# MREŽNO RAČUNARSTVO

Soketi za servere

(iz glave 9,10 u 4. izdanju)

# Soketi za servere

- klijenti – programi koji otvaraju soket do servera koji osluškuje konekcije
- klijentski soketi nisu dovoljni
- klijenti nemaju mnogo svrhe ako ne komuniciraju sa serverom
- a klasa Socket nije dovoljna za pisanje servera
- za kreiranje Socket-a mora se znati host sa kojim želimo da se povežemo
- kada pišemo server, ne znamo unapred ko će nas kontaktirati, a čak i da znamo, ne znamo kada će to biti.
- drugim rečima, serveri su kao recepcionari koji sede kraj telefona i čekaju dolazeće pozive. Oni ne znaju ko će zvati i kada, samo da kada telefon zazvoni, oni treba da se jave i pričaju ko god da je sa druge strane. Ovakvo ponašanje se ne može isprogramirati samom klasom Socket.

- Za servere, koji prihvataju konekcije, Java obezbeđuje klasu ServerSocket koja predstavlja serverske sokete.
- U suštini, posao serverskog soketa je da sedi kraj telefona i čeka dolazeće pozive.
- tj. serverski soket se izvršava na serveru i osluškuje dolazeće TCP konekcije.
- svaki serverski soket osluškuje na određenom portu serverske mašine.
- Kada klijent sa udaljenog hosta pokuša da se konektuje na taj port, server se budi, pregovara o konekciji između klijenta i servera i vraća regularni Socket objekat koji predstavlja soket između dva hosta.
- Drugim rečima, serverski soket čeka na konekcije, dok klijentski soketi iniciraju konekcije.
- Nakon što je ServerSocket uspostavio konekciju, server koristi regularni Socket objekat da šalje podatke klijentu. Podaci uvek putuju preko regularnog soketa

# klasa ServerSocket

- sadrži sve što je potrebno za pisanje servera u Javi
- ima konstruktore, metode koji osluškuju konekcije na zadatom portu, metode za konfigurisanje raznih opcija serverskog soketa, i uobičajene razne metode kao što je toString()

# Životni ciklus serverskog programa

1. kreira se novi ServerSocket na zadatom portu korišćenjem konstruktora
2. ServerSocket osluškuje dolazeće pokušaje konekcija na tom portu koristeći svoj accept() metod. Metod accept() *blokira* dok klijent ne pokuša da napravi konekciju, kada accept() *vraća Socket objekat* koji povezuje klijenta i servera
3. u zavisnosti od tipa servera, getInputStream(), getOutputStream() ili oba ova metoda se pozivaju za Socket objekat kako bi se komuniciralo sa klijentom
4. server i klijent interaguju u skladu sa dogovorenim protokolom do zatvaranja konekcije
5. server, klijent, ili obojica zatvaraju konekciju
6. server se vraća na korak 2 i čeka narednu konekciju

- ako korak 4 traje neograničeno, tradicionalni Unix serveri kreiraju novi proces da rukuje svakom konekcijom tako da veći broj klijenata može biti istovremeno uslužen
- u tom slučaju, java programi kreiraju niti za interakciju sa klijentima, tako da server može da procesira narednu konekciju
- s druge strane, ako je protokol jednostavan i brz i dopušta da server zatvori konekciju kada završi, efikasnije je da server neposredno procesira zahtev klijenta, bez kreiranja niti

- I generisanje prevelikog broja niti može predstavljati problem. Za sistem sa oko 1GB RAM-a sve od približno 1000 niti će dramatično usporiti i izazvati da CPU često swap-uje podatke u i iz RAM-a.
- Generisanje prevelikog broja niti je jedan od nekoliko načina da se pouzdano sruši java VM
- ServerSocketChannel klasa obezbeđuje neblokirajući I/O zasnovan na kanalima, ne na tokovima. Sa kanalima, jedna nit može procesirati veći broj konekcija. To je prednost za velike servere. Za jednostavne servere, manje opterećene, treba koristiti tehnike opisane u ovom (10.) poglavlju

- O.S. smešta dolazeće konekcije za određeni port u FIFO red.
- podrazumevana veličina je 50, ali može da varira od O.S. do O.S.
- Neki O.S. imaju ovu dužinu 5 (onda je red vel. 5, ne 50)
- Nakon što se red napuni, hostovi odbijaju dodatne konekcije za taj port dok se ne oslobodi mesto u redu
- mnogi (ali ne svi) klijenti pokušavaju nekoliko puta da naprave konekciju ako inicijalni pokušaj bude odbijen
- OS rukuje dolazećim konekcijama i redom, mi o tome ne moramo da brinemo
- nekoliko ServerSocket konstruktora omogućuje promenu veličine ovog reda, ali nije moguće povećati red iznad maksimuma koji podržava O.S.

# Konstruktori

- public ServerSocket(int port) throws BindException, IOException
- public ServerSocket(int port, int queueLength) throws BindException, IOException
- public ServerSocket(int port, int queueLength, InetAddress bindAddress) throws IOException
- public ServerSocket() throws IOException
- zadaje se port, dužina reda, local network interface za koji treba da se veže

# public ServerSocket(int port) throws BindException, IOException

- kreira serverski soket na portu zadatom argumentom
- ako se prosledi o kao argument, sistem bira dostupan port za nas
- port koji izabere sistem, ponekad se naziva anonimnim jer ne znamo njegov broj
- za servere, anonimni portovi nisu od koristi, jer klijenti moraju unapred da znaju na koji port da se konektuju, međutim, ima nekoliko situacija kada anonimni port može biti koristan

# primer

- kreiranje serverskog soketa koji će koristiti HTTP server, na portu 80

```
try{
  ServerSocket httpd = new ServerSocket(80);
}
catch(IOException ex){
  System.err.println(ex);
}
```

- Konstruktor izbacuje IOException kada soket ne može biti kreiran i vezan za traženi port.
- IOException pri kreiranju ServerSocket-a skoro uvek znači dve stvari: *drugi serverski soket iz potpuno drugog programa već koristi traženi port, ili pokušavamo da se konektujemo na port 1 do 1023 na Unix-u bez root privilegija*

# primer 1

- varijacija PortScanner programa iz prethodnog poglavlja
- proveravaju se portovi na lokalnoj mašini pokušajem kreiranja ServerSocket objekata na njima i gledanjem na kojim portovima to ne uspeva
- Ako koristite Unix i niste logovani kao root, ovaj program radi samo za portove 1024 i više

# page 302 -...

- ostali konstruktori

# Prihvatanje i zatvaranje konekcija

- ServerSocket radi u petlji koja uzastopno prihvata konekcije
- svaki prolazak kroz petlju poziva metod accept(). On vraća Socket objekat koji predstavlja konekciju između udaljenog klijenta i lokalnog servera
- Interakcija sa klijentom dešava se kroz ovaj Socket objekat
- Kada se transakcija završi, server treba da pozove close() metod Socket-a
- Ako klijent zatvori konekciju dok server još radi, input i output stream-ovi koji povezuju server sa klijentom izbacuju InterruptedIOException prilikom sledećeg čitanja ili pisanja
- U svakom slučaju, server treba da bude spreman da obradi sledeću dolazeću konekciju.
- Međutim, kada server treba da se ugasi i ne procesira daljnje konekcije, treba pozvati close() metod ServerSocket objekta.

# public Socket accept() throws IOException

- kada server podesi šta treba i spremni smo da prihvatimo konekciju, pozivamo accept() metod ServerSocket objekta
- ovaj metod blokira, tj. stopira tok izvršavanja i čeka dok se klijent ne konektuje
- kada se klijent konektuje, metod vraća Socket objekat. Koristimo tokove vraćene metodima getInputStream() i getOutputSteram() Socket-a za komuniciranje sa klijentom

# primer

```
ServerSocket server = new ServerSocket(5776);
while(true){
  Socket connection = server.accept();
  OutputStreamWriter out = new
  OutputStreamWriter(connection.getOutputStream
  ());
  out.write("You've connected to this server. Bye-bye
  now.\r\n");
  connection.close();
}
```

- Ako ne želite da program stoji dok čeka na konekciju, stavite accept() u posebnu nit
- imamo i opciju da koristimo kanale i neblokirajući I/O umesto niti. Na nekim (ne svim) VM ovo je mnogo brže nego niti i tokovi
- *Bitno je razlikovati izuzetke koji verovatno gase server i izdaju poruku o grešci i izuzetke koji samo treba da zatvore aktivnu konekciju*
- Izuzeci izbačeni od accept() ili input ili output stream-a generalno ne treba da ugase server. Većina drugih izuzetaka verovatno treba. Da bi se ovo uradilo, neophodno je *ugnjezditi try-ove*.

- Konačno, većina servera želi da bude sigurna da su svi soketi koje su prihvatili zatvoreni kada se oni završe
- Čak i ako protokol zahteva da su klijenti odgovorni za zatvaranje konekcije, klijenti ne moraju uvek striktno da se pridržavaju protokola
- poziv close() takođe treba da bude unutar try-bloka koji hvata IOException
- Međutim, ako hvatate IOException kada zatvarate soket, ignorišite ga. To samo znači da je klijent zatvorio soket pre nego što je server uspeo.

# primer

```
try{
 ServerSocket server = new ServerSocket(5776);
 while(true){
  Socket connection = server.accept();
  try{
    Writer out = new OutputStreamWriter(connection.getOutputStream());
    out.write("You've connected to this server. Bye-bye now.\r\n");
    out.flush();
  }
  catch(IOException ex){
     // This tends to be a transitory error for this one connection;
     // e.g. the client broke the connection early. Consequently, you
     // don't want to break the loop or print an error message.
     // However, you might choose to log this exception in an error log.
  }
  finally{
    // Guarantee that sockets are close when complete
    try{
       if(connection!=null) connection.close();
    } catch(IOException ex){}
  }
 }
catch(IOException ex){
  System.err.println(ex);
}
```

# primer 2, daytime server

- implementira jednostavni daytime server kao RFC 867
- ovaj server šalje jednu liniju teksta kao odgovor na svaku konekciju, pa procesira svaku konekciju neposredno
- složeniji serveri bi trebalo da generišu nit za obradu svakog zahteva
- u ovom slučaju, overhead generisanja niti bi bio veći nego vreme potrebno da se obradi zahtev
- ako se program pokreće na Unix-u mora se pokrenuti kao root da bi se konektovalo na port 13 (ili promeniti broj porta na nešto iznad 1024)

# primer 2 objašnjenja

- primer je pravolinijski
- klasa java.util.Date obezbeđuje vreme pročitano sa serverovog internog sata
- konstanta DEFAULT_PORT postavljena je na dobro poznati port 13 daytime servera
- metod main() radi sav posao
- ako se port zada u komandnoj liniji, koristi se taj port, a inače podrazumevani

# primer 2 objašnjenja

- spoljni try blok hvata IOException koji mogu nastati kada se ServerSocket objekat server konstruiše na daytime portu ili kada prihvata konekcije
- unutrašnji try blok hvata izuzetke izbačene kada se prihvaćene konekcije procesiraju
- poziva se metod accept() unutar beskonačne petlje da pazi na nove konekcije
- kao mnogi serveri, ovaj program se nikada ne završava, već nastavlja da osluškuje do izbacivanja izuzetka ili dok ga ručno ne zaustavimo (Unix: Ctrl+C, kill pid)

# primer 2 objašnjenja

- Kada se klijent konektuje, accept() metod vraća Socket objekat, koji se smešta u lokalnu promenljivu connection, i program se nastavlja. Zove se getOutputStream() da vrati izlazni tok pridružen Socket-u i olančava na novi OutputStreamWriter, out.
- Novi Date objekat obezbeđuje tekuće vreme. Sadržaj se šalje klijentu pisanjem njegove string reprezentacije na out pomoću write().

# primer 2, objašnjenja

- Konačno, nakon što su podaci poslati ili je izbačen izuzetak, finally blok zatvara konekciju.
- Uvek zatvarajte soket nakon što ste završili sa njim.
- U prethodnom poglavlju je rečeno da klijent ne treba da računa da će druga strana konekcije zatvoriti soket. To se utrostručava za servere.
- Klijent istekne ili se sruši, korisnik otkaže transakcije, padne mreža... Zbog bilo kog od ovih ili mora drugih razloga ne možemo računati da će klijenti zatvoriti sokete, čak i kada protokol to zahteva od njih, što ovaj ne zahteva.

# primer 3, time server

- slanje binarnih, netekstualnih podataka nije značajno teže
- time server koji prati time protokol iz RFC 868
- kada se klijent konektuje, server šalje 4-bajtni, big endian, neoznačeni ceo broj koji određuje broj sekundi proteklih od 12:00 A.M. January 1, 1900 GMT (the epoch)
- Ponovo, tekuće vreme se određuje kreiranjem novog Date objekta. Međutim, pošto klasa Date računa milisekunde od 12:00 A.M. January 1, 1970 GMT a ne od 1900, neophodna je konverzija

# public void close()

- ako ste završili sa serverskim soketom, treba da ga zatvorite, posebno ako će program nastaviti još neko vreme da se izvršava
- ovo oslobađa port tako da mogu da ga koriste drugi programi, ako žele
- zatvaranje ServerSocket-a ne treba mešati sa zatvaranjem Socket-a
- zatvaranje ServerSocket-a osobađa port na lokalnom host-u, dopuštajući drugom serveru da se veže na njega. Takođe raskida sve trenutno otvorene sokete koje je ServerSocket prihvatio

- Serverski soketi se zatvaraju automatski kada program umre, pa nije apsolutno neophodno zatvarati ih u programima koji se završavaju kratko nakon što ServerSocket više nije potreban
- Ipak, to ne boli.
- Primer: LocalPortScanner može biti bolje napisan tako da trenutno ne zauzima većinu portova sistema

# primer

```
for(int port = 1; port <= 65535; port++){
  try{
    // the next line will fail and drop into the
   // catch block if there is already a server
   // running on the port
    ServerSocket server = new ServerSocket(port);
    server.close();
   }
   catch(IOException ex){
     System.out.println("There is a server on port " + port + ".");
   }
} // end for
```

- Nakon što je serverski soket zatvoren, ne može biti rekonektovan, čak ni na isti port
- public boolean isClosed()

  vraća true ako je serverski soket zatvoren
- … page 14 of 51 (o metodu isClosed())
- public boolean isBound() – da li je serverski soket (ikada) vezan za port (čak i ako je trenutno zatvoren)
- za testiranje da li je soket otvoren:

```
public static boolean isOpen(ServerSocket ss){
     return ss.isBound() && !ss.isClosed();
}
```

# get*() metodi

- public InetAddress getInetAddress()

  ako još nije vezan za mrežni interfejs, vraća null

- public int getLocalPort()

  ako još nije vezan za port, vraća -1