Fakultet Tehničkih nauka

Napredni računarski sistemi sa kritičnom misijom u elektroenergetici

Laboratorijski praktikum

Branislav Atlagić, Mita Čokić, Mihalj Šagi, recenzija: Nemanja Isakov
10/7/2015

DRAFT

Sadržaj

Skraćenice

SCADA- Supervisory control and data acquisition,

IP – Internet protocol,

TCP – Transmission control protocol,

UDP – User datagram protocol,

DSL – Digital subscriber line,

ISO OSI – International organization for standardization - open systems interconnection,

DNS – Domain name system,

SNMP – Simple network management protocol,

RIP – Routing information protocol,

DHCP – Dynamic host configuration protocol,

HTTP – Hypertext transfer protocol,

FTP – File transfer protocol,

SMTP – Simple mail transfer protocol,

POP3 – Post office protocol,

Uvod

Laboratorijski praktikum ima za cilj da kroz zadatke pruži razumevanje dSCADA sistema što je osnova za dublje razumevanje industrijskih akviziciono upravljačkih sistema. Znanja neophodna za praćenje vežbi su: (1) programski jezik C, (2) razumevanje statičkih i dinamičkih struktura podataka, (3) konkurentno programiranje i (4) računarske mreže. Praktikum neće pokrivati programski jezik C, strukture podataka i konkurentno programiranje.

U prvom poglavlju su predstavljene teorijske osnove iz oblasti međuračunarskih komunikacija sa zadacima. Razumevanje međuračunarskih komunikacija je krucijalno za razvoj akviziciono upravljačke funkcionalnosti bilo kog SCADA sistema. Cilj ovog poglavlja nije izučavanje računarskih mreža već podsetnik osnovnih principa potrebnih za vežbe iz ovog predmeta. Nećemo ulaziti u analizu dublje nego što je to potrebno a neki principi će namerno biti pojednostavljeni ili izostavljeni kako ne bismo izgubili fokus sa akviziciono upravljačkih sistema.

U drugom poglavlju ćemo kroz implementaciju pojedinih komponenti dSCADA sistema doći do znanja potrebnog za implementaciju jednostavnog SCADA sistema i simulatora Modubus protokola.

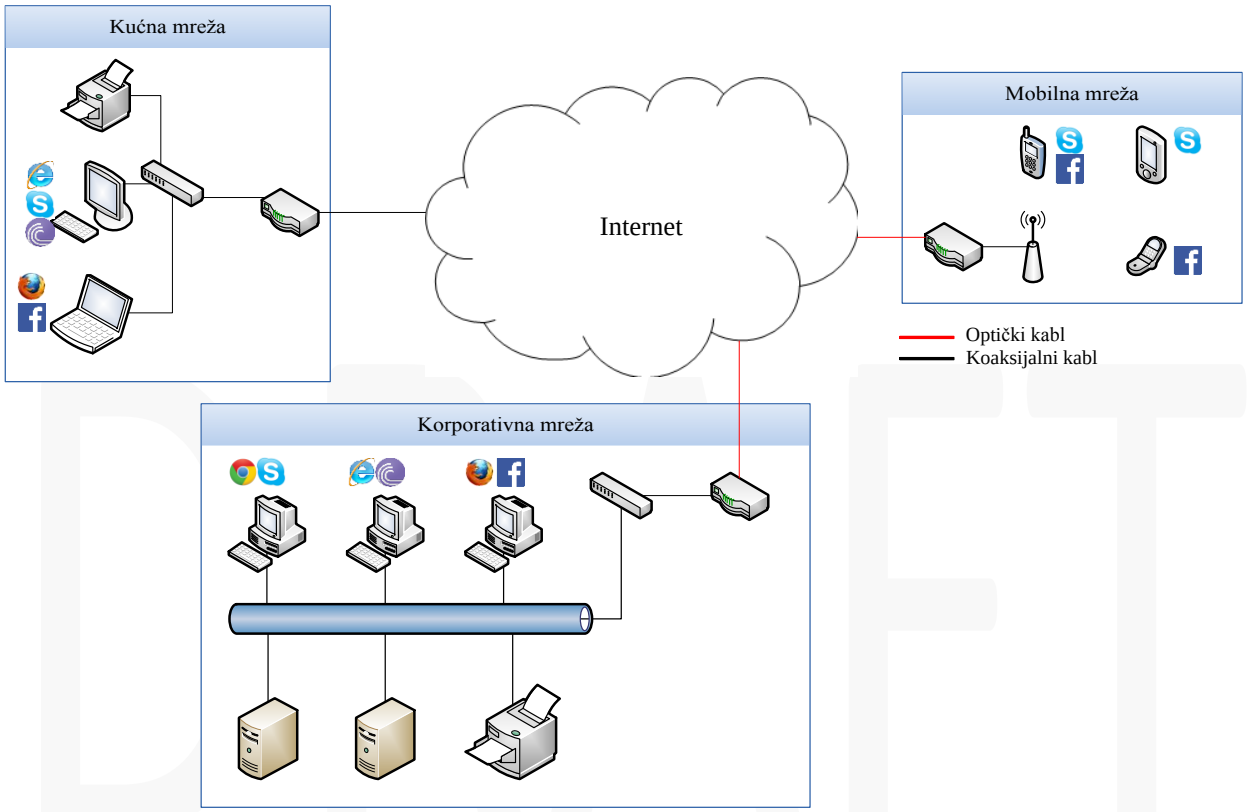
Na kraju ćemo dSCADA sistem prilagoditi za rukovođenje određenim industrijskim postrojenjem a simulator simulaciji postrojenja kako bismo rešenje istestirali.

Uz praktikum su isporučeni i sledeći laboratorijski materijali:

1. Kompletna dSCADA aplikacija sa urađenim primerima.
2. Primeri UDP klijent-server aplikacija koje koriste blokirajuće i neblokirajuće sokete.
3. Primer TCP klijent-server aplikacije korišćenjem blokirajućih soketa.

Međuračunarske komunikacije

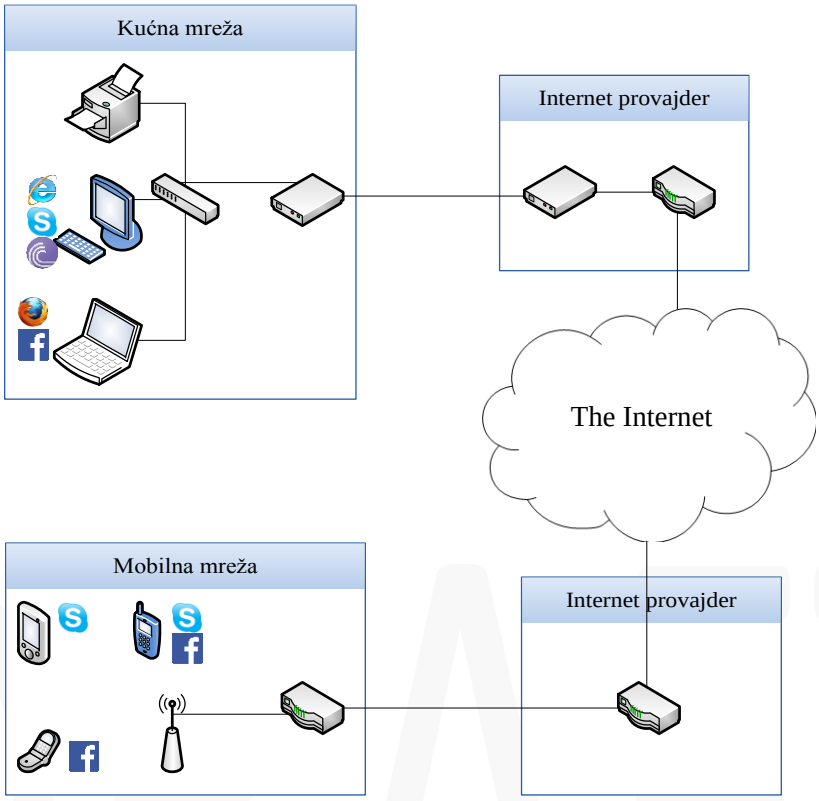
Svrha ovog poglavlja je da pruži osnovno znanje iz oblasti mređuračunarskih komunikacija. Prikaz dimenzija problema međuračunarskih komunikacija je dat na primeru heterogene mreže prikazane na sledećoj slici:



Slika 1: Heterogena računarska mreža

Više aplikacija na različitim uređajima razmenjuje podatke preko drugačijih fizičkih medija (koaksijalni kabl, optički kabl, bežično). Zatim, uređaji razmenjuju podatke između sebe i ako se ne nalaze u istoj računarskoj mreži. Podaci se razmenjuju između istih aplikacija (npr. Skype) koje mogu raditi na različitim operativnim sistemima. Na jednom uređaju može raditi istovremeno više drugih aplikacija koje koriste mrežne usluge.

Uzmimo na primer Skype poruku koju korisnik sa računara u kućnoj mreži želi da pošalje korisniku mobilnog telefona, što ilustije Slika 2. Predpostavimo da kućni računar ima pristup internetu preko DSL tehnologije. Poruka putuje od Skype aplikacije do DSL modema. Modemom se poruka šalje preko telefonskih parica do internet provajdera. Poruka se zatim usmerava do internet provajdera preko koga odredišni uređaj ima pristup internetu a koji će poruku proslediti do mobilnog operatera. Mobilni operater će poruku isporučiti do odredišnog uređaja nakon čega će biti isporučena odredišnoj aplikaciji.



Slika 2: Putanja Skype poruke od izvora do odredišta

Zaključujemo se da računarske mreže koriste širok spektar tehnologija iz različitih oblasti, u zavisnosti od toga koji segment posmatramo (od razvoja komunikacione platforme za razmenu poruka do propagacije svetlosnog signala kroz optički kabl).

Protokoli

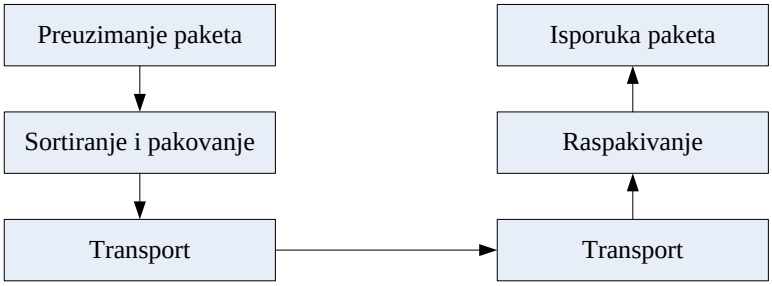
Komunikacija između računara, ili opštije rečeno između čvorova mreže se obavlja preko različitih mrežnih protokola. Protokol definiše skup pravila kojih se učesnici u razmeni podataka moraju pridržavati kako bi razmenjivali podatke. U svakodnevnom primeru jednostavan primer protokola je svetlosna sinhronizacija – semafor gde svako svetlo ima svoje značenje. Protokoli nude odgovarajuće usluge, kao što svetlosna signalizacija nudi uslugu sinhronizovanog pristupa deljenom resursu (raskrsnica) kako bi se izbegla kolizija (sudar).

Protokoli definišu tri elementa:

- Sintaksa – format podaka i signala,
- Semantika – pravila koja definišu da li su primljene informacije smislene u datom kontekstu,
- Vremenske kontrole – vremenski okviri i redosled događaja uključujući i brzinu razmene podataka.

Protokoli su slojevito organizovani a skup međuzavisnih protokola se naziva protokol stek. Slojevita organizacija protokola se može objasniti na primeru slanja paketa kurirskom službom [2]:

(1) pošiljalac navodi odredišnu adresu i paket preuzima kurir, (2) kurirska služba sve pakete iz iste organizacione jedinice (npr. grad, mesna zajednica) sakuplja, sortira prema odredištu i pakuje u transportne kontejnere (3) kontejneri se transportuju do odredišta (4) paketi se raspakuju iz kontejnera i stavljaju dostavljačima na raspolaganje, (5) kurir isporučuje paket na odredišnu adresu.



Slika 3: Slojevita organizacija kurirske sluzbe

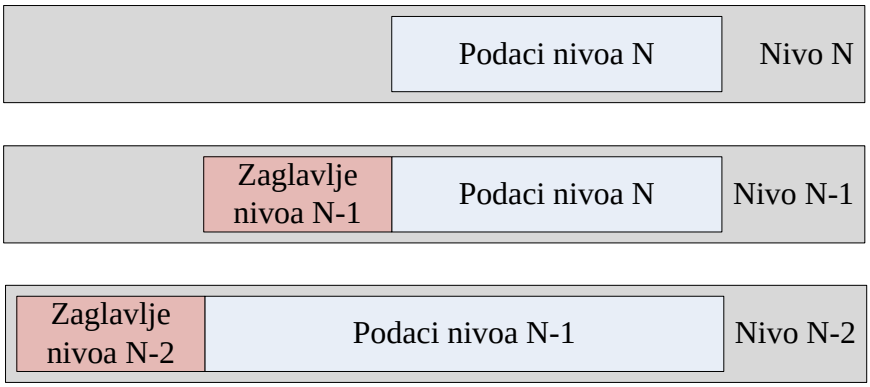
Slika 3 prikazuje proces od slanja do uručenja poštanskog pisma podeljen na akcije koje se obavljaju na izvorišnoj i odredišnoj lokaciji. Primećujemo da u transportu pisma imamo sledeće faze:

1. Preuzimanje i isporuka paketa.
2. Sortiranje, pakovanje odnosno raspakivanje.
3. Transport.

Ove faze se oslanjaju jedna na drugu, ali ne ulaze u detalje implementacije susednih faza (prilikom sortiranja, pakovanja i raspakivanja se ne ulazi u detalje da li se paketi transportuju brodom, avionom ili kamionom). Svaki sloj suštinski komunicira sa slojem iznad ili ispod a logički sa svojim slojem na odredištu.

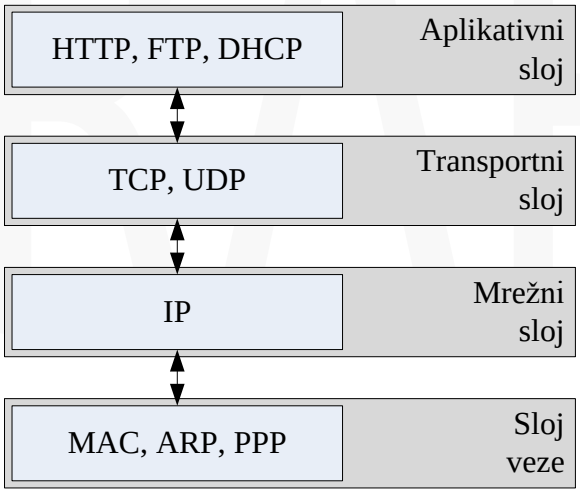
Pored slojevitosti, još jedna osobina mrežnih protokola je enkapsulacija podataka. Enkapsulacija predstavlja vid apstrakcije. Kada protokol nivoa N šalje podatke nivou N-1, on ih šalje u strukturi u kakvoj ih očekuje na odredištu (npr. dodaje zaglavlje). N-1 nivo će na te podatke staviti svoje zaglavlje i proslediti ih N-2 nivou.

Enkapsulacija se može uočiti i u poštanskoj isporuci paketa [2]. Svaki korisnik navodi odredišnu adresu paketa i u paket stavi njemu bitne predmete. U poštanskoj službi pakete koje su namenjeni istoj poštanskoj ekspozituri stavljaju u kutije sa adresom poštanske ekspoziture. Pri tome ne ulazi u detalje šta je koji pošiljalac poslao. Kutije koje putuju do istog grada se pakuju u kontejnere na koje se navodi adresa grada i pritom se ne ulazi pojedinačno u sadržaje kutija, niti paketa. Na odredištu se kontejneri raspakuju i kutije šalju do odgovarajućih poštanskih ekspozitura. Kutije se u ekspoziturama otvaraju i pojedinačni paketi isporučuju na odredišne adrese.



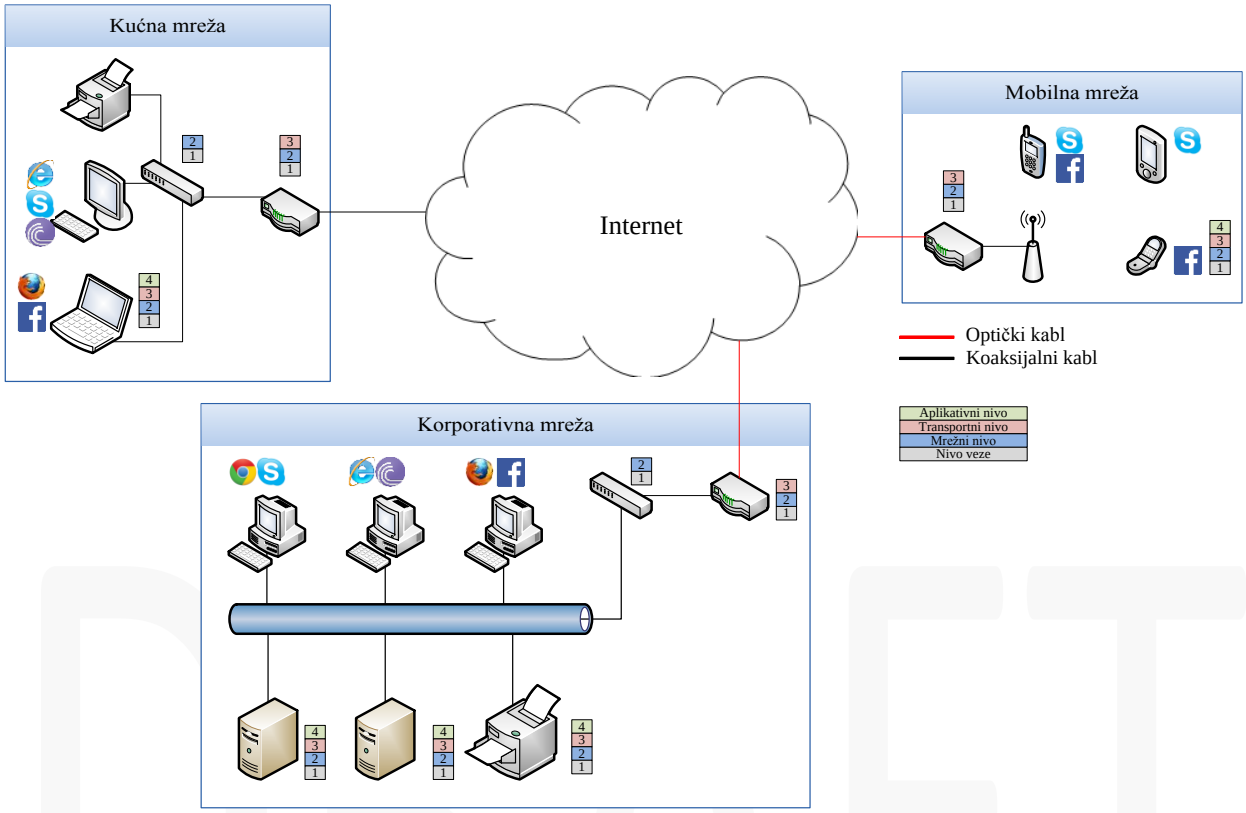
Slika 4: Enkapsulacija protokola

Identične principe imamo u računarskim mrežama. Dakle imamo stek protokola gde protokoli odgovarajućeg nivoa pružaju određene usluge a podatke primljene sa viših nivoa enkapsuliraju. Slika 4 ilustruje enkapsulaciju protokola. U terminologiji računarskih mreža se spominju dva modela slojevite organizacije protokola – ISO OSI i Internet protokol stek. Mi ćemo se fokusirati na Internet protokol stek iako su suštinske razlike male.



Slika 5: Stek internet protokola

Slika 5 prikazuje slojeve steka internet protokola. Nivoi protokola su logički podeljeni. Svaki protokol komunicira direktno sa protokolima iznad i ispod sebe a logički komunicira sa istim nivoom na drugom čvoru. Ne moraju svi čvorovi kroz koji poruka prolazi implementirati sve nivoe protokola. Sledeća slika prikazuje čvorove mreže kao i nivoe protokola koje svaki od čvorova (uglavnom) implementira:



Slika 6: Heterogena mreža sa prikazom implementiranih nivoa protokola na određenim uređajima

Mrežni sloj

Protokoli mrežnog sloja su zaduženi za isporuku paketa od izvornog do odredišnog uređaja. Neke od usluga koje protokoli mrežnog sloja mogu pružati su [2]:

- Garantovana isporuka paketa.
- Garantovana isporuka paketa u redosledu u kom su poslati.
- Garancija poštovanja različitih vremenskih okvira (maksimalno kašnjenje, brzina prenosa...).
- Bezbednost.
- Kontrola zagušenja.

Dve osnovne funkcionalnosti protokola mrežnog nivoa su:

- Prosleđivanje paketa (package forwarding).
- Usmeravanje paketa (package routing).

Prosleđivanje paketa je funkcija mrežnog rutera o usmeravanju poruke sa ulaznog na odgovarajući izlazni port. Informacije o prosleđivanju paketa se nalaze u mrežnim ruterima. Usmeravanje paketa se odnosi na funkciju usmeravanje odgovarajućeg paketa od izvora do odredišta kroz sve mrežne čvorove. Usmeravanje paketa se određuje algoritmima za usmeravanje paketa (routing algorithms). Kako se tabele

usmeravanja ažuriraju i detaljnije informacije o usmeravanju paketa prevazilaze domen ovog praktikuma. Dodatne informacije se mogu pročitati u [2].

Mrežni protokoli mogu funkcionisati sa ili bez prethodnom uspostavom veze. Mogu koristiti virtuelna kola (virtual circuit network) kada je transportni sloj svestan veze između dva čvora (informacija može biti upisana u ruterima) i svi paketi se kreću kroz unapred dogovorene čvorove. Nasuprot ovom pristupu, svaka poruka može biti individualna i posebno se usmeravati kroz mrežu (datagram network). U ovom slučaju paketi koji pripadaju istoj mreži ne moraju prolaziti kroz iste čvorove.

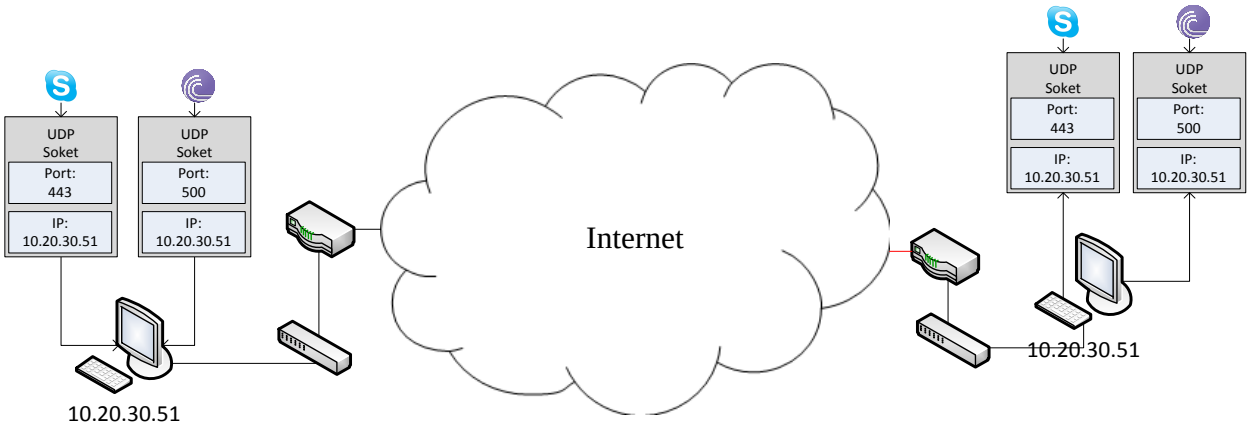
Internet stek protokola koristi u mrežnom sloju Internet Protokol (IP). Za potrebe ovog kursa je dovoljno apstrakovati da IP isporuči paket (datagram) od izvornog do odredišnog čvora bez garancije isporuke i prethodne uspostave veze.

Transportni sloj

Videli smo da IP protokol omogućava isporuku poruke od izvornog do odredišnog čvora. Kako na svakom računaru možemo imati više aplikacija koje žele da šalju i/ili primaju podatke nedostaje nam način kako da multipleksiramo/demultipleksiramo podatke aplikacija koje se izvršavaju na istom računaru kroz mrežni sloj. Upravo ovu uslugu nam pružaju protokoli transportnog sloja – User Datagram Protokol (UDP) i Transmission Control Protocol (TCP).

UDP i TCP ovo omogućuju uvođenjem soketa (socket). Soket, grubo rečeno predstavlja kombinaciju porta – celobrojnog 16-bitnog broja i IP adrese. IP adresa jednoznačno određuje računar u mreži a port aplikaciju na računaru. TCP soket ćemo dodatno precizirati prilikom obrađivanja TCP primera.

Kod svih mrežnih aplikacija imamo (bar) dve strane – jedna strana inicira komunikaciju i šalje zahteve koje druga strana obrađuje i šalje odgovore. Aplikacija koja inicira komunikaciju i šalje zahteve je klijent a aplikacija koja čeka zahteve i šalje odgovor je server. Server se operativnom sistemu prijavi da očekuje podatke na određenom portu (recimo port 5059). Klijent uz poruku koju šalje serveru pošalje port na kom server čeka podatke i IP adresu računara na kom je server pokrenut. Na ovaj način IP može da isporuči poruku do odredišnog računara a protokoli transportnog sloja do odredišne aplikacije. Opisan proces je prikazan na sledećoj slici:



Slika 7: Multipleksiranje i demultipleksiranje aplikativnih podataka

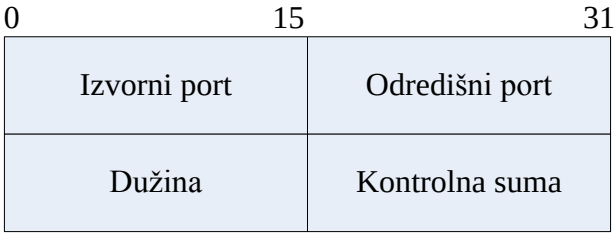
U nastavku ćemo izučiti osnovne principe na kojima UDP i TCP počivaju i njhove osobine. Zatim ćemo obraditi Berkeley API za socket programiranje uz primere i smernice koje preporučujemo. Poglavlje ćemo završiti programskim zadacima.

User Datagram Protocol

Za UDP se može reći da je jednostavno proširanje IP sa brojem izvornog i odredišnog porta. UDP socket je kombinacija adrese i porta. Slika 8 prikazuje sadržaj UDP zaglavlja. Osobine UDP protokola su:

- Poruke se šalju bez prethodne uspostave veze. Šalju se bez obzira da li ih neko očekuje.
- Nema stanja (state-less).
- Ne garantuje isporuku paketa.
- Ne garantuje isporuku paketa u redosledu u kom su poslati.
- Ne poseduje kontrole toka i zagušenja.

Zbog ovih osobina je UDP jednostavan i lagan protokol i kao takav je upotrebu našao u multimediji gde eventualno gubljenje dela poruke nije problematično, kao i osnova za druge protokole (DNS, SNMP, RIP, DHCP).



Slika 8: UDP zaglavlje

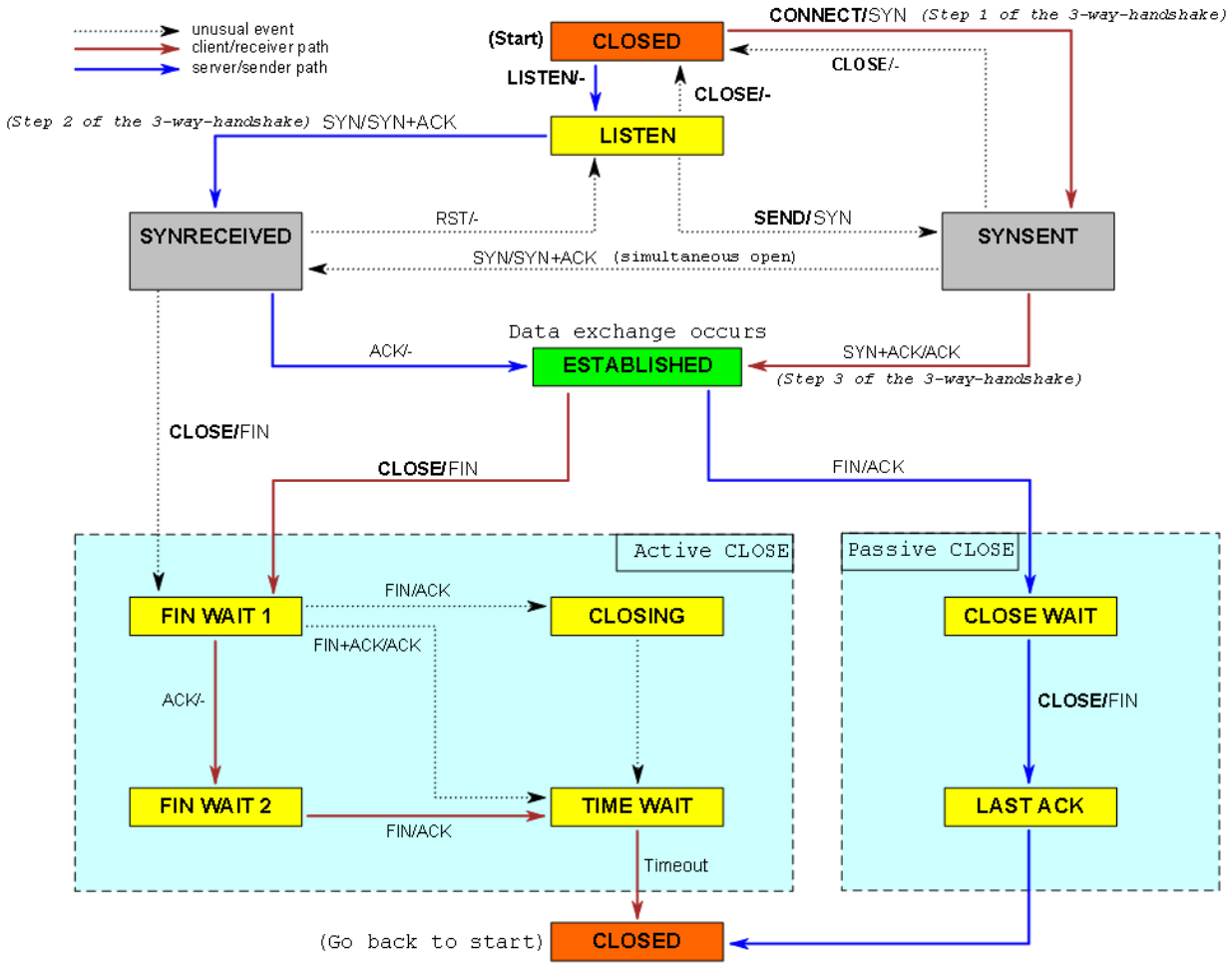
Transmission Control Protocol

TCP nudi pouzdan prenos podataka (reliable data transfer) sa prethodnom uspostavom veze (connection orientated) uz kontrole toka (flow control) i zagušenja (congestion control). U nastavku ćemo ukratko obraditi nabrojane funkcionalnosti TCP.

Protokol sa uspostavom veze

Uspostavljanje veze je spominjana kod protokola mrežnog nivoa. Pod ovim se podrazimeva da pre slanja aplikativnih podataka klijent i server uspostave vezu i međusobno razmene određene parametre koji će se koristiti prilikom razmene podataka. Proces uspostave veze se zove rukovanje (*handshake*).

Server obaveštava operativni sistem na kom portu želi da očekuje poruke i prelazi u stanje LISTEN. Klijent serveru šalje SYN paket na šta server njemu odgovara sa SYN ACK paketom. U poslednjem koraku klijent serveru odgovara sa ACK i veza je uspostavljena. Dijagram 1 Predstavlja traziciju TCP stanja.



Slika 9: TCP Dijagram stanja prilikom uspostave veze [9]

Informacije o vezi se nalaze na klijentu i serveru jer IP nije svestan veze. Postoje protokoli mrežnog nivoa koji jesu svesni veze.

Po uspostavi veze server će napraviti novi soket kroz koji će teći dalja komunikacija sa tim klijentom a socket koji je na početku bio u LISTEN stanju može nastaviti primanje novih klijenata. Kako kod TCP komunikacije možemo imati istovremeno pristizanje podataka na isti port ali od različitih izvora, TCP soket je određen:

- odredišnom adresom,
- odredišnim portom,
- izvornom adresom,
- izvornim portom.

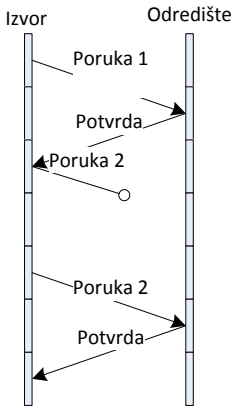
Nasuprot UDP soketu koji je određen samo odredišnom adresom i odredišnim portom.

Pouzdan prenos podataka

Nabrojaćemo osnovne elemente koji su neophodni za implementaciju pouzdanog prenosa poruka. Ako pošiljalac želi da zna da je njegova poruka stigla na odredište, odredište mora potvrditi da je poruka primljena. S' obzirom da se TCP oslanja na IP, može se desiti da se poslata poruka izgubi u transportu ili da izmenjena stigne na odredište. Zbog ovoga je neophodno nakon slanja poruke čekati potvrdu prijema određeno vreme i ukoliko poruka nije ispravno isporučena ili potvrda prijema nije stigla izvršiti ponovno slanje poruke (retransmission). Da bismo implementirali opisanu funkcionalnost potrebni su nam sledeći mehanizmi [2]:

- potvrda prijema,
- detekcija greške
- vremenska kontrola,
- ponovno slanje podataka.

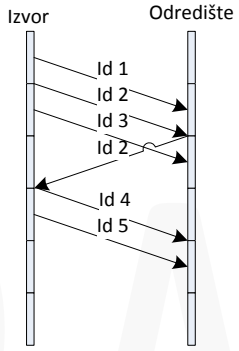
Prethodno opisani pristup, gde se nakon slanja svake poruke čeka potvrda o uspešnom prijemu poruke, naziva se stop-and-wait. Sledeći dijagram ilustruje slanje poruke korišćenjem *stop-and-wait* algoritma.



Slika 10: Ilustracija stop-and-wait algoritma

Kako se novi paketi ne šalju dok se potvrda za poslati paket ne primi S&W ima nisko iskorišćenje mrežne infrastrukture. Mehanizam koji ovaj problem rešava se naziva princip klizajućeg prozora (*sliding window*).

Kod klizajućeg prozora, svaki paket ima svoj identifikator. Pošiljalac pošalje dogovoren broj paketa jedan za drugim a nove pakete šalje tek kada je za najranije poslat paket primio potvrdu da ga je primio. Ovim pristupom je iskorišćenost mrežne infrastrukture drastično povećana. TCP implementira ovaj koncept a kao identifikator poruke koristi redni broj bajta u poruci. TCP podrazumeva da ukoliko dobije potvrdu isporuke određenog paketa, računa da su i svi paketi pre njega uspešno isporučeni. Ne očekuje za svaki pojedinačni poslati paket potvrdu o uspešnoj isporuci. Sa klijentske strane potvrde se šalju svakih 500ms. Šalje se samo potvrda za paket sa najvećim identifikatorom za koji su primljeni svi prethodni paketi. TCP princip klizajućeg prozora koristi se i za kontrolu toka o čemu će biti reči u sledećem poglavlju.



Slika 11: Ilustracija *sliding window* algoritma

Kontrola toka – flow controll

TCP je dvosmerni protokol – omogućava da obe strane šalju i primaju podatke istovremeno. Primljene podatke TCP smešta u prihvatnu memoriju iz koje ih aplikacija čita nakon čega se uklanjaju. Međutim, šta će se desiti ukoliko aplikacija podatke čita sporije nego što pristižu? Ukoliko se ništa ne bi preuzelo, prihvatna memorija bi se prepunila i nove poruke ne bi mogle biti smeštene. TCP na odredišnoj strani ovaj problem rešava tako što u svakom ACK paketu javlja koliko ima slobodnog mesta u prihvatnoj memoriji kako bi izvorna strana mogla da uspori slanje smanjenjem prozora za slanje na dužinu slobodnog mesta u prihvatnom baferu. Na ovaj način će izvorna strana slati manju količinu podataka.

Ovakvo ponašanje je često posledica programske greške ili pogrešnog pristupa prilikom razvoja aplikacije. Ukoliko obrada podataka nije trivijalna, preporuka je da nit koja čita podatke sa soketa pročitane podatke smešta u prihvatnu memoriju odakle će ih druga nit čitati i obrađivati. Na ovaj način će se izbeći nagomilavanje podataka na soketu.

UDP ne podržava kontrolu toka što znači da će podaci biti slati brzinom nezavisnom od brzine kojom ih odredište obrađuje.

Kontrola zagušenja – congestion controll

TCP paketi koji se šalju preko IP kroz mrežu prolaze kroz niz čvorova do odredišta. Čvorovi su uglavnom ruteri. U zavisnosti od njihovog opterećenja može doći do kašnjenja isporuke paketa ili do gubljenja paketa. Rečeno je ranije da IP ne nudi kontrolu zagušenja pa se TCP ne može osloniti na nju već je sam implementira. TCP koristi pakete potvrde kako bi ustanovio kada je došlo do zagušenja. Ukoliko TCP prima

ACK poruke regularno – zagušenja nema. Ukoliko se poruke gube – zagušenja ima. Za detekciju gubljenja poruka se koristi jedan od sledeća dva kriterijuma:

- Istekao timeout ;
- Za iste podatke se dobiju 4 ACK paketa. Kako dvostruki ACK može značiti da je došlo do primanja podataka van redosleda i da poruka nije zapravo izgubljen već da će se obraditi ali sa zakašnjenjem koji će generisati novi ACK paket. Ako se primi tri ili više duplikata iste poruke predpostavlja se da je segment izgubljen. Ovo je definisano standardom [4].

Ukoliko se primeti da zagušenja nema, TCP će povećavati brzinu slanja a ukoliko se primeti da zagušenja ima smanjiće brzinu slanja. TCP održava promenljivu koja se naziva prozor zagušenja (congestion window) koja govori koliko je moguće maksimalno poslati poruka istovremeno pre primanja potvrde. Širina prozora se smanjuje ili povećava u zavisnosti od toga da li ima zagušenja ili ne.

TCP je veoma složen protokol. Kroz ovo poglavlje smo samo zagrebali površinu kako bismo veoma pojednostavljeno predstavili osnovne principe na kojima se TCP zasniva. Više informacija o TCP se može naći u [2], [3] i [4].

Aplikativni sloj i razvijanje programske podrške

Protokoli aplikativnog sloja se oslanjaju na protokole transportnog sloja i oni su karakteristični za određene aplikacije: HTTP, FTP, SMTP, POP3, Skype, BitTorrent. U ovom i narednim poglavljima ćemo se fokusirati na razvoj programske podrške nad protokolima transportnog sloja uz razvoj protokola aplikativnog sloja.

Programska podrška će biti razvijana korišćenjem MS Visual Studio 2010 razvojnog okruženja i Windows Sockets 2 biblioteke (WinSock2). WinSock2 biblioteka pruža Berkeley sockets API za razvijanje mrežnih aplikacija na Windows operativnom sistemu. Koristićemo C programski jezik.

Sledi spisak najčešće korišćenih funkcija sa kratkim objašnjenjem. Detaljnije objašnjenje sa objašnjenjem parametara funkcija i povratnih vrednosti svake od funkcije je moguće naći u Microsoft biblioteci [4].

Funkcija	Opis	Primer

socket()	Formira soket vezan za određeni protokol, ali ga ne vezuje za port i adresu.
	Povratna vrednost je formirani SOCKET.
		`serverSocket =`
		`  socket(AF_INET, // IPv4 address famly`
		`  SOCK_DGRAM, // datagram supporting`
		`socket`
		`  IPPROTO_UDP); // UDP`

bind()	Povezuje soket formiran socket() funkcijom sa lokalnom adresom i portom.	`bind(serverSocket,`
		`  (LPSOCKADDR)&serverAddress,`
		`  sizeof(serverAddress));`

listen()	Prebacuje socket u režim čekanja uspostave veze.	`listen(ListenSocket, SOMAXCONN);`

accept()	Preuzima prvi zahtev za uspostavu veze sa reda čekanja i izvršava uspostavu veze. Formira novi SOCKET koji će se koristiti za komunikaciju sa klijentom	`clientSocket = accept(ListenSocket,`
		`  NULL,`
		`  NULL);`

	preko uspostavljene veze koji vraća kao povratnu vrednost.	
recvfrom()	Vrši čitanje datagrama sa soketa uz čitanje adrese i porta sa kojih je datagram poslat.	`recvfrom(serverSocket, accessBuffer, BUFFER_SIZE, 0, (LPSOCKADDR)&clientAddress, &sockAddrLen);`
sendto()	Vrši slanje datagrama na socket.	`sendto(clientSocket, outgoingBuffer, BUFFER_SIZE, 0, (LPSOCKADDR)&serverAddress, sockAddrLen);`
recv()	Čita podatke sa socketa.	`recv(ClientSocket, recvbuf, recvbuflen, 0);`
send()	Šalje podatke na socket.	`send(ConnectSocket, sendbuf, strlen(sendbuf), 0);`
closesocket()	Zatvara socket uz oslobađanje zauzetih resursa.	`closesocket(ListenSocket)`

Većina WinSock2 API funkcija u slučaju greške vraćaju informaciju da je do greške došlo ali ne i koja je greška tačno u pitanju. Nakon ustanovljenja da je do greške došlo na osnovu povratne vrednosti funkcije, tačne informacije o grešci se preuzimaju na sledeći način:

1. Kod greške se preuzima funkcijom `WSAGetLastError()`,

   ```
   int errorCode = WSAGetLastError();
   ```

2. Dobijeni kod greške potražiti na MSDN stranici: System Error Codes,
3. Ukoliko MSDN stranica ne pruža dovoljno informacija – koristiti Google za odgovarajući kod greške,

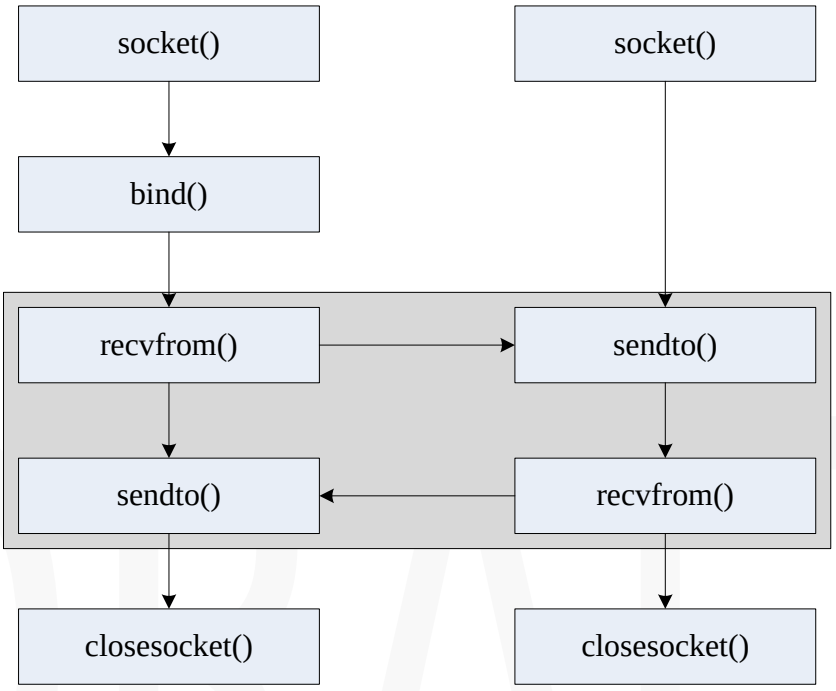
Preporuka je da se nakon svake WinSock2 funkcije vrši provera povratne vrednosti i ukoliko je došlo do greške ispiše kod greške kao i odrednica gde se greška tačno pojavila.

Soketi imaju dva režima rada – blokirajući i neblokirajući. U blokirajućem režimu se pozivi funkcija za čitanje i pisanje sa soketa neće završiti dok se ne stvore uslovi za izvršavanje. Pod ovim se podrazimeva da će se vraćanje iz funkcije čitanja soketa desiti tek kada ima nešto da se pročita a pisanje tek kada se pojavi mesto na internom baferu za prihvatanje podataka za slanje. U neblokirajućem režimu, pre početka slanja ili primanja podataka sa soketa se može proveriti da li će poziv biti blokiran ili ne.

UDP

Sledeći dijagram ilustruje dijagram sekvenci Berkli API funkcija na serverskoj i klijentskoj strani. Server formira odgovarajući soket i povezuje ga sa lokalnom adresom i protokolom (`socket()` i `bind()` funkcije). Zatim poruke prima i šalje sa `recvfrom()` i `sendto()`.

Klijent formira socket a poruke šalje i prima takođe sa `recvfrom()` i `sendto()` funkcijama. Obe strane će kada odluče da ne žele da primaju i šalju pakete zatvoriti soket sa `closesocket()` i time oslobode resurse vezane za taj soket.



Slika 12: Dijagram sekvenci za UDP sokete korišćenjem Berkli API

Primer UDP_Blocking

Ovaj primer predstavlja jednostavnu implementaciju servera i klijenta korišćenjem blokirajućih UDP soketa. Pokrenut server ispisuje tekst koji klijent pošalje i čeka poruku od sledećeg klijenta. Klijent pošalje uneseni tekst sa tastature serveru i završi se. Počnimo sa analizom servera (Server.cpp) i njegove `main()` funkcije. Serversku aplikaciju možemo podeliti na nekoliko segmenata:

1. Inicijalizacija lokalnih promenljivih i WinSock2 biblioteke uz inicijalizaciju soketa na kom će server čitati podatke poslate od strane klijenta. Soket se formira sa `socket()` funkcijom uz preciziranje tipa:

```
SOCKET serverSocket = socket(AF_INET,      // IPv4 address famly
                             SOCK_DGRAM,   // datagram socket
                             IPPROTO_UDP); // UDP
```

   Ranije smo rekli da je UDP soket određen adresom i portom. Vezivanje porta i IP adresa za soket obavlja `bind()` funkcija:

```
iResult = bind(serverSocket,(LPSOCKADDR)&serverAddress,sizeof(serverAddress));
```

2. Opsluživanje klijentskih zahteva – u našem primeru primanje poruke i njen ispis. UDP poruka se prima sa:

```
iResult = recvfrom(serverSocket,
                   accessBuffer,
                   ACCESS_BUFFER_SIZE,
```

```
                0,
                (LPSOCKADDR)&clientAddress,
                &sockAddrLen);
```

Prvi parameter predstavlja soket sa kog se podaci čitaju. Drugi i treći parametar predstavljaju prihvatni bafer i maksimalnu veličinu prihvatnog bafera, respektivno. Četvrti parametar ćemo preskočiti a u peti parametar će transportna infrastruktura smestiti adresu i port odakle podaci potiču. Setite se da UDP ne uspostavlja vezu tako da samo na osnovu prihvatnog soketa ne možemo znati ko je podatke poslao. *clientAddress* parametar će sadržati podatke koje server može iskorisiti za slanje odgovora.

3. Gašenje serverske aplikacije koje podrazumeva zatvaranje soketa, deinicijalizaciju WinSock2 biblioteke i ostalo oslobađanje resursa.

U `main()` funkciji klijentske aplikacije (*Client.cpp*) `bind()` se ne poziva. Implicitni bind se desi prilikom poziva `sendto()` funkcije kada se za soket vežu odredišna adresa servera i port servera. `sendto()` funkcija se koristi za slanje poruke:

```
iResult = sendto(clientSocket,
                 outgoingBuffer,
                 strlen(outgoingBuffer),
                 0,
                 (LPSOCKADDR)&serverAddress,
                 sockAddrLen);
```

Prvi parametar predstavlja soket na koji će se podaci slati. Drugi i treći predstavljaju podatke koji se šalju i veličinu podataka, respektivno. Četvrti parametar ćemo preskočiti a peti parametar predstavlja odredišnu adresu poruke.

Zadatak 1

Potrebno je izmeniti UDP_Blocking primer tako da server implementira sledeće funkcionalnosti:

1. Eho – primljeni tekst nakon ispisa pošalje nazad klijentu.
2. Ukoliko primi 'S' ili 's' ASCII znak kao prvi bajt poruke izvrši gašenje servera.

Kijent treba da implementira sledeću funkcionalnost:

1. Ispiše poruku primljenu od severa pre gašenja.

Primer UDP_Nonblocking

U prethodnom primeru smo koristili blokirajuće sokete. Vidi se da je (glavna) nit koja čeka podatke na soketu blokirana dok podaci ne stignu što i jeste glavni problem u ovom pristupu – ne postoji garancija da će podaci ikada stići. Da bi se izbeglo čekanje podataka na neodređeni period i omogućilo da nit koja čeka podatke obavlja još neke zadatke (obrada ulaza sa od drugih izvora kao na primer tastatura, ažuriranje ispisa ili slično) koriste se neblokirajući soketi. Glavna ideja je da se pre operacije nad soketom proveri da li će određena operacija nad soketom biti blokirana ili ne na osnovu čega programer može da odluči šta želi da izvrši.

Podrazumevan režim rada soketa je blokirajući. Režim rada soketa kao i različiti drugi atributi soketa se podešavaju sa `ioctlsocket()` funkcijom. Provera da li je soket spreman za čitanje ili pisanje se radi sa `select()` funkcijom.

Počnimo sa analizom servera (*Server.cpp*) i njegove `main()` funkcije. Kao i u prethodnom primeru, serversku aplikaciju možemo podeliti na nekoliko segmenata:

1. Inicijalizacija – slična je kao u prethodnom primeru sa jednom razlikom – serverski soket je postavljen u neblokirajući mod:

   ```
   unsigned long int nonBlockingMode = 1;
   iResult = ioctlsocket( serverSocket, FIONBIO, &nonBlockingMode );
   ```

2. Opsluživanje klijentskih zahteva se obavlja nakon što se ustanovi da se podaci mogu čitati sa soketa. Provera da li je određen soket raspoloživ za čitanje ili pisanje se proverava select() funkcijom:

   ```
   // Initialize select parameters
   FD_SET set;
   timeval timeVal;

   FD_ZERO( &set );
   // Add socket we will wait to read from
   FD_SET( serverSocket, &set );

   // Set timeouts to zero since we want select to return
   // instantaneously
   timeVal.tv_sec = 0;
   timeVal.tv_usec = 0;

   iResult = select( 0 /* ignored */, &set, NULL, NULL, &timeVal );
   ```

   `select()` funkcija prvi parametar ignoriše i tu je samo da API bude kompatibilan sa Berkely Socket API. Drugi parametar je skup soketa za koje je potrebno proveriti da li su spremni za čitanje. Manipulacija skupovima soketa se vrši makroima: `FD_SET`, `FD_ZERO`, `FD_SET`. Treći parametar, koji na serveru ne koristimo je skup soketa za koje je potrebno proveriti da li su spremni za pisanje.Četvrti parametar je skup soketa za koje potrebno proveriti da li su u stanju greške. Poslednji parametar je predstavlja koliko će maksimalno `select()` da bude blokiran ako ni jedan soket nije u stanju u kom se proverava. Ukoliko je on NULL, select poziv će biti blokirajući. Kada se `select()` funkcija završi, select će ažurirati ulazne skupove i u njima ostaviti samo one nad kojima je moguće izvršiti operaciju koja se proveravala, a kao povratnu vrednost će vratiti broj soketa koji su spremni za proveravane operacije.
   Čitanje sa soketa se nakon vraćanja `select()` funkcije i provere da je soket spreman za čitanje vrši identično kao i u prethodnom primeru.
3. Gašenje serverske aplikacije se vrši identično kao u prethodnom primeru.

Zadatak 2

Klijent u primeru UDP_Nonblocking koristi blokirajuće sokete. Izmeniti klijenta tako da koristi neblokirajuće sokete.

Zadatak 3

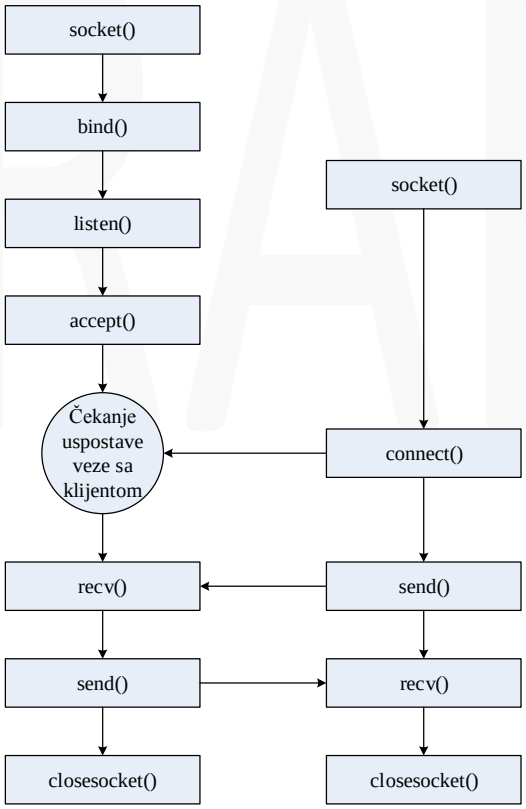
Potrebno je razviti klijent-server orijentisan kalkulator korišćnjem neblokirajućih UDP soketa. Server treba da podrži sledeće operacije:

- sabiranje,
- oduzimanje,
- kvadrat.

Klijent serveru šalje kod operacije i operande a server klijentu rezultat operacije.

TCP

Sledeći dijagram ilustruje redosled korišćenja Berkli API funkcija na serverskoj i klijentskoj strani:



Slika 13: Dijagram sekvenci za TCP sokete korišćenjem Berkli API

Nakon što se soket formira (funkcija `socket()`) i upari sa lokalnom adresom (funkcija `bind()`) neophodno ga je prebaciti u režim slušanja `listen()` funkcijom. Zatim sa `accept()` funkcijom server čeka klijenta. Rekli smo da je TCP protokol sa uspostavom veze. Za uspostavu veze između klijenta i servera je neophodno da server čeka klijenta a da klijent pozove `connect()` nad odgovarajućim soketom. Nakon

ovoga, obavljen je postupak rukovanja (*handshaking*), server i klijent su uspostavili vezu i mogu razmenjivati podatke.

Primer TCP_Blocking

Korišćenje TCP soketa ćemo objasniti na primeru *TCP_Blocking*. Ovaj primer sadrži jednostavan server koji čeka uspostavu veze sa klijentima i ispisuje tekst koji je klijent poslao. Server prima i ispisuje poruke od klijenta dok se veza ne prekine. Nakon toga može da opsluži novog klijenta.

Pravljenje TCP soketa se može videti u sledećem listingu:

```
SOCKET listenSocket = socket(AF_INET,      // IPv4 address famly
                             SOCK_STREAM,  // stream socket
                             IPPROTO_TCP); // TCP
```

Poziv `bind()` fukcije je nepromenjen u odnosu na UDP primere. Soket se prebacuje u režim slušanja pozivom na `listen()` funkciju:

```
iResult = listen(listenSocket, SOMAXCONN);
```

Prvi parametar listen funkcije predstavlja soket kom se menja režim a drugi koliko maksimalno istovremenih veza kroz njega želimo da dozvolimo ili `SOMAXCONN` ako želimo da dozvolimo maksimalan podržan broj veza. Sa `accept()` server čeka klijenta. Nakon što se veza uspostavi, povratna vrednost `accept()` funkcije je soket kroz koji će se obavljati komunikacija sa klijentom a `accept()` se može ponovo pozvati kako bi se čekao novi klijent. Česta praksa je da se opsluživanje klijenta nastavi u posebnoj niti kako bi server mogao da uspostavlja vezu sa novim klijentima.

```
SOCKET acceptedSocket = accept(listenSocket, NULL, NULL);
```

Sa druge strane, klijent uspostavlja vezu sa serverom pozivom na funkciju `connect()`.

```
connect(clientSocket, (SOCKADDR*)&serverAddress, sizeof(serverAddress))
```

Nakon uspostave veze, klijent i server podatke šalju i primaju korišćenjem `send()` i `recv()` funkcija.

```
iResult = recv(acceptedSocket, recvbuf, recvbuflen, 0);
iResult = send(clientSocket, sendbuf, (int)strlen(sendbuf), 0 );
```

Prvi parametar obe funkcije je soket kog se koristi za čitanje ili pisanje. Drugu parametar je adresa prihvatne memorije u slučaju primanja ili početna adresa memorijskog memorijskog segmenta koji treba poslati u slučaju slanja podataka. Treći parametar predstavlja veličinu prihvatne memorije u slučaju primanja podataka ili broj bajta koje je potrebno poslati preko soketa.

Važno je napomeniti da stanovišta aplikacije, TCP podatke ne šalje kao pojedinačne poruke već kao tok (stream) podataka. TCP ne nudi mogućnost razlikovanja gde se jedna aplikativna poruka završava a druga počinje već ta funkcionalnost mora biti implementirana na aplikativnom nivou.

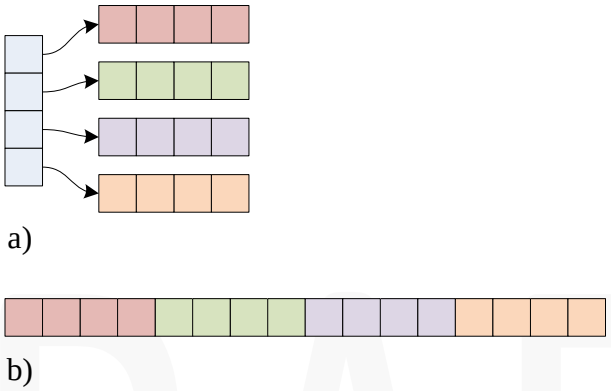
TCP omogućuje istovremenu razmenu podataka u oba smera (*full-duplex*) kroz isti soket kao i UDP.

Zadatak 4

Prepraviti primer TCP_Blocking tako da koristi neblokirajuće sokete.

Serializacija i deserializacija podataka

Videli smo da funkcije za slanje podataka preko soketa podrazumevaju da se podaci nalaze u sukcesivnim memorijskim lokacijama – nizu. Prilikom slanja podataka navodi se soket preko kog se podaci šalju, početna adresa memorijskog segmenta koji se šalje i dužina memorijskog segmenta. Na sledećoj slici, ilustracija a, je prikazan jedan način na koji dvodimenzionalni niz NxN, N=4, može biti predstavljen u memoriji:



Slika 14: a) jedna reprezentacija dvodimenzionalnog niza u memoriji, b) serializovan dvodimenzionalni niz

Za dvodimenzionalni niz NxN imamo niz pokazivača sa N elemenata, a svaki pokazivač pokazuje na jedan red niza. Jasno je da ovu strukturu ne možemo poslati preko mreže prosleđivanjem adrese prvog elementa niza i NxN bajta jer redovi niza nisu ne moraju biti jedan do drugog već su negde raspoređene po heap-u. Slanje pokazivača na kolone ne dolazi u obzir jer u odredišnom procesu neće biti validni. Rešenje je da se ovaj dvodimenzioni niz transformiše u jednodimenzioni niz pre slanja, takav pošalje do odredišta i tamo rekonstruiše u originalan dvodimenzionalni niz. Proces transformacije strukture podataka u jednodimenzionalnu reprezentaciju se naziva serijalizacija podataka a suprotan postupak deserijalizacija. Ovaj postupak ne postoji samo kod prenosa kroz mrežu već i krod bilo kakvog serijskog upisa (npr. upis u fajl).

Mogući postupak serijalizacije navedenog dvodimenzionalnog niza je opisan u sledećim koracima:

1. Slanje veličine niza odredištu kako bi znalo koliko memorije je potrebno pripremiti za čuvanje niza,
2. Dinamičko zauzimanje jednodimenzionalnog niza dužine NxN,
3. Upisivanje redova niza jedan za drugim u jednodimenzionalni niz,
4. Slanje pripremljenog serijalizovanog niza.

Postupak deserijalizacije koji odgovara prethodnoj serijalizaciji se može opisati u sledećim koracima:

1. Dimanička alokacija prihvatne memorije na osnovu poslate dužine niza,
2. Primanje serijalizovanog niza u pripremljenu prihvatnu memoriju,
3. Alociranje niza pokazivača dužine N,

4. Alokacija redova dužine N na koju će pokazivati pokazivači iz prethodne tačke,
5. Kopiranje reda po red iz prihvatnog niza u alocirane kolone.

Zadatak 5

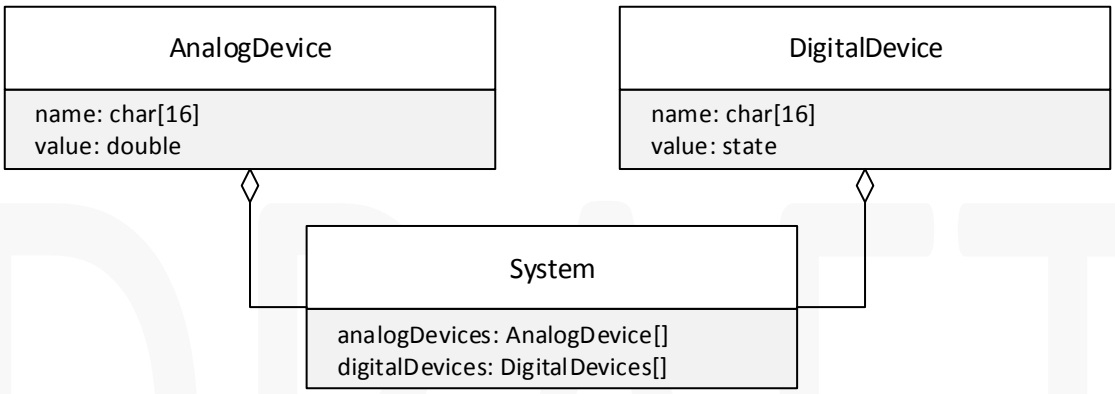
Implementirati klijent-server aplikaciju korišćenjem TCP soketa u kojoj server klijentu šalje dvodimenzionalni niz opisan u ovom poglavlju.

Kako je ovaj postupak moguće optimizovati?

Provera znanja

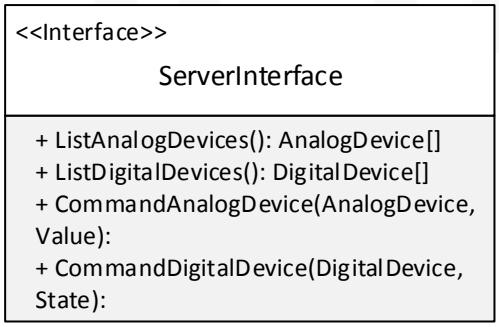
Provera 1

Potrebno je implementirati klijent-server aplikaciju sa sledećom specifikacijom:
Server sadrži model industrijskog postojenja – konfigurabilan broj analognih i digitalnih uređaja. Svaki analogni uređaj poseduje ime i vrednost tipa *double*. Svaki digitalni uređaj ima ime i stanje tipa *bool*. Imena uređaja mogu biti maksimalne dužine 16 znakova. Analogni i digitalni uređaji se čuvaju u dva niza i automatski generišu prilikom pokretanja servera na osnovu prosleđenog broja uređaja. Model podataka je prikazan na sledećoj slici:



Slika 15: UML ilustracija modela podataka

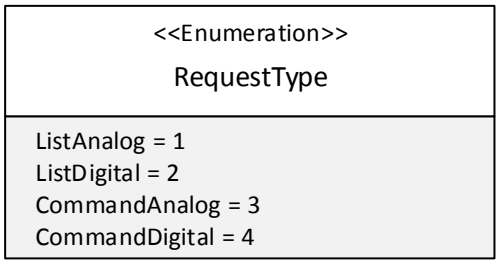
Server treba da omogući klijentu izlistavanje uređaja i komandovanje. Pod izlistavanjem uređaja se podrazumeva slanje imena i vrednosti/stanja uređaja. Komandovanje je jednostavna promena vrednosti/stanja odgovarajućeg uređaja.



Slika 16: Klijentska sprega sa serverom

Klijent treba da poseduje jednostavan tekstualni prikaz uređaja na serveru – tip, ime i vrednost/stanje uređaja. Klijent takođe treba da omogući komandovanje uređajem.

Sledi specifikacija komunikacionog protokola. Zahtevi koje šalje klijent trebaju biti sledećeg formata:
`[REQUEST TYPE][OPTIONAL REQUEST DATA]`
Sledeća slika prikazuje definiciju različitih tipova zahteva:



Slika 17: Definicija tipova zahteva

Tipovi zahteva i odgovora:

- Izlistavanje analognih uređaja – `ListAnalog`;
  Format zahteva: `[ListAnalog]`
  Format odgovora: `[ListAnalog][RESPONSE LEN][ANALOG DEVICES]`
- Izlistavanje digitalnih uređaja – `ListDigital`, celobrojna vrednost 1;
  Format zahteva: `[ListDigital]`
  Format odgovora: `[ListDigital][RESPONSE LEN][DIGITAL DEVICES]`
- Komandovanje analognim uređajem – `COMMAND_ANALOG`;
  Format zahteva: `[CommandAnalog][DEVICE_NAME][VALUE]`
  Format odgovora: nema odgovora.
- Komandovanje digitalnim uređajem – `COMMAND_DIGITAL`;
  Format zahteva: `[CommandDigital][DEVICE_NAME][VALUE]`
  Format odgovora: nema odgovora.

Klijent i server treba realizovati u programskom jeziku C kao dve konzolne aplikacije.

Dodatak

Izmeniti prethodni zadatak tako da se imena uređaja čuvaju u dinamički alociranim nizovima.

Literatura

1. Lectures in Advanced critical mission software for power grids,
   Branislav Atlagić
   Unpublished
2. Computer Networking
   James F. Kurose, Keith W. Ross,
   Pearson Education; International ed of 6th revised ed edition (May 1, 2012)
   ISBN-10: 0273768964
3. Transmission Control Protocol – RFC 793
   Darpa Internet Protocol,
   Information Sciences Institute University of Southern California, (September 1981)
4. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms – RFC 2001
   W. Stevens (January 1997)
5. ORM - Slajdovi
6. Windows Sockets 2, Microsoft Developer Network Library;
7. http://www.45nuclearplants.com, ilustracija nuklearne elektrane;
8. Transmission Control Protocol, Wikipedia Article;