Predmet	Internet Programiranje
Tema	Soket programiranje u javi
Autor	Goran Antić
Datum	07.12.2011.

Socket programiranje

Soketi predstavljaju interfejse na mreži preko kojih se dva uređaja "priključuju", time uspostavljaju komunikaciju i način razmene podataka. Da bi objasnili preciznije čemu služe i kako se koriste, dobro je osvrnuti se na razlog njihovog uvođenja.

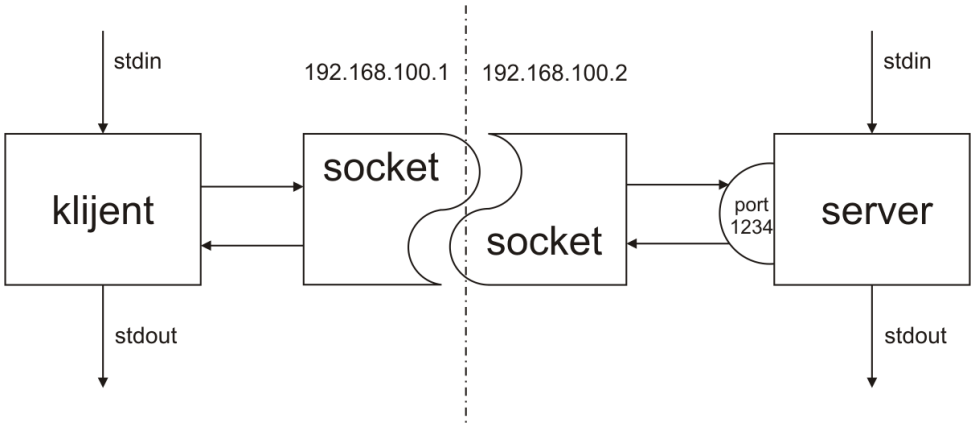
Datagrami i mrežni protokoli

Mrežna komunikacija se sastoji od razmene podataka mrežnih uređaja po datom protokolu kako bi se uređaji najbolje razumeli. Obično se podaci koji se razmenjuju dele na parčiće koje nazivamo **datagramima** (često i **paketima**). Tako podeljene podatke možemo lakše kontrolisati u transportu, ako se recimo desi greška u prenosu jednostavno šaljemo samo taj paket ponovo, a ne celu hrpu podataka koju smo prvobitno i podelili.

Datagrami se sastoje iz dva dela: **header** i **payload**. Payload je samo parče podataka koje želimo poslati, a header ima informacije koje definišu to parče, sadrže adresu i port uređaja kome se šalje, adresu pošiljaoca, redni broj paketa i još neke korisne informacije. Za kontrolu datagrama je potrebno prilično mnogo posla niskog nivoa, iz tog razloga postoje protokoli za komunikaciju (TCP, IP, UDP, ICMP itd.) i jedan apstraktan interfejs koji koristi te protokole i zove se **socket**.

Šta je soket?

Sa ugla programera, socket izgleda kao neka vrsta „priključka". Kada želimo ostvariti komunikaciju sa udaljenim serverom, mi jednostavno priključimo svoj socket sa socketom servera i kroz taj priključak se ostvaruje razmena podataka.



Nakon što „zakačimo" soket, komunikaciju tretiramo kao svaki drugi stream. Soket krije od programera sve što se dešava ispod soketa (može se reći i da štiti programera od svega ispod), a time i olakšava

njegov posao. Lokalni i udaljeni soketi koji su međusobno "priključeni" nose ime **soket par** koga karakterišu 4 podatka, IP i port lokalnog i IP i port udaljenog soketa.

Operativni sistem, mrežni protokoli i soketi

Operativni sistem mapira svaki soket sa odgovarajućim procesom ili nit koji vrši komunikaciju. Samim tim, soketi sadrže poseban identifikacioni broj koji poseduju i operativni sistem i aplikacija koja koristi soket. Soketi za komunikaciju koriste **socket adrese** koje se sastoje iz IP adrese i porta. Kada operativni sistem dobije datagram preko mreže, on iz header dela datagrama izvlači socket adresu i na osnovu toga payload deo datagrama prosleđuje odgovarajućoj mapiranoj aplikaciji.

Procesi koji pružaju servise nazivamo serverima i oni otvaraju sokete u stanju **slušanja**. Ovi soketi čekaju klijent aplikacije da im se jave kako bi ostvarili komunikaciju kao soket par.

Soket jedinstveno karakteriše kombinacija sledećih parametara:

- Local socket address – IP adresa i port u lokalu
- Remote socket address – koristi se samo za TCP sokete. TCP server može obsluživati više klijenata u istom trenutku koristeći istu lokalnu soket adresu, zato je potrebna i ova dodatna informacija
- Koji se transportni protokol koristi (na primer TCP ili UDP). S tim u vezi, TCP na portu 53 i UDP na portu 53 su različiti soketi

Za različite tipove protokola možemo izdvojiti različite vrste soketa. Za UDP protokol imamo **datagram sokete**, poznati i kao **soketi "bez konekcije"**. Paketi u takvom protokolu ne moraju stići u odgovarajućem redosledu, a dešava se i da ne stižu uopšte. Za protokole TCP ili SCTP se koriste **stream soketi** i to su soketi koji održavaju konekciju. **Raw soketi** (ili **raw IP soketi**) možemo videti u primeni kod mrežnih uređaja. Za raw sokete je karakteristično da zaobilaze transportni sloj. Primer je ICMP, poznatiji kao PING operacija. Pored ovih soketa, mogu se pomenuti i non-Internet soketi koji su implementirani u transportnom protokolu, ali se njima u ovom dokumentu nećemo baviti.

TCP i UDP serveri različito tretiraju soket konekcije.

- TCP serveri mogu obsluživati nekoliko klijenata istovremeno tako što se kreiraju različiti procesi za različite konekcije, a za svaku konekciju se pravi jedinstven soket. Za sokete sa uspostavljenom vezom kažemo da imaju *established* stanje. Server može napraviti više TCP soketa u *established* stanju na istom portu i lokalnoj IP adresi, svaki mapiran sa svojim klijentom. Operativni sistem ih tretira kao posebne sokete jer imaju različite adrese klijenata.
- UDP soketi ne mogu imati *established* stanje, jer UDP soketi nemaju održivu konekciju (to su soketi "bez konekcije"). Serveri koji koriste UDP protokol koriste isti proces za različite klijente, što je i razlog zašto ga identifikuje samo lokalna soket adresa, mada svaka poruka klijenta nosi adresu odakle dolazi.

Funkcionalnost soketa

Osnovne operacije koje se mogu vršiti nad soketima su:

- Konekcija sa udaljenom mašinom
- Slanje podataka
- Primanje podataka
- Zatvaranje konekcije
- Vezivanje za port (bind)
- Prisluškivanje dolazećih podataka
- Prihvatanje konekcija

Prve četiri koriste klijent i server, a ostatak koristi samo server.

Primeri korišćenja soketa u javi

Kao primeri soket komunikacije prvo su dati najjednostavniji mogući primeri sa kratkim pozivima kako bi se videla od prilike slika kako komuniciraju server i klijent. Nakon toga slede komplikovaniji primeri.

Datagram soketi (UDP)

Datagram soketi koriste UDP protokol. Server samo čeka datagrame a klijent ih šalje. Imamo klase klijenta i servera, server ima sledeće linije:

```
byte[] buffer = new byte[256];

DatagramSocket socket = new DatagramSocket(1234);

DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

socket.receive(packet);

for(byte b: packet.getData()) System.out.print((char) b);
```

buffer je mesto gde će se sadržaj paketa upisivati (payload) i on je tipa niza **byte**-ova. **DatagramSocket** je klasa koja simulira socket za UDP protokol, za server je dovoljno da se inicijalizuje sa odgovarajućim portom (druga linija), tu će soket prihvatati klijente. Objekat **packet** tipa **DatagramPaket** je paket koji se šalje ili prima, a u našem slučaju će nam služiti za primanje paketa od klijenta. Potrebno mu je dodeliti buffer tipa niza byte-ova kao i dužinu tog niza. Komandom **socket.recieve(packet)** dajemo instrukciju soketu da čeka klijenta na datagram. Kada klijent pošalje datagram, on se upisuje u packet, njegove podatke dobijamo kroz **packet.getData()**, što ćemo videti kod klijenta. Obratiti pažnju na **IOException** koji moramo uhvatiti.

Klijent ima sledeće linije:

```
DatagramSocket socket = new DatagramSocket();

DatagramPacket packet = new DatagramPacket(new
byte[]{'h','e','l','l','o','!'}, 6,InetAddress.getByName("localhost"),1234);

socket.send(packet);
```

U prvoj liniji pravimo objekat **socket** tipa **DatagramSocket**, ali za razliku od servera, mi ne navodimo port. Praznim konstruktorom se soket inicijalizuje na bilo kom slobodnom portu. Takođe, **packet** tipa **DatagramPacket** se razlikuje od paketa kod servera, jer pored bafera navodimo adresu i port gde šaljemo**. InetAddress.getByName(string name)** je statičan metod koji od hostname-a generiše IP adresu u string formatu. Na kraju preko **socket.send(packet)** dajemo instrukciju soketu da pošalje paket, adresa i port gde se šalje će biti prepoznato od strane operativnog sistema. Ovde takođe hvatamo **IOException**.

Stream soketi (TCP)

TCP soketi se ponašaju malo drugačije. Opet ćemo predstaviti primer kroz komunikaciju klijent/server. Sledi server deo:

```
byte [] buffer = new byte[256];

ServerSocket serverSocket = new ServerSocket(1234);

Socket socket = serverSocket.accept();

Thread.sleep(5000);

socket.getInputStream().read(buffer);

for(byte b: buffer){

     System.out.print((char) b);

}

socket.close();
```

Prvo kreiramo **buffer** tipa **byte** dužine 256 koji će nam biti prostor za primljene podatke. U drugoj liniji vidimo kako se kreira objekat **serverSocket** tipa **ServerSocket**. Kao argument mu šaljemo port na kome slušamo klijente. Objekat **socket** tipa **Socket** je klijentski socket i njega dobijamo kroz metodu **serverSocket.accept().** To je blokirajuća metoda i izvršava se tek kada se neki klijent javi. Kada se klijent javi, poželjno je malo pričekati dok klijent ne napiše nešto u soket i ta informacija ne stigne kao stream serveru, to se rešava kroz **Thread.Sleep(5000),** što znači da će server čekati 5 sekundi.

Objekat soketa ima svoje ulazne i izlazne strimove kojima možemo pristupiti kroz metode **getInputStream()** i **getOutputStream**. Mi smo u ovom primeru jednostavno pročitali input stream u **buffer** i nakon toga ga ispisali. Kada završimo sa komunikacijom, dobro je zatvoriti soket (**socket.close()**). Potrebno je uhvatiti **IOException**, a i za **Thread.Sleep()** se mora hvatati **InterruptedException**.

Klijentska klasa ima sledeće bitne komande:

```
Socket socket = new Socket(InetAddress.getByName("localhost"), 1234);

socket.getOutputStream().write(new byte[]{'h','e','l','l','o','!'});

socket.close();
```

U prvoj liniji kreiramo **socket** sa adresom servera i njegovim portom. U tom trenutku smo konektovani na server, a ako nismo, baciće se **IOException**. U slučaju da smo konektovani, možemo slati podatke, to radimo tako što pišemo podatke u output stream soketa, što se vidi u drugoj liniji. Kada završimo komunikaciju sa serverom, poželjno je zatvoriti soket sa komandom **socket.close()** . Moramo hvatati **IOException**.

Ovi primeri pokazuju samo skromno korišćenje soketa, ali je daleko od većine što soketi mogu uraditi. U narednim aplikacijama će se demonstrirati rad soketa na realnijem primeru.