Web programiranje – Vežbe 3
Mrežno programiranje

Primer – tekstualni protokol

Projekat **Mrezno programiranje** sadrži klase **Client** i **Server** u paketu **mrezno.programiranje**. Primer ilustruje prostu implementaciju klijent-server arhitekture korišćenjem ugrađenih Java klasa, na najnižem nivou. Klijent i server su programi koji se (najčešće) izvršavaju na dve različite mašine. Osnovni tok klijent-server komunikacije se odvija tako što klijent šalje zahteve serveru preko mreže, koji zatim obrađuje zahtev i šalje odgovor nazad klijentu.

Mašine se na mreži obično identifikuju preko **IP adrese**. Pošto se na jednoj mašini može izvršavati veliki broj programa, potrebno je i njih identifikovati. Svakom programu koji komunicira preko mreže se dodeljuje **port** u obliku 16-bitnog broja (0-65535). Neki portovi su rezervisani (npr. 80 za web servere, 53 za DNS itd). Na Unix sistemima kao što je Linux su potrebna administratorska prava za portove 0-1023.

Puna adresa programa je **ip_adresa:port** – npr **192.168.0.3:8080.**

Komunikacija preko mreže se u Javi vrši preko **soketa** koji su implementirani u ugrađenoj Javinoj klasi **Socket.** Soketi imaju ulazni tok (InputStream) koji sadrži podatke koji stižu na soket i izlazni tok (OutputStream), koji služi za slanje podataka. U komunikaciji uvek učestvuju barem dva soketa: jedan na klijentskoj strani i jedan na serverskoj strani. Podaci koji se upišu u izlazni stream jednog socketa će se naći u ulaznom streamu drugog. Ispod je dat isečak koda iz main metode klase **mrezno.programiranje.Client**:

```
// odredi adresu racunara sa kojim se povezujemo
InetAddress addr = InetAddress.getByName("127.0.0.1");

// otvori socket prema drugom racunaru
Socket sock = new Socket(addr, TCP_PORT);

// inicijalizuj ulazni stream
BufferedReader in = new BufferedReader(new InputStreamReader(
                                sock.getInputStream()));
// inicijalizuj izlazni stream
PrintWriter out = new PrintWriter(new BufferedWriter(
      new OutputStreamWriter(sock.getOutputStream())), true);
```

U ovom isečku koda se otvara klijentski soket ka adresi **127.0.0.1:9000**, što znači da se podaci šalju na tu adresu. Soketu će operativni sistem dodeliti IP adresu mašine na kojoj se izvršava i neki slobodan port (da bi server mogao da zna kome da vrati odgovor).

U sledećem isečku se preko izlaznog streama šalje string "HELLO" (**out.printLine**), pa se zatim čeka odgovor iz ulaznog streama (**in.readLine**). Metoda koja čita iz ulaznog streama je blokirajuća, što znači da će se izvršavanje u trenutnoj niti zaustaviti dok se ne pojave podaci u ulaznom streamu.

```
// posalji zahtev
System.out.println("[Client]: HELLO");
out.println("HELLO");

// procitaj odgovor
String response = in.readLine();
System.out.println("[Server]: " + response);
```

Implementacija servera se nalazi u klasi **mrezno.programiranje.Server**. Ispod je dat isečak iz main metode ove klase.

```
int clientCounter = 0;
// slusaj zahteve na datom portu
ServerSocket ss = new ServerSocket(TCP_PORT);
System.out.println("Server running...");
while (true) {
	Socket sock = ss.accept();
	System.out.println("Client accepted: " + (++clientCounter));
	handleRequest(sock, clientCounter);
}
```

Serverski soket funkcioniše malo drugačije: instancira se objekat klase **ServerSocket** koji sluša na portu 9000. Samo jedan program može slušati na nekom soketu u datom trenutku, tako da će se baciti izuzetak ako je port zauzet.

Server u beskonačnoj petlji poziva metodu **accept** klase ServerSocket, koja blokira izvršavanje dok prvi klijent ne pošalje podatke preko mreže. Rezultat ove metode je soket koji služi za komunikaciju sa klijentom koji je uspostavio vezu. Metoda **handleRequest** obrađuje zahtev.

Server se u ovoj verziji izvršava u jednoj niti, što znači da se zahtevi neće izvršavati u paraleli. Ovo predstavlja veliki problem pošto web servise može da koristi veliki broj klijenata u isto vreme. Najprostiji način za rešavanje ovog problema je pokretanjem nove niti za svaki zahtev.

NAPOMENA: u praksi se koriste sofisticiranija rešenja kao što su thread pooling i reactor/proactor šablon. Više o tome zašto nit po konekciji može predstaviti problem možete pročitati ovde (C10k problem).

Zadaci

1. Pokretanje niti po zahtevu
    a. Implementirati klasu **UserThread** koja nasleđuje klasu **Thread**
    b. Premestiti logiku iz metode **Server.handleRequest** u metodu **UserThread.run**
    c. Pokrenuti novi thread za svaki zahtev umesto prostog poziva metode handleRequest
2. Proširiti klasu Server tako da omogući dodavanje i listanje korisnika
    a. Komanda za dodavanje korisnika je oblika **ADD <username>**. Odgovor treba da sadrži string `"Success"`.
    b. Komanda za listanje korisnika je oblika **LIST.** Odgovor od servera treba da sadrži imena prethodno dodatih korisnika odvojena zarezom.
3. Proširiti klasu Server komandom za brisanje korisnika oblika **REMOVE <username>.** Odgovor treba da sadrži ime obrisanog korisnika ako postoji, u suprotnom `"User not found"`.

4. Proširiti klasu Server komandom za pretragu korisnika oblika **FIND <username>.** Ako postoji više korisnika sa istim imenom, vratiti prvog koji je pronađen. Odgovor treba da sadrži ime korisnika ako postoji, u suprotnom `"User not found"`.
5. Dodati komandu za dodavanje više korisnika odjednom oblika **ADD <username1>,<username2>,<username3>…** Odgovor treba da sadrži string `"Success"`.