

Internet

Autori:
Goran Savić
Milan Segedinac

1. Konkurentno programiranje

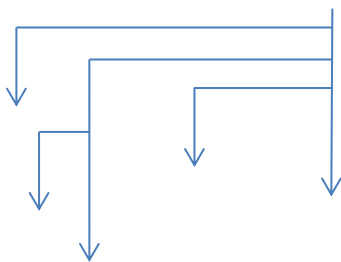
Ovde ćemo opisati principe i tehnologije koji se tiču računarskih mreža, mrežnih aplikacija i interneta. Pre nego što pređemo na ovu centralnu temu ovog dela teksta, upoznaćemo se sa konkurentnim programiranjem kao programskom paradigmom na koju se funkcionisanje mrežnih aplikacija često oslanja.

Konkurentno programiranje je način pisanja i izvršavanja programa tako da se program sastoji od više tokova izvršavanja koji se izvršavaju konkurentno. Ove tokove izvršavanja možemo posmatrati kao posebne programe uz naglasak da, za razliku od potpuno odvojenih programa, svi tokovi izvršavanja jedne aplikacije mogu da pristupaju zajedničkim memorijskim lokacijama i na taj način komuniciraju. Jedan tok izvršavanja zovemo **nit** (eng. *thread*).

Programi sa kojima smo se do sada susretali su bili sekvencijalni. Pri izvršavanju programa, nakon izvršavanja određene instrukcije, izvršavala se instrukcija koja sledi u tom toku programa. Kod konkurentnog programa, svaka nit ima svoj tok. Svaka nit izvršava klasičan programski kod. Sekvencijalni program možemo da posmatramo kao specijalan slučaj konkurentnog programa koji ima samo jednu nit. Razlika konkurentnog programa u odnosu na sekvencijalni je ilustrovana na slici.



sekvencijalni program



konkurentni program

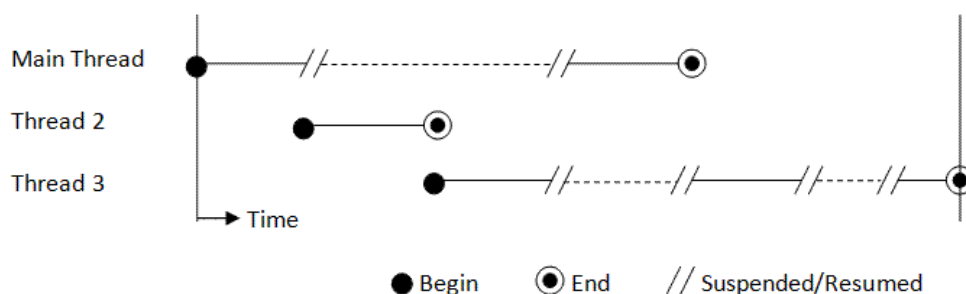
Kao što vidimo, konkurentni program sadrži više tokova izvršavanja. Svaki tok izvršavanja može da kreira nove tokove izvršavanja. Nema prepreke da različite niti izvršavaju isti programski kod. Tada imamo različite niti koje izvršavaju isto ponašanje. Ovo je slično kao kad pokrenemo više instanci iste aplikacije. Programski kod svih instanci je isti, ali su trenutno stanje i interni podaci drugačiji.

Na nivou jednog jezgra procesora, u jednom trenutku može da se izvršava samo jedna instrukcija. Pri izvršavanju konkurentnog programa, procesor izvršava određen broj instrukcija jedne niti, nakon čega prelazi na izvršavanje instrukcija druge niti i tako redom. Dakle, niti se ne izvršavaju istovremeno, nego konkurentno (konkurišu za procesor, pri čemu u svakom trenutku jedna nit dobija procesor). Ako je reč o procesoru sa više jezgara, tada je moguće da se u istom trenutku na različitim jezgrima izvršavaju različite niti programa. U tom slučaju, reč je o **paralelizaciji**. Ne treba gubiti iz vida da, zajedno sa nitima aplikacije, za procesor konkurišu i sve druge aplikacije koje se u tom trenutku izvršavaju. Trenutak dobijanja procesora, kao i period na koji je procesor dodeljen niti, je promenljiv, tako da je scenario preplitanja niti **nedeterministički**. Pri svakom izvršavanju aplikacije, može se desiti različit scenario preplitanja niti.

Konkurentno programiranje pruža dve osnovne prednosti. Kao prvo, moguće je razdvojiti logički odvojene delove programa tako da se tretiraju i izvršavaju praktično kao odvojeni programi, a da i dalje pripadaju istoj aplikaciji i da imaju pristup istim resursima, ukoliko je potrebno. Kao drugo, moguće je dobiti ubrzanje programa. Ako je moguće izvršiti paralelizaciju programa tada se različite

Internet

niti programa izvršavaju istovremeno, pa je očigledno da dolazi do ubrzanja izvršavanja u odnosu na scenario u kojem bi se isti kod izvršio sekvencijalno. Ubrzanje se može dobiti čak i kada se ne vrši paralelizacija izvršavanja. Naime, nit pri izvršavanju jedan deo vremena mora da provede blokirana čekajući na završetak operacija koje izvršavaju ulazno-izlazni uređaji. Na primer, ako je programski kod koji nit izvršava zatražio upis u fajl, ovu operaciju mora da izvede disk kao periferni uređaj. Periferni uređaji su znatno sporiji od procesora, tako da bi procesor morao da čeka besposlen dok se ne završi operacija na uređaju. Umesto toga, procesor može da za to vreme pređe na izvršavanje druge niti. Time se dobija na brzini, jer bi sekvencijalni program bio u blokadi za vreme izvršavanja ulazno-izlazne operacije i ne bi mogao da za to vreme pređe na drugu nit izvršavanja. Rad druge niti dok je jedna nit blokirana je ilustrovan na slici.



* preuzeto sa www3.ntu.edu.sg

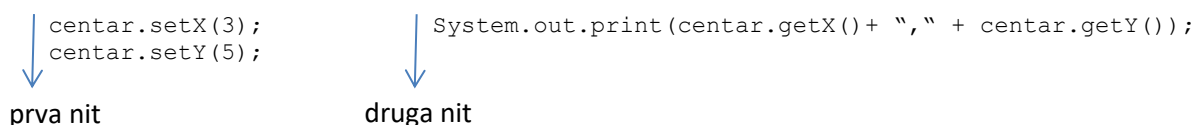
Vidimo da se dok je jedan deo programa blokirana, izvršava neki drugi deo organizovan u posebnu nit. Na ovaj način program se brže izvršio u odnosu na to da je postojala samo jedna nit u aplikaciji. U toj varijanti dužina izvršavanja programa bi bila jednaka zbiru dužina izvršavanja svakog pojedinačnog toka, jer procesor ne bi mogao da iskoristi blokiranost jednog dela programa za prelazak na izvršavanje nekog drugog dela.

Primena konkurentnog programiranja je raznovrsna. Najbolju primenu nalazi u scenarijima u kojima je moguće kreirati labavo povezane tokove programa takve da jedan tok može da se izvršava dok se čekaju uslovi za izvršavanje drugih tokova. Na primer, u editoru teksta, funkcionalnost automatske provere slovne i gramatičke ispravnosti teksta je moguće izdvojiti u posebnu nit, tako da interakcija sa korisnikom (koja je implementirana u glavnoj niti) ne bude nikad potpuno blokirana dok traje provera. Takođe, koristi se i da bi se dobilo na brzini raspoređivanjem različitih niti za istovremeno izvršavanje na različitim procesorskim jezgrima. Čak i kada nije moguće zbog hardverskih ograničenja dobiti na brzini, konkurentno programiranje obezbeđuje ravnopravan tretman različitih tokova izvršavanja. Na primer, ako govorimo o obradi korisničkih zahteva u internet aplikaciji, obrada zahteva svakog korisnika u posebnoj niti omogućuje ili ravnopravan tretman svih zahteva ili ubrzanje kroz istovremeno raspoređivanje na procesorska jezgra.

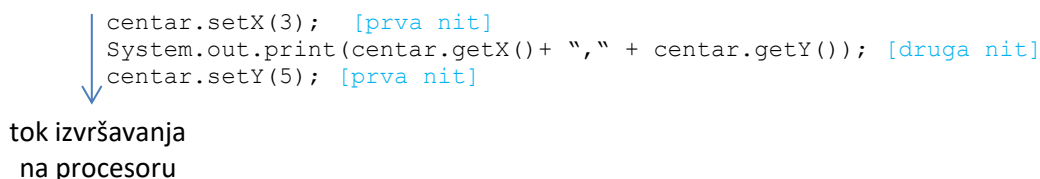
Pomenuli smo da niti jedne aplikacije mogu da pristupaju istim resursima. Na nivou programskog koda to bi značilo da različite niti mogu da imaju pristup istoj promenljivoj. Ovo omogućuje saradnju između niti, ali otvara i značajne probleme. Scenario preplitanja različitih niti pri izvršavanju nije predvidiv. Nit može da izgubi procesor nakon izvršavanja bilo koje instrukcije. To može dovesti do scenarija koji se naziva **štetno preplitanje**. Ilustrujmo ga na jednom primeru. Recimo da program vrši obradu pozicije kruga i da evidentira koordinatu centra kruga. Neka je trenutni centar kruga u tački (2, 4). Jedna nit treba da izvrši pomeranje kruga tako da centar bude na koordinati (3, 5). Druga nit

Internet

zadužena je za ispis koordinate centra. Pogledajmo programski kod ove dve niti u programskom jeziku Java, ako je *centar* objekat koji predstavlja centar kruga. Njegovim get i set metodama se postavljaju, odnosno preuzimaju x i y koordinate.



Objekat *centar* je deljena promenljiva kojoj obe niti imaju pristup. Ako procesor na kojem se prikazani program izvršava ima jedno jezgro, u svakom trenutku će se izvršavati samo jedna od niti na procesoru. Pogledajmo sledeći scenario preplitanja, koji je ilustrovan na slici.



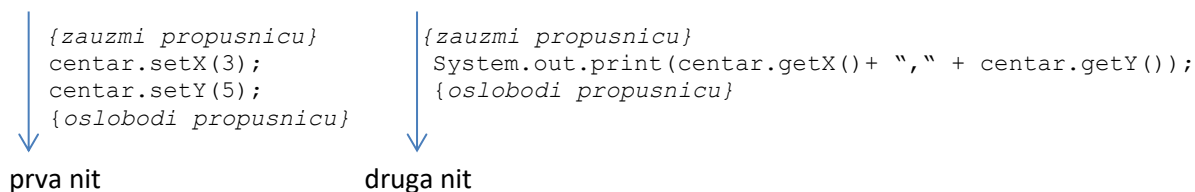
Dakle, procesor je izvršio jednu naredbu prve niti, nakon toga je prešao na izvršavanje naredbe druge niti, pa je tek na kraju izvršio i drugu naredbu prve niti. Naglasimo da svaka od Java naredbi podrazumeva veći broj procesorskih instrukcija, ali je ovde zbog jednostavnosti prikazan scenario u kojem je svaka od niti gubila procesor tačno nakon izvršavanja svih instrukcija od kojih se sastoji jedna Java naredba. U prikazanom scenariju, program kao koordinatu centra kruga ispisuje (3, 4) što je neispravna koordinata na kojoj se centar kruga nije nikada nalazio. Do neispravnog rada programa je došlo zbog štetnog preplitanja između niti programa.

Na sam scenario preplitanja ne možemo uticati i on nije programeru predvidiv jer zavisi od drugih programa koji se izvršavaju na procesoru, brzine i dostupnosti perifernih uređaja itd. To što je scenario preplitanja niti nedeterministički, i dalje ne znači da moramo da dobijemo program koji se ponaša nedeterministički. Konkurentno programiranje ne bi imalo smisla ako ne bismo imali mehanizme da obezbedimo ispravnost rada programa uz istovremeno korišćenje prednosti koje nam donosi konkurentno izvršavanje.

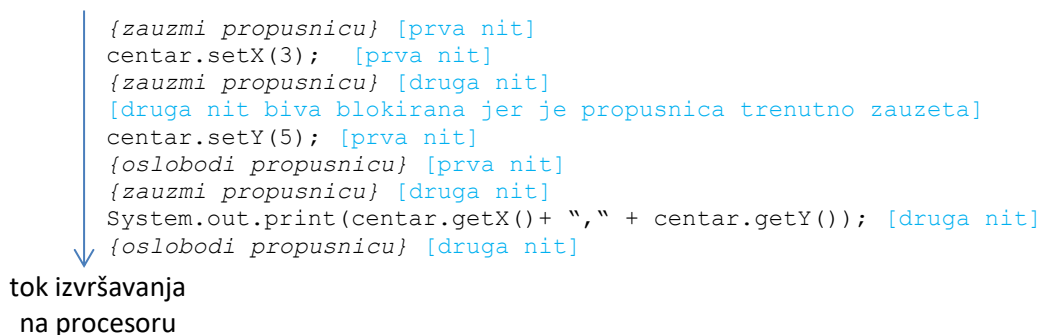
Problem štetnog preplitanja se rešava zabranom preplitanja pri izvršavanju delova koda u kojima preplitanje može da bude štetno. Ovakve delove koda nazivamo **kritična sekcija**. Program zabranjuje štetno preplitanje time što nit mora da zauzme deljenu propusnicu za ulazak u kritičnu sekciju. Kada jedna nit zauzme propusnicu i krene da izvršava programski kod kritične sekcije, druga nit čak i da dobije procesor neće moći da zauzme već zauzetu propusnicu, već će morati da sačeka da nit koja drži propusnicu završi rad u kritičnoj sekciji i oslobodi propusnicu. Ovo nazivamo **međusobna isključivost** niti pri pristupu deljenim resursima. Kritičnu sekciju u koju je ulaz zaštićen putem propusnice nazivamo **isključivi region**. Treba imati u vidu da se u okviru isključivog regiona izvršavanje programa sekvencionalizuje, što utiče na performanse. Iz tog razloga, trajanje isključivog regiona treba da bude minimalno. Potrebno je identifikovati koji delovi koda su zaista kritične sekcije i samo njih definisati kao isključive regione.

Internet

U prethodnom primeru kritična sekcija je bio deo koda sa postavljanjem novih x i y koordinata centra, kao i preuzimanje ovih koordinata pri ispisu. U nastavku je ponovo ilustrovan prethodni primer, ali ovog puta korišćenjem propusnice za ulazak u kritičnu sekciju.



Naredna slika ilustruje kako bi se program izvršavao da se desi isti scenario preplitanja kao u prethodnom primeru.



Vidimo da je druga nit bila blokirana jer nije bilo uslova da nastavi sa radom. Naime, propusnica koja joj je neophodna za rad je u tom trenutku bila zauzeta. Tako da će prva nit ponovo dočekati da dobije procesor, završiti svoje operacije i osloboditi propusnicu i tek nakon toga postoje uslovi da druga nit preuzme propusnicu i izvrši svoj kod. Sada će druga nit ispisati koordinate (3, 5), što je ispravno ponašanje.

Međusobna isključivost nam je obezbedila da u jednom trenutku samo jedna nit može da pristupa kritičnoj sekciji. Kojim redosledom će niti pristupati resursima i dalje zavisi od scenarija preplitanja. Ako ponašanje jedne niti zavisi od drugih niti, moguće je međusobno sinhronizovati izvršavanje niti. Razjasnimo to na sledećem primeru. Ranije je objašnjeno da je za uklanjanje nekorišćenih objekata u *heap* memoriji u Java aplikaciji zadužen *garbage collector*. Programski kod koji predstavlja *garbage collector* JVM izvršava u posebnoj niti. Kada se *garbage collector* izvršava, JVM zaustavlja sve niti aplikacije i drži ih blokirane dok god *garbage collector* ne završi svoj posao. Ovo je neophodno da bi se čišćenje memorije uspešno realizovalo. Kada se čišćenje završi, niti aplikacije nastavljaju svoj rad. Dakle, rad niti aplikacije mora biti sinhronizovan sa radom *garbage collector* niti. Konkurentno programiranje standardno pruža podršku za sinhronizaciju niti. Moguće je blokirati nit zbog neispunjenosti uslova za dalji rad niti. Kada se uslovi ispune, šalje se obaveštenje niti ili nitima da mogu da nastave rad.

Konkurentno programiranje u Javi

Pogledajmo sada podršku za konkurentno programiranje u programskom jeziku Java. Niti su u Javi predstavljene objektima klase Thread, koja je deo Java biblioteke klasa. Preciznije, programski kod koji treba da se izvršava u okviru posebne niti piše se u klasi naslednici klase Thread. Neophodno je redefinisati metodu `run()` i u njoj napisati programski kod. Ovako kreirana nit se pokreće

Internet

instanciranjem objekta ove klase naslednice i pozivanjem metode *start()*. Ova metoda će kreirati novi tok izvršavanja i pozvati metodu *run()* čiji kod će se izvršavati u ovom novom toku izvršavanja. Pogledajmo primer kreiranja i startovanja niti u programskom jeziku Java.



```
class IspisNit extends Thread {  
    public void run() {  
        System.out.println("Ispis iz niti");  
    }  
}
```

```
IspisNit nit = new IspisNit();  
nit.start();  
System.out.println("Ispis iz main metode");
```

Donji deo koda u prikazanom primeru je isečak iz main metode u kojoj se nit kreira i startuje. Nakon poziva metode *start()* u aplikaciji će postojati dve niti. Prva nit je nastala pokretanjem aplikacije i ona izvršava main metodu. Druga nit izvršava programski kod napisan u *run()* metodi klase *IspisNit*. Ovo znači da ne možemo unapred znati da li će prvo biti izvršen ispis u main metodi ili ispis iz niti. Nakon kreiranja druge niti, dve niti konkurentno postoje u aplikaciji i konkurišu za procesor tako da nije predvidiv scenario preplitanja, pa samim tim ni redosled izvršavanja programa. Na nivou pojedinačne niti, jasno je da svaka nit sekvencijalno izvršava svoj kod.

Pomenimo i to da je nad niti moguće pozvati i metodu *join()*. Pozivom ove metode pozivaoc ostaje blokiran dok god nit nad kojom je metoda pozvana ne završi svoj rad. Ova metoda se koristi kada je potrebno dva odvojena toka izvršavanja opet spojiti u jedan. Standardan scenario korišćenja je u situacijama kada je jednoj niti za dalji rad potreban rezultat koji treba da obezbedi druga nit. U tom slučaju će se pozivom metode *join()* sačekati na završetak rada niti čiji je rezultat potreban.

Međusobna isključivost se u Javi obezbeđuje korišćenjem ključne reči *synchronized*. Putem ove ključne reči nit pokušava zauzimanje propusnice koja se odnosi na određeni objekat. Ukoliko je propusnica slobodna, nit će zauzeti propusnicu i imati ekskluzivan pristup bloku koda koji reč *synchronized* definiše. Ako druga nit pokuša da zauzme propusnicu za isti objekat, moći će to da učine tek kada nit koja trenutno drži propusnicu završi *synchronized* blok. Ilustrujmo ovo na ranije prikazanom primeru sa ekskluzivnim pristupom objektu koji predstavlja centar kruga. Dat je Java kod koji implementira zauzimanje propusnice.

<pre>synchronized(centar) { centar.setX(3); centar.setY(5); }</pre>	<pre>synchronized(centar) { System.out.print(centar.getX() + "," + centar.getY()); }</pre>
	
prva nit	druga nit

Kao parametar reči *synchronized* se prosleđuje objekat na koji se propusnica odnosi. Samo jedna nit će u jednom trenutku moći da izvršava *synchronized* blok koji se odnosi na isti objekat. Oslobođanje propusnice se vrši automatski završetkom *synchronized* bloka. Dakle, *synchronized* blok predstavlja isključivi region nad kojim je putem propusnice obezbeđena međusobna isključivost.

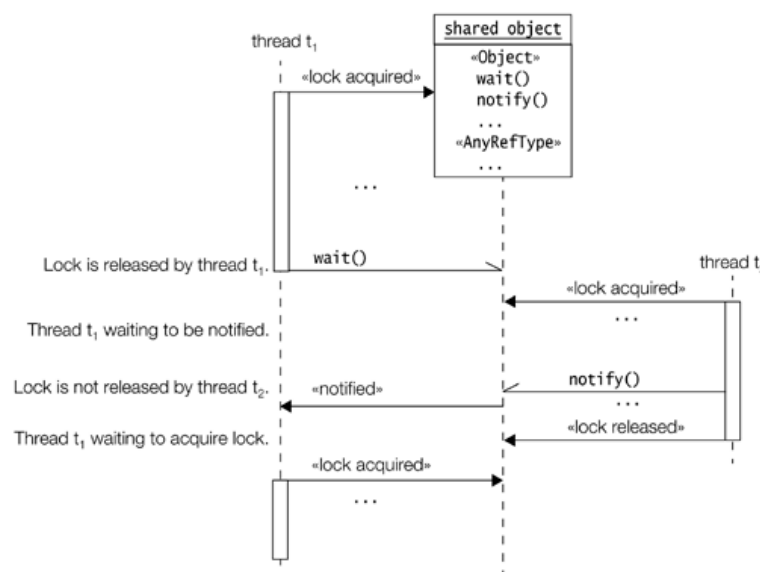
Internet

Ključna reč *synchronized* se može navesti i u zaglavlju metode. Tada ona označava da je kompletna metoda *synchronized* blok. Objekat na koji se propusnica odnosi je tada objekat nad kojim se metoda poziva. Ovo je ekvivalentno tome da se definiše *synchronized* blok koji se proteže na celu metodu i zauzima propusnicu koja se odnosi na objekat *this*. Dakle, ako posmatramo klasu koja modeluje centar kruga i njenu metodu za izmenu koordinate centra, dve prikazane metode klase realizuju istu funkcionalnost.

```
class Centar {
    private int x;
    private int y;
    public void pomeriCentarVerzija1(int novoX, int novoY) {
        synchronized(this) {
            this.x = novoX;
            this.y = novoY;
        }
    }

    public synchronized void pomeriCentarVerzija2(int novoX, int novoY) {
        this.x = novoX;
        this.y = novoY;
    }
}
```

Sinhronizacija između niti se u Javi vrši putem metoda *wait()*, *notify()* i *notifyAll()*. Ove metode su definisane u klasi *Object*, pa ih samim tim sadrži svaki Java objekat bez obzira na tip. Poziv metode *wait()* nad objektom ima za posledicu da se nit koja izvršava tu naredbu zaustavi. Nit će biti zaustavljena dok god neka druga nit nad istim objektom ne pozove metodu *notify()* ili *notifyAll()*. Među svim nitima koje su u blokadi, jer su nad objektom pozvale metodu *wait()*, poziv metode *notify()* će aktivirati jednu proizvoljnu nit. Sa druge strane, poziv metode *notifyAll()* će aktivirati sve niti koje su u blokadi jer su nad tim objektom pozvale metodu *wait()*. Naglašavamo da je metode *wait()*, *notify()* i *notifyAll()* nad objektom dozvoljeno pozivati samo iz *synchronized* bloka, dok trenutna nit drži propusnicu nad tim objektom. Rad metoda *wait()* i *notify()* je ilustrovan na slici.



Dakle, nit t_1 mora da zauzme propusnicu da bi pozvala metodu *wait()*. Nakon poziva metode *wait()*, nit prelazi u čekanje i ostaje u čekanju dok nit t_2 ne pozove metodu *notify()*. Treba primetiti da je za izlazak iz čekanja neophodno i da nit zauzme propusnicu. To je moguće tek kada druga nit propusnicu oslobodi.

Za kraj poglavlja o konkurentnosti u Javi, napomenimo još da za objekat koji garantuje ispravan rad u višenitnom okruženju kažemo da je *thread safe*. Ovakav objekat obezbeđuje međusobnu isključivost različitih niti koji pristupaju podacima objekta. Ovo treba imati u vidu kada koristimo objekat neke klase. Na primer, u Java Collections Framework koji sadrži klase koje se odnose na predstavljanje struktura podataka, samo neke od klasa su *thread safe*. Recimo, klasa *ArrayList* nije *thread safe*, a klasa *Vector* koja obezbeđuje praktično istu funkcionalnost jeste *thread safe*. Zavisno od scenarija, potrebno je koristiti odgovarajuću klasu. Ne treba gubiti iz vida da obezbeđivanje međusobne isključivosti uvodi dodatni kod u realizaciju (npr. zauzimanje i oslobađanje propusnice). Zato je u situacijama u kojima ne postoji opasnost od štetnog preplitanja bolje koristiti klase koje nisu *thread safe*.

2. Mrežna aplikacija

U ovom poglavlju ćemo se upoznati sa mrežnim aplikacijama. Sa ovakvim tipom aplikacije smo se već sreli. Setimo se rada sa bazama podataka i sistema za upravljanje bazom podataka. Kao SUBP smo koristili MySQL aplikaciju. Za komunikaciju sa MySQL SUBP-om smo koristili MySQL Workbench kao specijalizovanu aplikaciju. Pomenuli smo da ove dve aplikacije ne moraju biti na istom računaru, već je dovoljno da mogu da razmenjuju podatke putem računarske mreže. Aplikacija koja podatke može da šalje ili preuzima sa drugih računara se naziva **mrežna aplikacija**. Za razliku od mrežnih aplikacija, *stand-alone* aplikacije za rad ne moraju biti povezane na računarsku mrežu jer upravljaju podacima koji se nalaze na istom računaru kao i aplikacija.

Mrežne aplikacije su najčešće dizajnirane prema klijent-server arhitekturi. U ovakvoj arhitekturi, serverska aplikacija izvršava većinu funkcionalnosti i putem mreže obezbeđuje servise klijentskoj aplikaciji. Klijentska aplikacija je najčešće zadužena za interakciju sa korisnikom, slanje korisnikovih zahteva serveru i prikaz podataka koje server klijentu pošalje kao odgovor. Primer sa bazom podataka takođe prati ovakvu arhitekturu. MySQL SUBP je serverska aplikacija. Iz tog razloga se naziva i MySQL server. Funkcionalnosti skladištenja podataka, kao i obrade i izvršavanja SQL upita su odgovornost ove aplikacije. MySQL Workbench je klijentska aplikacija, koja je zadužena za interakciju sa korisnikom, slanje zahteva serveru i prijem odgovora od servera. Dakle, svi SQL upiti se izvršavaju na serveru, a MySQL Workbench obezbeđuje editor teksta za unos komandi koje zatim prosleđuje serveru koji ih izvršava.

Za transfer podataka između serverske i klijentske aplikacije ključno je rutiranje. Rutiranje u računarskoj mreži predstavlja proces izbora putanje od izvora podataka do odredišta. Za rutiranje su zaduženi specijalizovani hardverski uređaji koji se najčešće imenuju zajedničkim nazivom ruteri. Za realizaciju rutiranja je neophodno da cilj i odredište budu jednoznačno identifikovani, kao i da podaci koji se prenose sadrže informacije o putanji prenosa. Da bi aplikacije mogle da komuniciraju putem mreže neophodno je da postoji dogovor o načinu reprezentacije ovih informacija. Skup pravila koji obezbeđuje komunikaciju sistema putem razmene podataka nazivamo **protokol komunikacije**. Generalno govoreći, protokol može imati proizvoljnu formu. Važno je jedino da svi učesnici u

Internet

komunikaciji poznaju protokol. Ipak, kako bi se podržala jednostavna komunikacija raznovrsnih aplikacija, postoje standardni protokoli komunikacije.

Internet komunikacija se zasniva na **TCP/IP** protokolu. Ovaj protokol se koristi i za komunikaciju računara u lokalnoj računarskoj mreži (*local area network* – LAN). Kao što mu i ime kaže, ovaj protokol se sastoji od dva sloja. TCP sloj definiše format paketa koji se razmenjuju između učesnika u komunikaciji. Paket sadrži delove poruke koja se razmenjuje, kao i dodatne informacije neophodne za uspešan prenos podataka. IP (Internet Protocol) sloj specificira format adrese izvora i odredišta. Prema nazivu samog protokola, ove adrese se nazivaju **IP adrese**. Format adrese je takav da se sastoji od 32 bita organizovanih u 4 dela sa po 8 bitova. Ove delove nazivamo okteti. Zbog preglednijeg zapisa, oktete zapisujemo u decimalnom obliku i razdvajamo tačkom. Tako se IP adresa zapisana kao 192.168.0.12 pri prenosu podataka specificira kao 11000000 10101000 00000000 00001100. Obzirom da oktet sadrži 8 bita, najveća vrednost koju jedan oktet može da predstavlja je kada svih 8 bitova ima vrednost 1, a to je decimalni broj 255.

IP adresa identifikuje samo uređaj koji učestvuje komunikaciji. Izvor, odnosno odredište nisu određeni samo uređajem koji vrši komunikaciju. Izvor i odredište su potpuno definisani tačkom komunikacije (eng. *end point*). Tačka komunikacije je određena ne samo adresom uređaja, nego i portom na uređaju kroz koji se komunikacija odvija. Port je softverska apstrakcija koja identifikuje jedan tok komunikacije na uređaju. Jedan uređaj sadrži veći broj portova. Svaki od portova se koristi da bi neka od aplikacija primala ili slala podatke okruženju kroz taj port. Port je identifikovan svojim rednim brojem. Protokol predviđa 16 bitova za specifikaciju porta, što znači da su redni brojevi u rang od 0 do 65536. Port se najčešće zapisuje u nastavku zapisa IP adrese nakon znaka dve tačke. Na primer ako se sa računarom na adresi iz prethodnog primera komunicira na portu 8080, to se zapisuje kao 192.168.0.12:8080. Par koji čine IP adresa i port potpuno adresira izvor, odnosno odredište i naziva se **soket** (eng. *socket*). Adresiranje računara putem IP adrese i porta je prikazano na slici.



* preuzeto sa <http://telescript.denayer.wenk.be>

3. Mrežno programiranje

U ovom poglavlju ćemo prikazati kako se u programskom jeziku Java mogu implementirati mrežne aplikacije. Za uspostavljanje komunikacije između dve aplikacije, neophodno je da definišemo IP adrese računara na kojima se aplikacije izvršavaju, kao i portove na tim računarima kroz koje

Internet

aplikacije šalju, odnosno primaju podatke. Obzirom da IP adresa i port predstavljaju jedan soket, komunikacija je definisana parom soketa.

Java biblioteka klasa sadrži klase koje obezbeđuju podršku za mrežno programiranje. IP adresa je predstavljena klasom `InetAddress`. Dat je primer dobijanja objekta ove klase, ako je IP adresa `192.168.0.12`.

```
InetAddress address = InetAddress.getByName("192.168.0.12");
```

Soketi su u Javi predstavljeni klasom `Socket`. Moramo naglasiti da klasa `Socket` u Javi predstavlja par soketa. Dakle, jedan objekat klase `Socket` čuva informaciju o adresi i portu računara na kojem se aplikacija izvršava, ali i adresi i portu računara sa kojim se komunicira. Primer ilustruje kreiranje soketa putem kojeg će se vršiti komunikacija sa računarom na adresi `192.168.0.12`, pri čemu taj računar koristi port `9100` za komunikaciju.

```
Socket sock = new Socket(address, 9100);
```

Parametri navedeni u konstruktoru klase `Socket` se odnose na udaljeni računar sa kojim uspostavljamo komunikaciju. Lokalna adresa i lokalni port, koji su takođe uskladišteni u objektu klase `Socket`, će biti određeni automatski. Kada je bilo reči o fajlovima, objasnili smo da Java koristi ulazno-izlazne tokove kao uniforman način pristupa izvorima i destinacijama podataka. I razmena podataka sa udaljenim računarom koristi ulazno-izlazne tokove, pa je time ova komunikacija principijelno ista kao kod učitavanja, odnosno pisanja podataka u fajl. Neophodno je definisati ulazni, odnosno izlazni tok ka udaljenom računar. Dat je primer koda uspostavljanja ova dva toka.

```
BufferedReader in = new BufferedReader(new InputStreamReader(  
    sock.getInputStream()));
```

```
PrintWriter out = new PrintWriter(new BufferedWriter(  
    new OutputStreamWriter(sock.getOutputStream())), true);
```

Treba primetiti da se tok podataka uspostavlja preuzimanjem ulaznog, odnosno izlaznog toka iz objekta `sock` klase `Socket`.

Kada postoje definisani tokovi, moguće je slati podatke udaljenom računar upisom u izlazni tok, kao i primati podatke od udaljenog računara preuzimanjem podataka iz ulaznog toka. Dat je primer slanja i preuzimanja podataka.

```
out.println("Poruka za udaljeni računar");  
String response = in.readLine();
```

Kao i kod korišćenja tokova u radu sa fajlovima, i ovde je potrebno nakon završetka komunikacije zatvoriti izlazni i ulazni tok. Takođe, potrebno je zatvoriti sam soket.

```
out.close();  
in.close();  
sock.close();
```

Internet

TCP protokol predviđa da jedna strana u komunikaciji inicira komunikaciju, dok druga strana čeka da komunikacija bude inicirana. U klijent-server arhitekturi, klijent je taj koji komunikaciju inicira. Na nivou Java koda, klijent komunikaciju može inicirati otvaranjem soketa i uspostavljanjem tokova podataka ka serveru kao što je prikazano u prethodnim primerima koda. Serverska aplikacija očekuje klijentove zahteve i nakon što se klijent obrati uspostavlja soket i tokove ka klijentu. Ne treba gubiti iz vida da se klijentska i serverska aplikacija izvršavaju na različitim računarima i da komuniciraju razmenom podataka putem mreže.

Klasa `ServerSocket` omogućuje čekanje klijentovih zahteva na serverskoj strani. Odgovarajući kod je dat u primeru.

```
ServerSocket ss = new ServerSocket(9100);  
Socket sock = ss.accept();
```

Metodom `accept()` serverska strana očekuje obraćanje klijenta. Ova metoda blokira trenutni tok izvršavanja sve dok klijent ne inicira uspostavljanje komunikacije. Kada se to desi, metoda `accept()` vraća novi `Socket` objekat koji reprezentuje komunikaciju sa tim klijentom. Dakle, prikazani `sock` objekat će kao lokalnu adresu sadržati adresu serverskog računara, a kao lokalni port vrednost 9100. Adresa udaljenog računara će biti adresa računara na kojem se izvršava klijentska aplikacija, a port na tom računaru će biti redni broj porta kroz koji klijent šalje i prima podatke ka serveru. Taj port je automatski određen ako je klijent kreirao `Socket` kao u ranije prikazanom primeru koda. Pri ovakvom protokolu komunikacije jasno je da serverska aplikacija mora biti već pokrenuta kada klijent pokuša da joj se obrati.

Kada na serverskoj strani postoji `Socket` objekat koji reprezentuje komunikaciju, potrebno je i da server kreira tokove podataka kroz taj soket. Ovo se vrši na isti način kao i za ranije prikazani primer kod soketa na klijentskom računaru. Server sada iz ulaznog toka može preuzeti zahtev koji je klijent poslao i u izlazni tok poslati odgovarajući odgovor. Ovo je prikazano u primeru.

```
String request = in.readLine();  
out.println("Odgovor klijentu");
```

Na serversku aplikaciju se može povezati veći broj klijenata. Tada serverska aplikacija treba da ima petlju u kojoj opslužuje klijente. Sekvencijalno opsluživanje svih klijenata imalo bi za posledicu da klijent mora da sačeka odgovor dok zahtevi drugih klijenata budu potpuno obrađeni. Umesto toga, moguće je kreirati posebnu nit za obradu svakog zahteva. Na taj način se klijentski zahtevi obavljaju konkurentno. To obezbeđuje ravnopravnost u tretiranju klijenata, kao i ubrzanje performansi bilo kroz paralelizaciju izvršavanja, kao i prelaskom na obradu druge niti dok jedna nit komunicira sa ulazno-izlaznim uređajima. U nastavku je prikazan deo koda serverske aplikacije koja opslužuje svaki zahtev u posebnoj niti.

```
ServerSocket ss = new ServerSocket(9100);  
  
while (true) {  
    Socket sock = ss.accept();  
    ServerThread st = new ServerThread(sock);  
    st.start();  
}
```

Primetimo da svaka nit kao parametar dobija objekat klase Socket koji reprezentuje komunikaciju sa klijentom za kojeg je ta nit zadužena. Sada je nit zadužena da uspostavi tokove podataka ka klijentu, preuzme zahtev klijenta i vrati odgovarajući odgovor.

4. Internet

Internet je globalni sistem umreženih računara koji koriste TCP/IP protokol za komunikaciju. Putem interneta računari međusobno pružaju i koriste različite servise kao što su World Wide Web, elektronska pošta, direktno povezivanje računara (*peer-to-peer* veze) itd.

Najpopularnija primena interneta je za korišćenje World Wide Web servisa. World Wide Web (WWW) definiše način predstavljanja, prenosa i korišćenja informacija kojima se pristupa putem interneta. Često se internet i WWW pogrešno koriste kao sinonimi. Internet predstavlja globalnu računarsku mrežu organizovanu prema određenim pravilima, dok WWW specificira jedan način rada sa informacijama koje se skladište i obrađuju na računarima uvezanim u internet računarsku mrežu.

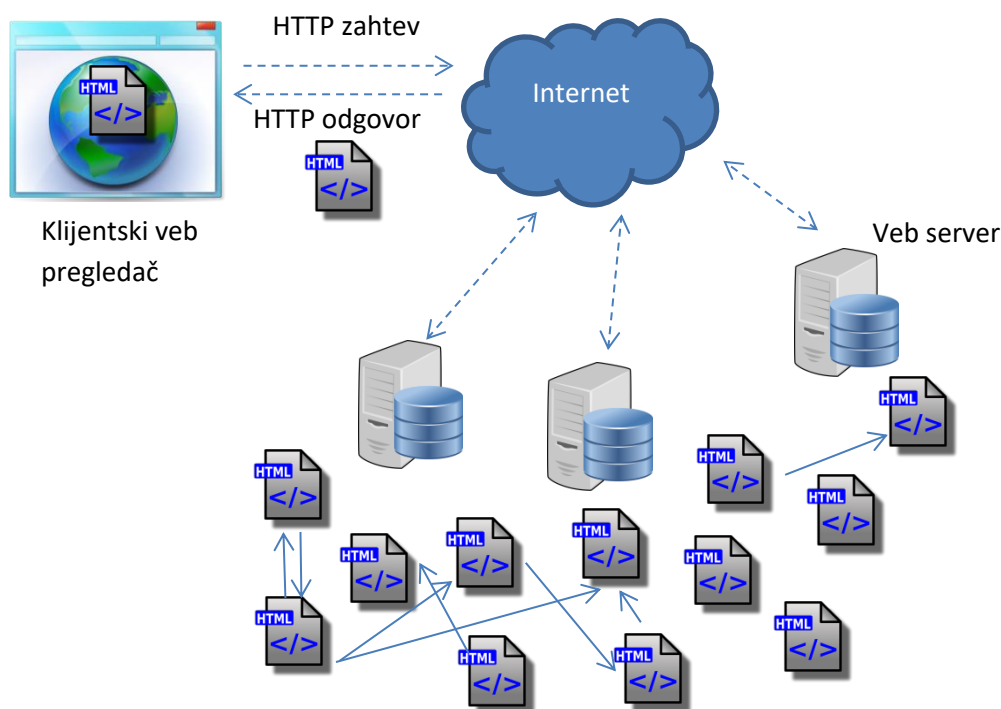
WWW je osmislio Tim Berners-Lee krajem osamdesetih godina prošlog veka na institutu CERN u Švajcarskoj. WWW specificira tri glavna principa pri upravljanju podacima:

- Podaci se nalaze u resursima koji imaju globalno jedinstvene identifikatore (Uniform Resource Identifier – URI) i mogu biti locirani navođenjem globalno jedinstvenog lokatora (Uniform Resource Locator – URL)
- Podaci su predstavljeni putem jezika HyperText Markup Language (HTML)
- Protokol za razmenu informacija je Hypertext Transfer Protocol (HTTP)

Osnovna ideja je da računari na internetu skladište dokumente koji mogu biti jednoznačno locirani i koji mogu da referenciraju druge dokumente, koristeći globalno jedinstvene identifikatore. Informacije u dokumentima su opisane na standardizovan način korišćenjem HTML jezika. Zahtevi za dokumentima, kao i odgovori na te zahteve šalju se po HTTP protokolu kao dogovorenom načinu komunikacije bez obzira na tip zahteva ili dokumenta.

Time je WWW standardizovao razmenu i interpretaciju podataka na internetu. Bilo koji računar koji ima aplikaciju koja prima zahteve i šalje odgovore po HTTP protokolu može biti veb server kojem se opet može obratiti bilo koji računar koji poštuje HTTP protokol. Dobijene informacije se mogu prezentovati u bilo kojoj aplikaciji koja može da prikaže HTML sadržaj. Ovaj tip aplikacija zovemo veb pregledač (eng. *web browser*). Osim što je specificirao način funkcionisanja World Wide Web, Tim Berners-Lee je implementirao i prvi HTTP veb server i prvi veb pregledač, čime je počela primena WWW ideje.

Na slici je ilustrovan inicijalni koncept World Wide Web. Vremenom je značajno evoluirao uvodeći podršku za druge tipove resursa, kao i za druge akcije nad resursima osim samog preuzimanja.



5. Http protokol

U prethodnom poglavlju smo naveli da World Wide Web podrazumeva korišćenje HTTP protokola za komunikaciju kroz internet infrastrukturu. Obzirom da smo naveli i da se internet komunikacija zasniva na TCP/IP protokolu, da ne bi bilo zabune, razjasnimo prvo odnos ova dva protokola. TCP/IP je protokol nižeg nivoa i specificira način transporta podataka putem organizacije podataka u pakete. Struktura paketa je definisana TCP/IP protokolom. Svaki paket sadrži samu poruku koju želimo da isporučimo primaocu, ali i zaglavlje paketa koje nosi informacije neophodne za uspešan prenos (npr. port primaoca). Protokoli nižeg nivoa su neutralni u odnosu na značenje poruke koja se prenosi. HTTP protokol stoji na višem nivou hijerarhije i bavi se informacijama sadržanim u samoj poruci koja se šalje drugom učesniku u komunikaciji. Ovaj protokol specificira koje sve informacije poruka mora da nosi i kako one moraju da budu strukturirane da bi druga korisnička aplikacija mogla da razume značenje poruke. Dakle, HTTP kao protokol aplikativnog nivoa specificira informacije koje druga korisnička aplikacija tumači, a ne mrežni uređaji i sistemski softver kao kod protokola nižeg nivoa.

U ovom poglavlju ćemo predstaviti više detalja o HTTP protokolu. Ovaj protokol specificira strukturu klijentovog zahteva i odgovora servera u komunikaciji između dva računara po klijent-server principu putem interneta. Zahtevi omogućuju sprovođenje akcija nad resursima identifikovanim putem URI-ja kako predviđa World Wide Web.

Internet

HTTP protokol predviđa da se svaki novi par zahtev-odgovor nezavisno tretira. Zato kažemo da je HTTP protokol *stateless*. Veb server u slučaju višestrukih zahteva od strane istog korisnika, svaki zahtev obrađuje nezvezano za prethodni. Dakle, veb server ne čuva informacije o korisniku koji je uputio zahtev, kako bi te informacije iskoristio pri obradi narednog zahteva. Ne treba mešati čuvanje stanja o korisniku koji je uputio zahtev sa stanjem podataka na serveru. Svakako da server skladišti različite podatke i da se stanje tih podataka može menjati pod uticajem zahteva klijenta.

HTTP je tekst bazirani protokol, što znači da je sadržaj poruke dat u formatu čitljivom za čoveka. Pogledajmo kako prema HTTP protokolu moraju da izgledaju zahtev i odgovor u komunikaciji.

Zahtev mora da sadrži sledeće linije:

- Prva linija navodi tip operacije nad resursom, resurs nad kojim se operacija vrši i verziju protokola
- Zaglavlje zahteva – niz dodatnih informacija o zahtevu. U svakoj liniji se navodi jedna informacija u formi *Ime parametra: Vrednost parametra*
- Prazan red
- Telo poruke (opciono, ako se još neki podaci šalju serveru)

Dat je primer sadržaja jednog HTTP zahteva.

```
GET www.w3.org/People/Berners-Lee/Longer.html HTTP/1.1
Host: www.w3.org
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:51.0) Gecko/20100101 Firefox/51.0
```

U prvom redu smo rečju GET označili da zahtevamo preuzimanje resursa sa servera. Identifikator resursa je naveden u nastavku zajedno sa verzijom HTTP protokola. Osim GET operacije, postoji mogućnost i da se izvrše druge operacije (npr. dodavanja ili brisanja veb resursa), ali sada ih nećemo objašnjavati obzirom da smo fokusirani samo na preuzimanje veb dokumenata. Drugi i treći red predstavljaju parametre zaglavlja. Prikazani primer nema telo poruke, jer je već u prvoj liniji i zaglavljima definisano sve što je potrebno da bi server obradio zahtev.

HTTP odgovor se sastoji od sledećih linija:

- Statusna linija – uključuje verziju protokola, statusni kod i objašnjenje statusnog koda. Statusni kod nosi informaciju o uspešnosti odgovora. Kod se sastoji od 3 cifre. Na primer, ako je zahtev uspešno obrađen vraća se status 200. Ako traženi resurs nije pronađen, dobićemo odgovor sa statusom 404. Objašnjenje statusnog koda je tekstualni opis statusa na engleskom jeziku. Na primer, za kod 200 opis je *OK*, a za kod 404 opis je *Not Found*.
- Zaglavlje odgovora – niz dodatnih informacija o odgovoru. U svakoj liniji se navodi jedna informacija u formi *Ime parametra: Vrednost parametra*
- Prazna linija
- Opciono telo odgovora. U telu odgovora se nalazi resurs koji je klijent tražio u zahtevu

Dat je primer jednog HTTP odgovora koji vraća HTML dokument sa tekстом Novi Sad.

```
HTTP/1.1 200 OK
Content-Length: 24569
Content-Type: text/html; charset=iso-8859-1
Date: Fri, 17 Feb 2017 20:35:00 GMT
```

Internet

```
<html>
  <body>
    Novi Sad
  </body>
</html>
```