Ivan Brčić i Neven Kmetić

PROGRAMSKI ALATI NA UNIX RAČUNALIMA

SEMINARSKI RAD

TCP/IP API

Zagreb, prosinac 2004.

Sadržaj

Popis tablica

Popis slika

Popis primjera

1. Uvod

1.1. Pregled TCP/IP protokola

1.1.1. TCP/IP i Internet

1969. godine ARPA, *Advanced Research Projects Agency*, osnovala je istraživčki i izvedbeni projek izrade eksperimentalne "packet-switching" mreže. Mreža, nazvana ARPAnet, bila je napravljena zbog proučavanja tehnika slanja robusnih i pouzdanih podataka. Mnoge tehnike modernih podatkovnih komunikacija razvijene su sa ARPAnet eksperimentalnom mrežom.

Eksperimentalna mreža bila je toliko uspješna da su je mnoge organizacije, povezane na nju, počele koristiti za svakodnevnu komunikaciju. 1975. godine ARPAnet je iz eksperimentalne postala vrlo razvijena "operativna" mreža i odgovornost administriranja je preuzela DCA (*Defense Communications Agency*) koja je kasnije promijenila ime u DISA (*Defense Information System Agency*).

TCP/IP protokol usvojen je kao vojni standard (MIL STD) 1983. godine, i svi "hostovi" spojeni na mrežu bili su primorani prijeći na novi protokol. Da bi olakšala prijelaz, DARPA[1], je zatražila Bolta, Beraneka i Newmana (BBN) da implementiraju TCP/IP protokol u Berkley (BSD) Unixu. Tako je započeao brak Unixa i TCP/IP-a.

Kako je s vremenom prihvaćen TCP/IP standard, termin *Internet* se počeo koristiti u svakodnevnoj uporabi. 1983. ARPAnet se podijelila na MILNET i novu manju ARPAnet. Internet termin se koristio za cijelokupnu mrežu: MILNET plus ARPAnet.

1.1.2. Podatkovni komunikacijski modeli

Arhitektonski model razvijen od *International Standards Organization* (ISO) je konstantno korišten za opisivanje strukture i funkcije podatkovnih komunikacijskih protokola. Ovaj model, nazvan *Open System Interconnect (OSI) Reference model*, predstavlja osnovnu referencu za promatranje komunikacija. Termini definirani ovim modelom su vrlo razumljivi i naveliko korišteni u komunikacijskoj organizaciji pa je nemoguće govoriti o podatkovnoj komunikaciji bez korištenja OSI terminologije.

[1] tijekom 1980. ARPA, kao dio *U.S. Department of Defense*, postala je DARPA (*Defense Advanced Research Projects Agency*) ali agencija i njena misija razvijanja naprednog istraživanja ostala je ista

OSI model sadrži sedam slojeva (*layers*) koji definiraju funkcije komunikacijskih protokola. Svaki sloj OSI modela predstavlja funkciju koja se "izvodi" kada se podaci transferiraju kroz kooperativne aplikacije i mreže. Slika 1.1. prikazuje svaki sloj s imenom i sadrži kratak opis. Gledajući sliku, protokoli su kao "blokovi" stavljeni jedan na drugi. Zbog toga se struktura često naziva *stack* ili *protocol stack*.

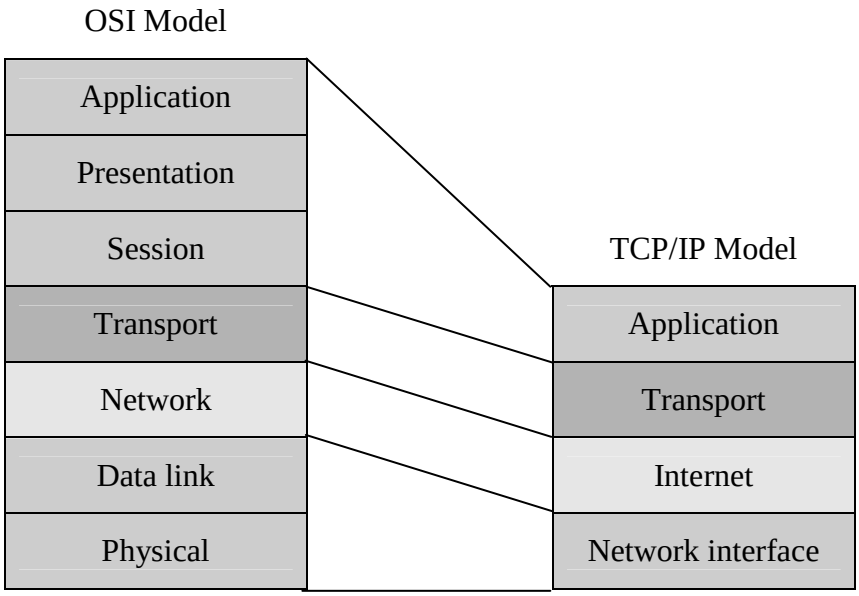
7. Aplikacijski sloj (APPLICATION) Mrežne primjene poput SMTP, HTTP, FTP i sl.
6. Prezentacijski sloj (PRESENTATION) Formatiranje podataka i zaštita
5. Sjednički sloj (SESSION) Uspostavljanje i održavanje sesija
4. Prijenosni sloj (TRANSPORT) Osiguranje prijenosa s kraja na kraj
3. Mrežni sloj (NETWORK) Isporuka jedinica informacije, uključujući usmjeravanje (routing)
2. Podatkovni sloj (DATA LINK) Prijenos jedinica informacije s provjerom greške
1. Fizički sloj (PHYSICAL) Prijenos binarnih podataka kroz medij

Slika 1.1. OSI model

Sloj ne definira jedinstven protokol. Svaki sloj definira podatkovne komunikacijske funkcije koje mogu biti korištene od bilo kojeg broja protokola. Dalje, svaki sloj može sadržavati više protokola gdje svaki provodi uslugu odgovarajući funkciji sloja. Na primjer, protokol za prijenos podataka (FTP) i elektronička pošta (SMTP) provode korisnikčke usluge i dio su aplikacijskog sloja (*Application Layer*).

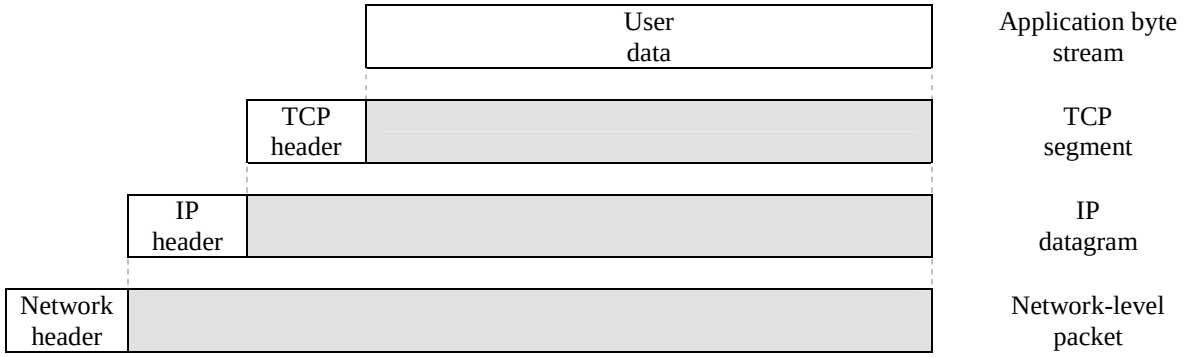
1.2. Arhitektura TCP/IP protokola

Generalno gledajući TCP/IP protokol je sastavljen od manjeg broja slojeva od onih korištenih u OSI modelu. Mnogi opisi TCP/IP definiraju sa tri do pet funkcionalnih slojeva arhitekture protokola. Četvero-slojni model prikazan na slici 1.2. (pokazuje korelaciju između OSI i TCP/IP modela). TCP/IP model je baziran na tri sloja (aplikacijski, prijenosni i mrežni) i dodatnim internet slojem.



Slika 1.2. Usporedba OSI i TCP/IP modela

Isto kao i u OSI modelu, podaci se šalju prema dolje niz *stack*, kada su poslani na mrežu, i gore uz *stack*, kada su primljeni sa mreže. Podaci se u TCP/IP-u šalju niz *protokol stack* iz aplikacijskog sloja sve do fizičke mreže. Svaki sloj u *stacku* dodaje kontrolnu informaciju da bi potvrdio sigurnu dostavu. Ova kontrolna informacija naziva se zaglavlje (*header*) jer je postavljen na početak podatka koji se treba poslati. Svaki sloj tretira sve informacije koje primi iz sloja iznad kao podatke i stavlja svoje zaglavlje na početak informacije. Kada su podaci primljeni, događa se suprotno, svaki sloj "skida" svoje zaglavlje prije nego pošalje podatke sloju iznad. Slika 1.3.



Slika 1.3.Prolaz podataka kroz slojeve

1.3. Networking na Linuxu

Linux kernel podržava mnogo različitih mrežnih arhitektura (TCP/IP je samo jedna od njih), s nekoliko implementiranih alternativnih algoritama za razvrstavanje mrežnih paketa, uključujući i programe koji olakšavaju sistemskim administratorima "postaviti" *router*, *gateway*, *firewall*, pa čak i jednostavan World Wide Web server, direktno na nivou kernela.

Trenutni kôd Net-4 je inspiriran orginalnom Berkley Unix implementacijom. Kako i samo ime govori, ovo je četvrta verzija *Linux networkinga*. Networking kôd je organiziran u slojeve (*layers*), gdje svaki od njih ima vrlo dobro definiran *interface*. Pošto podaci poslani mrežom nisu ponovo upotrebljivi, nema potreba za spremanjem u *cache*. Linux izbjegava kopiranje podataka unutar slojeva. Originalni podaci su spremljeni u memoriju (*memory buffer)*, koji je dovoljno velik da sadrži informacije potrebne svakom sloju.

1.4. Mrežne arhitekture

Mrežna arhitektura opisuje kako je pojedina računalna mreža napravljena. Arhitektura definira grupu slojeva gdje bi svaki trebao imati vrlo dobro definiranu namjenu. Programi u svakom sloju komuniciraju koristeći razmjenjive skupove pravila i konvencija (tzv. protokol).

Generalno gledajući, Linux podržava veliki broj različitih mrežnih arhitektura. Neke od njih su:

name	mrežna arhitektura i/ili protokol
PF_APPLETALK	Appletalk
PF_BLUETOOTH	Bluetooth
PF_BRIDGE	Multiprotocol bridge
PF_DECnet	DECnet
PF_INET	IPS's IPv4 protocol
PF_INET6	IPS's IPv6 protocol
PF_IPX	Novell IPX
PF_LOCAL, PF_UNIX	Unix domain socket (local communication)
PF_PACKET	IPS's IPv4/IPv6 protocol low-level access
PF_X.25	X25

Tablica 1.1 Podržane arhitekture pod Linux-om

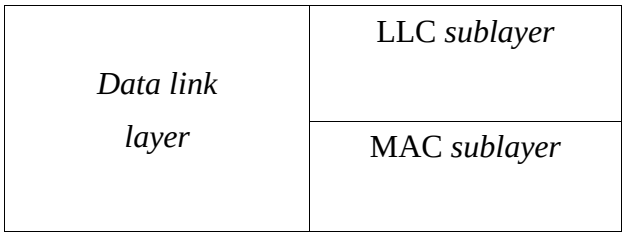
IPS (*Internet Protocol Suite*) je mrežna arhitektura Interneta, ponekad nazvana TCP/IP mrežna arhitektura zbog glavnih protokola.

1.5. Kratak opis etherneta

Ethernet je postao gotovo standard za lokalno umrežavanje. Postoji više standarda, a neki od njih su:

- 10baseT – koristi UTP kabel
- 100baseT – koristi UTP kabel
- 100baseFX – koristi optički kabel
- Gigabit ethernet (1000baseX) – postoji više verzija
    - 1000baseTX – koristi UTP kabel
    - 1000baseSX, 1000baseLX – koriste optički kabel

Data link sloj Ethernet protokola se sastoji od dva podsloja, *Media Access Control* sloja (MAC) i *Logical Link Control* sloja (LLC) kako je prikazano na slici.



Slika 1.4. Data link sloj Ethernet protokola

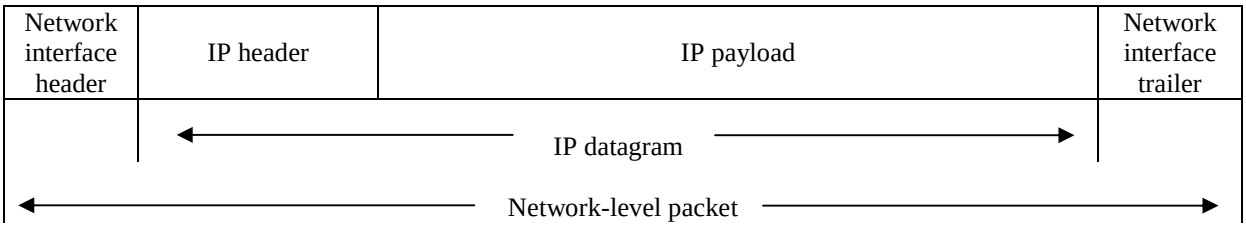
Podjelu je napravio Institute of Electrical and Electronics Engineers (IEEE). *Logical Link Control* (LLC) upravlja komunikacijom između mrežnih uređaja unutar jednog mrežnog spoja. LLC je definiran IEEE 802.2 specifikacijom te podržava spojne i bespojne servise. *Media Access Control* (MAC) upravlja pristupom mediju za prijenos podataka. IEEE MAC specifikacija definira jedinstvene MAC adrese tako da su mrežni uređaji jednoznačno identificirani na mreži. Svaka proizvedena mrežna kartica ima jedinstvenu MAC adresu koju je, djelomično, moguće softverski mijenjati.

Kada MAC sloj primi *transmit-frame* zahtjev za pripadajućom adresom i podacima iz LLC sloja, MAC počinje slati sekvencu (paket) na mrežu prebacivanjem LLC podataka u MAC *frame buffer*.

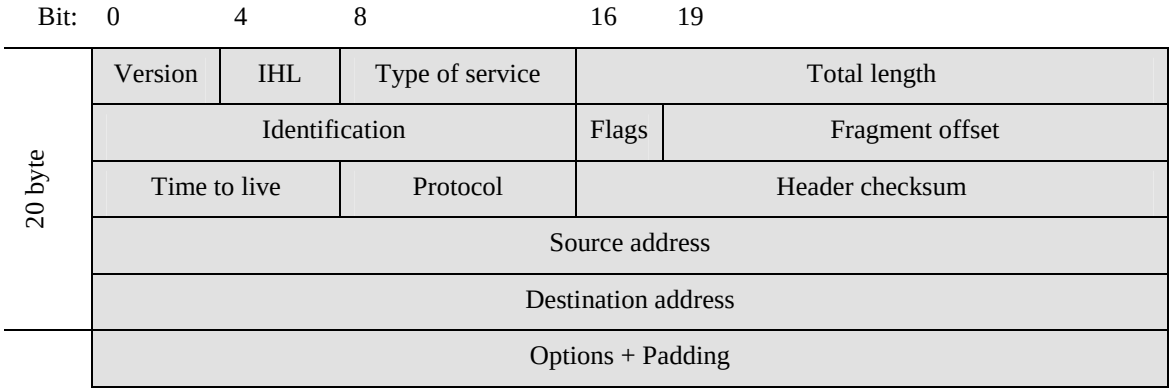
Svaka mrežna kartica prije slanja paketa na mrežu provjerava da li može poslati paket na mrežu ili treba čekati da se medij oslobodi. Protokol kojim se to obavlja se naziva Carrier Sense, Multiple Access with Collision Detection (CSMA/CD).

1.6. Internet protocol (IP)

Internet protocol (IP) se koristi za bespojnu komunikaciju između računala, a za prijenos ga koriste TCP i UDP. On je odgovoran za pravilno adresiranje računala i prosljeđivanje paketa, ali ne garantira njihovo dospjeće. IP može razbiti poruke na manje pakete te ih sastaviti na odredištu s time da svaki dio može krenuti drugim putem kroz mrežu. Ako paketi ne stignu pravim redosljedom IP protokol će ih sastaviti u ispravan paket.



Slika 1.5. IP datagram



Slika 1.6. IPv4 header

Version	4 bit	Verzija ip headera, trenutno se koristi verzija 4. Slijedeća verzija je Ipv6, koja se za sada koristi samo eksperimentalno
IP Header length	4 bit	Dužina IP headera (broj 32-bitnih riječi)
Type of service	8 bit	Kvaliteta veze
Total length	2 byte	Ukupna duljina IP paketa
Identification	2 byte	Identifikaciju specifičnog datagrama između pošiljatelja i primetelja. Za svaki sljedeći IP pakeet vrijednost se inkrementira
Flags	3 bit	Da li je paket pogodan za fragmentaciju, te da li je fragmentiran

Fragment offset	13 bit	Mjesto gdje se fragment nalazi u cijeloj poruci (potrebno za sastavljanje poruke)
Time to live	1 byte	Vrijeme života paketa, tj. Preko koliko prijelaza paket može proći dok se ne odbaci. Svaki IP router prilikom prosljeđivanja paketa dekrementira njegovu vrijednost, a ako je 0, paket se dalje ne prosljeđuje
Protocol	1 byte	Koji protokol se nalazi iznad, tj. Čiji paket IP paket sadrži
Header checksum	2 byte	Osigurava integritet headera
Source address	4 byte	Adresa pošiljatelja poruke
Destination address	4 byte	Adresa primatelja poruke
Options and Padding		Dodatne opcije i dopuna paketa da bi njegova veličina bila djeljiva sa 32

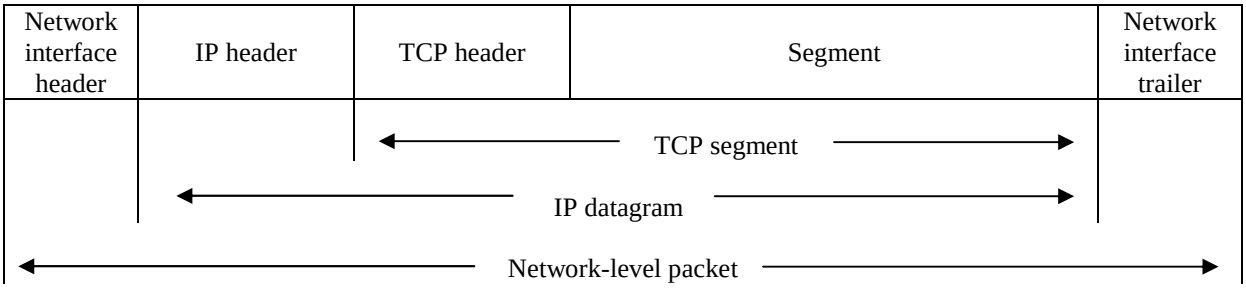
Tablica 1.2. Opis IPv4 header-a

IPv6 header:

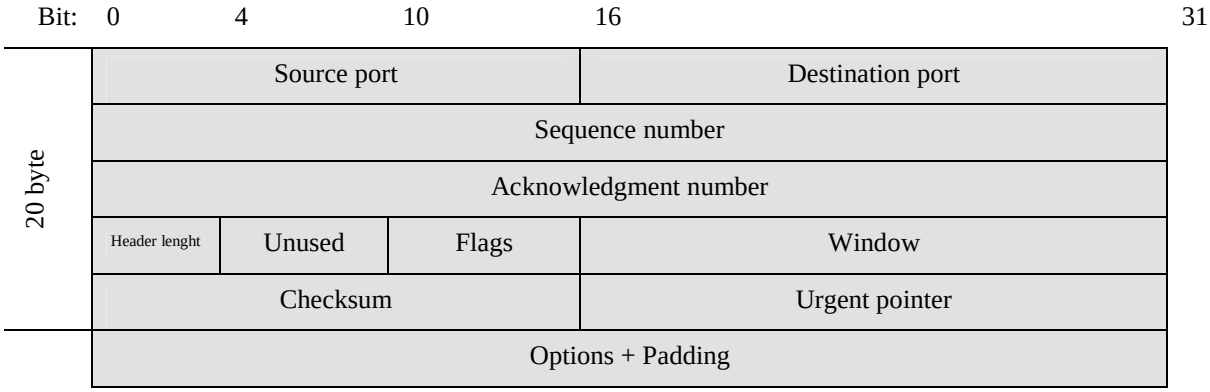
Slika 1.7. IPv6 header

1.7. TCP protokol

TCP je obostrani spojni protokol koji spaja 2 računala, te osigurava da paketi dolaze na odredište traženim redoslijedom. TCP koneckija može biti i unutar jednog računala (mnoge aplikacije komuniciraju na taj način, npr. X window system).



Slika 1.8. TCP segment



Slika 1.9. TCP header

Source port	2 byte	Socket s kojeg se šalje TCP segment, tj. Definira koja aplikacija na application layeru šalje segment, source socket zajedno s IP adresom jedinstveno definira mjesto sa kojeg je segment poslan
Destination port	2 byte	Socket na koji se šalje TCP segment, tj. Definira koja aplikacija na application layeru prima segment, source socket zajedno s IP adresom jedinstveno definira mjesto sa kojeg je segment poslan
Sequence number	4 byte	Redni broj početnog okteta segmenta
Acknowledgment number	4 byte	Broj slijedećeg okteta korisnikove poruke, ujedno i kumulativna potvrda
Data offset	4 byte	Početak podataka u paketu, ujedno označava veličinu TCP headera
Reserved	6 bit	Rezervirano za buduću uporabu (postavljeno na 0)
Flags	6 bit	URG, ACK, PSH, RST, SYN, FIN flagovi
Window	2 byte	Veličina buffera pošiljatelja poruke koji služi za primanje poruka.

		TCP/IP stack drugog sudionika treba slati pakete velike maksimalno do veličine prozora. Ako je poslana 0, poruke se dalje ne šalju, sve dok se ne pošalje nova poruka sa vrijednošću većom od 0.
Urgent pointer	2 byte	Lokacija hitnih podataka u segmentu
Options		Proizvoljne opcije koje se dodaju TCP headeru u paketima od 4 bytea

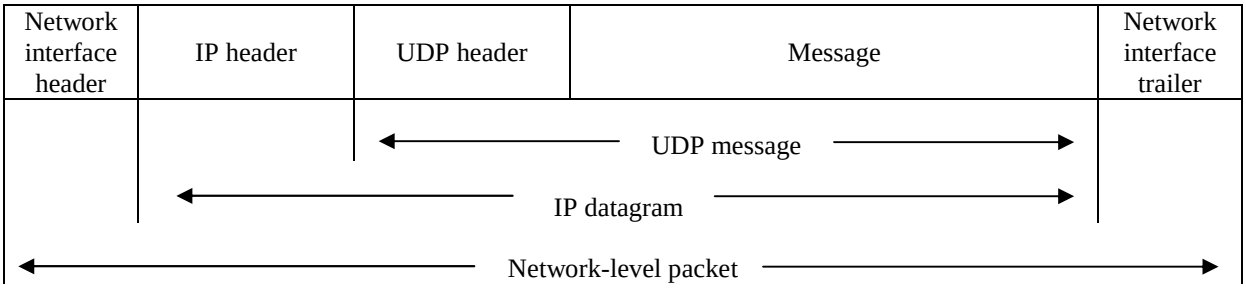
Tablica 1.3. Opis TCP header-a

20	FTP-data
21	FTP – File Transfer Protocol
22	SSH – Secure Shell
23	Telnet
25	SMTP – Simple Mail Transfer Protocol
80	HTTP – Hypertext Transfer Protocol
110	POP3 – Post Office Protocol version 3
119	NNTP – Network News Transfer Protocol
143	IMAP – Internet Message Access Protocol
194	IRC – Internet Relay Chat
389	LDAP – Lightweight Directory Access Protocol

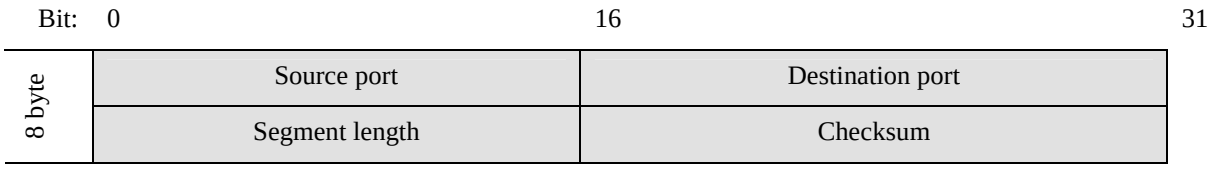
Tablica 1.4. Poznatiji TCP portovi

1.8. UDP protokol

UDP je bespojni protokol gdje nemamo informaciju da li je paket stigao ili nije. UDP samo prosljeđuje pakete protokolu iznad koji se mora brinuti za njihovo pravilno tumačenje. Kako nema overheada i odmah prosljeđuje pakete, znatno je brži od TCP-a.



Slika 1.10. UDP datagram



Slika 1.11. UDP header

Source port	2 byte	Port sa kojeg se šalje UDP poruka, tj. definira koja aplikacija na application layeru šalje poruku, polje je opcionalno te ako se ne koristi može biti 0
Destination port	2 byte	Port na koji stiže UDP poruka, tj. definira koja aplikacija na application layeru prima poruku, zajedno s IP adresom iz IP headera jedinstveno označava kojem procesu tj. aplikaciji je poruka poslana
Length	2 byte	Minimalna duljina je 8 byte-ova a teoretska maksimalna 65,515 (duljina ip paketa minus headeri). Prava maksimalna duljina je ograničena MTU-om linka preko kojeg se prenose paketi. Ovo polje je redudantno zato što se duljina paketa uvijek može izračunati iz duljine IP paketa
Checksum	2 byte	Polje koje osigurava integritet podataka

Tablica 1.5. Opis UDP header-a

Poznatiji UDP portovi:

- **53** DNS – Domain Name System
- **67** BOOTP client
- **68** BOOTP server (DHCP) – Dynamic Host Configuration Protocol

69	TFTP – Trivial File Transfer Protocol
137	NetBIOS Name Service
138	NetBIOS Datagram Service
161	SNMP – Simple Network Management Protocol
119	NNTP – Network News Transfer Protocol
520	RIP – Routing Information Protocol
1812, 1813	RADIUS – Remote Authentication Dial-In User Service

Tablica 1.6. Poznatiji UDP portovi

UDP (User data protocol) je niži nivo protokola od TCP-a. UDP ne garantira dostavu paketa na odredište. Također UDP ne garantira dostavu paketa u originalnom poretku, odnosno u onom poretku kojim su bili poslani. Korištenjem UDP socket-a naši programi mogu postati znatno složeniji nego što to mogu biti TCP socket-i. Evo savjeta i pod kojim uvjetima treba koristiti UDP socket-e:

- podaci koje trebamo poslati odgovaraju jednom fizičkom UDP paketu. UDP paketi su veliki 8192 byte-a, ali zbog podataka za header, preostaje nam jos 5000 byte-a
- neki podaci se mogu izgubiti bez narušavanja integriteta sistema
- svaki izgubljeni paket se nemora ponovno slati

Najveća prednost UDP protokola nad TCP-om je ta što je puno brži, ali mu je sigurnost prijenosa, odnosno dostave paketa mana.

1.9. Aplikacijska razina

Aplikacijska razina (*Application layer*) definira protokol koji korisnik koristi da bi razmjenjivao njemu suvisle informacije. Npr. poslao mail, pogledao sadržaj neke web stranice, udaljeno se spojio na drugo računalo i sl.
Primjeri su Telnet, FTP, SMTP, NFS, DNS, HTTP i sl.

2. IP adresiranje, stvaranje podmreža i usmjeravanje

2.1. IP adresiranje

IP protokol zahtjeva da svako računalo, točnije mrežni interface, na mreži ima jedinstvenu IP adresu. IP adresa se sastoji od 4 bytea, a mi je pisemo dekatski, svaki byte posebno odvojeno točkama. Prvi dio IP adrese je mrežna adresa, a drugi dio je adresa računala, a odnos između njihovih duljina ovisi o klasi mreže i subnet maski.

2.1.1. Klase IP adresa

IP adrese su podjeljene u klase, ovisno o primjeni i veličini organizacije kojoj se dodjeljuju. Najčešće korištene su A, B i C. Ovim klasama su definirane duljine mrežne adrese i duljine adrese računala. Klasa A koristi prvih 8 bitova za adresiranje mreže (od kojih je MSB (most significant bit) postavljen na 0), a zadnjih 24 za adresiranje računala i dodjeljuje se uglavnom ogromnim organizacijama. Klasa B koristi 16 bitova za adresu mreže (MS bitovi postavljeni na 10) i 16 bitova za adresu računala i dodjeljuje se uglavnom velikim organizacijama i tvrtkama (ISP-ovi i slično). Klasa C koristi 24 bita za adresu mreže (MS bitovi postavljeni na 110), a 8 za adresu računala. Dodjeljuje se većini organizacija i tvrtki. Ukoliko postoji potreba za više od 254 ali znatno manja od 65.534 računala, bolje je zakupiti više adresa klase C nego jednu B jer time štedimo adrese. Klasa D ima 4 najznačajnija bita postavljena na 1110, dok klasa E 4 najznačajnija bita ima postavljena na 1111.

```
Class Addresses
A 0.x.x.x to 126.x.x.x
B 128.0.x.x to 191.255.x.x
C 192.0.0.x to 223.255.255.x
D 224.0.0.1 to 239.255.255.255
E 240.x.x.x to 255.255.255.255
```

Ako u IP adresi sve bitove adrese računala postavimo na 0, dobili smo adresu mreže, a ako ih sve postavimo na 1 dobili smo broadcast adresu.

Npr.

199.10.2.0 – adresa mreže

199.10.2.255 – broadcast adresa mreže

199.10.2.4– jedno od racunala na mreži

2.1.2. Loopback adrese

Za testiranje i mrežnu komunikaciju između aplikacija na istom računalu koristi se specijalna loopback mrežna adresa 127.0.0.0.
Bilo koja regularna adresa na toj mreži je adresa samog računala, najčešće se koristi 127.0.0.1.

2.1.3. Privatne IP adrese

Za mreže koje se ne planiraju spajati na internet, ili se planiraju spajati na internet pomoću jednog gatewaya koji koristi NAT (Network Adress Translation, poznato jos kao i IP Masquarade, IP adresa se u tom slućaju maskira IP adresom gatewaya) predviđene su privatne IP adrese. Zapravo, za lokalnu mrežu mozemo koristiti bilo koje adrese, ali bolje je koristiti privatne da ne bi imali problema sa slučajnim routanjem tih paketa prema van, dok se privatne IP adrese teoretski ne routaju na Internetu.

Za svaku klasu imamo set privatnih mrežnih adresa:

```
A 10.0.0.0
B 172.16.0.0 - 172.31.0.0
C 192.168.0.0 - 192.168.255.0
```

2.2. Podmreže i mrežne maske

Pošto na istu mrežu nije moguće spojiti mnogo računala (nemogućnost hardverske izvedbe, loše performanse) velike mreže dijelimo na podmreže (*subnet*) pomoću subnet maski. Pomoću subnet maske određujemo (neovisno o klasi mreže) koji dio adrese usmjerivač (*router*) smatra za mrežnu, a koji dio za adresu računala, tj. prema njoj zna koje adrese usmjerava prema van, a koje zadržava na lokalnoj mreži.

Subnet maska je jednake velčcine kao IP adresa, tj. 32 bita. Adresa mreže je označena sa bitom 1, a adresa računala sa bitom 0, tako da ako recimo klasu B želimo podijeliti na 256 mreža dobijemo subnet masku 255.255.255.0.

Kako smo došli do toga:

`N = network`

`H = host`

Ovako izgleda IP adresa B klase:

`NNNNNNNN.NNNNNNNN.HHHHHHHH.HHHHHHHH`

Trenutna subnet maska je:

`11111111.11111111.00000000.00000000`

Kako je nama potrebno 256 mreža C klase, subnet maska će sad izgledati ovako:

`11111111.11111111.11111111.00000000` ili 255.255.255.0

Primjer 2:

Ako od adrese računala u klasi C uzmemo 3 bita možemo kreirati 8 mreža po 30 članova.

Subnet maska nam je:

`11111111.11111111.11111111.11100000`

Kada to prevedemo u standardni oblik dobijemo:

`255.255.255.224`

Sad imamo 8 mreža sa slijedećim adresama:

Network	Hosts	Broadcast address
192.168.1.0	192.168.1.1 – 192.168.1.30	192.168.1.31
192.168.1.32	192.168.1.33 – 192.168.1.62	192.168.1.63
192.168.1.64	192.168.1.65 – 192.168.1.94	192.168.1.95
192.168.1.96	192.168.1.97 – 192.168.1.126	192.168.1.127
192.168.1.128	192.168.1.129 – 192.168.1.158	192.168.1.159
192.168.1.160	192.168.1.161 – 192.168.1.190	192.168.1.191
192.168.1.192	192.168.1.193 – 192.168.1.222	192.168.1.223
192.168.1.224	192.168.1.225 – 192.168.1.254	192.168.1.255

2.2.1. Usmjeravanje (Routing)

Kako fizički nije moguće napraviti jednu veliku mrežu na koju će se spojiti sva računala, koriste se usmjerivači (*router*). Usmjerivač je uređjaj koji ovisno o svojoj routing tablici pakete sa jednog mrežnog sučelja proslijeđuje na drugo sučelje. Zapravo većina operativih sustava unutar samog TCP/IP stacka ima implementiranu osnovnu verziju usmjerivača kojeg onda konfiguriramo tako da mu upišemo adresu gatewaya (routera). U tom slučaju sve pakete za koje računalo ne zna gdje bi poslalo šalje na gateway. Napredniji operativni sustavi (svi UNIXoidi, noviji Windowsi) imaju integriran pravi usmjerivač. Kako na velikim mrežama postoji više puteva od jednog do drugog mjesta, tj. više IP adresa, usmjerivači tu uglavnom koriste neki od protokola za dinamičko usmjeravanje (rutanje).

Primjer routing tablice na Linux racunalu:

```
user@localhost:/$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.0.0     *               255.255.255.0   U     0      0        0 eth0
169.254.0.0     *               255.255.0.0     U     0      0        0 eth0
127.0.0.0       *               255.0.0.0       U     0      0        0 lo
0.0.0.0         192.168.0.1     0.0.0.0         UG    0      0        0 eth0
user@localhost:/$
```

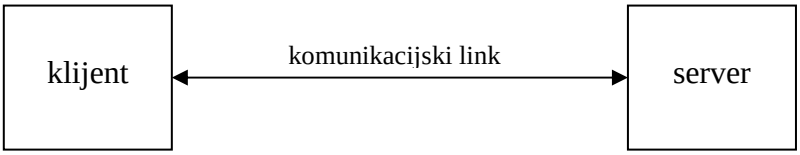
Primjer routing tablice na Windows XP računalu:

```
C:\>route print
===========================================================================
Interface List
0x1 ........................... MS TCP Loopback interface
0x10003 ...00 50 ba b9 03 54 ...... Realtek RTL8139 Family PCI Fast Ethernet NIC
===========================================================================
===========================================================================
Active Routes:
Network Destination        Netmask          Gateway       Interface  Metric
          0.0.0.0          0.0.0.0         10.0.0.1       10.0.0.11       20
         10.0.0.0    255.255.255.0        10.0.0.11       10.0.0.11       20
        10.0.0.11  255.255.255.255        127.0.0.1       127.0.0.1       20
   10.255.255.255  255.255.255.255        10.0.0.11       10.0.0.11       20
        127.0.0.0        255.0.0.0        127.0.0.1       127.0.0.1       1
        224.0.0.0        240.0.0.0        10.0.0.11       10.0.0.11       20
  255.255.255.255  255.255.255.255        10.0.0.11       10.0.0.11       1
Default Gateway:          10.0.0.1
===========================================================================
Persistent Routes:
  None
C:\>
```

3. Mrežno programiranje

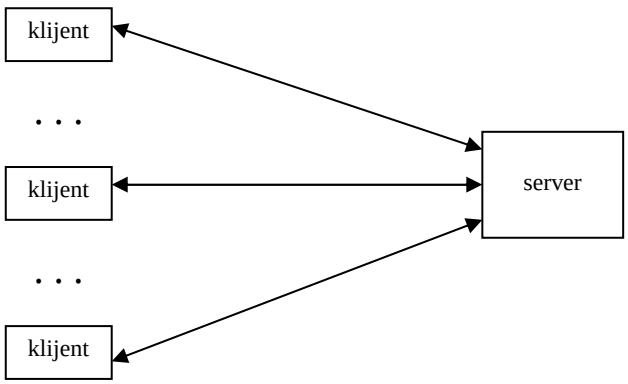
Mrežno programiranje obuhvaća pisanje programa koji komuniciraju sa drugim programima preko računalne mreže. Jedan program se uobičajeno naziva *klijent*, a drugi *server*.
Primjer klijent-server komunikacije je Web browser kao klijent i Web server kao server.

Većina mrežnih aplikacija se može podijeliti u dvije skupine: klijent i server.



Slika 3.1. Mrežna aplikacija: klijent i server

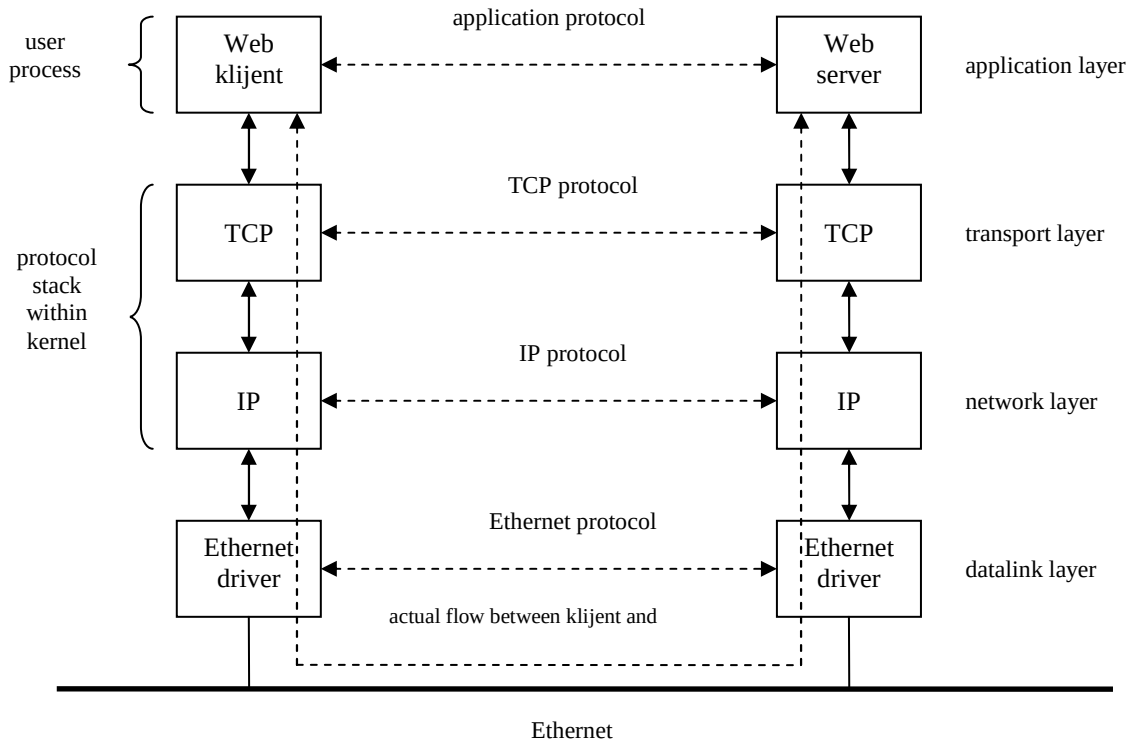
Klijenti (klijents) uobičajeno komuniciraju sa jednim serverom u isto vrijeme, premda koristimo Web browser kao primjer, možemo komunicirati sa više različitih servera. Sa serverske strane, sasvim je uobičajeno da server u jednom trenutku ima uspostavljenu vezu (komunicira) sa više različitih klijenata.



Slika 3.2. Upravljanje servera sa više klijenata istovremeno

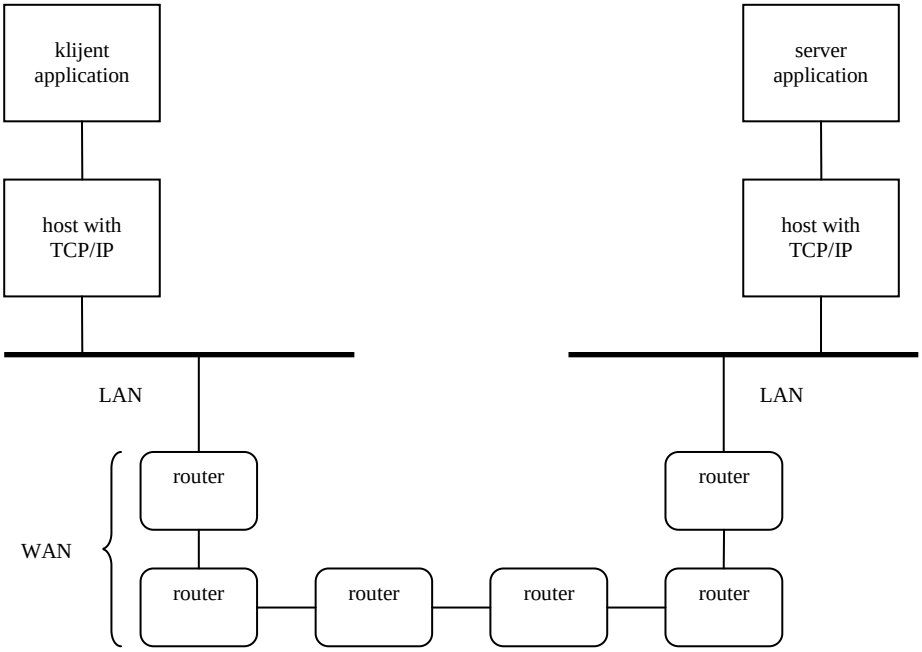
Klijent – server mogu se nalaziti u istoj lokalnoj mreži, na primjer Ethernet, ali također mogu biti i u različitim mrežama, što je u većini situacija slučaj, povezanim preko wide area network-a (WAN) koristeći routere.

Slike 1.3. i 1.4. prikazuju klijent – server vezu unutar iste mreže (sl. 1.3.) i klijent – server vezu kada se nalaze u različitim mrežama.



Slika 3.3. klijent i server u istom Ethernet, komuniciraju koristeći TCP

Bitno je napomenuti, iako izgleda da klijent i server komuniciraju koristeći application layer, transport layer komunicira koristeći TCP itd., da u stvarnosti to nije tako, već se sve spušta na najnižu razinu, a zatim se vrši prijenos podataka od najniže razine do odgovarajuće razine.



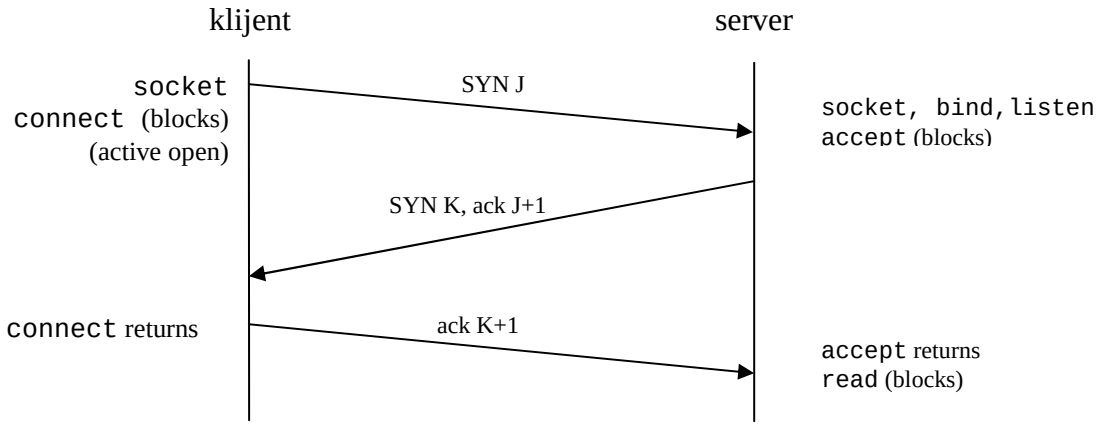
Slika 3.4. klijent i server u različitim LAN mrežama povezani preko WAN-a

3.1. Uspostavljanje i raskidanje veze

3.1.1. Uspostavljanje veze (three – way handshake)

1. server mora biti spreman prihvatiti nadolazeću vezu. To se uobičejeno ostvaruje pozivanjem `socket`, `bind` i `listen` funkcija i naziva se *passive open*.
2. klijent započinje aktivnim otvaranjem (*active open*) pozivanjem `connect` funkcije. To uzrokuje da TCP klijent šalje SYN segment (koji služi za sinhronizaciju) da se kaže serveru da je klijent inicijalizirao broj sekvence za podatke koji će se slati preko uspostavljene veze. Uobičajeno se ne šalju podaci kada se šalje SYN: on samo sadrži IP header, TCP header i moguće TCP opcije.
3. server mora potvrditi klijent-ov SYN, i server mora također poslati svoj SYN sadržavajući inicijalizirani broj sekvence za podatke koje će server slati na vezu. Server šalje svoj SYN i ACK na klijent-ov SYN u jednom segmentu.
4. klijent mora potvrditi serverov SYN

Minimalan broj potvrdnih paketa za ovu izmjenu je tri i zbog toga se uspostava TCP veze i naziva three – way handshake.
Slika 2.1. pokazuje opisan postupak.

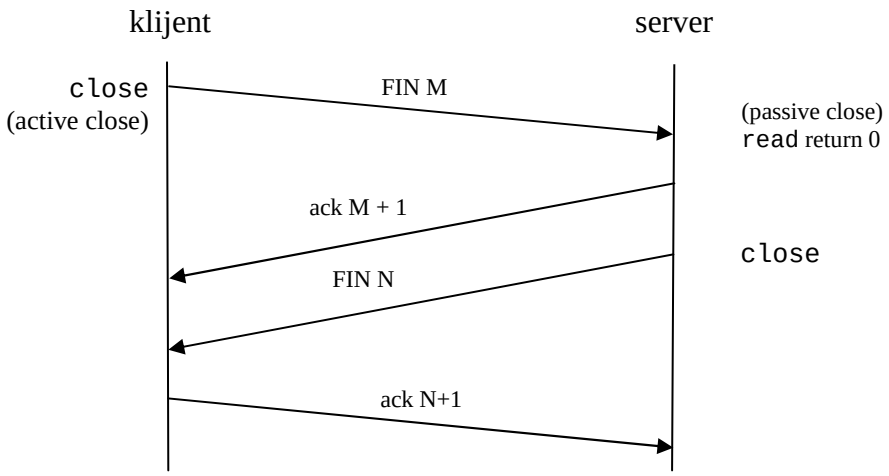


Slika 3.5. TCP three – way handshake

3.1.2. Raskidanje veze

Dok je za uspostavljanje veze bila potrebna tri segmenta, za raskid veze su potrebna četiri segmenta.

1. Jedna aplikacija prva zove `close`, kaže se da je kraj izvođenja *active close*. Ta strana šalje FIN segment, što znači da je završeno slanje podataka.
2. Druga strana koja primi FIN izvodi *passive close*. Primljeni FIN je potvrđen od TCP-a. Primitak FIN-a se također prosljeđuje aplikaciji kao end-of-file (EOF) (nakon svakog podatka koji je možda već u redu da ju primi aplikacija), budući da primitak FIN-a znači da aplikacija neće primit nit jedan suvišan podatak preko veze.
3. Malo kasnije aplikacija koja primi end-of-file će zatvoriti (`close`) socket. To uzrokuje da TCP pošalje FIN.
4. TCP na sistemu koji je primio taj konačni FIN (kraj sa *active close*) potvrđuje FIN.



Slika 3.6. Izmjena paketa prilikom raskida TCP veze

3.2. Socketi

Socket se definira kao jedan kraj komunikacijskog kanala. Za komunikaciju para procesa potreban je par socketa – za svaki proces po jedan. Socket je određen IP adresom i brojem port-a. Uglavnom socketi se koriste u klijent-server arhitekturi. Server čeka zahtjeve klijenata tako da "sluša" na određenom portu socketa.

Serveri obično implementiraju specifične servise na određenom portu(telnet-23, ftp-21, http-80). Svi brojevi portova ispod 1024 su uglavnom korišteni od strane poznatih aplikacija.

Kad se klijent spoji na socket dodjeljuje mu se broj porta koji je veći od 1024 npr 1625. Kad se još jedan klijent spoji na server aplikaciju njemu se dodjeljuje broj porta veći od 1024, ali različit od 1625. Na taj način uvijek imamo jedinstveni par socketa za svaki par klijent-server.

Dvije vrste socketa su najkorištenije:

- stream sockets
- datagram sockets

Stream socketi su pouzdani dvosmjerni komunikacijski kanal. Sve što se upisuje s jedne strane socketa izlazi na drugoj strani u istom obliku. Ako upišete string '1,2,3' na izlazu ćete pročitati '1,2,3'. Također sve moguće greške prilikom transporta paketa su ispravljene

tako da korisnik ima dojam da je socket "error free". Stream sockete koriste aplikacije poput **telnet**-a i web browsera koji koriste **http** protokol.
Datagram sockets se još nazivaju i connectionless. Nisu toliko pouzdani jer datagram koji je poslan može i ne mora stići, ne moraju stići istim redoslijedom kojim su poslani ali ako stigne biti će ispravan. Datagram socketi koriste IP ali ne i TCP. Nazivaju se connectionless jer se ne mora drugi kraj držati otvoren cijelo vrijeme nego se samo šalju paketi sa odredišnom adresom. Koriste ga aplikacije poput **tftp** i **bootp.**

Internet programi kao što su Telnet, rlogin, FTP, talk, i World Wide Web koriste socket za komunikaciju. Na primjer, možemo doći do www stranice preko Web servera koristeći socket za mrežnu komunikaciju. Da bi uspostavili vezu sa www serverom koristeći Telnet program, neka je stranica na koju se spajamo *www.imestranice.com*, tada ćemo prilikom spajanja navesti i port koji označava da se spajamo na www server, a on je 80. Port 80 je standardni port dodijenjen od IANA-e na kojem 'sluša' (listen) web server.

Primjer: *Dohvaćanje HTML-sourca pomću Telnet programa*

```
% telnet www.imestranice.com 80
    Trying 206.168.99.1...
    Connected to oznaka.imestranice.com (206.168.99.1)
   Escape character is '^]'.
   GET /
   <html>
   <head>
   <meta http-equiv = ″Content – Type″ content = ″text/html; charset – iso – 8859 – 1″>
   . . .
```

3.2.1. BSD socketi

Generalno gledajući, svaki operacijski sustav mora definirati odgovarajući *Application Programming Interface* (*API)* između korisničkog programa i “mrežnog kôda”. Linux mrežni API je baziran na *BSD socketima*. Predstavljeni su u Berkley UNIX 4.1cBSD i dostupni su u gotovo svakom operacijskom sustavu sličnom Unixu.

Socket je dvosmjerna komunikacijska jedinica koja može biti korištena za komuniciranje sa drugim procesima na istom računalu ili sa procesom pokrenutim na drugom računalu. Internet programi kao Telnet, rlogin, talk ili World Wide Web koriste sockete. Socket je komunikacija između tzv. ulaznih vrata i krajnje točke kanala koji povezuje dva procesa. Podaci se pošalju na “ulazna vrata” i nakon nekog vremena pojave se na drugoj

strani (izlazu). Komunikacijski procesi mogu biti na različitim računalima. Mrežni kôd Kernela prosljeđuje podatke između krajnjih točaka.

Linux implementira BSD sockete kao datoteke koji pripadaju u *sockfs special filesystem*. Preciznije, za svaki novi BSD socket, Kernel kreira novi *inode* u *sockfs* filesistemu. Atributi BSD socketa su spremljeni u `socket` strukturi podataka koja je objekt uključen u polju `u.socket_i` *sockfs inode-a*.

Metode BSD socket objekta	
metoda	**opis**
`release`	zatvara socket
`bind`	dodjeljuje lokalnu adresu (ime)
`connect`	uspostavlja vezu (TCP) ili dodjeljuje *remote adress* (UDP)
`socketpair`	kreira par socketa za dvosmjernu razmjenu podataka
`accept`	čeka zahtjev za spajanje
`getname`	vraća lokalnu adresu
`ioctl`	implementira `ioctl()`komande
`listen`	inicijalizira socket da prihvati zahtjev za spajanjem
`shutdown`	zatvara pola ili obje polovice *full-duplex* veze
`setsockopt`	postavlja vrijednosti socket flagovima
`getsockopt`	uzima vrijednosti socket flagova
`sendmsg`	šalje paket socketu
`recvmsg`	prima podatak od socketa
`mmap`	*file memory-mapping* (ne koriste mrežni socketi)
`sendpage`	kopira podatke direktno iz/prema fajlu (`sendfile()` sistemski poziv)

Tablica 3.1. Metode BSD socket objekta

3.2.2. INET socketi

INET socketi su strukture podataka tipa `struct sock`. Svaki BSD socket koji pripada IPS mrežnoj arhitekturi sadrži adrese INET socketa u `sk` polju `socket` objekta.

INET socketi su potrebni jer `socket` objekti (opisujući BSD sockete) uključuju samo polja koja su značajna za sve mrežne arhitekture. Ali, kernel mora također zapamtiti i nekoliko drugih informacija za svaki socket bilo koje mrežne arhitekture. Na primjer, u svakom INET socketu, kernel zapisuje lokalne i udaljene IP adrese, lokalne i udaljene brojeve portova, relativni transportni protokol, podatke o primljenim paketima, podatke o paketima koji čekaju da budu poslani socketu, te nekoliko tablica metoda za slanje paketa do socketa. Ovi atributi su spremljeni, zajedno sa mnogim drugima, u INET socketima.

INET socket objekt također definira neke metode specifične tipu transporta protokola (TCP ili UDP). Metode spremljene u strukturi podataka tipa `proto`:

metode INET socket objekta	
metoda	**opis**
`close`	zatvara socket
`connect`	uspostavlja vezu ili dodjeljuje *remote adress*
`disconnect`	prekida uspostavljenu vezu
`accept`	čeka zahtjev za spajanje
`ioct1`	implementira `ioctl()`komande
`init`	INET socket objekt konstruktor
`destroy`	INET socket objekt destruktor
`shutdown`	zatvara pola ili obje polovice *full-duplex* veze
`setsockopt`	postavlja vrijednosti socket flagovima
`getsockopt`	uzima vrijednosti socket flagova
`sendmsg`	šalje paket socketu
`recvmsg`	prima podatak od socketa
`bind`	dodjeljuje lokalnu adresu (ime)
`backlog_rcv`	funkcija pozvana kada prima paket
`hash`	doda INET socket *per-protocol hash* tablici
`unhash`	makne INET socket iz *per-protocol hash* tablice
`get_port`	dodjeljuje broj porta INET socketu

Tablica 3.2. Metode INET socket objekta

`sock` objekt sadrži otprilike 80 polja od kojih su mnoga pokazivači na druge objekte, tablice, metode ili druge strukture podataka.

3.3. Koncepti socketa

Kada kreiramo socket moramo definirati tri parametra:

- način komuniciranja (*communication style*),
- namespace,
- i protokol.

3.3.1. Communication style

Način komuniciranja kontrolira kako socket tretira poslane podatke i specificira broj komunikacijskih partnera. Kada su podaci poslani kroz socket, raspakirani su u dijelove nazvane paketima. Stil komuniciranja određuje kako su ti paketi podržani i kako su adresirani od pošiljaoca do primatelja.

- *Connection style* garantira isporuku svih paketa po redu kako su poslani. Ako su paketi izgubljeni ili je promijenjen redoslijed zbog problema na mreži, primatelj automatski zahtjeva njihovo ponovno slanje od pošiljaoca. *Connection-style socket* je kao telefonski poziv: adresa pošiljatelja i primatelja je fiksna na početku komunikacije kada je veza uspostavljena.
- *Datagram style* ne garantira isporuku ili redoslijed dolaska paketa. Paketi mogu biti izgubljeni ili promijenjenog redoslijeda zbog problema na mreži. Svaki paket mora biti označen sa svojom destinacijom i nije zagarantirano da će biti isporučen. Sustav garantira samo “najbolju volju”, tako paketi mogu nestati ili stići u krivom redoslijedu u odnosu na to kako su poslani. *Datagram-style sockets* način slanja je moguće usporediti sa slanjem pisma poštom: pošiljatelj specificira adresu primatelja za svaku individualnu poruku.

3.3.2. Namespace

Socket *namespace* određuje kako su zapisane adrese socketa. Adresa socketa određuje jedan kraj “veze” socketa. Na primjer, adrese socketa u “*local namespaceu*” su obična imena datoteka. U “*Internet namespaceu*” adresa socketa je sastavljena od Internet adrese (*Internet Protocol adress* ili *IP address*) računala spojenog na mrežu i broj porta. Broj porta raspoznaje višestruke sockete na istom računalu.

3.3.3. Protokol

Protocol specificira kakvim socketom se podaci prenose.
Neki protokoli su TCP/IP, primarni mrežni protokol korišten kod Interneta; Apple Talk mrežni protokol; UNIX lokalni komunikacijski protokol.

3.4. Socket buffer

Svaki pojedini paket poslan kroz mrežnu jedinicu sastoji se od nekoliko dijelova. Svi slojevi mreže, dodaju neke kontrolne informacije na osnovni *payload*, tj. podatak koji se prenosi putem mreže. Format paketa se mijenja prolaskom kroz TCP/IP slojeve, a prikazan je na slici 1.3..

Cijeli paket je napravljen u nekoliko dijelova korištenjem različitih funkcija. Na primjer, UDP/TCP zaglavlje i IP zaglavlje su sastavljeni od funkcija zavisno transportnom i mrežnom sloju IPS arhitekture, dok su zaglavlje i *trailera*, napisani odgovarajućom metodom specificiranom mrežnom karticom.

Mrežni kôd Linuxa sadrži svaki paket u velikom memorijskom području nazvanom *socket buffer*. Svaki socket buffer je definiran sa strukturom podatka tipa `sk_buff` koja sadrži, uz mnoge druge stvari, pokazivače na slijedeće strukture podataka:

- *socket buffer*
- *payload* – podaci korisnika (unutar socket buffera)
- *the data link trailer* (unutar socket buffera)
- *INET socketi* (`sock` object)
- deskriptor zaglavlja transportnog sloja
- deskriptor zaglavlja mrežnog sloja
- deskriptor *data link layer* zaglavlja
- *the destination cache entry* (`dest_entry` object)

Struktura podataka `sk_buff` sadrži ”identifikator” mrežnog protokola korišten za slanje podataka (*checksum field*) i vrijeme dolaska primljenih paketa.

Kernel izbjegava kopiranje podataka već jednostavno proslijeđuje `sk_buff` *descriptor pointer* svakom mrežnom sloju. Na primjer, kada priprema paket za slanje,

transportni sloj počinje kopirati *payload* iz korisničkog *buffera* na višu poziciju *socket buffera*; tada transportni sloj dodaje svoje TCP ili UDP zaglavlje prije *payloada*. Nakon toga, kontrola se predaje mrežnom sloju, koji prima *socket buffer descriptor* i dodaje IP zaglavlje prije transportnog zaglavlja. Konačno, *data link* sloj dodaje svoje zaglavlje i *trailer*, čime pripremi paket za slanje.

3.5. Serveri

Životni ciklus servera ovisi o kreiranju *connection-style* socketa, dodjelivši mu adresu na njegov socket, postavljajući poziv na `listen` koji omogućuje veze do socketa, postavljajući pozive na `accept` dolazećim vezama, i na kraju zatvarajući socket. Podaci nisu čitani i pisani direktno preko server socketa. Umjesto toga program svaki puta prima novu vezu; Linux kreira zasebne sockete za korištenje prijenosa podataka preko te veze.

U ovom odlomku ćemo objasniti `bind`, `listen` i `accept`. Adresa mora biti dodjeljena socketu servera koristeći `bind` tako da se klijent može spojiti na taj socket. Prvi argument `bind`-a je socket *file descriptor*. Drugi argument je pokazivač na adresnu strukturu socketa. Treći argument je dužina adresne strukture, u byte-ovima. Kada je adresa dodijeljena *connection-style* socketu, potrebno je pozvati `listen` čime će pokazati da je server. Prvi argument `listen` metode je socket *file descriptor*. Drugi argument određuje koliko je prihvaćenih veza u ”redu” (queue). Ako je ”red” pun, dodatna spajanje biti će odbijena. To ne ograničava ukupan broj veza koje server može obraditi. Ograničava jedino broj klijenata koji se pokušavaju spojiti a nisu još bili prihvaćeni.

Server prihvaća zahtjev za spajanjem od klijenta pokretanjem `accept`-a. Prvi argument `accept`-a je socket *file descriptor*. Drugi argument pokazuje na adresnu strukturu socketa koja je napunjena klijentovim adresama socketa. Treći argument je dužina adresne strukture socketa u byte-ovima. Server može koristiti klijentovu adresu da utvrdi hoće li stvarno komunicirati sa klijentom. Poziv za prihvaćanje kreira novi socket za komunikaciju sa klijentom i vraća odgovarajući *file descriptor*. Originalni server socket nastavlja primati nove klijentove veze. Za čitanje podataka iz socketa bez da ga se ukloni iz ulaznog ”reda”, koristimo `recv`. Koristimo iste argumente kao `read`, plus dodatni `FLAGS` argument. Flag MSG_PEEK omogućava čitanje podataka ali ne i micanje iz ulaznog ”reda”.

3.6. Lokalni socketi

Proces spajanja socketa na istom računalu može koristiti lokalni *namespace* prezentiran kao sinonim `PF_LOCAL` i `PF_UNIX`. Zovu se *lokalni socketi* ili *UNIX-domain socketi*. Njihove socket adrese, specificirane po imenima datoteka, su korištene samo kod kreiranja veza.

Socketima je ime specificirano u `struct sockaddr_un`. Moramo postaviti `sun_family` polje na `AF_LOCAL`, čime određujemo da se radi o lokalnom *namespaceu*. Polje `sun_path` određuje ime datoteke koja će se koristiti i može biti, najviše, 108 byte-ova dugačko. Može biti korišteno bilo koje ime, a proces mora imati prava ”pisanja” u direktoriju jer trebamo dodavati datoteke u direktorij. Da bi se spojili na socket, proces mora imati prava čitanja iz datoteke. Makar različita računala mogu dijeliti iste file-sisteme, samo procesi pokrenuti na istom računalu mogu komunicirati sa lokalnim *namespace socketima*.

Jedini dozvoljeni protokol za lokalni *namespace* je 0. Zbog toga što postoji u file-sistemu, lokalni socket se može izlistati kao i datoteka. Na primjer:

```
$ls –l /tmp/socket
srwxrwx—x   1 user       group      0  Nov 13 18:18  /tmp/socket
```

U dodatku C može se pogledati primjer komunikacije dva procesa preko socket-a.

3.7. IPv4 Socket adresna struktura

IPv4 socket adresna struktura, često nazivana ″Internet socket adresna struktura″, je nazvana `sockaddr_in` i definirana je u `<netinet/in.h>` header-u.

Posix definicija strukture je slijedeća:

```
struct in_addr {
      in_addr_t   a_addr;            /* 32 - bit IPv4 adress */
};                                   /* network byte ordered */

struct sockaddr_in {
      uint8_t           sin_len;     /* lenght of structure (16) */
      sa_family_t       sin_family;  /* AF_INET */
      in_port_t         sin_port;    /* 16-bit TCP or UDP port number */
                                     /* network byte ordered */
      struct in_addr    sin_addr;    /* 32-bit IPv4 adress */
                                     /* network byte ordered */
      char              sin_zero[8]; /* unused */
};
```

Socket adresna struktura se može prosljeđivati od procesa prema kernelu, i obratno, od kernela prema procesu. Četiri socket funkcije prosljeđuju socket adresnu strukturu od procesa prema kernelu, a to su: `bind`, `connect`, `sendto`, i `sendmsg`. Pet funkcija kojima kernel prosljeđuje socket adresnu strukturu su: `accept`, `recvfrom`, `recvmsg`, `getpeername`, i `getsockname`.

Socket adresna struktura se uvijek šalje po referenci, kada je prosljeđujemo kao argument bilo koje socket funkcije.

Kako bi se ostvarila generička socket adresna struktura, neovisna o protokolu bilo je potrebno osmisliti kako će se kroz jednu funkciju slati strukture različitih protokola. U ANSI C-u rješenje bi bilo vrlo jednostavno korištenjem `void *` (void pointera), koji je generički tip pointera. Ali kako su socket funkcije rađene prije nego što je donešen ANSI C, bilo je potrebno pronaći nekakvo rješenje, koje je izabrano 1982. godine kao generička socket adresna struktura i koja je definirana u `<sys/socket.h>` header-u a izgled strukture je slijedeći:

```
struct sockaddr {
      uint8_t           sa_len;
      sa_family_t       sa_family;        /* address family: AF_xxx value */
      char              sa_data[14];      /* protocol-specific address */
};
```

Socket funkcije su definirane tako da uzimaju pointer na generičku strukturu, kao što je prikazano u ANSI C prototipu funkcije za `bind` funkciju:

```
int bind (int, struct sockaddr *, socklen_t);
```

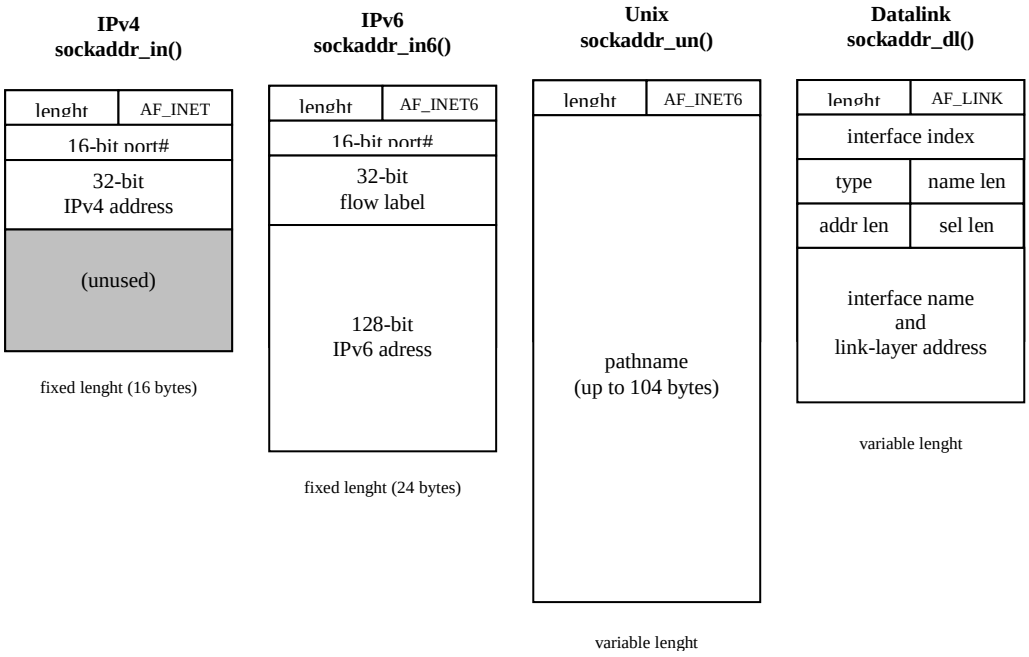
Ovo dređuje da za svaki poziv ove funkcije mora se cast-ati pointer socket adresne strukture specifičnog protokola na pointer generičke socket adresne strukture.

Na primjer,

```
int sockfd;
struct sockaddr_in serv;
      ...
bind (sockfd, (struct sockaddr *) &serv, sizeof(serv));
```

Ako izostavimo cast `(struct sockaddr *)` compiler generira upozorenje.

Slika 3.7. prikazuje izgled struktura `sockaddr_in` (IPv4), `sockaddr_un` (Unix), i `sockaddr_dl` (Datalink).



Slika 3.7. Usporedba različitih socket adresnih struktura

3.8. Osnovne funkcije za rad sa socket-ima (System Calls)

`socket` - kreiranje socket-a
`close` - uništavanje socket-a
`connect` - uspostavljanje veze između dva socket-a
`bind` - označavanje server socket-a sa adresom
`listen` - konfigurira socket za prihvaćanje uvjeta
`accept` - prhvaća vezu i kreira novi socket za komunikaciju

Socket je predstavljen file descriptor-om.

3.8.1. Pregled osnovnih funkicija

3.8.1.1. socket funkcija

Za izvršavanje mrežne komunikacije, prvu stvar koju proces mora napraviti je pozvati funkciju `socket`, odrediti tip komunikacijskog protokola (TCP koristeći Ipv4, UDP koristeći Ipv6, Unix protokol, itd.).

```
#include <sys/socket.h>

int socket (int family, int type, int protocol);
```

Returns: nonnegative descriptor if OK, -1 on error

family - određuje vrstu protokola
type - tip podataka koji se prenose socket-om (tip socket-a)
protocol - protokol

Za *family* argument koristimo jednu od ponuđenih konstanti:

family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols
AF_ROUTE	Routing sockets
AF_KEY	Key sockets

Tablica 3.3. Protokol family konstante za socket funkciju

Za *type* argument koristimo slijedeće konstante:

family	Description
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_RAW	raw socket

Tablica 3.4. tip socket-a za socket funkciju

Za *protocol* argument koristimo nulu, osim ako ne koristimo raw socket.

Nisu podržane sve kombinacije parametara. Tablica 3.3. prikazuje pregled kombinacija koje su podržane.

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP	TCP	Yes		
SOCK_DGRAM	UDP	UDP	Yes		
SOCK_RAW	IPv4	IPv6		Yes	Yes

Tablica 3.3. Podržane kombinacije family-a i type-a za socket funkciju

3.8.1.1.1. AF_xxx vs. PF_xxx

AF_ prefiks dolazi od ″address family″, a PF_ prefiks dolazi od ″protocol family″. U povijesti je bila intencija da jedna porodica protokola može podržavati više adresnih porodica i za to je korištena PF_ vrijednost za kreiranje socket-a, dok je AF_ vrijednost korištena u socket address strukturama. Trenutno je PF_ vrijednost postavljena na AF_, ali nema garancije da se to u budućnosti neće promijeniti.

3.8.1.2. connect funkcija

connect funkciju poziva TCP klijent da bi uspostavio vezu sa TCP serverom.

```
#include <sys/socket.h>

int connect (int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);

                                                    Returns: 0 if OK, -1 on error
```

sockfd je socket deskriptor koji nam vrati funkcija `socket`. Drugi i treći argument je pointer na socket address strukturu i veličina te strukture.
socket address mora sadržavati IP adresu i broj porta servera. Klijent ne mora zvati `bind` funkciju prije `connect` funkcije. Kernel će izabrati oboje, polazan port i izvor IP adrese ako je to potrebno.
U slučaju TCP socket-a, veza se inicijalizira u tri sinhronizacijska koraka (three – way handshake).

Ako se veza uspostavi funkcija vraća nulu, inače ako se dogodi error ili se veza ne može uspostaviti funkcija vraća -1.

3.8.1.3. bind funkcija

`bind` funkcija označava socket sa lokalnom adresom protokola.

```
#include <sys/socket.h>
int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
                                                    Returns: 0 if OK, -1 on error
```

myaddr argument je pointer na socket adresnu strukturu kojom se određuju osnovne informacije pojedinog protokola, a treći argument je veličina adresne strukture.
`bind` funkcija je zapravo dodjeljivanje adrese serveru. Adresa koja se postavi ovom funkcijom je ujedno i adresa na koju se klijenti spajaju kada žele uspostaviti vezu sa serverom.

3.8.1.4. listen funkcija

`listen` funkciju zove jedino server i izvršava dvije akcije:

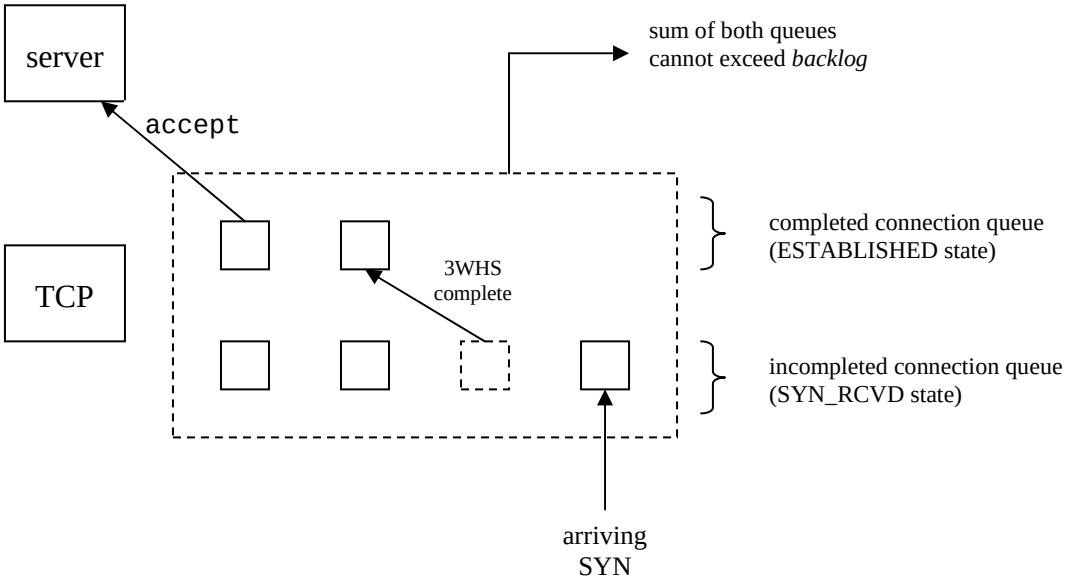
1. Kada je socket kreiran pomoću `socket` funkcije tada je predodređen da bude aktivan socket, to je, klijentov socket će pozvati `connect`. `listen` funkcija pretvara nepovezan socket u pasivan socket, naznačuje da kernel mora osluškivati ima li zahtjeva za uspostavljanjem veze i ako ima da ih poveže na taj socket. U TCP terminima to znači prelazak socket-a iz stanja CLOSED u stanje LISTEN.
2. Drugi argument funkcije određuje maksimalan broj veza koje kernel treba poredati za uspostavljanje vez za taj socket.

```
#include <sys/socket.h>
int listen (int sockfd, int backlog);
                                                    Returns: 0 if OK, -1 on error
```

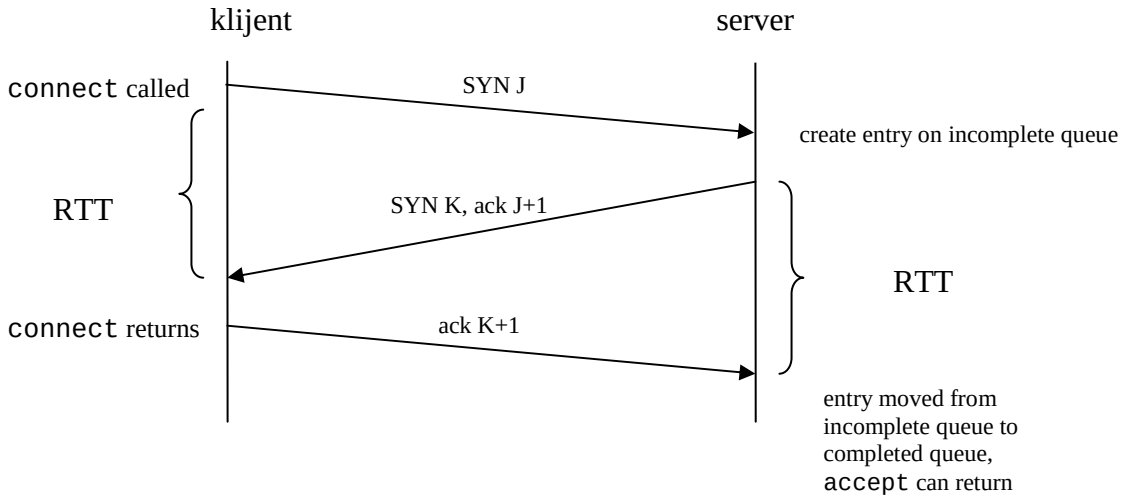
`listen` funkcije se uobičejeno zove nakon oba poziva `socket` i `bind` funkcija i mora biti pozvana prije poziva `accept` funkcije.

Kernel mora podržati dva reda:

1. *incomplete connection* queue, koji sadržava ulaz za svaki SYN koji dolazi od klijenta za koje server čeka da se kompletira TCP three – way handshake. Takvi socket-i su u SYN_RCVD stanju.
2. *completed connection* queue, koji sadržava ulaz za svakogklijenta sa kim je TCP three – way handshake završen. Ovakvi socket-i su ESTABLISHED stanju.



Slika 3.8. Dva reda podržana od TCP-a za listening socket



Slika 3.9. TCP three – way handshake i dva reda za listening socket

3.8.1.4.1. Koju vrijednost izabrati za *backlog*?

Kod odabira vrijednost *backlog-a* najbolje bi bilo provjeriti posljednju dokumentaciju pojedinog operacijskog sustava.

U tablici 3.4. nalaze se trenutne vrijednost za *backlog*.

	Maximum actual number of queued connections					
backlog	AIX 4.2, BSD/OS 3.0	Dunix 4.0, Linux 2.0.27, Uware 2.1.2	HP-UX 10.30	SunOS 4.1.4	Solaris 2.5.1	Solaris 2.6
0	1	0	1	1	1	1
1	2	1	1	2	2	3
2	4	2	3	4	3	4
3	5	3	4	5	4	6
4	7	4	6	7	5	7
5	8	5	7	8	6	9
6	10	6	9	8	7	10
7	11	7	10	8	8	12
8	13	8	12	8	9	13
9	14	9	13	8	10	15
10	16	10	15	8	11	16
11	17	11	16	8	12	18
12	19	12	18	8	13	19
13	20	13	18	8	14	21
14	22	14	19	8	15	22

Tablica 3.6. Trenutni broj veza za vrijednost backlog-a

3.8.1.5. accept funkcija

`accept` funkcija zove TCP server, prihvaća slijedeću završenu vezu iz reda completed connection queue. Ako je red završenih veza prazan, proces prelazi u sleep stanje (blocking socket). Kada server prihvati novu vezu, kreira se i novi socket preko kojeg se nastavlja komunikacija između servera i klijent-a.

```
#include <sys/socket.h>

int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

Returns: nonnegative descriptor if OK, -1 on error

3.8.1.6. close funkcija

`close` funkcija se koristi da bi se zatvorila i terminirala TCP veza.
Podrazumijevana akcija `close` funkcije je da označi socket kao zatvoren za dajlnju komunikaciju. Socket deskriptor više nije valjan i ne može se koristiti kao argument funkcija `read` i `write`. Jedina nezgodna stvar kada koristimo `close` funkciju je što će naša TCP veza i dalje pokušavati (neko određeno vrijeme) poslati odnosno primiti podatke. `close` funkcija se ponaša kao dekrementirajuća, dok se funkcija `accept` ponaša kao inkrementirajuća funkcija. Ako drugoj strani želimo odmah poslati FIN signal, kako bi joj dali do znanja da se veza raskida tada ćemo koristiti funkciju `shutdown`.

```
#include <sys/socket.h>

int close (int sockfd);
```

Returns: 0 if OK, -1 on error

4. Primjer korištenja socket-a za ostvarivanje komunikacije klijent – server

Osnovna ideja je slijedeća: server mora biti cijelo vrijeme u radu i ″osluškivati″ da li ima clienat-a koji se žele spojiti na server (kao čovjek na informacijama, čeka da zazvoni telefon). Ako postoji netko tko se želi spojiti na server, server mu to mora omogućiti (čovjek na informacijama podiže slušalicu i preusmjerava poziv na odgovarajuću liniju kojom će se vršiti komunikacija). Ako postoji još linija koje se žele spojiti, server će im to omogućiti samo u slučaju da nisu sve linije već zauzete, to je u našem slučaju broj koji smo naveli u *backlog* kod `listen` funkcije. Ako su sve linije zauzete i ne možemo prihvatiti niti jednu nadolazeću vezu, tada klijent-ova strana javlja klijent-u da je server zauzet ili da se ne može spojiti na određeni server. Klijent-ova strana može imati definirani vremenski interval u kojemu će se klijent pokušavati spojiti na server.

4.1. Trenutno vrijeme na serveru

Jednostavni primjeri *socket-klijent.c* i *socket-server.c* pokazuju najjednostavnije spajanje klijent-a sa serverom. Nakon spajanja, ako spajanje uspije, server šalje klijent-u trenutno vrijeme, vrijeme kada se klijent spojio na server.

socket-klijent.c

```
#include <stdio.h>
#include <sys/types.h>        /* osnovni sistemski tipovi podataka */
#include <sys/socket.h>       /* osnovne socket definicije */
#include <netinet/in.h>       /* sockaddr_in{} i druge Internet definicije */
#include <unistd.h>
#include <string.h>

int main (void)
{
        int sockfd;
        struct sockaddr_in servaddr;
        char buffer[64] = {0};

        /* kreiranje klijent-ovog socket-a za komunikaciju */
        sockfd = socket (PF_INET, SOCK_STREAM, 0);
        if (sockfd == -1) {
                printf("socket() error!\n");
                return 1;
        }

        /* dogovorno se struktura inicijalizira na nulu */
        memset(&servaddr, 0, sizeof(servaddr));

        /* osnovne informacije servera */
        servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons(8000);
        servaddr.sin_addr.s_addr = INADDR_ANY;
```

```
        /* spajanje klijent-a sa serverom, i to na onaj koji je određen adresnom strukturom */
        if ( connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) == -1 ) {
                printf("connect() error!\n");
                return 2;
        }

        /* primamo podatke koje nam je server poslao */
        recv(sockfd, buffer, sizeof(buffer), 0);

        puts(buffer);

        /* zatvaramo socket, kraj komunikacije */
        close(sockfd);

        return 0;
}
```

Primjer 4.1. Spajanje klijent-a sa serverom

socket-server.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

int main (void)
{
        int sockfd, klijent;
        struct sockaddr_in servaddr, cliaddr;
        char buffer[64] = {0};
        char * begin;
        int addr_sz;
        time_t tmp_time;

        /* kreiranje serverovog socket-a */
        sockfd = socket (PF_INET, SOCK_STREAM, 0);
        if (sockfd == -1) {
                printf("socket() error!\n");
                return 1;
        }

        memset(&servaddr, 0, sizeof(servaddr));

        /* informacije servera */
        servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons(8000);
        servaddr.sin_addr.s_addr = INADDR_ANY;

        /* označavanje da je ovo server */
        if (bind(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) == -1) {
                printf("bind() error!\n");
                return 2;
        }

        /* osluškivanje konekcije */
        if (listen(sockfd, 2) == -1) {
                printf("listen() error!\n");
                return 3;
        }

        /* prihvaćanje konekcije koja želi uspostaviti vezu */
        addr_sz = sizeof(cliaddr);
        if ( (klijent = accept(sockfd, (struct sockaddr *) &cliaddr, &addr_sz)) == -1) {
                printf("accept() error!\n");
                return 4;
        }

        /* dohvaćanje trenutnog vremena */
        tmp_time = time (NULL);
        begin = ctime(&tmp_time);
```

```
        strcpy(buffer, begin);

        /* pošalji klijent-u vrijeme spajanja */
        send(klijent, buffer, strlen(buffer), 0);

        /* zatvaranje privremenog komunikacijskog socket-a */
        close(klijent);

        /* zatvaranje serverovog socket-a */
        close(sockfd);

        return 0;
}
```

Primjer 4.2. Jednostavan server program

Primjeri *socket-server.c* i *klijent-server.c* su u potpunosti pojednostavljeni kako bi što bolje pokazali najjednostavnije spajanje klijent-a sa serverom. U stvarnosti serverski program je onaj koji radi bez prestanka i tako omogućuje klijent-ima da se spoje u bilo koje vrijeme.

Prvo se pokreće serverov program koji kreira socket i zatim čeka na klijent-a sa funkcijom `accept`. Kada se klijent spoji na serverov socket, `accept` funkcija vrati novi socket preko kojega će se vršiti komunikacija. Nakon toga server klijent-u šalje svoje trenutno vrijeme i završava sa radom. Klijent dobiva informaciju od servera i također završava sa radom.

4.2. Konkurentni server

Ako želimo da naš server program bude što sličniji pravom serveru tada bi `accept` funkciju stavili u beskonačnu `while` petlju zajedno sa `send` funkcijom koja šalje klijent-u vrijeme servera. Sada je sasvim očit jedan veliki problem, a to je što ako imamo puno klijenata koje se žele spojiti u isto vrijeme? Neki od klijenata bi bili prisiljeni da čekaju one koji imaju uspostavljenu vezu sa serverom. Kako nitko ne voli čekati, a pogotovo klijent, postoji rješenje ovoga problema, a to je konkurentni server, server koji može u jednom trenutku izmjenjivati podatke sa više klijenata. Konkurentan server radi na način da svaku komunikaciju sa pojedinim klijentom vrši u zasebnom thread-u. Pomoću thread-ovo dobivamo paralelnu komunikaciju servera sa klijentom. Jedino što treba paziti kod multithread aplikacija je sinhronizacija pristupa dijeljenim resursima i da neke funkcije ne garantiraju ispravan rad u thread funkciji.
Primjer 4.3. prikazuje kako server komunicira sa više klijenata u isto vrijeme pomoću thread-ova.

server-thread.c

```
#include <stdio.h>
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include <pthread.h>          /* thread definicije */

/* thread funkcija, za sve klijente jednaka */
void * ThreadFun (void *arg)
{
        /* klijent-ov socket */
        int clifd = *((int *) arg);
        time_t tmp_time;
        char buffer[64] = {0};
        char * begin;
        int i;

        /* oslobađamo zauzete resurse */
        free(arg);

        /* dohvaćanje trenutnog vremena, (ctime() ne garantira ispravan rad) */
        tmp_time = time (NULL);
        begin = ctime(&tmp_time);
        strcpy(buffer, begin);

        /* šaljemo trenutno vrijeme klijent-u */
        send(clifd, buffer, strlen(buffer), 0);

        /* svaki thread opterećujemo dugotrajnom operacijom */
        for (i = 0; i < 1000000; i++)
                printf("Ovo je thread %d\n", i);

        /* zatvaramo komunikacijski socket */
        close(clifd);

        return NULL;
}

int main (void)
{
        int sockfd;
        struct sockaddr_in servaddr, cliaddr;
        int addr_sz;
        pthread_t tid;

        sockfd = socket (PF_INET, SOCK_STREAM, 0);
        if (sockfd == -1) {
                printf("socket() error!\n");
                return 1;
        }

        memset(&servaddr, 0, sizeof(servaddr));

        servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons(7000);
        servaddr.sin_addr.s_addr = INADDR_ANY;

        if (bind(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) == -1) {
                printf("bind() error!\n");
                return 2;
        }

        if (listen(sockfd, 10) == -1) {
                printf("listen() error!\n");
                return 3;
        }

        addr_sz = sizeof(cliaddr);
        printf("Server ceka na klijent-a!\n");
        while (1) {
                /* za svaku novu konekciju dinamički alociramo resurse */
                int *clifd = (int *) malloc (sizeof(int));

                if ( (*clifd = accept(sockfd, (struct sockaddr *) &cliaddr, &addr_sz))
                                == -1)
                {       printf("accept() error!\n");
```

```
                return 4;
        }

        /* za svaku prihvaćenu vezu kreiramo novi thread */
        pthread_create(&tid, NULL, ThreadFun, (void *) clifd);
    }

    /* kraj rada servera */
    close(sockfd);

    return 0;
}
```

Primjer 4.3. Server ostvaren pomoću thread funkcije

Klijent-ov kod se nije mijenjao, on se i dalje spaja na server kao i prije promjene, samo sada se ne može dogoditi zastoj, jer server svaku novu konekciju pokreće u zasebnom thread-u. Sad server može biti samo zauzet ako se dosegne maksimalan broj thread-ova koje server može opsluživati.

Kod korištenja thread-ova imamo mali problem, a to je, ako server ostvari veliki broj konekcija, odnosno, u nekom trenutku imamo puno klijenata koji su trenutno spojeni na naš server i ako iz nekog razloga se naš server sruši, klijenti koji se obrađuju u zasebrni thred-ovima, oni postaju zombiji, ostaju bez onoga tko ih je stvorio. Da bi se izbjegla takva situacija koriste se procesi. Sada ćemo za svaku uspostavljenu vezu umjesto kreiranja thread-a kreirati novi proces. Jedina mana ovakvom pristupu je ponesto sporiji odziv servera prema klijentu i zauzimanje nešto više osnovnih resursa. Za kreiranje procesa pod UNIX-om koristi se funkcija `fork()`.

4.2.1. fork i exec funkcije

Jedini način da kreirate novi proces pod UNIX-om je korištenjem fork funkcije. Njezin prototip je vrlo jednostavan i lako za koristiti, samo je osnovno razumijevanje malo složenije.

```
#include <unistd.h>

pid_t fork (void);
```

Returns: 0 in child, process ID of child in parent, -1 on error

Ako do sada niste vidjeli ovu funkciju, najteži dio za shvatiti je da se `fork` funkcija poziva jednom, a da njezin poziv izaziva vrijednosz dva puta. Vraća jednom rezultat pozivajućem procesu (roditelju) i to ID novonastalog procesa, i jednom vraća vrijednost novonastalom procesu (djetetu) i to vrijednost nula. Vraćena vrijednost nam ujedno služi da bi znali da li se naš program trenutno izvršava kao roditelj ili kao dijete. Ono što je najznačajnije kada

koristimo `fork` funkciju je to da svi deskriptori otvoreni prije samog poziva fork funkcije su dijeljeni između roditelja i dijeteta.

Dva tipična načina korištenja `fork` funkcije:

1. Proces radi vlastitu kopiju tako da jedna kopija može upravljati osnovnim operacijama, dok druga kopija može služit kao pomoćni task za obavljanje nekih drugih poslova.
2. Proces želi izvršavati drugi program. Jedini način za kreiranje novog procesa je pozivanjem `fork` funkcije. Proces prvo poziva `fork` da bi kreirao vlastitu kopiju, a tada jedna od kopija (najčešće proces djeteta) poziva `exec` da bi zamijenio sebe sa novim programom. To je tipično za shell programe.

```
#include <unistd.h>

int execl (const char *pathname, const char *arg0, ... /* (char *) 0 */ );

int execv (const char *pathname, char *const argv[]);

int execle (const char *pathname, const char *arg0, ... /* (char *) 0, char *const envp[] */ );
```

```
int execve (const char *pathname, char *const arv[], char *const envp[]);

int execlp (const char *filename, const char *arg0, ... /* (char *) 0*/ );

int execvp (const char *filename, const char argv[]);
```

All six function returns -1 on error, no return on success

Kako tema ovoga seminara nije interprocesna komunikacija, već TCP/IP API i socket-i, tako da se nećemo upuštatit u detaljna razmatranja ovih funkcija. Za detalje pogledajte neku od odgovarajućih referenci ili man stranice.

Jedina bitna napomena, i to vrlo važna, je to da se često u ponekim knjigama spominje da upravo ovih šest funkcija služe za kreiranje novih procesa ili pokretanje novih programa, to je donekle točno, ali može biti vrlo krivo shvaćeno. Naime, ako ne koristimo `fork` funkciju, a koristimo jednu od navedenih `exec` funkcija tada moramo biti svjesni da smo kreirali novi proces unutar adresnog prostora procesa koji pokušava kreirati novi proces. Za kreiranje novog procesa u zasebnom adresnom prostoru, moramo koristiti prvo `fork` funkciju, a tek tada jednu od navedenih `exec` funkcija.

server-fork.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
```

```c
#include <string.h>
#include <time.h>
#include <signal.h>
#include <sys/wait.h>

/* Funkcija koja hvata signal od childa i poziva waitpid da se ocisti memorija
 * od child procesa */
void sigchld_handler(int signum)
{
	int status;
	waitpid(WAIT_ANY, &status, WNOHANG);
	printf("PARENT: Child exited with signal %d and status %d\n", signum, status);
}

int main(void)
{
	int sockfd, client;
	struct sockaddr_in servaddr, cliaddr;
	char buffer[64] = {0};
	char * begin;
	int address_size;
	time_t tmp_time;
	pid_t childpid; /* variable to store the child's pid */

	/* kreiranje serverovog socket-a */
	sockfd = socket (PF_INET, SOCK_STREAM, 0);
	if (sockfd == -1)
	{
		printf("socket() error!\n");
		return 1;
	}
	memset(&servaddr, 0, sizeof(servaddr));

	/* informacije servera */
	servaddr.sin_family = AF_INET;
	servaddr.sin_port = htons(8000);
	servaddr.sin_addr.s_addr = INADDR_ANY;

	/* oznavanje da je ovo server */
	if (bind(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) == -1)
	{
		printf("bind() error!\n");
		return 2;
	}

	/* osluškivanje konekcije */
	if (listen(sockfd, 2) == -1)
	{
		printf("listen() error!\n");
		return 3;
	}

	/* prihvaanje konekcije koja zeli uspostaviti vezu */
	address_size = sizeof(cliaddr);
	while(1)
	{
		if ( (client = accept(sockfd, (struct sockaddr *) &cliaddr, &address_size))
			== -1)
		{
			printf("accept() error!\n");
			return 4;
		}

		/* Forkanje osnovnog procesa */
		if ((childpid = fork())== -1)
		{
			perror("fork"); /* display error message */
			exit(0);
		}

		/* Obrada unutar child procesa */
		if (childpid == 0)
		{
			/* dohvaanje trenutnog vremena */
			tmp_time = time (NULL);
			begin = ctime(&tmp_time);
			strcpy(buffer, begin);
```

```
                        /* posalji client-u vrijeme spajanja */
                        send(client, buffer, strlen(buffer), 0);

                        sleep(10);
                        strcpy(buffer, "Zdravo!!!\n");
                        send(client, buffer, strlen(buffer), 0);

                        /* zatvaranje privremenog komunikacijskog socket-a */
                        shutdown(client,SHUT_RDWR);
                        _exit(0);
                }
                /* Serverski proces */
                else
                {
                        /* Hvatanje signala od Childa i njegovog exit statusa */
                        signal(SIGCHLD, sigchld_handler);
                        printf("PARENT: Childpid is %d\n", childpid);
                }
        }
        /* zatvaranje serverovog socket-a */
        close(sockfd);

        return 0;
}
```

Primjer 4.4. *Server ostvaren korištenjem fork funkcije*

4.2.2. Procesi vs. Thread-ova

Za neke programe je teško odlučiti kada koristiti thread-ove, a kada procese. Evo nekoliko savjeta koji mogu olakšati taj izbor:

- svi thread-ovi u programu moraju biti pokretani unutar istog izvršnog programa. Djetetov proces, sa druge strane, može pokretati različite izvršne programe koristeći exec funkcije.
- "lutajući" thread-ovi mogu naškoditi drugim thread-ovima unutar istog procesa zato što thred-ovi dijele isti virtualni memorijski prostor i druge resurse. Na primjer, jedan thread može pisati po neautoriziranom dijelu memorije, koja je možda dostupna u drugom thread-u unutar istog procesa. Sa druge strane "lutajući" procesi to ne mogu zato što svaki proces ima vlastitu kopiju memorijskog prostora.
- Kopiranje memorije za novi proces je skuplja operacija, od operacije kreiranja thread-a, zato što se kod kreiranja novog thread-a kopira manji broj potrebnih informacija nego sto je to slučaj kod kreiranja novog procesa.
- Thread-ovi bi se trebali koristiti za programe kojima je potrebniji finiji paralelizam, dok bi se procesi trebali koristiti tamo gdje je potrebniji grublji paralelizam.

Dijeljenje podataka između thread-ova je jednostavnije zato što thread-ovi koriste istu memoriju. Jedinu brigu koju moramo paziti kada radimo sa thread-ovima je utrka za resursima. Dijeljenje memorije između procesa je ponešto složenije, ali zato kod njih nema

toliko problema oko dijeljenih resursa i puno su rijeđi bugovi koji se pojavljuju kod konkuretnosti.

Dodatak A: Web server

Napravili smo jedan rudimentalni web server kako bi barem malo dočarali kako se pomoću TCP/IP API-a izrađuju komunikacijski programi.

Ovaj web server radi po HTTP 1.1 standardu (opisanom u RFC-u 1216) na portu 8000 (tako da ga može pokrenuti i običan korisnik, a ne samo *root*) i isporučuje samo HTML ili obične *plain text* datoteke. Ostale funkcionalnosti (isporučivanje slika i sl.) nismo stavljali jer nije predmet ovog seminara, a samo bi otežalo razumjevanje koda.

Želite li isprobati kako ovaj server radi, kompajlirajte kod, napravite direktorij htdocs i u njega stavite datoteku index.html. Pokrenite web server, te u web preglednik na tom istom računalu upišite http://localhost:8000. Naravno, ako serveru pristupate sa nekog drugog računala, tada umjesto localhost upisujete ime servera ili njegovu IP adresu. Ako je sve prošlo kako treba, trebali bi dobiti index.html web stranicu u svom web pregledniku.

BiK Web Server:

```
/*bik-httpd.c*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include <signal.h>
#include <sys/wait.h>

#define BUF_SIZE 16384 /* Velicina buffera */
#define SRV_NAME "BiK Web server v0.2" /* Ime servera */

/* Funkcija koja hvala signal od childa i poziva waitpid da se ocisti memorija
 * od child procesa */
void sigchld_handler(int signum)
{
	int status;
	waitpid(WAIT_ANY, &status, WNOHANG);
	printf("Child exited with signal %d and status %d\n", signum, status);
}

/* postavljanje parametara protokola HTTP 1.1 koji ce se vratiti klijentu u
 * ovisnosti o statusu
 */
void parametri_odgovor(int status, char* buff)
{
	char param1[BUF_SIZE/4] = {0}; /* Buffer za parametre na pocetku */
	char param2[BUF_SIZE/4] = {0}; /* Buffer za parametre na kraju */
	time_t tmp_time;

	/* ovisno o statusu se vracaju odredjeni parametri klijentu */
	switch(status)
	{
		case 404:
			strcpy(param1,"HTTP/1.1 404 Not Found\n");
			strcpy(param2,"Transfer-Encoding: chunked\n");
			break;
		case 200:
			strcpy(param1,"HTTP/1.1 200 OK\n");
//			strcat(param2, "Last-Modified: Mon, 22 Nov 2004 12:02:11 GMT\n");
			strcat(param2, "Accept-Ranges: bytes\n");
			strcat(param2, "Content-Length: 4096\n");
			break;
		default:
			strcpy(param1,"HTTP/1.1 400 Bad Request\n");
			strcpy(param2,"Connection: close\n");
	}

	strcpy(buff, param1);
	strcat(buff, "Date: ");
```

```
        tmp_time = time (NULL);
        strcat(buff, ctime(&tmp_time));
        strcat(buff, "Server: ");
        strcat(buff, SRV_NAME);
        strcat(buff, "\n");
        strcat(buff, param2);
        strcat(buff, "Content-Type: text/html; Charset: UTF-8\n\n");
}

/* Obrada HTTP zahtjeva i vracanje odgovora klijentu */
int obrada(int client)
{
        char param[BUF_SIZE/4] = {0}; /* Buffer za parametre HTTP protokola */
        char html[BUF_SIZE] = {0}; /* Buffer za isporuku HTML stranice */
        char buf1[BUF_SIZE/4] = {0}; /* Privremeni buffer za prihvacanje zahtjeva */
        char tmpbuf[BUF_SIZE] = {0}; /* Privremeni buffer za potrebe obrade zahtjeva */
        char fpath[128] = {0}; /* relativni path i ime datoteke koja se servira */
        char *cmd; /* Parametri zahtjeva koji se ovdje samo ispisuju na strani servera, ali
                      inace sluze HTTP protokolu */
        FILE *datoteka;

        /* Dohvacanje komande od klijenta */
        do {
                recv(client, tmpbuf, sizeof(tmpbuf), 0);
                if(tmpbuf[0]=='\r')
                        break;
                strcat(buf1,tmpbuf);
        } while(tmpbuf[strlen(tmpbuf)-3]!='\n');

        /* ako u prvoj liniji nije GET onda se vraca bad request */
        if (strncmp(buf1, "GET /", 5))
        {
                memset(param, 0, sizeof(param));
                parametri_odgovor(400, param);
                strcat(html, "12a\n");
                strcat(html, "<!DOCTYPE HTML PUBLIC \"-//IETF//DTD HTML 2.0//EN\">\n");
                strcat(html, "<HTML><HEAD>\n");
                strcat(html, "<TITLE>400 Bad Request</TITLE>\n");
                strcat(html, "</HEAD>\n<BODY>\n");
                strcat(html, "<H1>Bad Request</H1>\n");
                strcat(html, "Your browser sent a request that this server could not
                                understand.\n<P>");
                strcat(html, "Invalid URI in request ");
                strcat(html, buf1);
                strcat(html, "</P>\n<HR>\n<ADDRESS>");
                strcat(html, SRV_NAME);
                strcat(html, " at localhost Port 8000</ADDRESS>\n");
                strcat(html, "</BODY></HTML>\n");
                strcat(html, "\n0\n");
                send(client, param, sizeof(param), 0);
                send(client, html, sizeof(html), 0);
                shutdown(client,SHUT_RDWR);
                return 0;
        }

        /* parsiranje komandi koje su dobivene */
        strcpy(tmpbuf, buf1);
        cmd = strtok(tmpbuf, "\n");
        /* Uzimanje imena filea iz trazenog zahtjeva
         * ako je filename / onda se postavlja na index.html
         */
        sscanf(strstr(cmd, "/"), "%s", &tmpbuf);
        if (strlen(tmpbuf)==1)
                strcpy(tmpbuf, "/index.html");
        strcpy(fpath, "htdocs");
        strcat(fpath, tmpbuf);
        /* nastavak parsiranja s time da se parametri klijenta u ovoj
         * verziji ne uzimaju u obzir vec se samo ispisuju na strani
         * servera na stdout
         */
        while(cmd = strtok(NULL,"\n"))
        {
                printf("%i: %s\n", strcmp(cmd,"\r"), cmd);
        }

        /* Otvaranje trazene datoteke, ili vracanje greske ako trazena
         * datoteka ne postoji
```

```
         */
        if((datoteka=fopen(fpath, "r"))==NULL)
        {
                memset(param, 0, sizeof(param));
                parametri_odgovor(404, param);
                strcat(html, "10a\n");
                strcat(html, "<!DOCTYPE HTML PUBLIC \"-//IETF//DTD HTML 2.0//EN\">\n");
                strcat(html, "<HTML><HEAD>\n");
                strcat(html, "<TITLE>404 Not Found</TITLE>\n");
                strcat(html, "</HEAD>\n<BODY>\n");
                strcat(html, "<H1>Not Found</H1>\n");
                strcat(html, "<P>The requested URL ");
                strcat(html, tmpbuf);
                strcat(html, " was not found on this server.");
                strcat(html, "</P>\n<HR>\n<ADDRESS>");
                strcat(html, SRV_NAME);
                strcat(html, " at localhost Port 8000</ADDRESS>\n");
                strcat(html, "</BODY></HTML>\n");
                strcat(html, "\n0\n");
        }
        else
        {
                memset(param, 0, sizeof(param));
                parametri_odgovor(200, param);

                /* Citanje iz trazene datoteke i slanje klijentu */
                while(!feof(datoteka))
                {
                        memset(tmpbuf, 0, sizeof(buf1));
                        fgets(tmpbuf, BUF_SIZE, datoteka);
                        strcat(html, tmpbuf);
                        /* zadnju liniju dva puta ispisuje!!! */
                }
                fclose(datoteka);
        }

        /* slanje HTTP komandi klijentu */
        send(client, param, strlen(param), 0);
        send(client, html, strlen(html), 0);

        /* zatvaranje privremenog komunikacijskog socket-a */
        shutdown(client,SHUT_RDWR);
        /* close(client); */

        return 0;
}

int main (void)
{
        int sockfd, client;
        struct sockaddr_in servaddr, cliaddr;
        char * begin;
        int address_size;
        pid_t childpid; /* variable to store the child's pid */

        /* kreiranje serverovog socket-a */
        sockfd = socket (PF_INET, SOCK_STREAM, 0);
        if (sockfd == -1)
        {
                printf("socket() error!\n");
                return 1;
        }
        memset(&servaddr, 0, sizeof(servaddr));

        /* informacije servera */
        servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons(8000);
        servaddr.sin_addr.s_addr = INADDR_ANY;

        /* oznavanje da je ovo server */
        if (bind(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) == -1)
        {
                printf("bind() error!\n");
                return 2;
        }

        /* osluskivanje konekcije */
```

```
        if (listen(sockfd, 2) == -1)
        {
                printf("listen() error!\n");
                return 3;
        }

        /* prihvaanje konekcije koja želi uspostaviti vezu */
        address_size = sizeof(cliaddr);
        while(1)
        {
                if ( (client = accept(sockfd, (struct sockaddr *) &cliaddr, &address_size))
                                == -1)
                {
                        printf("accept() error!\n");
                        return 4;
                }
                if ((childpid = fork())== -1)
                {
                        perror("fork"); /* display error message */
                        exit(0);
                }
                if (childpid == 0)
                {
                        /* Obrada kompletnog sessiona u posebnoj funkciji */
                        obrada(client);
                        _exit(0);
                }
                else
                {
                        /* serverski proces */
                        printf("Childpid is %d\n", childpid);
                        /* Hvatanje signala od Childa i njegovog exit statusa */
                        signal(SIGCHLD, sigchld_handler);
                }
        }
        /* zatvaranje serverovog socket-a */
        close(sockfd);

        return 0;
}
```

Dodatak B: Klijent – server ostvaren pomoću UDP-a i korištenjem gethostbyname funkcije

Klijent:

```
/*  uuc.c - UNIX UDP Client Code  */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>


int main(int argc, char *argv[])
{
  struct sockaddr_in their_addr;
  struct sockaddr_in my_addr;
  struct hostent *he;
  int sockfd;
  int addr_len;
  char data[32]={0};

  int nRet=0;

  if (argc != 3)
   {
    printf("usage: application celsius hostname\n");
    return EXIT_FAILURE;
   }

  sprintf(data, "%s", argv[1]);

  he = gethostbyname(argv[2]);
  if (he == NULL)
   {
    printf("Error in gethostbyname\n");
    return EXIT_FAILURE;
   }

  sockfd = socket(AF_INET, SOCK_DGRAM, 0);
  if (sockfd == -1)
   {
    printf("Error Creating Socket\n");
    return EXIT_FAILURE;
   }
  their_addr.sin_family = AF_INET;
  their_addr.sin_addr = *((struct in_addr *)he->h_addr);
  their_addr.sin_port = htons(1092);


   if (sendto(sockfd, data, strlen(data),
       0, (struct sockaddr *)&their_addr,
       sizeof(struct sockaddr)) == -1)
       {
     printf("Error in sendto\n");
         return EXIT_FAILURE;
       }


    memset(data, 0, sizeof data);

    addr_len = sizeof(struct sockaddr);
    if (recvfrom(sockfd, data, sizeof data,
               0, (struct sockaddr *)&their_addr, &addr_len) == -1)
       {
     printf("Error in recvfrom\n");
```

```
        return EXIT_FAILURE;
       }

  printf("%s Celsius = %s Fahrenheight\n",argv[1],  data);


  shutdown(sockfd,2);
  return EXIT_SUCCESS;
}
```

Server:

```
/*  uss.c - UNIX UDP Server Code  */

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <netinet/in.h>

volatile sig_atomic_t done;


void inthandler(int Sig)
{
 done = 1;
}

int CheckForData(int sockfd)
 {
  struct timeval tv;
  fd_set read_fd;
  tv.tv_sec=0;
  tv.tv_usec=0;
  FD_ZERO(&read_fd);
  FD_SET(sockfd, &read_fd);
  if(select(sockfd+1, &read_fd,NULL, NULL, &tv)== -1)
  {
   return 0;
  }
   if(FD_ISSET(sockfd, &read_fd))
   {
    return 1;
   }
 return 0;
}


int main(int argc, char **argv)
 {
  int   sock;
  struct sockaddr_in my_addr;
  struct sockaddr_in their_addr;
  int address_length;
  char data[32]= {0};
  int celc=0;
  int farh=0;


  if(signal(SIGINT, SIG_IGN) != SIG_IGN)
   {
    signal(SIGINT, inthandler);
   }

  if(signal(SIGTERM, SIG_IGN) != SIG_IGN)
   {
    signal(SIGTERM, inthandler);
```

```
   }

  my_addr.sin_family = AF_INET;
  my_addr.sin_addr.s_addr = INADDR_ANY;
  my_addr.sin_port = htons(1092);


  if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
   {
    printf("Error Creating Socket\n");
    return EXIT_FAILURE;
   }



  if (bind(sock, (struct sockaddr *)&my_addr,sizeof(struct sockaddr)) ==-1)
   {
    printf("Unexpected error on bind\n");
    shutdown(sock,2);
    return EXIT_FAILURE;
   }

  address_length = sizeof(struct sockaddr_in);


  while(!done)
  {
   if(0 != CheckForData(sock))
   {
    memset(data, 0, sizeof data);

        if (recvfrom(sock, data,
                        sizeof data,
                        0,   (struct sockaddr *)&their_addr,
                        &address_length) == -1)
        {
     printf("Error on recvfrom\n");
        }


    celc=atoi(data);
    farh=(celc*2)+32;

        memset(data, 0, sizeof data);

        sprintf(data, "%d", farh);

   if (sendto(sock, data, strlen(data), 0,
                (struct sockaddr *)&their_addr,
                       sizeof(struct sockaddr)) == -1)
   {
    printf("Error on sendto\n");
   }

        printf("%s\n", data);

   }
  }

   shutdown(sock,2);
   printf("User requested program to halt.\n");

   return EXIT_SUCCESS;
}
```

Dodatak C: Local Namespace Socket

Server:

```
/* (socket-server.c) Local Namespace Socket Server */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

/* Čita tekst sa socket-a i ispisuje ga na standardni izlaz. Ponavljaj
 * dok se socket ne zatvori. Vraća broj različit od nule ako klijent
 * pošalje "quit" poruku, nulu u suprotnom.
*/
int server (int client_socket)
{
       while (1) {
       int length;
       char* text;

       /* Prvo, čitamo duljinu tekst poruke sa socket-a. ako
       read vrati nulu, klijent je zatvorio konekciju. */
       if (read (client_socket, &length, sizeof (length)) == 0)
              return 0;
       /* Alociramo buffer dovoljne veličine. */
       text = (char*) malloc (length);
       /* Čitamo tekst te ga ispisujemo. */
       read (client_socket, text, length);
       printf (“%s\n”, text);
       /* Oslobađamo zauzete resurse. */
       free (text);
       /* Ako klijent pošalje "quit" poruku, završavamo */
       if (!strcmp (text, “quit”))
              return 1;
       }
}

int main (int argc, char* const argv[])
{
       const char* const socket_name = argv[1];
       int socket_fd;
       struct sockaddr_un name;
       int client_sent_quit_message;

       /* Kreiramo socket. */
       socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);

       /* Označavamo da je ovo server. */
       name.sun_family = AF_LOCAL;
       strcpy (name.sun_path, socket_name);
       bind (socket_fd, &name, SUN_LEN (&name));

       /* Osluškujemo konekcije. */
       listen (socket_fd, 5);

       /* Uzastopno prihvaćamo konekcije.
              Ponavljamo dok klijent ne pošalje "quit". */
       do {
              struct sockaddr_un client_name;
              socklen_t client_name_len;
              int client_socket_fd;
              /* Prihvaćamo konekciju. */
              client_socket_fd = accept (socket_fd, &client_name, &client_name_len);
              /* Obrada konekcije. */
              client_sent_quit_message = server (client_socket_fd);
              /* Zatvaramo našu stranu konekcije. */
              close (client_socket_fd);
       } while (!client_sent_quit_message);

       /* Brišemo socket file. */
       close (socket_fd);
```

```
        unlink (socket_name);

        return 0;
}
```

Klijent:

```
/* (socket-client.c) Local Namespace Socket Client */

#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

/* Pišemo na socket dan imenom file deskriptora SOCKET_FD. */
void write_text (int socket_fd, const char* text)
{
        /* Zapisujemo broj byte-ova stringa, uključujući
        NUL-terminator. */
        int length = strlen (text) + 1;
        write (socket_fd, &length, sizeof (length));
        /* Pišemo string na socket. */
        write (socket_fd, text, length);
}

int main (int argc, char* const argv[])
{
        const char* const socket_name = argv[1];
        const char* const message = argv[2];
        int socket_fd;
        struct sockaddr_un name;

        /* Kreiramo socket. */
        socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);
        /* Spremamo serverovo ime u socket adresnu strukturu. */
        name.sun_family = AF_LOCAL;
        strcpy (name.sun_path, socket_name);
        /* Spajamo se na socket. */
        connect (socket_fd, &name, SUN_LEN (&name));
        /* Pišemo tekst sa komandne linije na socket. */
        write_text (socket_fd, message);

        close (socket_fd);

        return 0;
}
```

Dodatak D: Primjer korištenja socketa pod Windows-ima

Za znatiželjne čitatelje (oni koji žele znati više) dat je jedan jednostavan primjer komunikacije klijent – server pod windows-ima.

Server program otvara socket na portu 3490 i čeka da se netko spoji na njega. Nakon što se neki klijent spoji na socket ispisuje se string "Hello world!" i zatvara socket.

Server:

```
/*
** server.c -- a stream socket server demo
*/

#include <winsock2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MYPORT 3490    // the port users will be connecting to

#define BACKLOG 10     // how many pending connections queue will hold

int main(void)
{
   int sockfd, new_fd;  // listen on sock_fd, new connection on new_fd
   struct sockaddr_in my_addr;    // my address information
   struct sockaddr_in their_addr; // connector's address information
   int sin_size;
   int yes=1;
   WSADATA wsaData;

   if (WSAStartup(0x202,&wsaData) == SOCKET_ERROR) {
      fprintf(stderr,"WSAStartup failed with error %d\n",WSAGetLastError());
      WSACleanup();
      return -1;
   }

   if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
      perror("socket");
      exit(1);
   }

   my_addr.sin_family = AF_INET;         // host byte order
   my_addr.sin_port = htons(MYPORT);     // short, network byte order
   my_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
   memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct

   if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr))
      == -1) {
      perror("bind");
      exit(1);
   }

   if (listen(sockfd, BACKLOG) == -1) {
      perror("listen");
      exit(1);
   }

   sin_size = sizeof(struct sockaddr_in);
   if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr,
      &sin_size)) == -1) {
         perror("accept");
   }
   printf("server: got connection from %s\n", inet_ntoa(their_addr.sin_addr));
   if (send(new_fd, "Hello, world!\n", 14, 0) == -1)
      perror("send");
   closesocket(new_fd);  // parent doesn't need this
```

```
   WSACleanup();
   return 0;
}
```

Klijent:

```
/* client.c -- a stream socket client demo */
#include <winsock2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define PORT 3490 // the port client will be connecting to

#define MAXDATASIZE 100 // max number of bytes we can get at once

int main(int argc, char *argv[])
{
   int sockfd, numbytes;
   char buf[MAXDATASIZE];
   struct hostent *he;
   struct sockaddr_in their_addr; // connector's address information
   WSADATA wsaData;

   if (argc != 2) {
      fprintf(stderr,"usage: client hostname\n");
      exit(1);
   }

   if (WSAStartup(0x202,&wsaData) == SOCKET_ERROR) {
      fprintf(stderr,"WSAStartup failed with error %d\n",WSAGetLastError());
      WSACleanup();
      exit(1);
   }


   if ((he=gethostbyname(argv[1])) == NULL) {  // get the host info
      fprintf(stderr,"gethostbyname");
      WSACleanup();
      exit(1);
   }

   if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
      fprintf(stderr,"socket");
      WSACleanup();
      exit(1);
   }

   their_addr.sin_family = AF_INET;     // host byte order
   their_addr.sin_port = htons(PORT);  // short, network byte order
   their_addr.sin_addr = *((struct in_addr *)he->h_addr);
   memset(&(their_addr.sin_zero), '\0', 8);  // zero the rest of the struct

   if (connect(sockfd, (struct sockaddr *)&their_addr,
      sizeof(struct sockaddr)) == -1) {
      fprintf(stderr,"connect");
      WSACleanup();
      exit(1);
   }

   if ((numbytes=recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1) {
      fprintf(stderr,"recv");
      WSACleanup();
      exit(1);
   }

   buf[numbytes] = '\0';

   printf("Received: %s",buf);

   closesocket(sockfd);
```

```
    WSACleanup();
    return 0;
}
```

5. Reference

1. W. Richard Stevens: UNIX Network Programming – Networking APIs: Sockets and XTI *Volume 1 Second Edition*
2. Kurt Wall, Mark Watson, Mark Whitis: Linux Programming Unleashed
3. Mark Mitchell, Jeffrey Oldham, Alex Samuel: Advanced Linux Programming
4. Craig Hunt: TCP/IP Network Administration
5. William Stallings: High-Speed Networks: TCP/IP and ATM Design Principles
6. Matija Čupen: Sockets – unix domain, network (Seminarski rad)
7. Saša Dragić: TCP/IP (Seminarski rad)
8. Viktor Marohnić – Operacijski sustavi (Socketi – 8. predavanje)
9. Internet Assigned Numbers Authority (IANA): http://www.iana.org
10. The World Wide Web Consortium (W3C): http://www.w3c.org
11. The Internet Engineering Task Force (IETF): http://www.ietf.org
12. Requests For Comments
    a. RFC 791: http://www.faqs.org/rfcs/rfc791.html
    b. RFC 792: http://www.faqs.org/rfcs/rfc792.html
    c. RFC 793: http://www.faqs.org/rfcs/rfc793.html
    d. RFC 1180: http://www.faqs.org/rfcs/rfc1180.html
    e. RFC 1365: http://www.faqs.org/rfcs/rfc1365.html
    f. RFC 1700: http://www.faqs.org/rfcs/rfc1700.html
    g. RFC 2616: http://www.faqs.org/rfcs/rfc2616.html