Glava 3: Nivo transporta

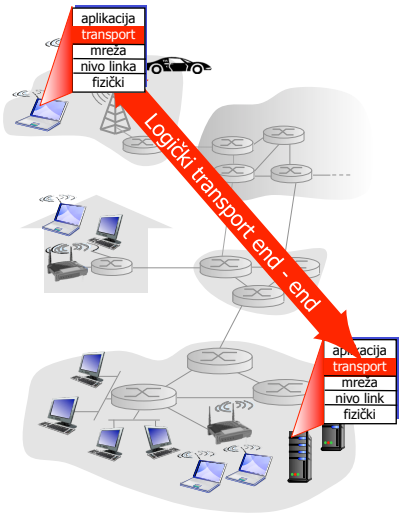
Ciljevi:

- Shvatiti principe na kojima počivaju servisi nivoa transporta:
  - Multipleksiranje/ demultipleksiranje
  - Pouzdan prenos podataka
  - Kontrola protoka
  - Kontrola zagušenja
- Protokoli transportnog nivoa na Internetu:
  - UDP: nekonektivni transport
  - TCP: konektivni transport i kontrola zagušenja

Glava 3: Sadržaj

Transportni servisi i protokoli

- obezbjeđuju *logičku komunikaciju* između aplikacija koje se odvijaju na različitim hostovima
- transportni protokoli se implementiraju na krajnjim sistemima
  - Predajna strana transportnog protokola: dijeli poruke u segmente, prosleđuje ih mrežnom nivou
  - Prijemna strana transportnog protokola: desegmentira segmente u poruke, i prosleđuje ih nivou aplikacije
- Više od jednog transportnog protokola je na raspolaganju aplikacijama
  - Internet: TCP i UDP



Poređenje transportnog i mrežnog nivoa

- *Mrežni nivo:* logička komunikacija između hostova
- *Transportni nivo:* logička komunikacija između procesa
  - Oslanja se na servise mrežnog nivoa i poboljšava njihove osobine

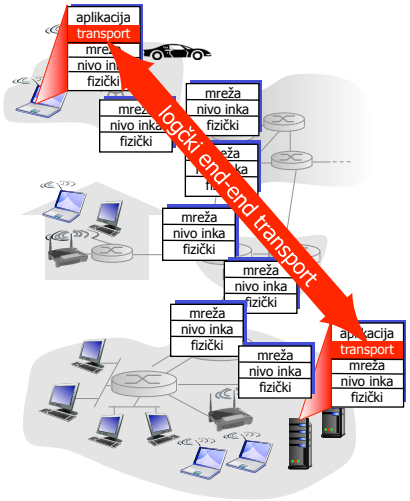
Analogija:

12 ljudi šalje pisma za 12 ljudi

- procesi = ljudi
- poruke = poruke u kovertama
- hostovi = kuće u kojima ljudi žive
- transportni protokol = zapis na koverti
- mrežni protokol = poštanski servis

Internet protokoli transportnog nivoa

- pouzdana, redosledna isporuka (TCP)
  - Kontrola zagušenja
  - Kontrola protoka
  - Uspostavljanje veze
- nepouzdana, neredosledna isporuka: UDP
  - Bez unapređenja"best-effort" pristupa IP
- Servisi koji se ne pružaju:
  - Garantovano kašnjenje
  - Garantovana propusnost



Glava 3: Sadržaj

3.1 Servisi nivoa transporta

3.2 Multipleksiranje i demultipleksiranje

3.3 Nekonektivni transport: UDP

3.4 Konektivni transport: TCP
- Struktura segmenta
- Pouzdani prenos podataka
- Kontrola protoka
- Upravljanje vezom

3.5 TCP kontrola zagušenja

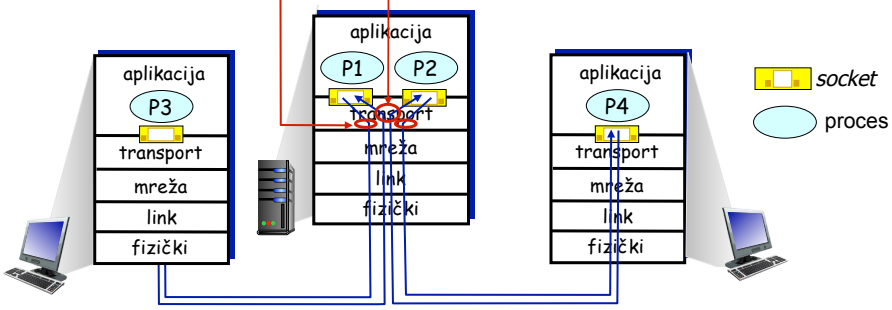
Multipleksiranje/demultipleksiranje

Multipleksiranje na predaji:

Manipulisanje podacima iz više *socket*-a, dodavanje transportnog zaglavlja (koristi se za demultipleksiranje)

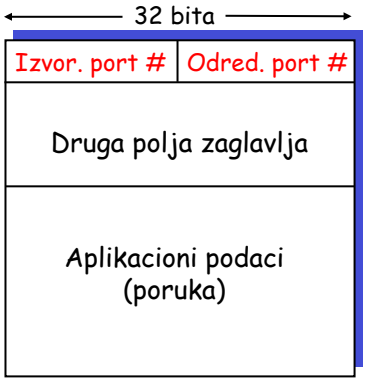
Demultipleksiranje na prijemu:

Koristi zaglavlje za predaju primljenih segmenata pravom *socket*-u



Kako funkcioniše demultipleksiranje?

- host prima IP datagrame
  - Svaki datagram ima izvorišnu IP adresu, odredišnu IP adresu
  - Svaki datagram nosi 1 segment nivoa transporta
  - Svaki segment ima izvorišni i odredišni broj porta
    - 16 bitni broj (0-65535)
    - 0-1023 su tzv"dobro poznati" portovi koji su unaprijed rezervisani (RFC1700, www.iana.org)
- host koristi IP adrese & brojeve portova da usmjeri segment na odgovarajući socket



TCP/UDP format segmenta

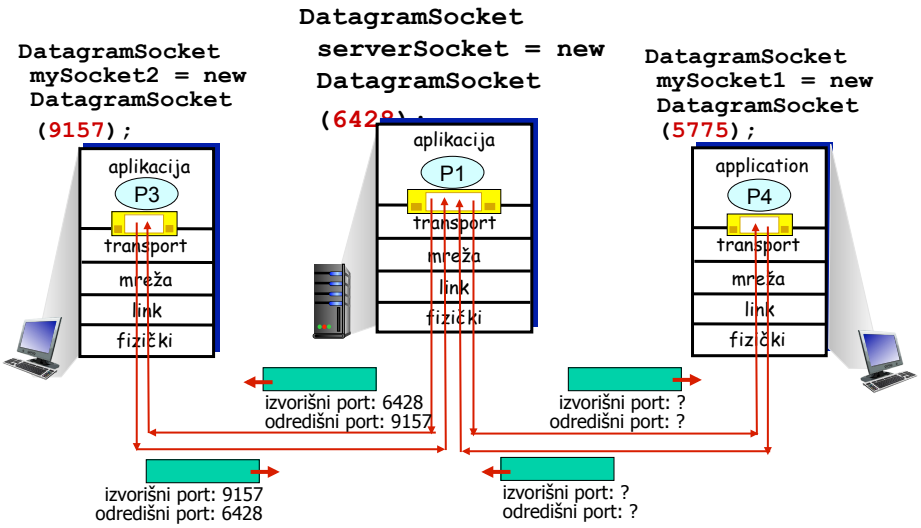
Nekonektivno demultipleksiranje (UDP)

- Kada se kreira UDP *socket* transportni nivo mu odmah dodjeljuje broj porta koji ne koristi neki drugi UDP *socket* na hostu
- Klijentska strana transportnog protokola obično *socket*-u dodjeljuje ne "dobro poznate" portove 1024-65535
- UDP *socket* identifikuju dva podatka:

(IP adresa odredišta, broj porta odredišta)

- Kada host primi UDP segment:
  - Provjerava odredišni broj porta u segmentu
  - Usmjerava UDP segment u *socket* koji ima taj broj porta
- IP datagrami sa različitim izvorišnim IP adresama i/ili izvorišnim brojevima portova se usmjeravaju na isti *socket*

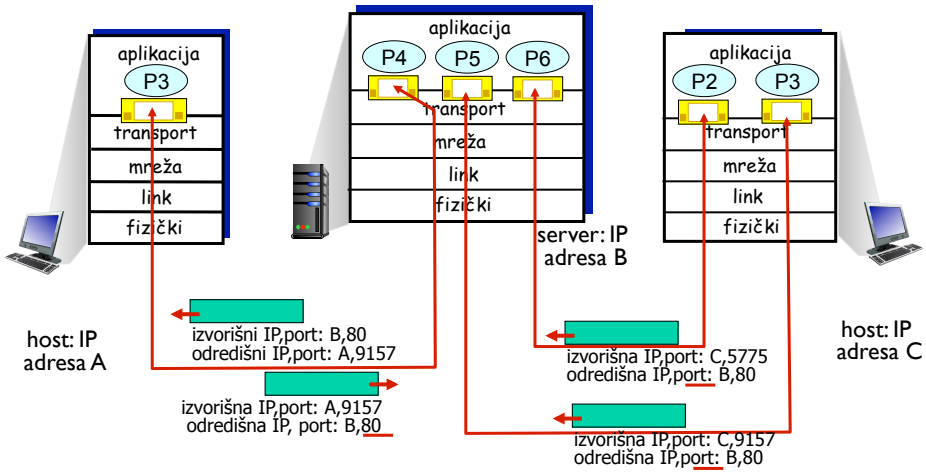
Nekonektivno multipleksiranje



Konektivno demultipleksiranje

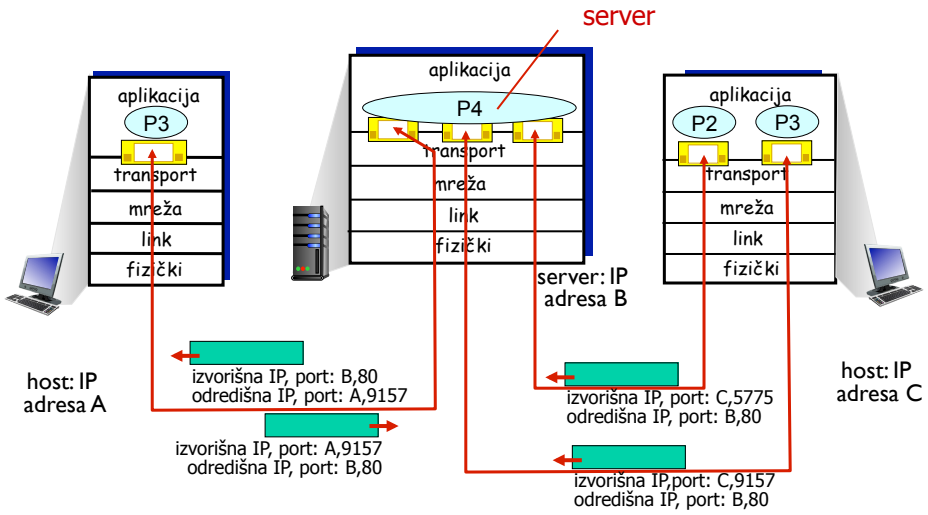
- TCP *socket* identifikuju 4 parametra:
  - Izvorišna IP adresa
  - Izvorišni broj porta
  - Odredišna IP adresa
  - Odredišni broj porta
- Prijemni host koristi sve četiri vrijednosti za usmjeravanje segmenta na odgovarajući socket
- Server host može podržavati više simultanih TCP *socket*-a:
  - svaki *socket* je identifikovan sa svoja 4 parametra
- Web serveri imaju različite *socket*-e za svakog povezanog klijenta
  - ne-perzistentni HTTP će imati različite *socket*-e za svaki zahtjev

Konektivno demultipleksiranje



tri segmenta, adresirana na IP adresu: B, dest port: 80 se demultipleksiraju na različite sokete

Konektivno demultipleksiranje



Glava 3: Sadržaj

UDP: User Datagram Protocol [RFC 768]

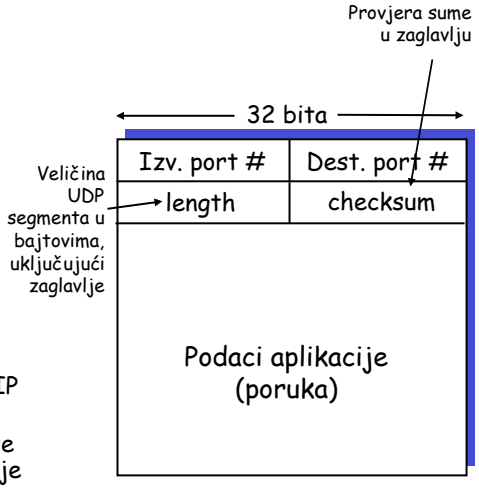
- Nema poboljšanja koja se nude Internet protokolu
- "*best effort*" servis, UDP segmenti mogu biti:
  - izgubljeni
  - neredosledno predati
- nekonektivni:
  - nema uspostavljanja veze *(handshaking)* između UDP pošiljaoca i prijemnika
  - svaki UDP segment se tretira odvojeno od drugih

Zašto onda UDP?

- Nema uspostavljanja veze (koja povećava kašnjenje)
- jednostavnije: ne vodi se računa o stanju veze
- manje zaglavlje segmenta (8B u odnosu na 20B kod TCP-a)
- nema kontrole zagušenja: UDP može slati podatke onom brzinom kojom to aplikacija želi

UDP: više

- Često se koristi za "*streaming*" multimedijalne aplikacije
  - Tolerantne u odnosu na gubitke
  - Osjetljive na brzinu prenosa
- drugi UDP korisnici
  - DNS
  - SNMP (zbog toga što mrežne menadžment aplikacije funkcionišu kada je mreža u kritičnom stanju)
  - RIP (zbog periodičnog slanja RIP update-a)
- Pouzdani prenos preko UDP: mora se dodati pouzdanost na nivou aplikacije
  - Oporavak od greške na nivou aplikacije
- Problem kontrole zagušenja je i dalje otvoren!



Format UDP segmenta RFC 768

Glava 3: Sadržaj

3.1 Servisi nivoa transporta

3.2 Multipleksiranje i demultipleksiranje

3.3 Nekonektivni transport: UDP

3.4 Konektivni transport: TCP
- Struktura segmenta
- Pouzdani prenos podataka
- Kontrola protoka
- Upravljanje vezom

3.5 TCP kontrola zagušenja

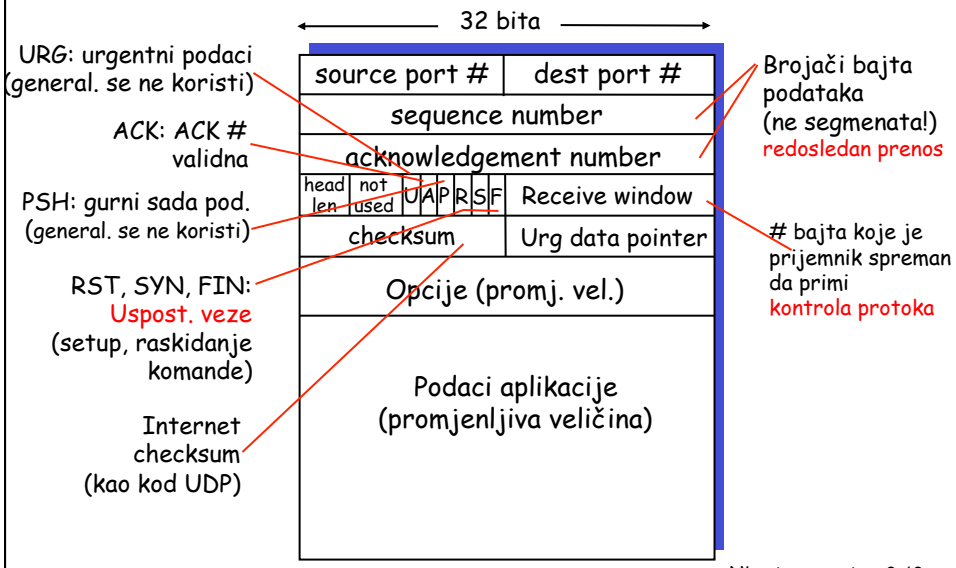
TCP: Pregled

RFC-ovi: 793, 1122, 1323, 2018, 2581

- tačka-tačka:
  - Jedan pošilj, jedan prij.
- pouzdan, redosledan prenos bajta:
  - nema "granica poruka"
- "pipelined":
  - TCP kontrola zagušenja i protoka podešava veličinu prozora
- *Baferi za slanje & prijem*

- "full duplex" prenos:
  - U istoj vezi prenos u dva smjera
  - MSS: maksimalna veličina podataka sloja aplikacije u segmentu (1460B, 536B, 512B)
- konektivan:
  - "*handshaking*" (razmjena kontrolnih poruka) inicira je pošiljalac, razmjenjuje stanja prije slanja
- kontrola protoka:
  - Pošiljalac ne može "zagušiti" prijemnika

TCP struktura segmenta (21B-1480B)



TCP brojevi u sekvenci, ACK-ovi

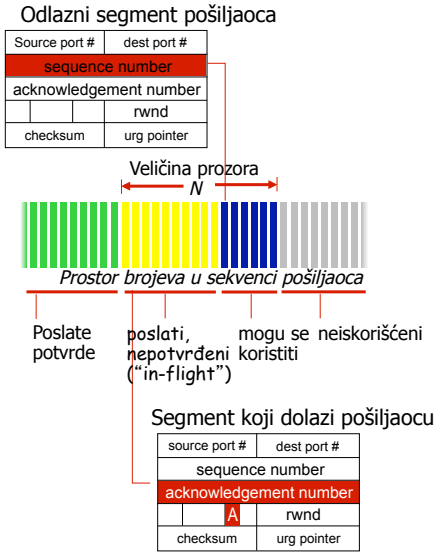
Brojevi u sekvenci:

- Dodjeljuje se broj prvom bajtu iz sadržaja segmenta
- Inicijalne vrijednosti se utvrđuju na slučajan način.

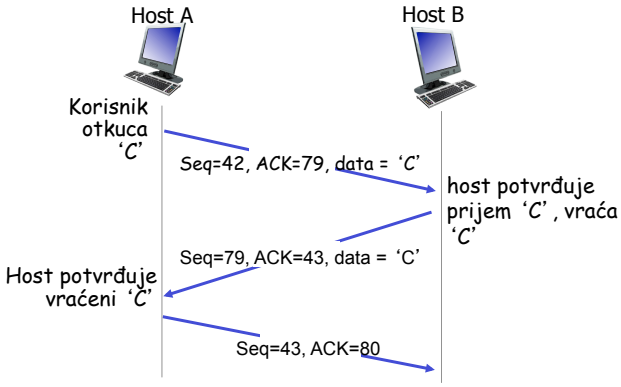
Potvrde (ACK):

- Broj sekvence sledećeg bajta koji se očekuje sa druge strane
- kumulativni ACK

Q: Kako se prijemnik ponaša prema *out-of-order* segmentima?



TCP brojevi u sekvenci, potvrde



jednostavni Telnet scenario

TCP pouzdani prenos podataka

- TCP kreira rdt servis po IP nepouzdanom servisu
- "*Pipelined*" segmenti
- Kumulativne potvrde
- TCP koristi jedan retransmisioni tajmer
- Retransmisije su trigerovane sa:
  - *timeout* događajima
  - duplim ack-ovima
- Na početku treba razmotriti pojednostavljenog TCP pošiljaoca:
  - Ignorišu se duplirani ack-ovi
  - Ignorišu se kontrole protoka i zagušenja

Događaji vezani za TCP pošiljaoca

1. Podaci primljeni od aplikacije:

- Kreiranje segmenta sa sekvencom brojeva
- Broj u sekvenci je *byte*-stream broj prvog bajta podataka u segmentu
- Startuje se tajmer ako to već nije urađeno
- Interval *timeout*-a se izračunava po odgovarajućoj formuli

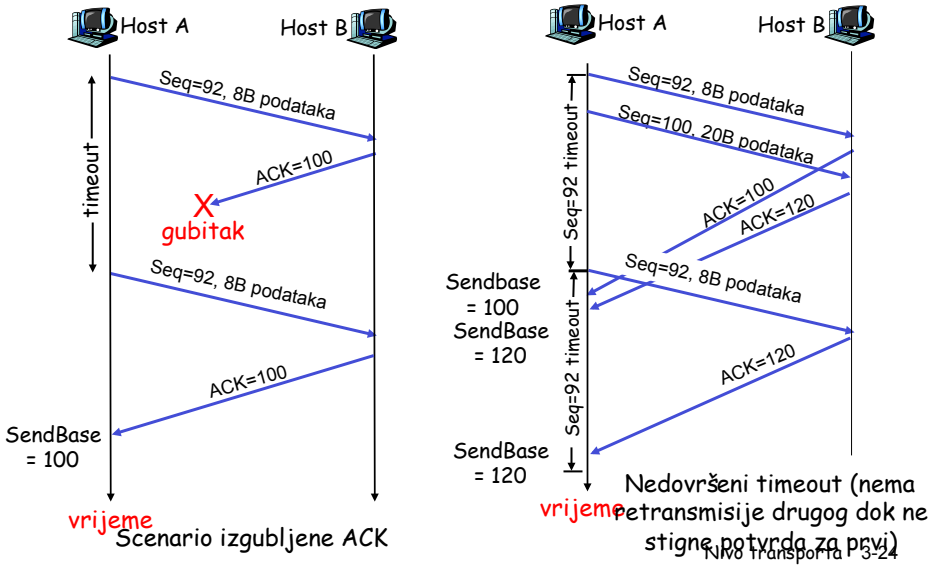
2. timeout:

- Ponovo se šalje segment koji je izazvao timeout
- restartovati tajmer

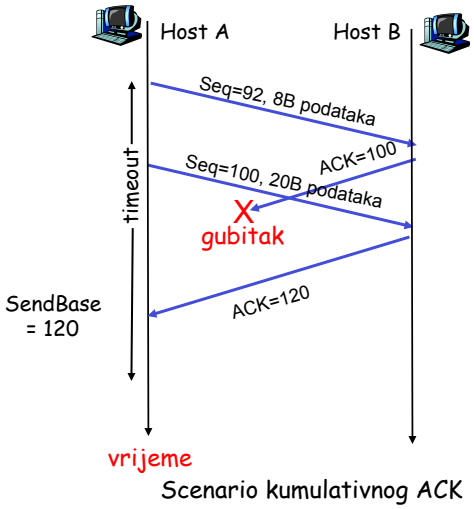
3.Ack primljen:

- Ako se potvrdi prijem ranije nepotvrđenog segmenta
  - Napraviti odgovarajući *update*
  - startovati tajmer ako postoje segmenti koji čekaju

TCP: scenariji retransmisije



TCP: scenariji retransmisije



Scenario kumulativnog ACK

U slučaju kada istekne *timeout* period, TCP se više ne pridržava ranije pomenute formule za izračunavanje timeout intervala. Umjesto nje TCP duplira raniju vrijednost *timeout* intervala.

TCP Round Trip Time i Timeout

P: kako postavitiTCP vrijeme timeout-a?

- Duže od RTT-a
  - ali RTT varira
- Suviše kratko: prerani timeout
  - nepotrebne retransmisije
- Previše dugo: spora reakcija na gubitak segmenta
- Potrebna je aproksimacija RTT-a

P: kako aproksimirati RTT?

- `SampleRTT`: mjeriti vrijeme od slanja segmenta do prijema ACK
  - Ignorisati retransmisije
  - Radi se za samo jedan nepotvrđeni segment
- `SampleRTT` će varirati, želja je za što boljom estimacijom RTT
  - Više mjerenja, a ne samo trenutno `SampleRTT`

P: Da li SampleRTT vezivati za svaki nepotvrđeni segment?

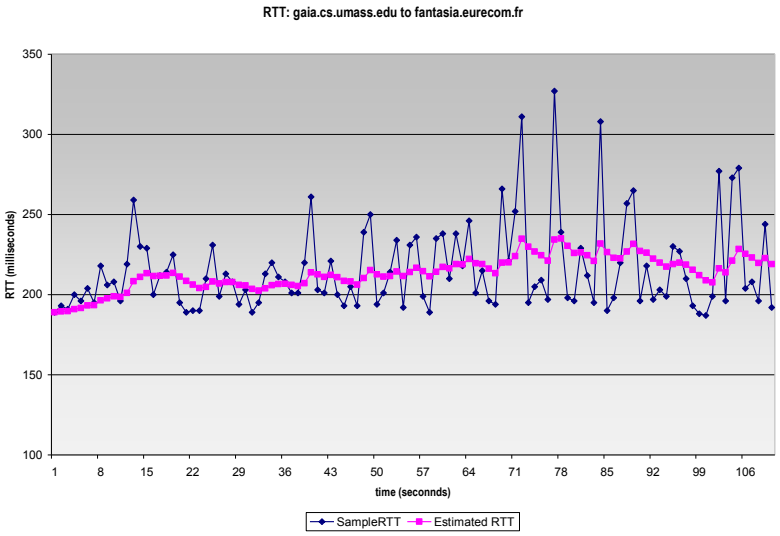
P: Zašto ignorisati retransmisije?

TCP Round Trip Time and Timeout

EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT

- Uticaj prošlosti opada po eksponencijalnoj raspodjeli
- *Exponential weighted moving average* (EWMA) ili eksponencijalno ponderisani klizni prosjek
- Tipična vrijednost: α = 0.125

Primjer RTT estimacije:



TCP Round Trip Time i Timeout

Setovanje timeout-a

- `EstimatedRTT` + "sigurnosna margina"
  - Velika varijacija u `EstimatedRTT` -> velika sigurnosna margina
- Prvo se estimira koliko SampleRTT odstupa od EstimatedRTT:

```
DevRTT = (1-β)*DevRTT +
              β*|SampleRTT-EstimatedRTT|
```

(tipično, β = 0.25) EWMA od ove razlike

Tada se setuje timeout interval:

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```


TCP generisanje ACK [RFC 1122, RFC 2581]

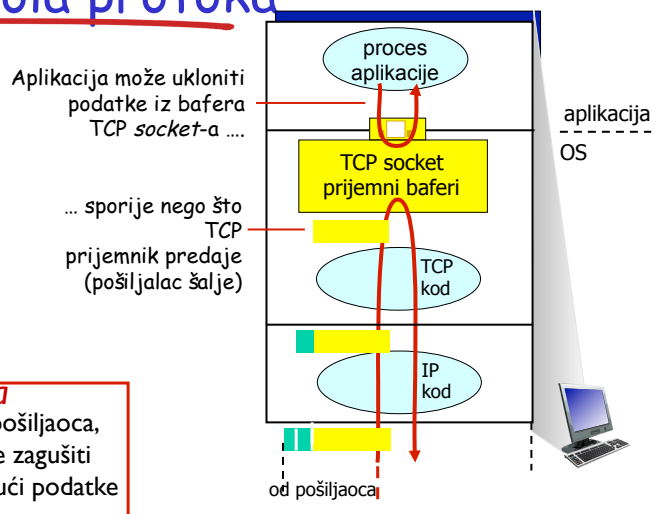
Događaj na prijemu	TCP akcije prijemnika
Dolazak in-order segmenta sa očekivanim brojem u sekvenci. Svi podaci do očekiv. broja su potvrđ.	ACK sa kašnjenjem. Čeka do 500ms za sledeći segment. Ako nema sledećeg, šalje ACK.
Dolazak in-order segmenta sa očekiv. brojem u sekvenci. Potvrđ. prijema drugog segmenta u toku.	Odmah šalje jednu kumulativnu ACK, potvrđujući oba in-order segmenta
Dolazak out-of-order segmenta sa većom vrijednosti broja u sekv. od očekivane. Detektovan prekid.	Odmah šalje duplikat ACK, indicirajući broj u sekvenci očekivanog bajta.
Dolazak segmenta koji djelimično ili potpuno popunjava prekid.	Odmah šalje ACK, omogućavajući da segment popuni prekid

"Fast Retransmit"

- *Time out* period je često predug:
  - Dugo kašnjenje prije slanja izgubljenog paketa
- Detekcija izgubljenog segmenta preko dupliranih ACK-ova.
  - Pošiljalac često šalje mnogo segmenata
  - Ako je segment izgubljen, najvjerovatnije će biti dosta dupliranih ACK-ova.
- Ako pošiljalac primi 3 ACK za iste podatke, pretpostavlja se da je segment poslije potvrđenog izgubljen:
  - "fast retransmit": ponovno slanje segmenta prije nego što je tajmer istekao

P: Da li TCP ima GBN ili "*selective repeat*" kontrolu greške?
P: Zašto 3 a ne dva ACK?

TCP kontrola protoka

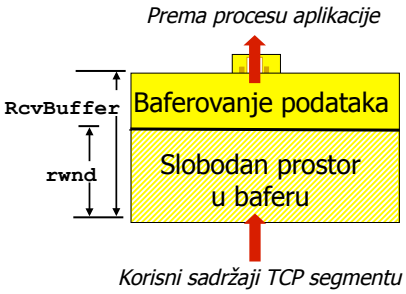


prijemnikov protokol *stack*

Kontrola protoka
Prijemnik kontroliše pošiljaoca, tako da pošiljalac neće zagušiti prijemnikov bafer šaljući podatke velikom brzinom

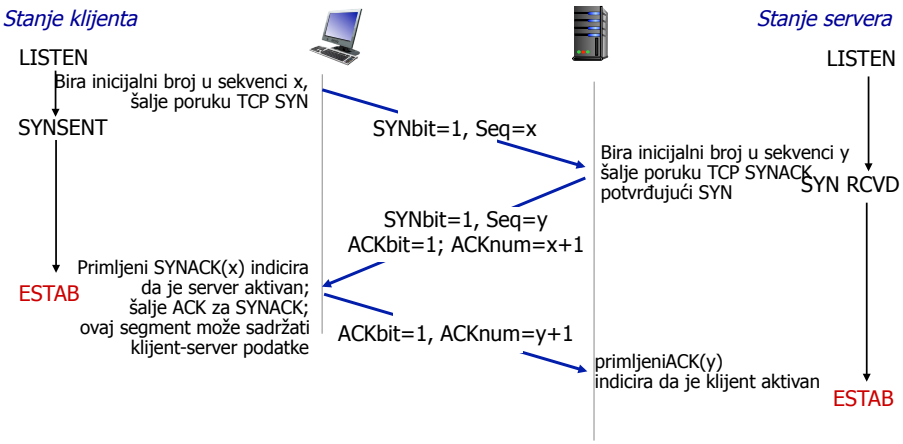
TCP flow control

- Prijemnik oglašava slobodan prostor u baferu podešavanjem vrijednosti u polje **`rwnd`** u zaglavljuTCP segmenta
  - **`Veličina RcvBuffer`** se podešava u opcijama *socket*-a (tipična vrijednost 4096B)
  - Mnogi OS podešavaju **`RcvBuffer`**
- Pošiljalac ograničava broj nepotvrđenih ("in-flight") podataka na vrijednost prijemnikovog **`rwnd`**
- Garantuje da se ne prepuni bafer

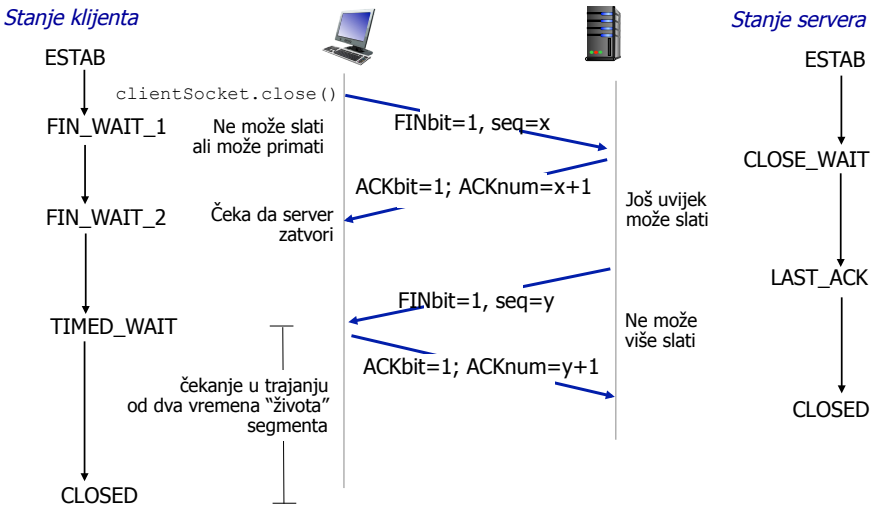


Baferovanje na prijemnikovoj strani

TCP 3-way handshake



TCP: zatvaranje konekcije



Glava 3: Sadržaj

3.1 Servisi nivoa transporta

3.2 Multipleksiranje i demultipleksiranje

3.3 Nekonektivni transport: UDP

3.4 Konektivni transport: TCP

- Struktura segmenta
- Pouzdani prenos podataka
- Kontrola protoka
- Upravljanje vezom

3.5 TCP kontrola zagušenja

TCP kontrola zagušenja

- Kontrola od kraja do kraja (bez učešća mreže)
- Pošiljalac ograničava slanje:

  `LastByteSent-LastByteAcked`
  `≤ CongWin`
- Približno,

$$\text{brzina} = \frac{\text{CongWin}}{\text{RTT}} \quad \text{b/s}$$

- **CongWin** je dinamička funkcija detekcije zagušenja mreže

Kako pošiljac otkriva zagušenje?

- gubitak = timeout *ili* 3 duplirane potvrde
- TCP pošiljalac smanjuje brzinu (**CongWin**) poslije gubitka

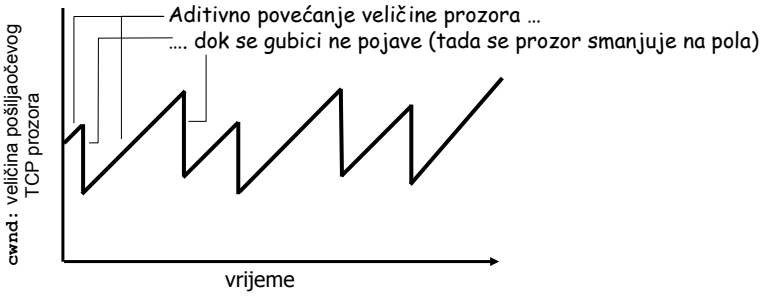
tri mehanizma:

- AIMD
- "slow start"
- konzervativan poslije timeouta

TCP AIMD

Multiplikativno smanjenje: smanjuje **CongWin** na pola u slučaju gubitka

Aditivno povećanje: povećava **CongWin** za 1 MSS svaki RTT u odsustvu gubitka: *sondiranje*



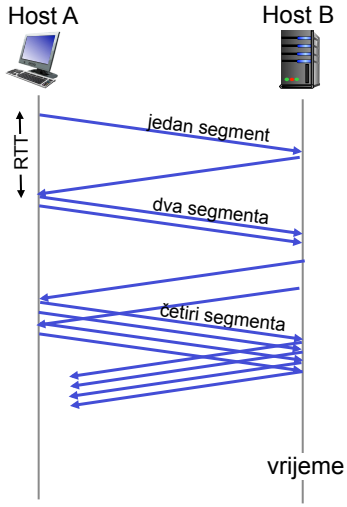
Duga TCP veza

TCP Slow Start

- Kada veza počne, `CongWin` = 1 MSS
  - Primjer: MSS = 500 B & RTT = 200 ms
  - Inicijalna brzina =20 kb/s
- Dostupna propusnost može biti>> MSS/RTT
  - Poželjno je brzo podešavaje na željenu brzinu
- Kada veza počne, povećava brzinu eksponencijalno do prvog gubitka

TCP Slow Start (više)

- Kada veza počne, eksponencijalno povećanje brzine do gubitka :
  - Udvostručuje se `CongWin` svaki RTT
  - Inkrementira se `CongWin` sa svakim primljenim
  - ACK Sumarum: inicijalna brzina je niska ali brzo raste



Ponavljanje

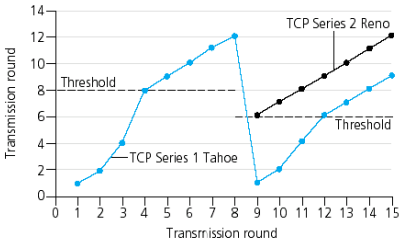
- Poslije 3 duplirane ACK:
  - **CongWin** se smanjuje na pola
  - Prozor raste linearno
- Ali posle timeout-a:
  - **CongWin** = 1 MSS;
  - Prozor raste eksponencijalno do praga a zatim linearno

Filozofija:

• 3 duple ACK indicira da je mreža sposobna da šalje
• timeout prije 3 duple ACK je "alarmantan"

Ponavljanje (više)

P: Kada eksponencijalna prelazi u linearnu?



Implementacija:

- Varijabilni prag (Tahoe)
- U slučaju gubitka, prag se postavlja na 1/2 vrijednosti CongWin prije gubitka
- U slučaju gubitka CongWin se smanjuje na pola(Reno)

TCP Tahoe

- "Slow Start", izbjegavanje kolizije
- Detektuje zagušenje kroz isticanje timeout-a i trostruke potvrde
- Inicijalizacija
  - CongWin=1;
  - Threshold=1/2 Max(Win)
- Poslije timeouta i trostruke potvrde
  - Threshold= 1/2 CongWin, CongWin= 1
  - Ulazi u slow start

TCP Reno

- "Fast Retransmit", "Fast recovery"
- Detektuje zagušenje kroz timeout-e i duplikate ACK-ova
- Kada se primi trostruki duplikat nekog ACK
  - Izbjegava slow start i ide direktno u fazu izbjegavanja kolizije
  - Threshold = 1/2 CongWin; Congwin = Threshold (Koristi AIMD)
- Kada se pojavi timeout
  - "Slow start"

Izbjegavanje zagušenja

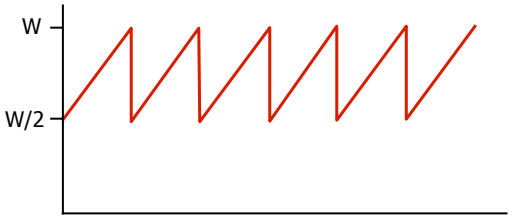
- Na bazi rutera
  - RED, (DecBIT)
- Na bazi izvora
  - TCP Vegas

Sumarum: TCP kontrola zagušenja

- Kada je **CongWin** ispod **Threshold**, pošiljalac je u slow-start fazi, prozor raste eksponencijalno.
- Kada je **CongWin** iznad **Threshold**, pošiljalac je u fazi izbjegavanja kolizije, prozor raste linearno.
- Kada se trostruki duplirani ACK pojavi, **Threshold** se setuje **CongWin/2** a **CongWin** se setuje na **Threshold**.
- Kada se pojavi timeout, **Threshold** se setuje na **CongWin/2** i **CongWin** se setuje na 1 MSS.

TCP propusnost

- Koliko iznosi srednja propusnost TCP-a u funkciji veličine prozora i RTT?
  - Ignoriše se slow start
- Neka je W veličina prozora kada nastaju gubici.
- Kada je veličina prozora W, propusnost je W/RTT
- Poslije gubitka, veličina prozora pada na W/2, propusnost na W/2RTT.
- Srednja propusnost: .75 W/RTT



Budućnost TCP-a

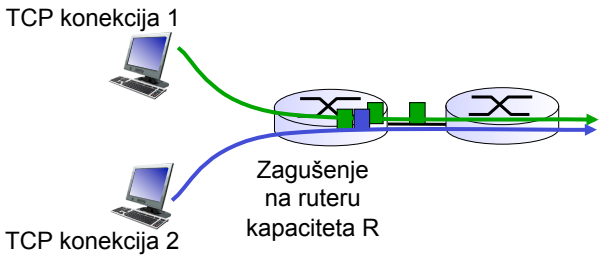
- primjer: 1500 B segmenti, 100ms RTT, želi se 10 Gb/s propusnost
- Zahtijeva se veličina prozora od W = 83,333 segmenata
- Srednja propusnost u zavisnosti vjerovatnoće gubitka:

$$\frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

- Vjerovatnoća gubitka ➜ L = $2 \cdot 10^{-10}$
- Potrebne su nove verzije TCP-a za high-speed potrebe!
- http://netlab.caltech.edu/FAST/

Korektnost TCP

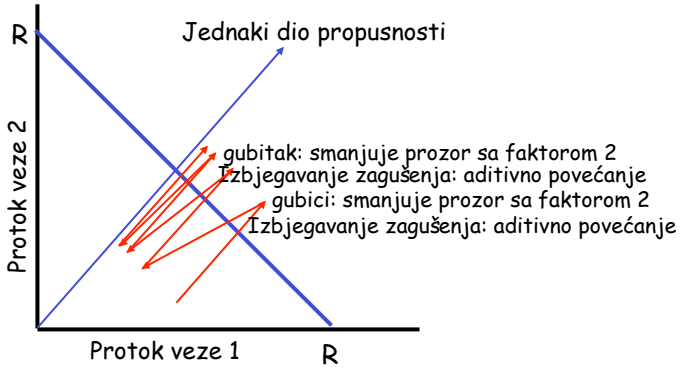
Cilj korektnosti: ako K TCP sesija dijele isti zagušeni link propusnosti R, svaki bi trebao da ima srednju propusnost od R/K



Zašto je TCP korektan?

Dvije sučeljene sesije:

- Aditivno povećanje daje porast za 1, tako da protok raste
- Multiplikativno smanjenje smanjuje protok proporcionalno



Korektnost (više)

Korektnost i UDP

- Multimedijalne aplikacije često ne koristeTCP
  - Ne žele da kontrola zagušenja ograniči kapacitet
- Umjesto toga se koristi UDP:
  - Ubacuje audio/video konstantnom brzinom, toleriše gubitak paketa
- Oblast istraživanja: TCP

Korektnost i paralelne TCP konekcije

- Nema prevencije da aplikacija otvori paralelne veze između 2 hosta.
- Web browser-i to rade
- Primjer: link propusnosti R podržava 9 veza;
  - nova aplikacija pita za 1 TCP vezu, a dobija propusnost od R/10
  - Nova aplikacija pita za 11 novih TCP veza, i dobija više od R/2!