Mrežno programiranje pomoću POSIX socket i Winsock programskog sučelja

Banić, Valerija Iva

Master's thesis / Diplomski rad

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:155448

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Valerija Iva Banić

MREŽNO PROGRAMIRANJE POMOĆU POSIX SOCKET I WINSOCK PROGRAMSKOG SUČELJA

Diplomski rad

Voditelj rada:
prof. dr. sc. Mladen Jurak

Zagreb, 2023.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

Računala, mobiteli i raznovrsni drugi uređaji međusobno komuniciraju putem interneta. U ovom radu proučava se pozadina te komunikacije i razvija jednostavni program. U prvom poglavlju opisuju se mrežni protokoli - standardi koji omogućavaju komunikaciju. U drugom poglavlju opisuje se adresiranje u mreži pomoću Internet Protocol-a. U trećem poglavlju opisuje se što su to utičnice i programska sučelja za njih. U četvrtom poglavlju bavimo se prijenosom podataka TCP protokolm. U petom poglavlju ukratko se upisuje HTTP. I na kraju, u šestom poglavlju koristimo POSIX socket programsko sučelje za razvoj krakog programa.

Poglavlje 1

Mrežni protokoli

Mreža računala je skup računala i uređaja koja mogu međusobno komunicirati. Čvor mreže je svako mjesto u mreži koje može slati podatke. Kako bi komunikacija tekla glatko moraju se dogovoriti standardi koji ju definiraju. Te standarde nazivamo mrežni protokoli. Protokoli su organizirani u slojeve. Svaki sloj pruža usluge sloju iznad i svaki viši sloj može se oslanjati na sloj ispod bez da zna kako radi.

1.1 OSI model

Prvi model za složeni mrežni protokol bio je Open Systems Interconnection model (OSI model) koji ima sedam slojeva:

1. Fizički sloj: Sloj fizičke komunikacije stvarnog svijeta. U ovom sloju imamo specifikacije za na primjer napon Ethernet kabla i radio frekvencija Wi-Fi-ja.

2. Sloj veze podataka: Ovaj sloj se nastavlja na fizički. Bavi se komunikacijom između dva izravno povezana čvora.

3. Mrežni sloj: Ostvaruje vezu između čvorova koji nisu fizički povezani, podaci prolaze kroz više međusobno vezanih čvorova. Bavi se adresiranjem i usmjeravanjem. Ovom sloju pripada Internet Protocol (IP).

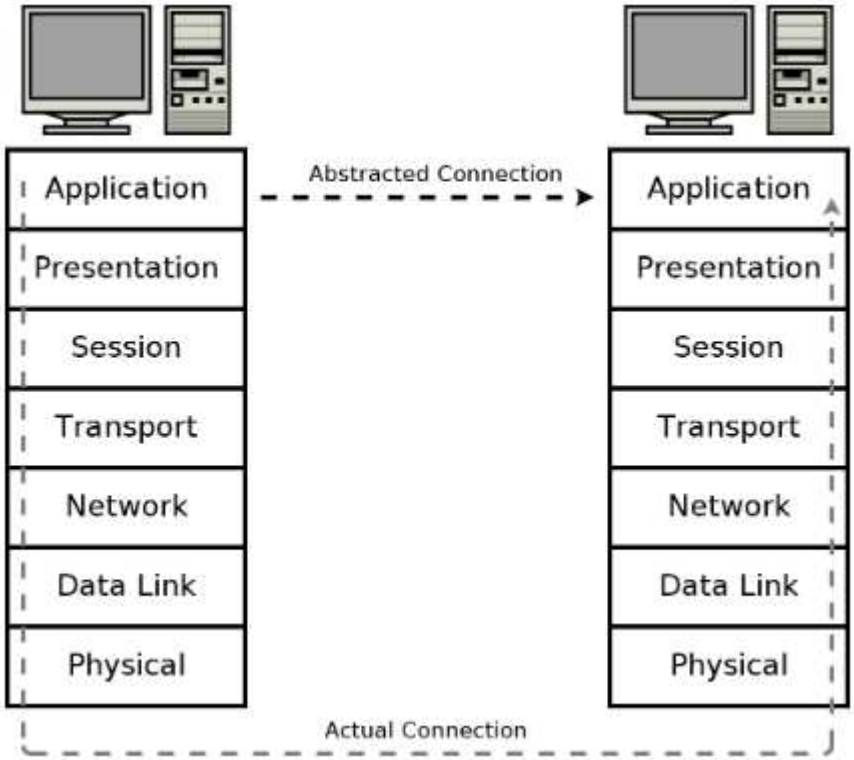
4. Transportni sloj: ostvaruje komunikaciiju između krajnih računala. Bavi se podijelom podataka, otkrivanjem i ispravljanjem pogrešaka, upravljanjem tokom i drugo. Transmission Control Protocol (TCP) i User Datagram Protocol (UDP) pripadaju ovom sloju.

5. Sloj sesije: Nastavlja se na transportni sloj dodajući metode za uspostavu, suspenziju, nastavak i završetak dijaloga.

6. Prezentacijski sloj: Najniži sloj koji se ne bavi razmjenom podataka, nego određuje prikaz i značenje informacije, definirane su strukture podataka i prezentacija aplikacije.

7. Aplikacijski sloj: Sadrži protokole kojii pružaju uslugu korisniku (na primjer internet preglednici i email klijent). HTTP pripada ovom sloju.

Podaci se stvaraju na razini aplikacije i prolaze kroz sve slojeve stoga prilikom slanja i primanja. Protokol iz aplikacijskog sloja ne mora znati kako se šalju podaci. To je ilustrirano dijagramom na slici 1.1

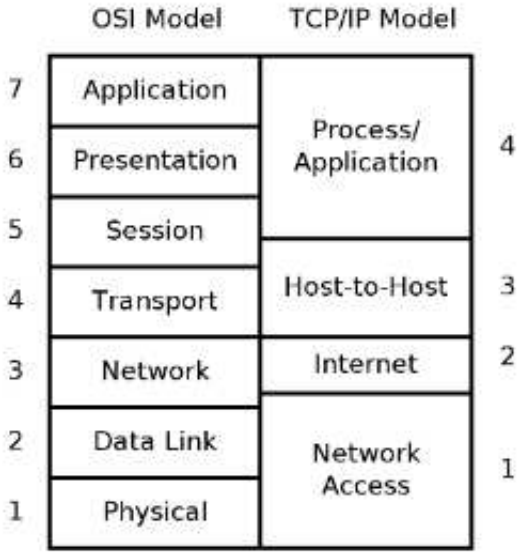


Slika 1.1: Razlika apstraktne i stvarne veze u OSI modelu

1.2 TCP/IP stog protokola

TCP/IP stog protokola najčešći je komunikacijski model. Za razliku od OSI modela, TCP/IP ima samo četiri sloja. Na slici 1.2 možemo vidjeti kako slojevi TCP/IP mo-

dela odgovaraju OSI modelu. Oba modela imaju iste funkcionalnosti, samo su drugačije raspoređene. TCP/IP model razvio se nakon što su protokoli TCP i IP već bili u upotrebi.



Slika 1.2: Usporedba slojeva OSI i TCP/IP modela

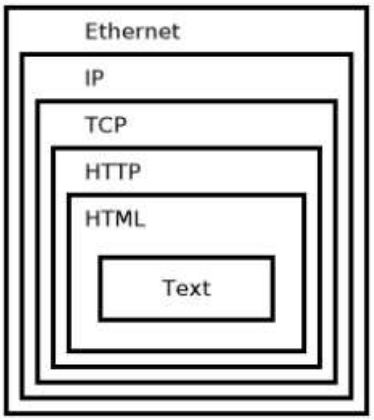
Slojevi TCP/IP modela su:

1. Mrežno sučelje - Ethernet, Wi-Fi
2. Internetski sloj - IP
3. Transportni sloj - TCP, UDP
4. Aplikacijski sloj - HTTP, SMTP, FTP

1.3 Enkapsulacija podataka

Protokoli nižih slojeva enkapsuliraju podatke za više slojeve. Prednost ove apstrakcije je da prilikom programiranja aplikacije koristimo samo protokol najviše razine. Na primjer, web preglednik implentira samo protokole koji su specificirani za web stranice - HTTP, HTML, CSS i tako dalje. Ne treba implementirati TCP/IP niti razumjeti kako se enkodiraju Ethernet paketi. Može se oslanjati na implementacije nižih slojeva.

Primjer 1.3.1. *Pogledajmo primjer web stranice koja sadrži nekoliko odlomaka teksta. Web poslužitelj ne šalje samo tekst. Tekst mora biti unutar HTML strukture, a šalje se kao*



Slika 1.3: Enkapsulacija teksta web stranice

dio HTTP odgovora. HTTP je dio TCP sesije, a TCP paket se usmjerava pomoću IP paketa koji se šalje žicom u Ethernet paketu. Primjer ovakve enkapsulacije vidimo na slici 1.3.

Poglavlje 2

Internet Protocol

Kako bi komunikacija u mreži mogla funkcionirati moramo imati jedinstven sistem adresiranja. Internet Protocol (IP) najvažniji je mrežni protokol koji definira IP adrese. Postoje dvije inačice protokola IP: IPv4 i IPv6. Danas je IPv4 najraširenij, a IPv6 se postupno uvodi. IPv4 koristi 32-bitne adrese što ograničava adresiranje na najviše 2^{32} adresa. IPv6 sa 128-bitnim adresama ima veće mogućnosti adresiranja.

2.1 IPv4

IP-adresa je identifikator koji globalno i jednoznačno označava čvor u mreži. U slučaju IPv4 taj je identifikator duljine 4 okteta, odnosno 32 bita. Uobičajeno IP-adresu zapisujemo tako da svaki oktet adrese pretvorimo u dekadski broj i odvojimo ih točkom.

Primjer 2.1.1. *Primjeri IPv4 adresa*

- *0.0.0.0*
- *127.0.0.1*
- *10.0.0.0*
- *172.16.0.5*
- *255.255.255.255*

Za raspon adresa u kojemu je početni dio adrese fiksan taj fiksni dio nazivamo prefiks, a varijabilni dio sufiks. CIDR (Classless Inter Domain Routing) notacija zapisuje raspon adresa kao uređeni par IP adrese i broja bitova u prefiksu koje odvaja kosom crtom "/". Na primjer raspon adresa od 10.0.0.0. do 10.255.255.255 u CIDR notaciji zapisujemo 10.0.0.0/8.

IPv4 rezervira neke adrese za posebnu namjenu. Adressa 0.0.0.0 odnosi se na ”ovo računalo”. Adresa 255.255.255.255 je takozvana limited broadcast adresa i odnosi se na sva računala lokalne mreže. Adrese 127.0.0.0/8 su loopback adrese koje služe za testiranje softvera, a najčešće se koristi 127.0.0.1. Tri bloka adresa rezervirana su za komunikaciju unutar privatne mreže. Tim adresama mrežni administrator slobodno raspolaže jer one ne moraju biti globalno jedinstvene. U tablici 2.1 možemo vidjeti koje su to adrese.

Raspon adresa	CIDR notacija
10.0.0.0 - 10.255.255.255	10.0.0.0/8
172.16.0.0. - 172.31.255.255	172.16.0.0/12
192.168.0.0-192.168.255.255	192.168.0.0/16

Tablica 2.1: IP-adrese za privatne mreže

2.2 IPv6

IPv6 razlikuje se od IPv4 ne samo po većoj mogućnosti adresiranja nego i mnogim drugim dodatnim svojstvima. Normalno je da IPv6 mrežna sučelja imaju nekoliko IPv6 adresa. IPv6 adrese koriste se drugačije od IPv4 adresa.

Adrese u IPv6 su dugačke 128 bitova i zapisujemo ih kao 8 grupa 4 heksadecimalnih znakova odvojenih točkom. Heksadecimalni znakovi su 0-9 i a-f. Standardno se koriste mala slova za IPv6 adrese.

Primjer 2.2.1. *Primjeri IPv6 adresa*

- *0000:0000:0000:0000:0000:0000:0000:0001*
- *2001:0db8:0000:0000:0000:ff00:0042:8329*
- *fe80:0000:0000:0000:75f4:ac69:5fa7:67f9*
- *ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff*

Postoje pravila za skraćivanje IPv6 adresa. Prvo pravilo kaže da u svakoj grupi vodeće nule možemo izostaviti. Drugo pravilo dopušta da uzastopne grupe nula zamijenimo dvostrukom dvotočkom (::). Drugo pravilo može biti upotrebljeno najviše jednom u adresi.

Primjer 2.2.2. *Koristeći pravila za skraćivanje IPv6 adresa, adrese iz primjera 2.2.1 možemo skratiti ovako:*

- *::1*
- *2001:db8::ff00:42:8329*
- *fe80::75f4:ac69:5fa7:67f9*
- *ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff*

IPv6, kao i IPv4, ima loopback adresu. To je ::1.

Posebna klasa IPv6 adresa mapira IPv4 adrese. Ove adrese počinju s 80 nula koje slijedi 16 jedinica i završava s 32-bitnom IPv4 adresom. Ovaj blok možemo zapisati u CIDR notaciji kao ::ffff:0:0/96. Ove adrese uobičajeno zapisujemo na način da je prvih 96 bitova u IPv6 formatu, a ostalih 32 bita u IPv4. Primjere takvih adresa možemo vidjeti u tablici 2.2.

IPv6 adresa	Mapirana IPv4 adresa
::ffff:10.0.0.0	10.0.0.0
::ffff:172.16.0.5	172.16.0.5
::ffff:192.168.0.1	192.168.0.1
::ffff:192.168.50.1	192.168.50.1

Tablica 2.2: IPv6 adrese i njihove mapirane IPv4 adrese

Site-local adrese su IPv6 adrese iz raspona fec0::/10 i koriste se za privatne lokalne mreže. Te adrese su sada zastarjele i ne smiju se koristiti za nove mreže, ali mnoge postojeće implementacije ih još uvijek koriste.

Link-local adrese su u bloku adresa IPv4 169.254.0.0/16 ili IPv6 fe80::/10. One se mogu koristiti samo putem lokalnog linka. Usmjernik nikada na prosljeđuje pakete s ovih adresa. Korisne su za upotrebu auto-konfiguracijskih funkcija prije nego je dodijeljena IP adresa.

2.3 DNS

Osim IP adrese, računala imaju i simboličku adresu u tekstualnom obliku. Sustav imena domena (engl. Domain Name System, DNS) zadužen je za međusobno pretvaranje simboličkih adresa i IP adresa. Razlučivanje adrese (address resolution) je postupak prevođenja simboličke adrese u IP adresu unutar DNS-a. Pri svakom pozivu internetske usluge u kojem se rabi simboličko ime poslužitelja provodi se razlučivanje simboličke adrese na način da klijent pošalje UDP poruku DNS poslužitelju i traži ga AAAA zapis za domenu. Ako ima takav zapis, DNS poslužitelj šalje IPv6 adresu klijentu koji onda može upostaviti vezu

s poslužiteljem na dobivenoj adresi. Ako nema AAAA zapisa, onda klijent ponovo šalje poruku DNS poslužitelju, ovaj put tražeći A zapis. Ako DNS poslužitelj ima A zapis, onda šalje IPv4 adresu tražene domene.

Poglavlje 3

Utičnice

Utičnica (engl. socket) je softverska struktura koja služi kao krajna točka komunikacije dva čvora u mreži. Specifikacije utičnica definirane su u aplikacijskom programskom sučelju (engl. Application Programming Interface, API). Berkeley sockets prvi je takav API. Uz male modifikacije, postale su dio POSIX (Portable Operating System Interface) standarda. Za programski jezik C, POSIX socket programsko je sučelje za rad sa utičnicama u UNIX sustavima. Windowsov API za rad s utičnicama je Winsock i modeliran je prema Berkeley sockets.

3.1 Razlike između POSIX socket i Winsock

Sada ćemo promotriti neke od razlika između POSIX socketa i Winsocka. Prve razlike koje možemo vidjeti su u zaglavlju. Na Windowsima uključujemo zaglavlja winsock.h i ws2tcpip.h i moramo definirati _WIN32_WINNT. Dodajemo i pragma komentar koji će povezati biblioteku ws2_32.lib.

```
#ifndef _WIN32_WINNT
#define _WIN32_WINNT 0x0600
#endif
#include <winsock2.h>
#include <ws2tcpip.h>
#pragma comment(lib, "ws2_32.lib")
```

Dok na Unixu zaglavlje izgleda ovako:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <errno.h>
```

Dok je Berkley sockets API spreman za korištenje, Winsock je potrebno inicijalizirati.

```
WSADATA d;
if (WSAStartup(MAKEWORD(2, 2), &d)) {
    fprintf(stderr, "Failed to initialize.\n");
    return 1;
}
```

Makro MAKEWORD() koristimo kako bismo specificirali verziju Winsock-a. Koristimo verziju 2.2. Također, na kraju programa moramo pozvati funkciju WSACleanup().

Na Unixu, socket descriptor je standard file descriptor. To znači da se mogu koristiti sve standarne UNIX file I/O funkcije. To nije moguće na Windowsima.

Sljedeće razlike odnose se na funkciju socket(). Na Unixu funkcija socket() vraća int i to 0 ako je poziv bio uspješan, a za nesuspjeh negativnu vrijednost. Na Windowsima vraća SOCKET - typedef za unsigned int, a ako je poziv bio neuspješan vraća INVALID_SOCKET.

Ako je došlo do greške u socket funkcijama kao što su socket(), bind() i accept(), kod pogreške možemo saznati pozivom funkcije WSAGetLastError() na Windowsima, dok je na Unixu posljednja greška spremljena u globalnoj varijabli errno.

Zatvaranje utičnice još je jedna bitna razlika između POSIX socketa i Winsocka. Na Windowsima se korsiti funkcija closesocket(), a na Unixu close().

3.2 Vrste utičnica

Utičnice mogu biti bezspojne i spojno-orijentirane, što ovisi o tipu protokola koji se koristi. User Datagram Protokol (UDP) implementira bezspojnu komunikaciju što znači da se svaka poruka šalje neovisno. Poruke mogu doći drugačijim redoslijedom nego što su poslane te nema garancije da će svaka poruka stići. UDP se koristi kada nije bitno dođe li svaka poruka nego je bitnija brzina prijenosa. Na primjer kod videa izravnog prijenosa nema smisla ispravljati greške, propušteni podaci više nisu važni. Za razliku od UDP-a, Transmission Control Protocol (TCP) implementira spojnu komunikaciju. Tek nakon što se uspostavi veza, šalju se podaci. TCP garantira da će podaci doći redom kojim su poslani, bez gubitaka i duplikacije. O TCP-u ćemo više govoriti u sljedećem poglavlju.

3.3 Socket funkcije

Programska sučelja za utičnice imaju mnoge funkcije, ovdje navodimo neke:

- socket() - stvara i inicijalizira utičnicu;
- bind() - povezuje utičnicu s određenom lokalnom IP adresom i portom;
- listen() - koristi se na poslužitelju kako bi TCP utičnica slušala nove veze;
- connect() - koristi se na klijentu da se postavi udaljena adresa i port. U slučaju TCP-a, uspostavlja vezu;
- accept() - kreira novu utičnicu za nadolazeću TCP vezu, koristi se na poslužitelju;
- send() - šalje podatke utičnicom;
- recv() - prima podatke utičnicom;
- sendto() - šalje podatke utičnicom bez povezane udaljene adrese;
- recvfrom() - prima podatke utičnicom bez povezane udaljene adrese;
- close() (Berkeley sockets) i closesocket() (Winsock sockets) - koriste se za zatvaranje utičnice. Također zatvara TCP vezu;
- shutdown() - zatvara jednu stranu TCP veze;
- select() - čeka događaj na jednoj ili više utičnica;
- setsockopt() - koristi se za promjenu nekih opcija utičnice;
- fcntl() (Berkeley sockets) i ioctlsocket() (Winsock sockets) - koristi se za promjenu nekih opcija utičnice.

Poglavlje 4

TCP

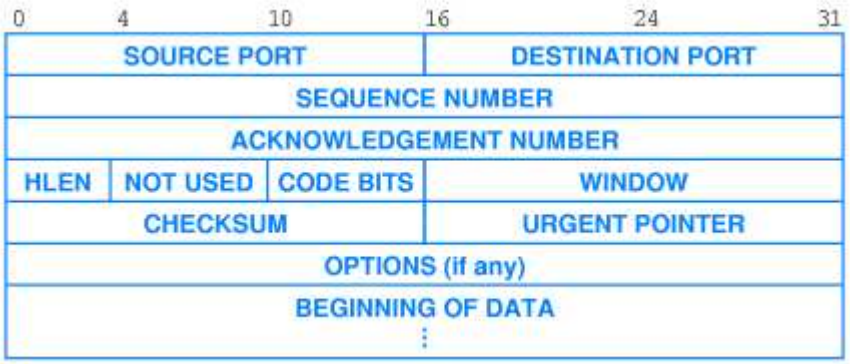
TCP (Transmission Control Protokol) je protokol koji upravlja prijenosom podataka. Osobine TCP-a:

- Spojna usluga: Prijenos podataka počinje tek nakon što se uspostavila veza.
- Pouzdanost: TCP koristi razne tehnike kako bi osigurao da će podaci doći redom kojim su poslani, bez gubitaka i duplikacije.
- Puni dupleks: U svakom trenutku oba čvora mogu slati podatke.
- Stream Interface: Pristigli podaci čine kontinuirani niz. Primatelj ih čita neovisno o tome kako su bili podijeljeni tijekom slanja.
- Pouzdano otvaranje veze: Oba čvora moraju pristati na komunikaciju da bi se veza uspostavila.
- Pouzdano zatvaranje veze: Svi poslani podaci moraju biti dostavljeni prije zatvaranja veze.
- Kontrola zagušenja: Kada dođe do kašnjenja TCP dijeli podatke u male pakete, usporava dinamiku slanja i zatim postupno povećava veličinu paketa.

Port je 16-bitna transportna adresa koja definira logičku vezu između dva procesa. Za standardne internetske usluge definirani su brojevi tzv. well-known ports koji pripadaju rasponu 0-1023. Npr. port 21 rezerviran je za FTP, 22 za SSH, 25 za SMTP, 80 za HTTP, 110 za POP3, itd.

4.1 TCP-segment

TCP-paket naziva se TCP-segment. Sastoji se od zaglavlja i podatkovnog polja u kojem prenosi podatke aplikacijskog sloja. Na slici 4.1 možemo vidjeti format TCP-segmenta.



Slika 4.1: Format TCP-segmenta

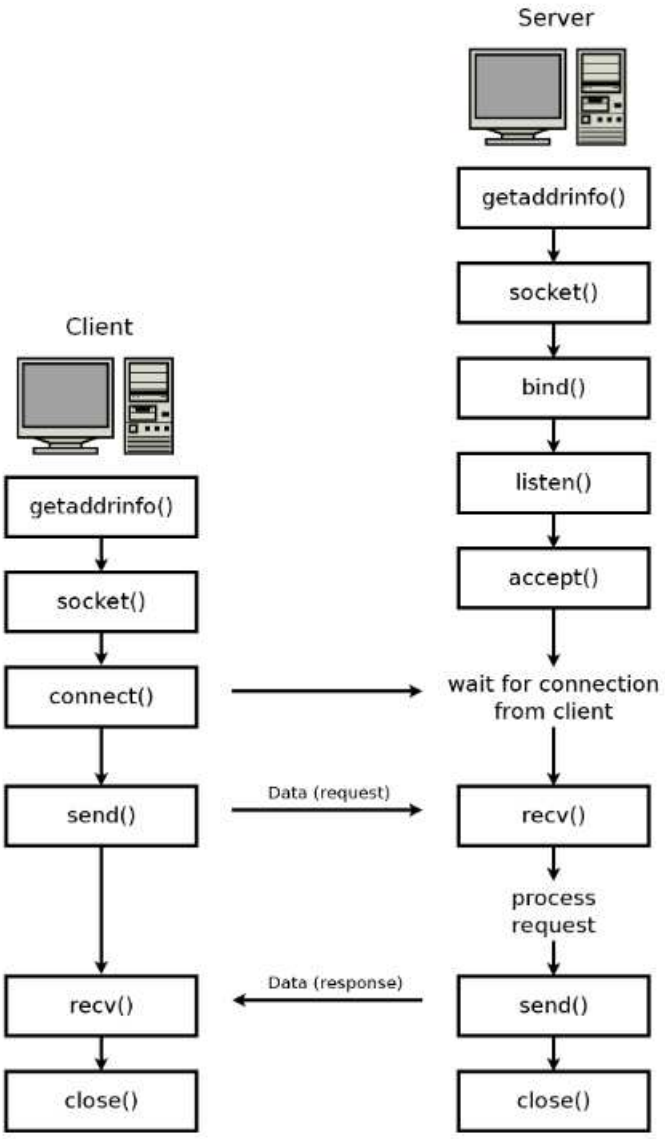
TCP-zaglavlje sadrži:

- source port (16 bita) - broj porta procesa pošiljatelja;
- destination port (16 bita) - broj porta procesa primatelja;
- sequence number (32 bita) - redni broj prvog okteta segmenta koji se šalje;
- acknowledgment number (32 bita) - redni broj sljedećeg paketa;
- hlen (4 bita) - duljina zaglavlja;
- not used (6 bita) - trenutno se ne koriste, rezervirani za buduće upotrebe;
- code bits (6 bita) - bitovi za potvrdu i upravljanje transportom;
- window (16 bita) - najveći dopušteni broj okteta koje primatelj smije poslati prije primitka potvrde;
- checksum (16 bita) - kontrolna suma za TCP zaglavlje i podatke, pomaže u otkrivanju pogreške;
- urgent pointer (16 bita) - pokazivač na početak podatka koji zahtijeva prioritetnu obradu;
- options - posebne mogućnosti

4.2 TCP programski tok

TCP klijent mora znati adresu TCP poslužitelja. Klijent uzima tu adresu i pretvara ju u strukturu struct addrinfo koristeći funkciju getaddrinfo(). Zatim, pozivom funkcije socket(), stvara utičnicu. Onda klijent stvara novu TCP vezu pozivom connect(). Sada klijent može slati i primati podatke koristeći funkcije send() i recv().

TCP poslužitelj prvo, koristeći getaddrinfo(), inicijalizira strukturu struct addrinfo s odgovarajućom IP adresom i portom. Zatim stvara utičnicu pozivom funkcije socket(). Pozivom funkcije bind(), poslužitelj povezuje utičnicu s IP adresom i portom. Poslužitelj poziva funkciju listen() koja stavlja utičnicu u stanje slušanja novih veza. Poslužitelj onda poziva funkciju accept() koja čeka dok klijent ne uspostavi vezu te tada vraća novu utičnicu. Dok prva utičnica nastavlja slušati, nova utičnica može izmijenjivati podatke funkcijama send() i recv(). Na slici 4.2 se nalazi grafički prikaz programskog toka TCP klijenta i poslužitelja. Nema pravila koja strana prva poziva send() ili recv(), ni koliko puta.



Slika 4.2: Programski tok

Poglavlje 5

HTTP

HyperText Transfer Protocol (HTTP) aplikacijski je protokol koji se koristi za komunikaciju web klijenta i web poslužitelja. HTTP koristi TCP vezu preko porta 80. U HTTP-u prvo web klijent šalje HTTP zahtjev web poslužitelju. Zatim web poslužitelj šalje HTTP odgovor.

HTTP zahtjev

Nekoliko najčešćih metoda HTTP zahtjeva:

- GET se koristi kada klijent zahtijeva određeni podatak;
- HEAD se koristi kada klijent traži samo informaciju o podatku, a ne i sami podatak;
- POST se koristi kada klijent šalje podatke poslužitelju.

HTTP zahtjev sastoji se od sljedećih dijelova:

- zahtjev: počinje metodom zahtjeva, nakon koje slijedi jednostruki razmak (SP), zatim URI, ponovno jednostruki razmak, zatim verzija protokola i završava s CRLF;
- zaglavlje: svaka linija zahtjeva sastoji se od naziva atributa, nakon kojeg slijedi dvotočka (":"), opcionalna praznina, vrijednost atributa i opcionalna praznina;
- tijelo poruke (nije obavezno).

HTTP odziv

HTTP odziv počinje statusnom linijom koja se sastoji od verzije protokola i koda HTTP odziva. Nakon statusne linije, kao i kod HTTP zahtjeva slijedi zaglavlje i na kraju tijelo poruke, ako je potrebno. Ako je HTTP zahtjev bio tipa HEAD, tijelo poruke se ne šalje.

Ako je zahtjev bio uspješan, poslužitelj odgovara kodom iz klase 2xx, na primjer:

- 200 OK: Zahtjev je uspio, poruka sadrži traženi objekt;
- 201 Created: Zahtjev je uspio i kreiran je jedan ili više novih podataka;
- 202 Accepted: Zahtjev je zaprimljen, ali obrada nije još završena;
- 204 No Content: Zahtjev je uspio i nema dodatnog sadržaja u tijelu poruke.

Ako je podatak premješten, poslužitelj može odgovoriti kodom iz klase 3xx, na primjer:

- 301 Moved Permanently: Traženi objekt je premješten na novu trajnu lokaciju, zaglavlje sadrži atribut Location čija je vrijednost novi trajni URI. Svi budući zahtjevi trebaju koristiti novu lokaciju;
- 307 Temporary Redirect: Traženi objekt je premješten na novu privremenu lokaciju, zaglavlje sadrži atribut Location čija je vrijednost novi trenutni URI. Budući zahtjevi trebaju koristiti originalnu lokaciju.

Ako je došlo do greške, poslužitelj odgovara kodom iz klase 4xx za greške klijenta ili iz klase 5xx za greške poslužitelja, na primjer:

- 400 Bad Request: Poslužitelj ne razumije klijentov zahtjev;
- 401 Unauthorized: Klijent nije autoriziran za traženi objekt;
- 403 Forbidden: Klijentu je zabranjeno pristupiti traženom objektu;
- 404 Not Found: Traženi objekt se ne nalazi na poslužitelju;
- 500 Internal Server Error: Došlo je do greške na poslužitelju prilikom izvršavanja zahtjeva;
- 503 Service Unavailable: Poslužitelj trenutno nije u mogućnosti obraditi zahtjev. Poslužitelj može u zaglavlju dodati atribut Retry-After u kojem predlaže klijentu koliko da pričeka prije ponovnog slanja zahtjeva.

Poglavlje 6

Implementacija

Implementirati ćemo TCP klijenta i TCP poslužitelja sobe za čavrljanje prema [6]. Koristit ćemo POSIX socket programsko sučelje.

6.1 Zaglavlje

Za početak trebamo uključiti API datoteke zaglavlja.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <errno.h>

#define ISVALIDSOCKET(s) ((s) >= 0)
#define CLOSESOCKET(s) close(s)
#define SOCKET int
#define GETSOCKETERRNO() (errno)
```

Trebaju nam i standardne datoteke zaglavlja.

```
#include <stdio.h>
#include <string.h>
```

6.2 Klijent

Adresu servera i port primamo kao argumente komandne linije. Zato trebamo prvo provjeriti broj argumenata.

```
int main(int argc, char *argv[]) {
    if (argc < 3) {
        fprintf(stderr, "usage: tcp_client hostname port\n");
        return 1;
    }
```

Zatim konfiguiramo adresu koristeći getaddrinfo(). Budući da želimo TCP vezu postavljamo hints.ai_socktype = SOCK_STREAM. Ne moramo specificirati koristi li se protokol IPv4 ili IPPv6 jer getaddrinfo() može odlučiti koji protokol treba koristiti. Adresu i port šaljemo kao argumente komande linije. Ako je funkcija getaddrinfo() vratila 0 znači da je sve dobro prošlo i adresa je spremljena u peer_address. Inače ispisujemo grešku i završavamo program.

```
    printf("Configuring remote address...\n");
    struct addrinfo hints;
    memset(&hints, 0, sizeof(hints));
    hints.ai_socktype = SOCK_STREAM;
    struct addrinfo *peer_address;
    if (getaddrinfo(argv[1], argv[2], &hints, &peer_address)) {
        fprintf(stderr, "getaddrinfo() failed. (%d)\n",
                GETSOCKETERRNO());
        return 1;
    }
```

Zatim kreiramo utičnicu.

```
    printf("Creating socket...\n");
    SOCKET socket_peer;
    socket_peer = socket(peer_address->ai_family,
            peer_address->ai_socktype, peer_address->ai_protocol);
    if (!ISVALIDSOCKET(socket_peer)) {
        fprintf(stderr, "socket() failed. (%d)\n",
                GETSOCKETERRNO());
        return 1;
    }
```

Postavljamo udaljenu adresu i port te uspostavljamo vezu. Funkcija connect() vraća 0 ako je sve uspjelo.

```
printf("Connecting...\n");
if (connect(socket_peer,
            peer_address->ai_addr, peer_address->ai_addrlen)) {
    fprintf(stderr, "connect() failed. (%d)\n",
            GETSOCKETERRNO());
    return 1;
}
freeaddrinfo(peer_address);

printf("Connected.\n");
printf("To send data, enter text followed by enter.\n");
```

Sada program stavljamo u beskonačnu petlju u kojoj provjeravamo je li korisnik unio novi podatak u terminal, jesmo li primili novi podatak od poslužitelja te šaljemo eventualne podatke iz terminala poslužitelju.

```
while(1) {

    fd_set reads;
    FD_ZERO(&reads);
    FD_SET(socket_peer, &reads);
    FD_SET(0, &reads);

    struct timeval timeout;
    timeout.tv_sec = 0;
    timeout.tv_usec = 100000;

    if (select(socket_peer+1, &reads, 0, 0, &timeout) < 0) {
        fprintf(stderr, "select() failed. (%d)\n",
                GETSOCKETERRNO());
        return 1;
    }

    if (FD_ISSET(socket_peer, &reads)) {
        char read[4096];
        int bytes_received = recv(socket_peer, read, 4096, 0);
        if (bytes_received < 1) {
```

```
                printf("Connection closed by peer.\n");
                break;
            }
            printf("Received (%d bytes): %.*s",
                    bytes_received, bytes_received, read);
        }
        if(FD_ISSET(0, &reads)) {

            char read[4096];
            if (!fgets(read, 4096, stdin)) break;
            printf("Sending: %s", read);
            int bytes_sent = send(socket_peer, read,
                                  strlen(read), 0);
            printf("Sent %d bytes.\n", bytes_sent);
        }
    }
```

Na kraju zatvaramo utičnicu i završavamo program.

```
    printf("Closing socket...\n");
    CLOSESOCKET(socket_peer);

    printf("Finished.\n");
    return 0;
}
```

6.3 Server

Dohvaćamo svoju lokalnu adresu, kreiramo utičnicu, pozivamo bind() kako bi povezali utičnicu s lokalnom adresom te pozivamo funkciju listen() kako bi utičnica ušla u stanje slušanja.

```
#include <ctype.h>

int main() {
    printf("Configuring local address...\n");
    struct addrinfo hints;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
```

```
hints.ai_flags = AI_PASSIVE;

struct addrinfo *bind_address;
getaddrinfo(0, "8080", &hints, &bind_address);

printf("Creating socket...\n");
SOCKET socket_listen;
socket_listen = socket(bind_address->ai_family,
        bind_address->ai_socktype, bind_address->ai_protocol);
if (!ISVALIDSOCKET(socket_listen)) {
    fprintf(stderr, "socket() failed. (%d)\n",
            GETSOCKETERRNO());
    return 1;
}

printf("Binding socket to local address...\n");
if (bind(socket_listen,
            bind_address->ai_addr, bind_address->ai_addrlen)) {
    fprintf(stderr, "bind() failed. (%d)\n", GETSOCKETERRNO());
    return 1;
}
freeaddrinfo(bind_address);

printf("Listening...\n");
if (listen(socket_listen, 10) < 0) {
    fprintf(stderr, "listen() failed. (%d)\n",
            GETSOCKETERRNO());
    return 1;
}
```

Definiramo fd_set strukturu koja ce spremati sve aktivne utičnice. Koristit ćemo varijablu max_socket u kojoj ćemo čuvat najveći socket deskriptor. Budući da trenutno imamo samo jednu utičnicu, postavljamo max_socket na socket_listen.

```
fd_set master;
FD_ZERO(&master);
FD_SET(socket_listen, &master);
SOCKET max_socket = socket_listen;
```

Zatim ispisujemo statusnu poruku i ulazimo u glavnu petlju. Kopiramo fd_set master u reads kako ne bismo izgubili podatke pri pozivu funkcije select(). Funkciji select()

prosljeđujemo 0 kao timeout value kako se program ne bi vratio dok utičnica iz skupa master nije spremna za čitanje. Na početku programa master sadrži samo socket_listen, no tijekom izvođenja programa, dodajemo svaku novu vezu u master.

```
printf("Waiting for connections...\n");


while(1) {
    fd_set reads;
    reads = master;
    if (select(max_socket+1, &reads, 0, 0, 0) < 0) {
        fprintf(stderr, "select() failed. (%d)\n",
                GETSOCKETERRNO());
        return 1;
    }
```

Sada prolazimo u petlji kroz sve utičnice i provjeravamo je li select() označio da je neka utičnica spremna. FD_ISSET() je istina samo za utičnice koje su spremne za čitanje.

```
    SOCKET i;
    for(i = 1; i <= max_socket; ++i) {
        if (FD_ISSET(i, &reads)) {
```

Ako je socket_listen spremna za čitanje onda imamo novu vezu koju uspostavljamo s accept(). Koristimo FD_SET() kako bismo dodali utičnicu od nove veze u master.

```
            if (i == socket_listen) {
                struct sockaddr_storage client_address;
                socklen_t client_len = sizeof(client_address);
                SOCKET socket_client = accept(socket_listen,
                        (struct sockaddr*) &client_address,
                        &client_len);
                if (!ISVALIDSOCKET(socket_client)) {
                    fprintf(stderr, "accept() failed. (%d)\n",
                            GETSOCKETERRNO());
                    return 1;
                }

                FD_SET(socket_client, &master);
```

```
                    if (socket_client > max_socket)
                        max_socket = socket_client;

                    char address_buffer[100];
                    getnameinfo((struct sockaddr*)&client_address,
                            client_len,
                            address_buffer, sizeof(address_buffer),
                            0, 0,
                            NI_NUMERICHOST);
                    printf("New connection from %s\n",
                            address_buffer);
```

Za ostale utičnice, koje su spremne za čitanje, koristimo recv() kako bismo pročitali podatke koje zatim šaljemo svim ostalim utičnicama.

```
                } else {
                    char read[1024];
                    int bytes_received = recv(i, read, 1024, 0);
                    if (bytes_received < 1) {
                        FD_CLR(i, &master);
                        CLOSESOCKET(i);
                        continue;
                    }

                    SOCKET j;
                    for (j = 1; j <= max_socket; ++j) {
                        if (FD_ISSET(j, &master)) {
                            if (j == socket_listen || j == i)
                                continue;
                            else
                                send(j, read, bytes_received, 0);
                        }
                    }
                }
            } //if FD_ISSET
        } //for i to max_socket
    } //while(1)
```

Iako program neće izaći iz while petlje, program završavamo zatvaranjem socket_listen.

```
    printf("Closing listening socket...\n");
    CLOSESOCKET(socket_listen);

    printf("Finished.\n");

    return 0;
}
```

Sada kada imamo programe možemo ih testirati. Prvo ih trebamo kompajlirati ovako:

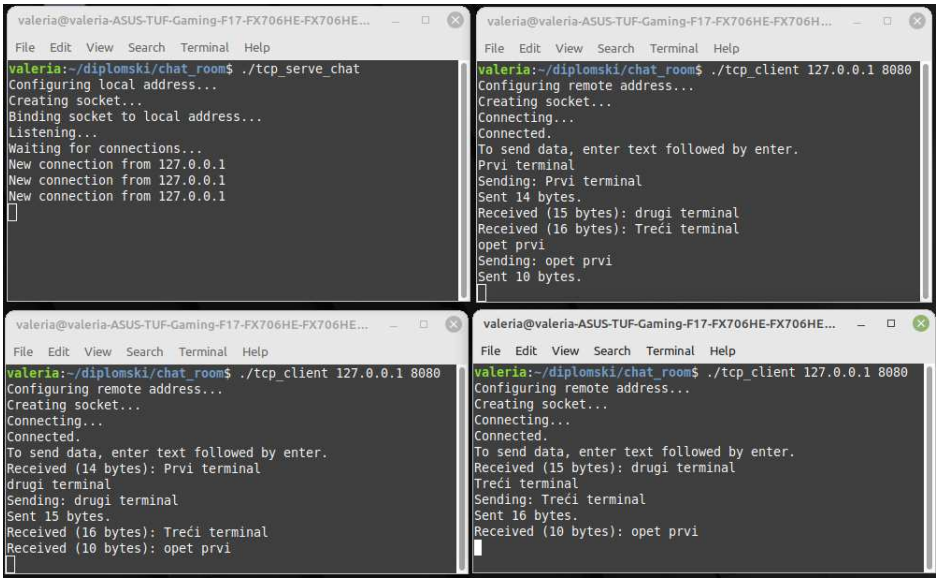
```
gcc tcp_client.c -o tcp_client
gcc tcp_serve_chat.c -o tcp_serve_chat
```

Zatim možemo pokrenuti program poslužitelja u jednom terminalu:

```
./tcp_serve_chat
```

Za testiranje su nam potrebna dva ili više klijentskih programa koje pokrećemo u drugim terminalima. Potrebni su argumenti adresa i port, koristimo loopback adresu i port 8080:

```
tcp_client 127.0.0.1 8080
```



Slika 6.1: Testiranje programa

Na slici 6.1 testirano je tako da je u gornjem lijevom terminalu pokenut poslužitelj. Zatim su pokrenuti klijentski programi u gornjem desnom i donjem lijevom terminalu te

je prvi od njih poslao poruku. Onda je pokrenut još jedan klijentski program u donjem desnom terminalu. Zatim svi klijenti šalju poruku. Možemo vidjeti da su sve poruke prosljeđene svim spojenim klijentima.

Bibliografija

[1] Alen Bažant, Gordan Gledec, Željko Ilić, Gordan Ježić, Mladen Kos, Marijan Kunštić, Ignac Lovrek, Maja Matijašević, Branko Mikac i Vjekoslav Sinkov, *Osnovne arhitekture mreža*, Element, 2003.

[2] R. Fielding i J. Reschke, *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*, (2014), `https://datatracker.ietf.org/doc/html/rfc7230`.

[3] ———, *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*, (2014), `https://datatracker.ietf.org/doc/html/rfc7231`.

[4] Ignac Lovrek, Maja Matijašević, Gordan Ježić i Dragan Jevtić, *Komunikacijske mreže*, 2014.

[5] Mario Radovan, *Računalne mreže (1)*, Digital point tiskara, 2010.

[6] Lewis Van Winkle, *Hands-On Network Programming with C: Learn socket programming in C and write secure and optimized network code*, Packt Publishing Ltd, 2019.

Sažetak

U ovom radu opisuje se mrežno programiranje. Rad daje uvid u mrežne protokole kroz slojeve te detaljnije opisuje IP, TCP i HTTP protokole. Opisuju se programska sučelja POSIX socket i Winsock. U radu su još implementirni klijent i poslužitelj sobe za čavrljanje.

Summary

This thesis describes network programming. It provides insight into network protocols and its layers and describes the IP, TCP and HTTP protocols in more detail. POSIX socket and Winsock programming interfaces are described. The chat room client and server are also implemented in this thesis.

Životopis

Valerija Iva Banić rođena je 13. listopada 1996. u Zagrebu, gdje završava osnovnu školu i prirodoslovno-matematičku gimnaziju. Preddiplomski studij Matematika na Prirodslovno-matematičkom fakultetu Sveučilišta u Zagrebu upisuje 2015., a 2018. upisuje diplomski studij Računarstvo i matematika.