusnog opsega zagušenog linka? Objasniti.

51. Razmotrite mrežu opisanu u prethodnom problemu. Sada pretpostavite da ove dve TCP veze, C_1 i C_2, imaju isto RTT vreme. Pretpostavimo da je u vremenskom trenutku t_0, veličina prozora zagušenja C1 15 segmenata, ali da je veličina prozora zagušenja C2 10 segmenata.

a. Kolike su veličine njihovih prozora zagušenja posle 2 200ms?

b. Da li će ove dve veze dugoročno imati isti udeo u propusnom opsegu zagušenog linka?

c. Kažemo da su dve veze sinhronizovane, ako obe veze dostižu u isto vreme maksimalnu i minimalnu veličinu njihovih prozora. Da li će dugoročno ove dve veze na kraju biti sinhronizovane? Ako je tako, kolike će biti maksimalne veličine njihovih prozora?

d. Da li ovo sinhronizovanje pomaže u poboljšanju iskorišćenja deljenog linka? Zašto? Ukratko opišite neku ideju koja bi prekinula ovu sinhronizaciju.

P52. Posmatrajte modifikaciju algoritma kontrole zagušenja protokola TCP. Umesto aditivnog povećanja, možemo da upotrebimo multiplikativno povećanje. TCP pošiljalac povećava svoju veličinu prozora za malu pozitivnu konstantu a $(0 < a < 1)$, svaki put kad primi ispravnu ACK potvrdu. Pronađite funkcionalnu vezu između verovatnoće gubitaka L i maksimalnog prozora zagušenja W. Obrazložite da je za ovaj izmenjen protokol TCP, bez obzira na prosečnu propusnu moć protokola TCP, TCP vezi potrebna uvek ista količina vremena za povećanje veličine prozora zagušenja sa $W/2$ na W.

P53. Prilikom razmatranja budućnosti protokola TCP u odeljku 3.7 napomenuli smo da bi protokol TCP za postizanje propusne moći od 10 Gb/s mogao da podnese verovatnoću gubitaka segmenata od samo 2 · 1010 (što odgovara jednom događaju gubitka na svakih 5 000 000 000 segmenata). Pokažite kako se izvode vrednosti 2 · 1010 i jedan od 5 000 000 000 za vrednosti RTT i MSS, date u odeljku 3.7. Ukoliko bi TCP trebalo da podrži vezu od 100 Gb/s, kolike bi gubitke mogao da podnese?

P54. U našem opisu TCP kontrole zagušenja u odeljku 3.7 podrazumevali smo da TCP pošiljalac uvek ima podatke za slanje. Razmotrite sada slučaj kada TCP pošiljalac pošalje veliku količinu podataka, a zatim ne radi ništa (pošto nema podataka za slanje) od trenutka $t1$. TCP ostaje besposlen relativno duže vreme, a zatim hoće ponovo da šalje podatke u trenutku $t2$. Kakve su prednosti i mane, ako TCP upotrebi vrednosti `cwndi ssthresh` iz trenutka $t1$, kada počne da šalje podatke u trenutku $t2$? Koje drugo rešenje biste preporučili? Zašto?

P55. U ovom problemu istražujemo da li protokoli UDP i TCP obezbeđuju ikakvu proveru autentičnosti krajnje tačke.

a. Razmatramo slučaj da server primi zahtev UDP paketom i na taj zahtev odgovori UDP paketom (na primer, kao što rade DNS serveri). Ukoliko klijent sa IP adresom X prikrije svoju adresu adresom Y, gde će server poslati svoj odgovor?

b. Pretpostavimo da server primi SYN segment sa IP adresom izvora Y, a nakon što odgovori segmentom SYNACK, primi ACK potvrdu sa IP adresom izvora Y sa ispravnim brojem potvrde prijema. Pretpostavljajući da server slučajno bira početni redni broj i da nije u pitanju napad „čoveka u sredini", može li server da bude siguran da je klijent zaista na adresi Y (a ne na nekoj drugoj adresi X, koju prikriva adresom Y)?

P56. U ovom problemu razmatramo kašnjenje koje nastaje zbog faze sporog starta protokola TCP. Razmotrimo slučaj u kome su klijent i server neposredno povezani linkom brzine R. Pretpostavimo da klijent želi da preuzme objekat čija je veličina tačno jednaka 15 S, gde je S najveća veličina segmenta (MSS). Označimo povratno vreme između klijenta i servera sa RTT (pretpostavljamo da je nepromenljivo). Zanemarujući zaglavlja protokola, odredite vreme potrebno za preuzimanje objekta (uključujući vreme za uspostavljanje veze) kada je:

a. $4\,S/R > S/R + RTT > 2S/R$

b. $8\,S/R > S/R + RTT > 4S/R$

c. $S/R > RTT$.

Programerski zadaci

Izrada protokola za pouzdan prenos podataka

U ovom programerskom zadatku napisaćete kôd za pošiljaoca i primaoca transportnog sloja kojim se ostvaruje jednostavan protokol za pouzdan prenos podataka. Postoje dve verzije ove vežbe, verzija protokola sa naizmeničnim bitovima i GBN verzija. Ova vežba bi trebalo da bude zabavna – vaše ostvarenje bi trebalo veoma malo da se razlikuje od onog što bi se zahtevalo u stvarnim okolnostima.

Pošto verovatno nemate samostalne mašine (sa operativnim sistemom koji možete da menjate), vaš kôd će morati da se izvršava u simuliranom hardversko-softverskom okruženju. Međutim, programski interfejs za vaše rutine, tj. kôd koji poziva vaše entitete odozgo i odozdo veoma liči na ono što se zaista događa u okruženju UNIX. (Tačnije, softverski interfejsi, opisani u ovom programerskom zadatku, mnogo su realniji od pošiljalaca i primalaca sa beskonačnim petljama, koji su opisani u drugim udžbenicima.) Zaustavljanje i pokretanje tajmera se takođe simulira, a tajmerski prekidi pokretaće rutinu obrade slučaja izazvanog tajmerom.

Cela laboratorijska vežba, zajedno sa kodom koji je potrebno da kompajlirate sa Vašim kodom nalazi se na veb stranici ove knjige http://www.awl.com/kurose-ross.

Wireshark laboratorijske vežbe: istraživanje protokola TCP

U ovoj vežbi upotrebićete svoj veb pretraživač za pristupanje datotekama sa veb servera. Kao i u prethodnim *Wireshark* vežbama, koristićete *Wireshark* za prikupljanje paketâ koji stižu do Vašeg računara. Za razliku od prethodnih vežbi, *takođe* ćete moći da sa veb servera sa kojeg ste preuzeli određenu datoteku preuzmete zapise paketâ koje je moguće očitati programom *Wireshark*. U tim podacima, zapisima preuzetim sa servera, pronaći ćete pakete koji su nastali Vašim pristupanjem veb serveru. Analiziraćete podatke o zapisima paketâ sa klijentske i serverske strane, da biste istražili ponašanje protokola TCP. Posebno ćete proceniti performanse TCP veze između Vašeg računara i veb servera. Pratićete ponašanje prozora protokola TCP i izvoditi zaključke o gubitku paketâ, ponovnim slanjima, kontroli toka i kontroli zagušenja, kao i o procenjenom trajanju povratnog puta.

Kao i kod svih *Wireshark* laboratorijskih vežbi, potpuni opis ove vežbe dostupan je na veb adresi ove knjige http://www.awl.com/kurose ross.

Wireshark laboratorijske vežbe: istraživanje protokola UDP

U ovoj kratkoj vežbi, prikupićete pakete i analizirati ponašanje Vaše omiljene aplikacije koja koristi protokol UDP (na primer, DNS ili multimedijalnu aplikaciju kao što je *Skype*). Kao što smo naučili u odeljku 3.3, UDP je jednostavan protokol, bez suvišnih delova. U ovoj laboratorijskoj vežbi, istražujete polja zaglavlja u UDP segmentu, kao i to kako se izračunava kontrolni zbir.

Kao i kod svih *Wireshark* laboratorijskih vežbi, potpuni opis ove vežbe dostupan je na veb adresi ove knjige http://www.awl.com/kurose ross.