Izrada poslužitelja za povezivanje ravnopravnih čvorova u sustavu za strujanje videa

Zovko, Dragan

Undergraduate thesis / Završni rad

2019

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:199884

Rights / Prava: Attribution 3.0 Unported/Imenovanje 3.0

Download date / Datum preuzimanja: **2024-03-29**



Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository







SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Dragan Zovko

IZRADA POSLUŽITELJA ZA POVEZIVANJE RAVNOPRAVNIH ČVOROVA U SUSTAVU ZA STRUJANJE VIDEA

ZAVRŠNI RAD

Varaždin, 2019.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Dragan Zovko

Matični broj: 37045/08–I

Studij: Informacijski sustavi

IZRADA POSLUŽITELJA ZA POVEZIVANJE RAVNOPRAVNIH ČVOROVA U SUSTAVU ZA STRUJANJE VIDEA

ZAVRŠNI RAD

Mentor:

Doc. dr. sc. Nikola Ivković

Varaždin, travanj 2019.

Dragan Zovko

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

U ovom radu se opisuje izrada poslužitelja koji se treba izvoditi na operativnom sustavu Linux, a poslužitelj treba biti napravljen u C++ programskom jeziku sa naglaskom na socket programiranje i višedretveno programiranje. Poslužitelj je dio aplikacije koja se izvodi na ravnopravnim krajnjim čvorovima (p2p), a služit će za uspostavu komunikacije krajnjih čvorova koji ne mogu direktno komunicirati putem interneta jer se jedan ili oba nalaze u lokalnim mrežama iza mehanizma za prevođenje lokalnih adresa (engl. *network address translation*, NAT). Poslužitelj, kojega možemo nazvati poslužiteljem za sastajanje, je dio sustava i koristi protokol koji je definiran u radu N.Ivković, I.Magdalenić, L.Milić, „An Ad-Hoc Smartphone-to-Smartphone Live Multimedia Streaming Application with Real-Time Constraints", *Journal of Advances in Computer Networks,* vol. 4, no. 1, March 2016. Taj poslužitelj treba imati poznatu javnu IP adresu, a prijenos svih aplikacijskih poruka odvija se putem transportnog protokola UDP. Komunikacija se uspostavlja između dva krajnja čvora od kojih je jedan izvor videa (multimedije), a drugi prijemnik. Povezivanje se obavlja putem jednokratnog koda preko poslužitelja za sastajanje (povezivanje). U sustavu postoji i još jedan poslužitelj koji služi za posredovanje u prijenosu videa ako nije moguća direktna komunikacija između krajnjih čvorova.

Ključne riječi: p2p, socket programiranje, poslužitelj, sustav ravnopravnih čvorova

Sadržaj

1 Uvod....1
2 Arhitektura sustava i primjer komunikacije....3
3 Poslužitelj za prijavu (pronalaženje)....5
3.1 Faza inicijalizacije....5
3.1.1 Zašto UDP, a ne TCP?....11
3.1.1.1 Multipleksiranje i demultipleksiranje....13
3.1.1.2 UDP segment....13
3.2 Faza radnog režima....15
3.2.1 Sinkronizacija dretvi s pomoću mutexa....21
3.2.2 Sinkronizacija dretvi s pomoću uvjetnih varijabli....23
3.2.3 Klasa KruzniSpremnik....24
3.2.4 Poruke aplikacijskog sloja....26
3.2.5 Primjer obrade poruke u klasi PorukaMajstor....30
3.2.6 Baza podataka implementirana pomoću asocijativnog spremnika....34
4 Pomoćni program za testiranje poslužitelja....35
5 Zaključak....37
6 Popis literature....38
7 Popis slika....39
8 Popis tablica....40

1 Uvod

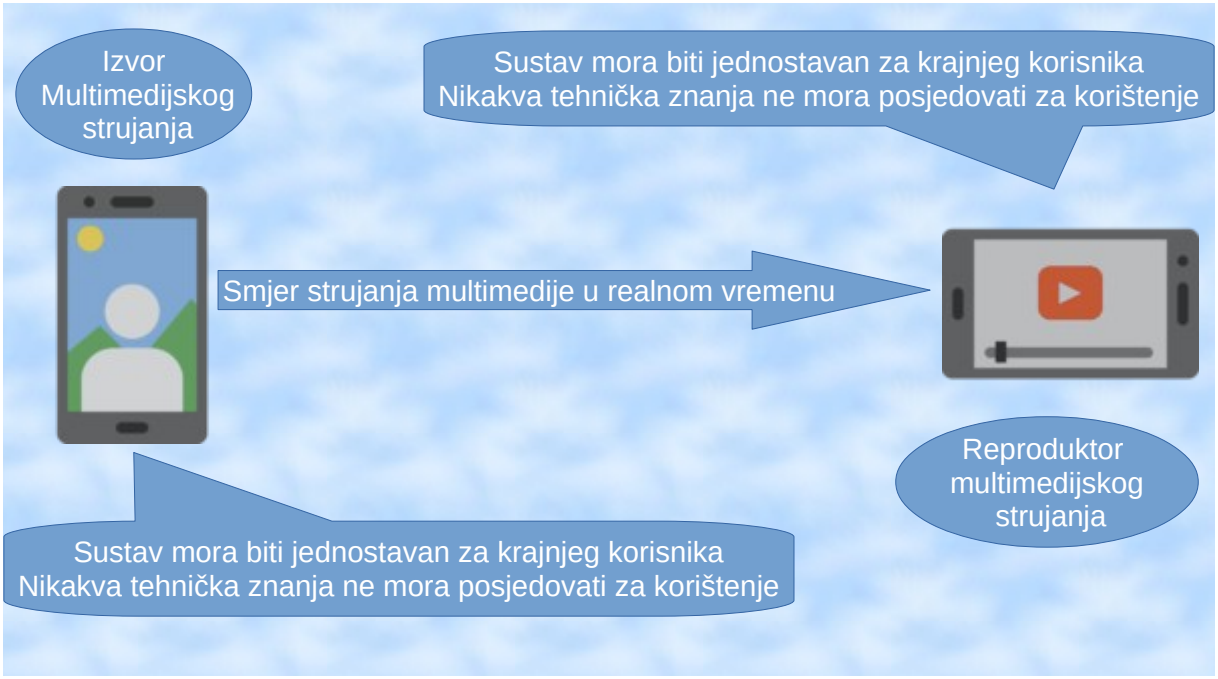
Uspostavljanje komunikacije između dva uređaja koja se nalaze iza mehanizma za prevođenje mrežnih adresa (engl. *network address translation*, NAT) nije jednostavan zadatak, ali je od presudne važnosti za aplikaciju za strujanje videa. Glavni cilj ovog završnog rada je izraditi poslužitelj koji će djelovati u sustavu za strujanje multimedije, a omogućiti će pronalaženje izvora multimedijskog strujanja te pomoć oko uspostave komunikacije. Poslužitelj je potrebno izraditi u programskom jeziku C++ jer mora moći obraditi veliki broj zahtijeva u realnom vremenu, a iz tog razloga je odabran i Linux kao operativni sustav. Da bi se zadovoljili strogi zahtjevi za rad u realnom vremenu kao protokol za prijenos podataka je izabran transportni protokol UDP, a i razvijen je aplikacijski protokol za razmjenu poruka:

Kao što je navedeno u [1] sustav se sastoji od:

1. Aplikacije koja je izvor/reproduktor multimedijskog strujanja
2. Poslužitelja za pronalaženje (prijavu)
3. Poslužitelja posrednika

Sustav je zamišljen tako da izvor multimedijskog strujanja koji može biti pametni telefon ili neki sličan uređaj koji putem interneta šalje video (i zvuk) nekom drugom pametnom telefonu koji je reproduktor multimedije. Izvor može biti i računalo sa kamerom koje se nalazi na dronu, a ima mogućnost spajanja na internet putem GSM mreže ili na neki drugi način bežično.

Kao što se vidi na slici 1 krajnji korisnik vidi samo izvor ili reproduktor multimedijskog strujanja. Vrijeme kašnjenja koje je dozvoljeno mora biti odabrano tako da se ne izgubi osjećaj realnog vremena jer uvijek postoji nekakvo kašnjenje kao što je rečeno u [2, str. 35]. Smisao postojanja takvog sustava je da osoba koja gleda video mora biti sigurna da gleda video u realnom vremenu. Kao kod automobila koji imaju kameru za parkiranje ili kod nekih drugih sustava gdje se na osnovu gledanja videa u realnom vremenu upravlja kretanjem nekakvih objekata.



Slika 1: Izvor i reproduktor

Ako prikaz multimedije kasni za unaprijed određeni vremenski iznos na reproduktoru to mora biti jasno prikazano, zvučno i vizualno kao na slici 2.



Slika 2: Obavijest o kašnjenju vizualno i kratki zvučni signal

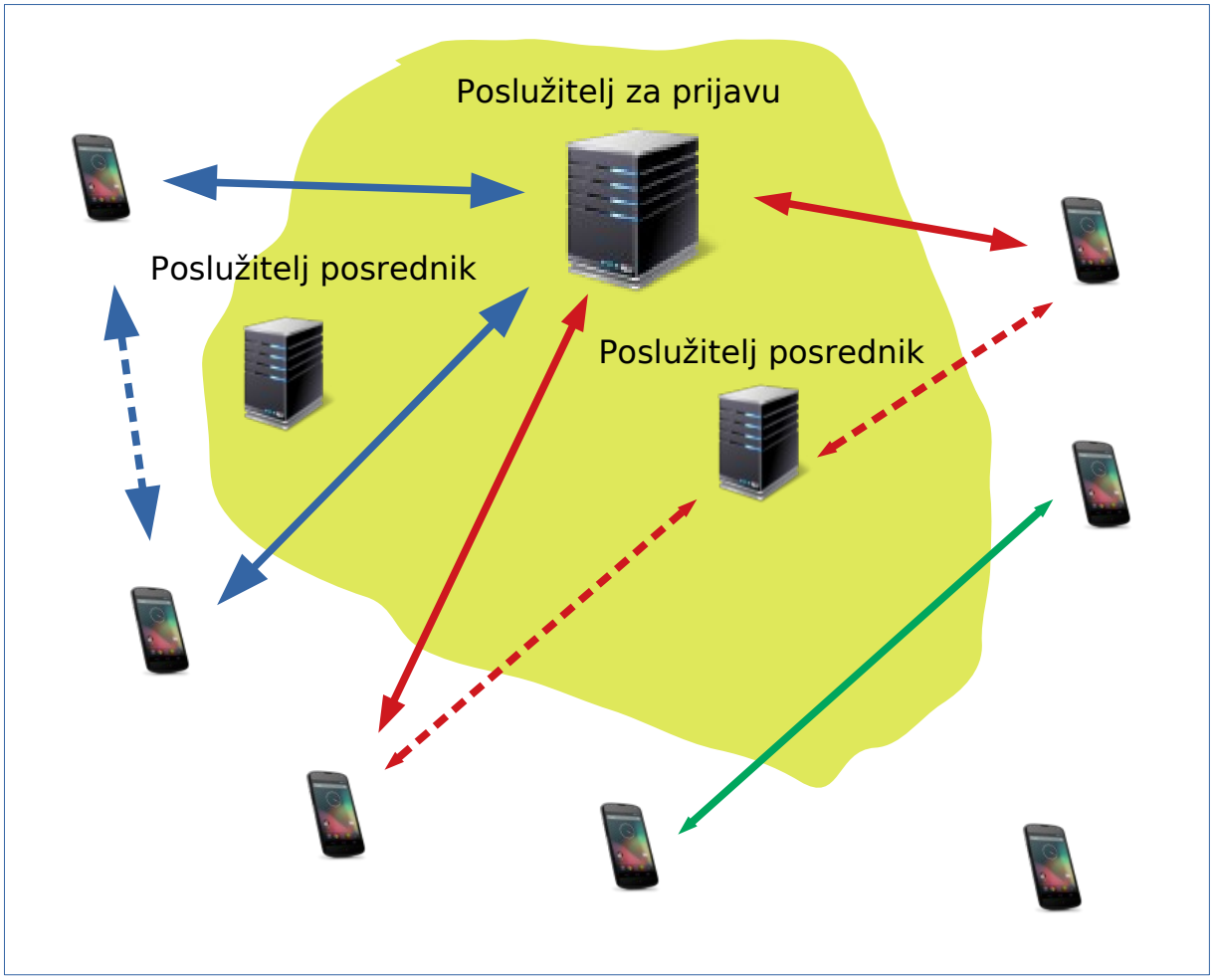
2 Arhitektura sustava i primjer komunikacije

Za uspostavu komunikacije najbitniji je poslužitelj za pronalaženje (prijavu) koji je i tema ovog rada. Izvor strujanja (multimedije) prijavljuje se na poslužitelj za pronalaženje sa jedinstvenim identifikatorom strujanja. Identifikator strujanja se određuje ili na osnovu koda kojeg korisnik unosi u aplikaciju ili na osnovu IP adrese i porta na kojem radi izvor strujanja. Poslužitelj ima javnu i fiksnu IP adresu (ili i ime) i broj porta i zato ga izvor i reproduktor mogu pronaći. Korisnik reproduktora, koji je u međuvremenu dobio jedinstveni identifikator strujanja od korisnika izvora, sa dijelom aplikacije reproduktor također se prijavljuje na poslužitelj za pronalaženje gdje im poslužitelj pomaže u razmijeni potrebnih informacija da se uspostavi strujanje multimedije. Cijeli sustav mora funkcionirati bez obzira na korištenje mehanizma za prevođenje mrežnih adresa, NAT uređaja, mora biti otporan na ispade ili nesuradnju pojedinih čvorova i izbjegavati nepotrebna posredovanja u komunikaciji [1]

Kao što se vidi na slici 3 postoji nekoliko slučajeva i različitih načina na koje se ostvaruje komunikacija i razmjena multimedije. Na slici 3 sa punim linijama su označene putanje poruka za uspostavu komunikacije, a isprekidanim linijama putanje multimedije i signalizacije povezane sa strujanjem multimedije. Punim plavim strelicama je prikazano kako se uspostavlja komunikacija u slučaju ako je onemogućeno direktno pronalaženje izvora i reproduktora. Kada putem Poslužitelja za prijavu izvor i reproduktor dobiju informacije koje su im potrebne razmjena multimedije se odvija direktno što je na slici prikazano isprekidanom plavom linijom.

Slučaj u kojem i nakon dobivanja potrebnih podataka nije moguća direktna komunikacija između izvora i reproduktora prikazan je crvenom bojom. U tom slučaju razmjena multimedije se odvija preko Poslužitelja posrednika.

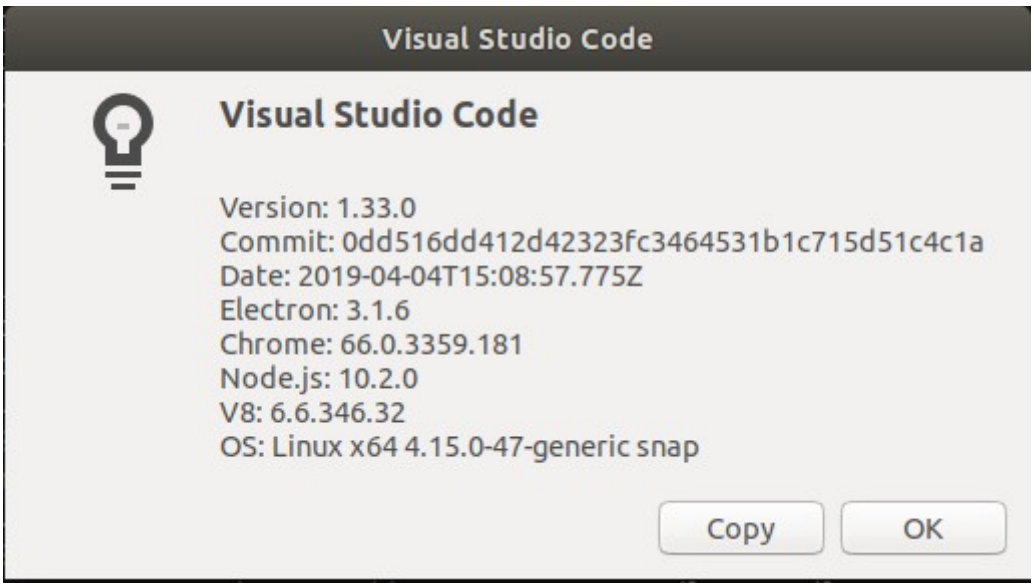
Zelenom bojom je prikazan slučaj u kojem iz koda za pronalaženje reproduktor dobiva IP adresu i broj porta izvora pa mu pristupa direktno i direktno razmjenjuju multimediju putem mreže. U ovom slučaju se u izvor multimedije ne unosi kod za pronalaženje nego ga aplikacija izvor sama generira na osnovu IP adrese i broja porta izvora i prikazuje korisniku na ekranu uređaja.



Slika 3: Arhitektura sustava [1, str.7]

3 Poslužitelj za prijavu (pronalaženje)

Program sam pisao u Visual Studio Code razvojnoj okolini i C++ programskom jeziku na operativnom sustavu Ubuntu 18.04.2 LTS. Na slici 4 se vidi verzija razvojne okoline Visual Studio Code.



Slika 4: Visual Studio Code

3.1 Faza inicijalizacije

U dokumentu sa detaljnim specifikacijama sustava [2] je određeno da prilikom pokretanja poslužitelj započinje raditi u fazi inicijalizacije. Iz konfiguracijske datoteke treba učitati IP adresu i broj porta na koje će poslužitelj povezati svoj UDP socket. Konfiguracijska datoteka je u tekstualnom obliku (ASCII) tako da se lako može pregledavati i uređivati kao što je traženo u dokumentu sa specifikacijama [2].

U konfiguracijskoj datoteci se nalaze IP adresa servera koja je za potrebe testiranja postavljena na „0.0.0.0”, broj porta koji je u ovom slučaju 12000 i popis poslužitelja posrednika. U popisu poslužitelja posrednika se nalaze IP adrese u IPv4 i IPv6 obliku jer su to trenutno važeći načini adresiranja kao i imena računala "3;gooffe.com:12000", "3;foi.com:12000". Poslužitelj za prijavu šalje ping poruku i kao potvrdu rada poslužitelja

posrednika treba dobiti odgovarajući odgovor. Sa gore navedenim imenima računala se testira da su dostupne informacije povezane sa domenskim nazivima (engl. *Domain Name System*, DNS). „goffe.com” je nepostojeće ime, a „foi.com:12000” se putem DNS sustava pronađe i pinga, ali ne pošalje odgovarajući odgovor jer foi.com:12000 nije poslužitelj posrednik.

Konfiguracijska datoteka, kod (konfiguracija.txt):

```
//IP adresa servera

IPadresa = "0.0.0.0"

//IPadresa = "::0%wlp16s0"

//IPadresa = "fe80::ce1a:2ee1:c8b4:cde1%wlp16s0"

brojPorta = "12000"

relayPosluzitelji = "2;[fe80::5c45:f89d:375c:bb8e%wlp16s0]:14000", "2;
        [fe80::c514:bef6:387e:114a%wlp16s0]:12000", "2;
        [fe80::172a:4e29:a78d:6af9%wlp16s0]:14000", "1;192.168.1.77:14000",
        "1;192.168.5.109:14000", "1;192.168.1.77:12000",
        "3;gooffe.com:12000", "3;foi.com:12000", "1;192.168.5.111:14000",
        "1;192.168.5.109:12000"
```

Na slici 5 se nalazi početak main() funkcije.

```
672  int main()
673  {
674      const int vrijemeCekanjaUSecReceiveFrom = 1;
675      const int vrijemeCekanjaUMiliSecReceiveFrom = 0;
676      Poco::Timespan timeSpanZaPrijem;
677      timeSpanZaPrijem.assign(vrijemeCekanjaUSecReceiveFrom, vrijemeCekanjaUMiliSecReceiveFrom);
678      u_char poljeZaPrijem[1032];
679
680      //1. FAZA INICIJALIZACIJE
681          //učitavanje parametara iz konfiguracijske datoteke u objekt citac
682
683      UcitavanjeKonfiguracije citac;
684      citac.IspisiSveParametre();
685          //priprema posluzitelja za komunikaciju
686      SocketAddress sa(citac.DajParametar(IPadresa), citac.DajParametar(port));
687      DatagramSocket ds(sa);
688      ds.setReceiveTimeout(timeSpanZaPrijem); //podešavanje koliko će dugo server čekati na prijemu
689      cout << " Ovo je moj posluzitelj: "<< sa.toString()
690           << "\n" << "----------------------------------------" << endl;
```

Slika 5: Početak main() funkcije

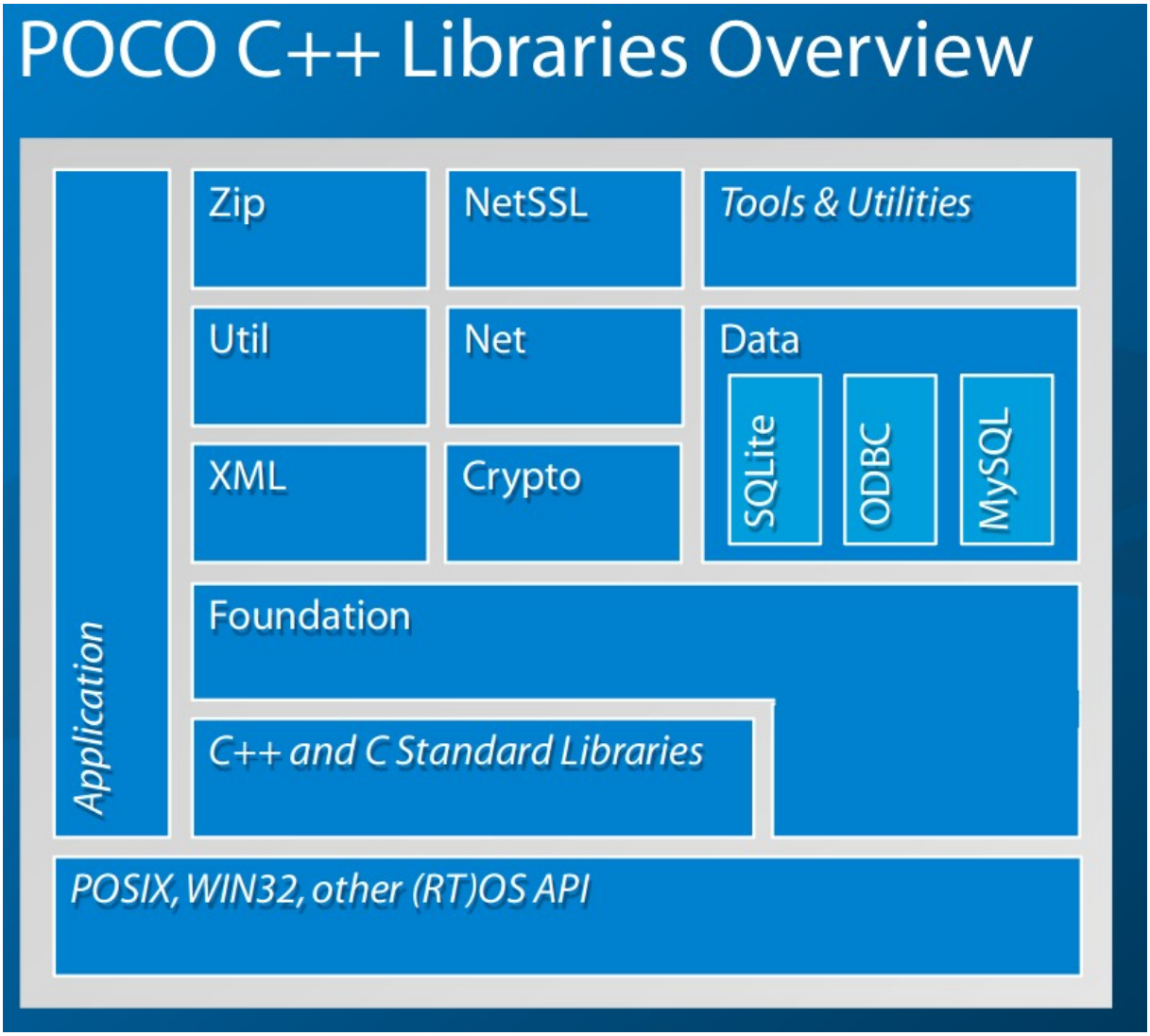
Na početku sam deklarirao nekoliko varijabli potrebnih za podešavanje objekta Pocco::Timespan timeSpanZaPrijem. Nakon toga pomoću objekta citac klase UcitavanjeKonfiguracije učitao sam podatke iz konfiguracijske datoteke u isti objekt i sa citac.IspisiSveParametre() ispisao ih.

Izvorni kod: početak main() funkcije

```
int main()
{
const int vrijemeCekanjaUSecReceiveFrom = 1;
const int vrijemeCekanjaUMiliSecReceiveFrom = 0;
Poco::Timespan timeSpanZaPrijem;
timeSpanZaPrijem.assign(vrijemeCekanjaUSecReceiveFrom,
vrijemeCekanjaUMiliSecReceiveFrom);
u_char poljeZaPrijem[1032];

//1. FAZA INICIJALIZACIJE
//učitavanje parametara iz konfiguracijske datoteke u objekt citac
UcitavanjeKonfiguracije citac;
citac.IspisiSveParametre();
//priprema posluzitelja za komunikaciju
SocketAddress sa(citac.DajParametar(IPadresa), citac.DajParametar(port));
DatagramSocket ds(sa);
ds.setReceiveTimeout(timeSpanZaPrijem); //podešavanje koliko će dugo server
                                        //čekati na prijemu
cout << " Ovo je moj posluzitelj: "<< sa.toString()
<< "\n" << "----------------------------------------" << endl;
```

Ovaj program je pisan sa naglaskom na socket programiranje i višedretvenu (višenitnu) arhitekturu. Za socket programiranje sam koristio POCO C++ Libraries [3], a na slici 6 je pregled iste biblioteke.



Slika 6: Poco C++ Libraries Overview [3]

https://pocoproject.org/slides/000-IntroAndOverview.pdf

Prvo sam inicijalizirao objekt „sa" klase SocketAddress i pomoću njega objekt „ds" klase DatagramSocket i nakon toga podesio vrijeme čekanja na prijem sa podacima iz objekta „timeSpanZaPrijem".

```
SocketAddress sa(citac.DajParametar(IPadresa), citac.DajParametar(port));
```

Klasa SocketAddress predstavlja jednu internet (IP) krajnju točku/socket adresu. Ta adresa može pripadati IPv4 ili IPv6 obitelji adresa i sastoji se od adrese računala (engl. *host*) i broja porta [3].

Funkcijski članovi su addr, af, family, host, init, length, operator !=, operator <, operator =, operator ==, port, resolveService, toString

```
DatagramSocket ds(sa);
```

Klasa DatagramSocket omogućava sučelje prema UDP socketu (utičnici).

```
ds.setReceiveTimeout(timeSpanZaPrijem);
```

Unutar klase DatagramSocket imamo više parametara sa kojima možemo podešavati rad UDP socketa. Sa setReceiveTimeout sam podesio vrijeme čekanja na prijem podataka.

Ostali funkcijski članovi su kao što je navedeno u [3]:

Member Functions:

bind, connect, getBroadcast, operator =, receiveBytes, receiveFrom, sendBytes, sendTo, setBroadcast

Inherited Functions:

address, available, close, getBlocking, getKeepAlive, getLinger, getNoDelay, getOOBInline, getOption, getReceiveBufferSize, getReceiveTimeout, getReuseAddress, getReusePort, getSendBufferSize, getSendTimeout, impl, init, operator !=, operator <, operator <=, operator =, operator ==, operator >, operator >=, peerAddress, poll, secure, select, setBlocking, setKeepAlive, setLinger, setNoDelay, setOOBInline, setOption, setReceiveBufferSize, setReceiveTimeout, setReuseAddress, setReusePort, setSendBufferSize, setSendTimeout, sockfd, supportsIPv4, supportsIPv6

Kao što se vidi u nastavku izvornog koda pomoću objekta provjeraRelayPosluzitelja klase ProvjeraRelayPosluzitelja sam izvršio provjeru dostupnosti Posluzitelja posrednika i u objektu se nakon toga nalazi lista aktivnih Poslužitelja posrednika koja će se u radnoj fazi po potrebi slati krajnjim korisnicima (izvor/reproduktor multimedije).

Izvorni kod nastavak:

```
cout << "Faza ucitavanja liste relay posluzitelja i testiranje dostupnosti"
<< endl << endl;

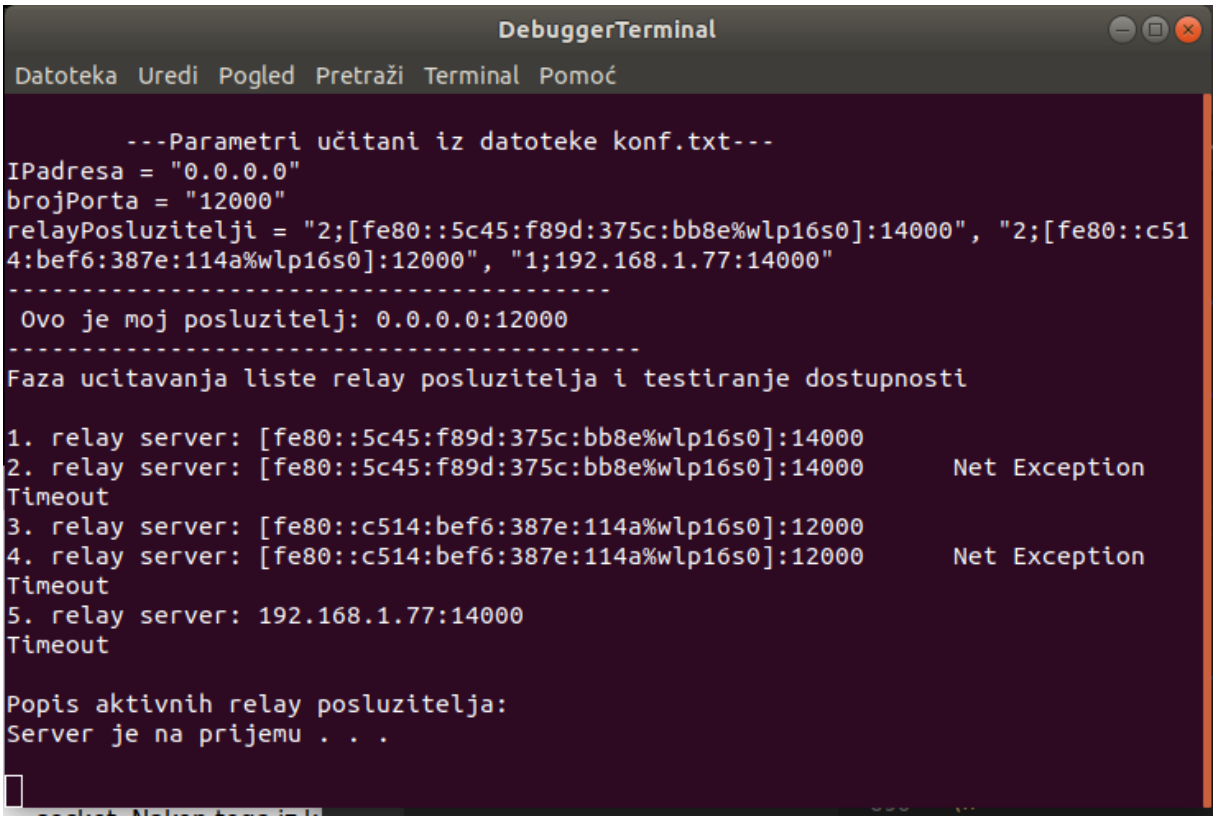
ProvjeraRelayPosluzitelja provjeraRelayPosluzitelja(citac.DajParametar(rel

ayServeri));

provjeraRelayPosluzitelja.Provjera(ds);

cout << "\nPopis aktivnih relay posluzitelja: " <<
provjeraRelayPosluzitelja.DajPopisAktivnihPosluzitelja() << endl;
```

Na slici 7 se vidi izgled ekrana nakon što je poslužitelj izvršio fazu inicijalizacije i ušao u radnu fazu.



Slika 7: Izgled ekrana nakon što je server izvršio inicijalizaciju i ušao u radnu fazu

3.1.1 Zašto UDP, a ne TCP?

Za razmjenu podataka (poruka) koristimo internet za koji su razvijeni razni protokoli, a svi ti protokoli pripadaju nekom od slojeva. Kao što se vidi na slici jedan imamo pet slojeva: Fizički, Veze, Mrežni, Transportni i Aplikacijski sloj. Promjenom pojedinog protokola na nekom sloju ne utječe se na ostale slojeve jer između slojeva postoje standardizirana sučelja koja svaki protokol mora poštivati. Raspored slojeva se može vidjeti u tablicama 1 i 2 zavisi da li govorimo o pet slojnom internet protokol modelu ili sedam slojnom ISO OSI referentnom modelu.

Table 1: Pet slojni internet protokol model

5. Aplikacijski
4. Transportni
3. Mrežni
2. Sloj veze
1. Fizički sloj

Na aplikacijskom sloju će raditi zadana aplikacija sa svojim protokolom za komunikaciju koji komunicira putem svojih 19 različitih poruka, a biti će opisan prilikom opisa pojedinih poruka.

Kao što je rečeno u [4] internet transportni sloj prenosi poruke pomoću IP protokola između krajnjih točaka zadane aplikacije omogućujući logičku komunikaciju između procesa aplikacije na različitim domaćinima (računalima, engl. *hosts*). Na transportnom sloju možemo birati između TCP protokola ili UDP protokola. Poruke na transportnom sloju nazivamo segmentima makar se u pojedinoj literaturi za UDP poruke koristi naziv datagram. UDP (engl. *User Datagram Protocol*) omogućuje jednu nepouzdanu, bezkonekcijsku uslugu za aplikaciju koja ga koristi. TCP (engl. *Transmission Control Protocol*) omogućuje pouzdanu, na vezu orijentiranu na uslugu. [4]

Table 2: Sedam slojni ISO OSI referentni model

Slojevi	Jedinica	Protokoli
7. Aplikacijski Mrežni procesi vezani za aplikaciju	Podatak Message	
6. Prezentacijski Enkriptiranje i kodiranje podataka	Podatak	
5. Sloj sesije Uspostavljanje sesije krajnjih korisnika	Podatak	
4. Transportni Veza, pouzdanost, transport	Segment, za UDP ponekad datagram	**TCP, UDP**
3. Mrežni Logičko adresiranje i prespajanje (engl. routing)	Paket Datagrams	IP i ostali protokoli
2. Sloj veze Fizičko adresiranje, pristup mediju	Okvir Frames	
1. Fizički sloj Prijenos signala	Bit Signali	

Na mrežnom sloju je uz ostale protokole i IP protokol. IP model usluge je usluga u najboljoj namjeri (engl. *best-effort delivery service*). Ne garantira isporuku paketa, poredak paketa, a ni integritet podataka i kao takva se smatra nepouzdanom uslugom. [4] Poruke na mrežnom sloju nazivamo datagrami kao što se i vidi u tablici 2.

IP protokol radi sa IP adresama i zadatak IP protokola je isporuka paketa između računala (krajnjih točaka).

3.1.1.1 Multipleksiranje i demultipleksiranje

Zadatak transportnog sloja, a time i TCP i UDP protokola je isporuka segmenata između procesa (aplikacija) povezanih računala. Svaki proces ima jedan ili više socketa koji imaju svoj jedinstveni broj i jedinstveni broj porta da bi se znalo koju poruku treba proslijediti kojem procesu. Zato nam uz IP adresu treba i broj porta. IP adresa jedinstveno određuje računalo, a port se veže uz socket i proces. Preuzimanje podataka od aplikacije, ubaciivanje podataka u UDP segment i u UDP segment upisivanje odredišnog i izvorišnog porta i predavanje segmenta mrežnom sloju zove se multipleksiranje. Demultipleksiranje, (engl. *transport-layer multiplexing i demultiplexing*) je preuzimanje segmenta iz datagrama i određivanje na osnovu broja odredišnog porta kojem socketu treba predati podatke iz segmenta.

3.1.1.2 UDP segment

UDP segment se sastoji od 4 polja i svaki je velik po dva bajta što će reći da je veličina zaglavlja UDP segmenta osam bajtova. To je osjetno manje od zaglavlja TCP segmenta koji je 20 bajtova i to je jedan od razloga korištenja UDP-a, a ne TCP-a. Četiri polja su izvorišni i odredišni port, dužina segmenta i kontrolna suma (engl. *checksum*). Polje dužine segmenta je potrebno zato što nisu svi segmenti jednake dužine. Kontrolna suma služi za provjeru integriteta segmenta. Ako u segmentu ima grešaka može se automatski odbaciti ili se može aplikaciji poslati upozorenje. U tablici 3 vidimo strukturu UDP segmenta.

Table 3: Struktura UDP segmenta

----------16 bitova----------

1. Izvorišni port
2. Odredišni port
3. Dužina segmenta
4. Kontrolna suma

I UDP i TCP izvršavaju provjeru integriteta segmenata. Što se tiče UDP-a to su jedine dvije usluge koje izvodi UDP. Provjera integriteta segmenata i isporuka segmenata između pojedinih procesa različitih računala.

TCP omogućava i pouzdan prijenos podataka. Točan poredak paketa, a pakete koji nedostaju, ili su oštećeni ponovo šalje. Omogućava i kontrolu zagušenja spojnog puta i zbog toga je TCP segment osjetno veći i na prvi pogled bi bilo bolje koristiti TCP zbog sigurnosti koju pruža, ali prilikom prijenosa multimedije u ovoj aplikaciji jako je bitan osjećaj realnog vremena i u tome je UDP puno bolji. U aplikacijski protokol se ugradi kontrola kašnjenja i odbacivanje paketa koji su prekasno došli. Nema uspostave veze u UDP-u. Segmenti se odmah počinju slati pa je i kašnjenje manje. Poslužitelji koji rade sa UDP-om su puno manje opterećeni jer ne moraju održavati sve moguće pojedinačne veze, a i samo zagušenje spojnog puta je manje zbog manjeg zaglavlja UDP segmenta.

3.2 Faza radnog režima

U fazi radnog režima poslužitelj prima poruke, obrađuje ih i šalje odgovore. Kao što je rečeno u dokumentu sa specifikacijama [2] Poslužitelj za prijavu mora biti organiziran tako da može istovremeno opsluživati veći broj zahtijeva i slati odgovore na poruke. Ideja je da se prijem poruka odvija u jednoj dretvi (niti, engl. *thread*), a obrada u dvije ili više njih jer je obrada složeniji postupak i da duže obrade ne koče obrade koje se mogu brzo izvesti. Prilikom stvaranja dretvi za obradu zahtijeva postoji mogućnost da se za svaki zahtjev stvara nova dretva, a nakon obrađenog zahtjeva uništava, ili da se stvori jedan određeni broj niti na početku faze radnog režima i da se dretve ne uništavaju nakon obrađenog zahtjeva nego da se stvori jedan red čekanja zahtjeva na obradu iz kojeg svaka dretva uzima zahtjev koji je na redu za obradu, obradi ga, pošalje odgovor i iz reda čekanja uzima slijedeći. Za početak sam napravio dvije dretve za obradu zahtjeva radi testiranja ispravnosti rada, ali bez problema se može u program ubaciti veći broj, a koji broj bi bio optimalan zavisi i od prometa koji se očekuje i od građe računala (poslužitelja) na kojem će se program izvoditi. Najviše ovisi o broju procesorskih jezgri koje posjeduje računalo.

Za kreiranje dvije dodatne dretve koristio sam službenu podršku koja je uvedena sa standardom C++ 11:

```
include<thread>

thread t1{Trosilo};
```

Kao što se vidi u sljedećem nastavku izvornog koda, na početku faze radnog režima sam objekt ds klase DatagramSocket podesio da je konstantno na prijemu. Nakon toga sam kreirao dvije dretve u kojima će se obrađivati poruke. Dretve se kreiraju kreiranjem objekta klase thread, a kao prvi parametar se prosljeđuje ime funkcije u kojoj će se izvršavati svaka pojedinačna dretva, a ostali parametri po redu su argumenti pozvane funkcije. Struktura PrijemnaPoruka služi za pohranu svih oblika prijemnih poruka i podataka koji su kasnije potrebni za obradu ulaznih (prijemnih) poruka.

Nastavak izvornog koda:

```
//2. FAZA RADNOG REŽIMA
timeSpanZaPrijem.assign(0, 0);
ds.setReceiveTimeout(timeSpanZaPrijem);
cout << "Server je na prijemu . . ." << endl << endl;


thread t1{Trosilo, 1};
thread t2{Trosilo, 2};


PrijemnaPoruka prijemnaPoruka;
PrijemnaPoruka* pokPrijemnaPoruka;
PrijemnaPoruka2 prijemnaPoruka2;
int i = 0;
int brojPrimljenihBajtova{0};
```

U beskonačnoj petlji objekt ds sa svojim funkcijskim članom receiveFrom prima poruke u obliku niza bajtova i zapisuje ih u varijablu poljeZaPrijem koja je deklarirana kao polje char poljeZaPrijem[1024] koje mora biti veliko minimalno koliko je u bajtovima velika najveća poruka koju će primiti server, ali može biti i veće.

Nastavak izvornog koda:

```
while(true){
try
        {
 brojPrimljenihBajtova = ds.receiveFrom(poljeZaPrijem,sizeof(poljeZaPrijem),
                prijemnaPoruka2.adresaIzDatagramSocketaSaKojeJeDoslaPoruka);
        }
catch(const std::exception& e)
        {
        std::cerr << e.what() << '\n';
        t1.join();
```

```
        t2.join();
        }
```

Struktura PrijemnaPoruka je namijenjena da se u nju zapisuju podaci iz ulaznih poruka i da se objekti u kojima su ti podaci zapisani prosljeđuju pojedinim dretvama (nitima) programa na daljnju obradu. U prvom redu sljedećeg koda pokazivač pokPrijemnaPoruka preusmjeravam na početak polja u kojem su zapisani primljeni bajtovi, a u drugom redu objekt prijemnaPoruka preusmjeravam na adresu na koju pokazuje pokPrijemnaPoruka.

Nastavak izvornog koda:

```
pokPrijemnaPoruka = (PrijemnaPoruka*)&poljeZaPrijem[0];
prijemnaPoruka = *pokPrijemnaPoruka;
```

Iz objekta prijemnaPoruka prepišem podatke u objekt prijemnaPoruka2 jer u strukturi PrijemnaPoruka2 imam već zapisanu adresu pošiljatelja pa samo dopišem prijemnu poruku. Mogao sam to riješiti i na drugačiji način, ali u trenutku pisanja koda ovako mi je izgledalo logično.

U kodu ispod vidi se struktura strukture PrijemnaPoruka2.

```
struct PrijemnaPoruka2{
uint8_t tipPoruke;
uint64_t identifikatorStrujanja;
IdentifikacijaSocketa javnaAdresa;
IdentifikacijaSocketa lokalnaAdresa;
SocketAddress adresaIzDatagramSocketaSaKojeJeDoslaPoruka;
};
```

Na kraju beskonačne petlje pozivam funkcijski član Dodaj objekta cirkularniBafer klase KruzniSpremnik i prosljeđujem mu objekt prijemnaPoruka2 čime je prijem pojedinačne poruke završio i u beskonačnoj petlji se prima slijedeća poruka i šalje na obradu.

Nastavak izvornog koda:

```
prijemnaPoruka2.tipPoruke = prijemnaPoruka.tipPoruke;
prijemnaPoruka2.identifikatorStrujanja =
prijemnaPoruka.identifikatorStrujanja;
prijemnaPoruka2.javnaAdresa.tipArdese =
prijemnaPoruka.javnaAdresa.tipArdese;
prijemnaPoruka2.javnaAdresa.IPAdresa = prijemnaPoruka.javnaAdresa.IPAdresa;
prijemnaPoruka2.javnaAdresa.port = prijemnaPoruka.javnaAdresa.port;
prijemnaPoruka2.lokalnaAdresa.tipArdese =
prijemnaPoruka.lokalnaAdresa.tipArdese;
prijemnaPoruka2.lokalnaAdresa.IPAdresa =
prijemnaPoruka.lokalnaAdresa.IPAdresa;
prijemnaPoruka2.lokalnaAdresa.port = prijemnaPoruka.lokalnaAdresa.port;
cirkularniBafer.Dodaj(prijemnaPoruka2);
```

U funkciji Trošilo kreiram objekt porukaMajstor klase PorukaMajstor u kojoj je sva logika što se tiče obrade poruka. Za obradu služi funkcijski član obradaPoruke kojem se kao argument prosljeđuje element (prijemnaPoruka2) kružnog spremnika cirkularniBafer.

```
void Trosilo(){                         //dretva koja prazni spremnik
PorukaMajstor porukaMajstor;
        while(true)
                porukaMajstor.obradaPoruke(cirkularniBafer.Sljedeci());
}
```

CirkularniBafer je globalni spremnik poruka kojem mogu pristupiti sve niti programa.

```
KruzniSpremnik<PrijemnaPoruka2, 100> cirkularniBafer; //globalni spremnik
                                                      //ulaznih poruka
```

Klasa KruzniSpremnik je posebno zanimljiva za prokomentirati jer u njoj je riješena problematika istovremenog pristupanja različitih dretvi programa istim podacima.

Glavni dio klase KruzniSprmnik je polje koje se sastoji od elemenata određenog tipa i kapaciteta. Zato sam klasu napisao kao predložak da ju mogu koristiti za različite elemente i veličine polja [5]. U polje se elementi stavljaju pomoću funkcijskog člana void Dodaj(const Elem& elem), a iz polja se vade pomoću Elem Sljedeci() [5]. Korištenje tih funkcijskih članova je već prije spomenuto prilikom dodavanja i uzimanja ulaznih poruka.

```
template<typename Elem, size_t kapacitet>
class KruzniSpremnik
{
private:
        std::array<Elem, kapacitet> niz;
        size_t pocetak;               //index početka cirkularnog spremnika
        size_t broj;                  //trenutni broj elemenata
        std::mutex brava;
        std::condition_variable daLiJePun;
        std::condition_variable daLiJePrazan;
public:
        KruzniSpremnik() : pocetak{0}, broj{0} { }
        void Dodaj(const Elem& elem){
                std::unique_lock<decltype(brava)> zasun(brava);
                daLiJePun.wait(zasun,[this](){ return broj != kapacitet;});
                niz[DajKraj()] = elem;
```

```
                ++broj;
                daLiJePrazan.notify_one();
                }
        Elem Sljedeci(){
                std::unique_lock<decltype(brava)> zasun(brava);
                daLiJePrazan.wait(zasun, [this](){ return broj != 0; });
                Elem rez = niz[pocetak];
                pocetak = ++pocetak % kapacitet;
                --broj;
                daLiJePun.notify_one();
                return rez;
                }
+       uint64_t ObrisiNajstarijiAkoJeProsloViseOd(int sekunde){
        }
+       bool NadjiIIdentifikatorStrujanjaStaviNaNulu(uint64_t identStruj){
        }
+       void IspisCirkularnogSpremnika(){
        }
private:
                //pomoćna funkcija za izračunavanje indeksa kraja niza
        size_t DajKraj(){

                return (pocetak + broj) % kapacitet;
        }
};
```

3.2.1 Sinkronizacija dretvi s pomoću mutexa

Više različitih dretvi u isto vrijeme će pokušavati pristupiti istom polju i zato sam morao primijeniti sinkronizaciju dretvi.

Suština sinkronizacije dretvi jest spriječiti da više dretvi istovremeno izvodi problematični dio koda koji se naziva kritični odsječak (engl. *Critical section*). To se postiže pomoću globalnog objekta kojim se kontrolira pristup kritičnom odsječku iz različitih dretvi i brave (engl. *lock*) koja se postavlja na početak kritičnog odsječka. Prva dretva koja naiđe na bravu na početku kritičnog odsječka će „zaključati" globalni objekt, tj. preuzet će vlasništvo nad njim. Time će spriječiti ostalim dretvama da uđu u taj dio koda. Naiđe li neka druga dretva na dio koda koji je zaključan, njeno izvođenje će se zaustaviti sve dok prva dretva ne završi s izvođenjem kritičnog odsječka, otključa globalni objekt i time dozvoli pristup drugim dretvama. [5]

```
#include<mutex>
mutex brava;
...
brava.lock();
... //kritični odsječak
brava.unlock();
...
```

Zaključani objekt se uvijek na kraju kritičnog odsječka mora otključati!

U slučaju ako u kritičnom odsječku bude bačena iznimka ili ako zaboravimo otključati zaključani odsječak program će zablokirati, brava će ostati zaključana zauvijek. Standardna biblioteka predložaka definira predložak klase lock_guard i unutar nje zapakiramo klasu mutex, a prilikom izvođenja desturuktora klase lock_guard poziva se funkcijski član unlock() objekta koji joj je proslijeđen kao argument konstruktoru.

```
#include<mutex>
mutex brava;
...
{
        lock_guard<mutex> kalauz(brava);
        ... //kritični odsječak
}
...
```

Ako deklaraciju pišemo:

```
{
        lock_guard<decltype(brava)> kalauz(brava);
```

kod će biti još manje podložan greškama jer kada umjesto klase mutex upotrijebimo neku drugu klasu za sinkronizaciju nećemo ništa u kodu trebati mijenjati.[5]

Postoji još nekoliko vrsta mutexa:

```
        std::recursive_mutex brava;
```

Ova vrsta mutexa dopušta onoj dretvi koja ga je zaključala da ga može ponovo zaključavati. Koliko puta ga je zaključala toliko puta ga mora i otključati da bi ga neka druga dretva vidjela kao otključanog. Često se koristi kod rekurzivnih poziva.

```
        timed_mutex i recursive_timed_mutex
```

Ove dvije klase služe kada želimo da dretva ne čeka beskonačno kad će se mutex otključati nego da možemo postaviti nekakvo vremensko ograničenje. Uz funkcijske članove lock() i unlock() postoje još:

- try_lock() - dretva pokušava zaključati objekt. Ako je objekt zaključala neka druga dretva onda vraća false, inače ga zaključava i vraća true

- try_lock_for() - ako timed_mutex nije trenutno zaključan (ili recursive_timed_mutex), pozivajuća dretva ga zaključava i ostaje zaključan do poziva unlock() funkcijskog člana. Ako ga je zaključala neka druga dretva, pozivajuća dretva će čekati vrijeme određeno kao argument ovog funkcijskog člana i ako se u međuvremenu timed_mutex otključa pozivajuća dretva će ga zaključati i vratiti true, inače vraća false
- try_lock_until() - ako objekt nije zaključan funkcijski član vraća true i zaključava ga do poziva unlock(). Ako je objekt zaključala neka druga dretva pozivajuća dretva će čekati do određenog vremenskog trenutka koji je proslijeđen kao argument i ako se u međuvremenu objekt otključao zaključava ga i vraća true, u suprotnom vraća false

Za automatsko otključavanje može se koristiti ovojnica unique_lock koja je slična klasi lock_gurad, ali klasa unique_lock ima i nekoliko funkcijskih članova. Definira i eksplicitni operator bool kojim se može provjeriti da li je objekt u vlasništvu brave.[5]

```
std::recursive_timed_mutex bravaPokusaja;
...
{
        std::unique_lock<declType(bravaPokusaja)>
                                vremenskiZasun{bravaPokusaja,try_to_lock};
        if(vremenskiZasun){...}
        else{...}
}
...
```

3.2.2 Sinkronizacija dretvi s pomoću uvjetnih varijabli

U klasi KruzniSpremnik sam koristio i uvjetne varijable (engl. *condition variables*). Uvjetne varijable su varijable koje omogućavaju zaustavljanje dretvi dok čeka na neki događaj u nekoj drugoj dretvi ili na neki brojač vremena (engl. *timer*).[5]

```
        std::condition_variable daLiJePun;
        std::condition_variable daLiJePrazan;
```

3.2.3 Klasa KruzniSpremnik

U programu koristim dva globalna objekta klase KruzniSpremnik kojima mogu pristupati različite dretve programa. Jedan je cirkularniBafer i u njemu se nalaze objekti klase PrijemnaPoruka2, a u cirkularniSpremnikVremenaUpisa se nalaze objekti u kojima su podaci o tome kad je koje računalo registriralo svoj identifikacijski broj.

```
KruzniSpremnik<PrijemnaPoruka2, 100> cirkularniBafer;
KruzniSpremnik<VrijemeUpisa, 10000> cirkularniSpremnikVremenaUpisa;
```

Na primjeru objekta cirkularniBafer ću objasniti rad klase KruzniSpremnik.

Linijom koda:

```
cirkularniBafer.Dodaj(prijemnaPoruka2);
```

koja se nalazi u na kraju beskonačne petlje glavne dretve (prijem poruka) prijemnaPoruka2 se pozivom funkcijskog člana Dodaj upisuje u polje objekta cirkularniBafer.

Na početku funkcijskog člana Dodaj se nalazi objekt tipa unique_lock koji se u slijedećem redu prosljeđuje uvjetnoj varijabli daLiJePun, tj. njezinom funkcijskom članu wait kao prvi argument, a drugi argument je lambda izraz pomoću kojeg provjeravamo da li je polje std::array<Elem, kapacitet> niz popunjeno do kraja. Ako polje nije popunjeno mutex će ostati zaključan i dretva će nastaviti sa izvođenjem, upisom elementa u polje, inkrementiranjem varijable broj++ i izlaskom iz funkcijskog člana Dodaj() se otključava mutex brava.

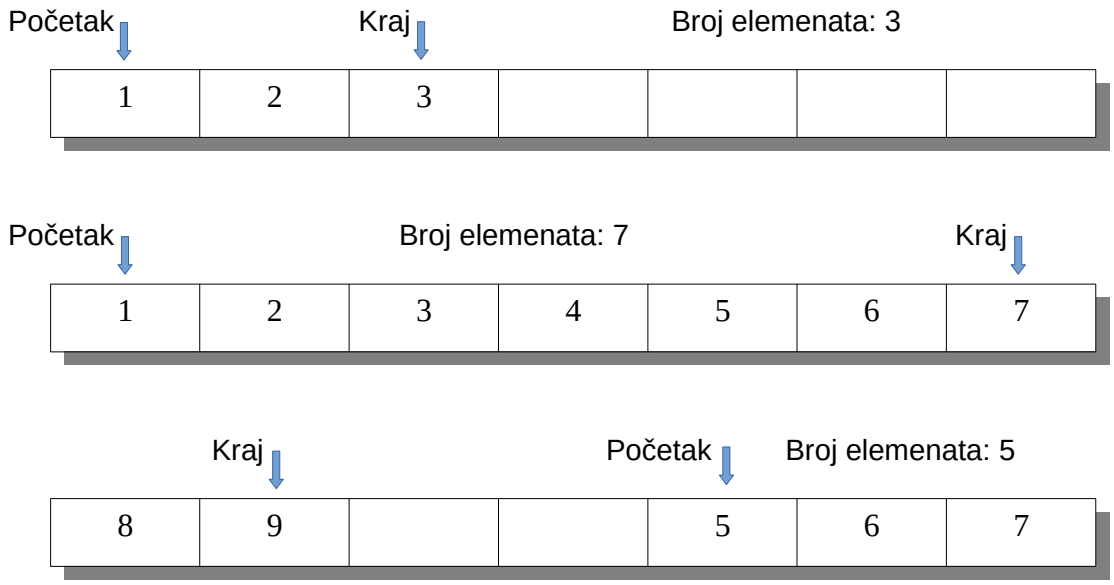
Ako je niz popunjen, a za to nam i služi uvjetna varijabla daLiJePun, otključati će se mutex brava i zaustaviti dretvu na uvjetnoj varijabli daLiJePun (lambda izraz je vratio false). Druga dretva koja vadi elemente iz polja pozivom funkcijskog člana Sljedeci() će vrijednost elementa na kojeg pokazuje varijabla pocetak prepisati u lokalnu varijablu rez, promijenit će vrijednost varijable pocetak izrazom pocetak = ++pocetak % kapacitet, smanjiti broj

elemenata dekrementiranjem varijable broj i izrazom daLiJePun.notify_one(), dojaviti promjenu stanja.

Nakon dojave o promjeni stanja od druge dretve prva dretva koja je zaustavljena u funkcijskom članu Dodaj() na uvjetnoj varijabli daLiJePun, ponovo će sa funkcijskim članom wait() provjeriti predikat i ako on vraća true dretva nastavlja sa izvođenjem i ponovno se zaključava mutex brava u Dodaj() te dodaje element u niz na poziciju koja je kraj niza, a koju mu vraća funkcijski član DajKraj() i inkrementira varijablu broj.

Na isti način funkcionira i uvjetna varijabla daLiJePrazan u funkcijskom članu Sljedeci(). Provjerava da li u nizu ima elemenata za obradu i ako ih nema zaustavlja dretvu i čeka na dojavu o promjeni stanja iz funkcijskog člana Dodaj(), u kojem na kraju je daLiJePrazan.notify_one(). Možda bi bolje ime za uvjetnu varijablu bilo daLiImaElemenata.

Na slici 8 je prikazan raspored elemenata u cirkularnom spremniku koji je izveden pomoću polja jednako kao u klasi KruzniSpremnik. U polje na slici stane sedam elemenata. Na početku se elementi stavljaju od prve do zadnje pozicije. Kada se cijelo polje popuni dretva koja stavlja elemente mora čekati da se barem jedno mjesto isprazni.



Slika 8: Cirkularni spremnik

Elementi koji su prvi upisani u spremnik se i prvi vade van. Prvi unutra prvi van. Varijabla Početak uvijek pokazuje na element koji je na redu za vađenje, a kraj dobivamo

pomoću funkcijskog člana DajKraj(). U varijablu Broj upisujemo trenutni broj elemenata. Na takav način pomoću punjenja i pražnjenja možemo upisati beskonačan broj elemenata. Ako se elementi prebrzo upisuju, ili prebrzo prazne balans možemo postići sa optimalnim brojem dretvi.

3.2.4 Poruke aplikacijskog sloja

Klasa PorukaMajstor sa svojom metodom obradaPoruke() obrađuje prijemne poruke koje kao argument dobiva iz cirkularnog bafera i šalje potrebne odgovore.

U tablici 4 se nalaze sve poruke aplikacijskog sloja. Svaka poruka započinje sa oznakom tipa poruke koja je osam bitna vrijednost, a poruka ima ukupno 19.

Table 4: Poruke aplikacijskog protokola (sloja)

Naziv poruke	Vrijednost(tip)
MSG_PING	1
MSG_PONG	2
MSG_PONG_REG_REQ	3
MSG_STREAM_ADVERTISEMENT	4
MSG_STREAM_REGISTERED	5
MSG_IDENTIFIER_NOT_USABLE	6
MSG_FIND_STREAM_SOURCE	7
MSG_STREAM_SOURCE_DATA	8
MSG_STREAM_REMOVE	9
MSG_MULTIMEDIA	10
MSG_REQUEST_STREAMING	11
MSG_FORWARD_PLAYER_READY	12
MSG_PLAYER_READY	13
MSG_SOURCE_READY	14
MSG_REQ_RELAY_LIST	15
MSG_RELAY_LIST	16
MSG_SHUTTING_DOWN	17
MSG_PLEASE_FORWARD	18
MSG_REGISTER_FORWARDING	19

(Izvor: Ivković, Magdalenić, Milić [1])

Identifikator strujanja se javlja kod većine poruka i on je 64-bitni broj. Mora biti jedinstven i generira ga izvor multimedije.

Kod većine poruka se koriste tri uzastopna polja za identifikaciju socketa u kojima je na prvom mjestu tip adrese kao što se može vidjeti na slici 9. U tablici 5 se vide mogući tipovi. Drugo polje označava adresu i promjenjive je duljine, a na trećem mjestu je broj porta. Ako je adresa IPv4 onda je duljina adrese 32-bitni broj.

Table 5: Vrste adresa, vrijednosti u polju za tip adrese i duljina polja adrese

Vrsta adrese	Vrijednost	Duljina adrese
ADR_IPv4	1	32-bitni broj
ADR_IPv6	2	128-bitni broj
ADR_HOSTNAME	3	promjenjiva

Tip adrese (osam bitni broj)	Lokalna IP adresa (32-bitni broj za IPv4)	Lokalni broj porta (16 – bitni broj)

Slika 9: Tri polja za identifikaciju socketa

U fazi inicijalizacije poslužitelj šalje poruku MSG_PING i dobiva (ili ne dobiva) odgovor porukom MSG_PONG ili MSG_PONG_REG_REQ.

Poslužitelj u radnoj fazi mora moći obraditi pet vrsta poruka i poslati odgovarajuće odgovore.

Prva poruka koju poslužitelj dobiva je Stream_advertisement. Na slici 10 to je prva poruka. Lokalna IP adresa i lokalni broj porta u poruci Stream_advertisement pripadaju trenutnom pošiljatelju, tj. izvoru multimedijskog strujanja. Kada primi poruku Stream_advertisement posluižitelj pročita identifikator strujanja i provjeri da li ga već ima zapisanog u bazi.:

- Ako nema zapisuje ga i pošiljatelju šalje poruku Stream_registered. Na slici 11 to je prva poruka. Zapis u bazi treba ostati zapisan neko minimalno vrijeme nakon čega se

automatski briše, a treba negdje zapisati i vrijeme upisa u bazu radi automatskog brisanja. Javna IP adresa i javni broj porta su adrese trenutnog pošiljatelja, a TTL u sekundama je vrijeme nakon kojeg će zapis u bazi biti automatski obrisan

- Ako postoji zapis u bazi provjerava se da li je zapis registrirao trenutni pošiljatelj i ako je šalje mu se ista poruka Stream_registered i ažurira se vrijeme upisa u bazu
- a ako postoji zapis u bazi koji je registrirao drugi pošiljatelj šalje se poruka Identifier_not_usable, a zapis u bazi se ne dira niti se vrijeme ažurira

MSG_STREAM_ADVERTISEMENT (poruka br. 4)	Identifikator strujanja	Tip adrese	Lokalna IP adresa	Lokalni broj porta

MSG_STREAM_REMOVE (poruka br. 9)	Identifikator strujanja

MSG_FIND_STREAM_SOURCE (poruka br. 7)	Identifikator strujanja

MSG_FORWARD_PLAYER_READY (poruka br. 12)	Identifikator strujanja	Tip adrese	Javna IP adresa	Javni broj porta	Tip adrese	Lokalna IP adresa	Lokalni broj porta

MSG_REQ_RELAY_LIST (poruka br. 15)

Slika 10: Pet vrsta poruka aplikacijskog protokola koje server mora moći obraditi

Ako poslužitelj zaprimi poruku Stream_remove (druga poruka na slici 10) provjerava da li je stream registrirao trenutni pošiljatelj i ako je briše zapis u bazi bez slanja bilo kakvog odgovora. Ako nije ignorira poruku.

MSG_STREAM_REGISTRED (poruka br. 5)	Identifikator strujanja	TTL u sekund. (u_int16_t)	Tip adrese	Javna IP adresa	Javni broj porta

MSG_IDENTIFIER_NOT_USABLE (poruka br. 6)	Identifikator strujanja	Tip adrese	Javna IP adresa	Javni broj porta

Slika 11: Dvije odlazne poruke kao odgovori na Stream_advertisement

Ako poslužitelj zaprimi poruku Find_stream_source (treća poruka na slici 10) to znači da reproduktor multimedije traži podatke o izvoru multimedije. Poslužitelj na osnovu identifikatora traži zapis u bazi i ako zapis postoji kreira poruku tipa Stream_source_data (slika 12), a ako ne postoji ignorira poruku.

MSG_STREAM_SOURCE_DATA (poruka br. 8)			Identifikator strujanja	Tip adrese	Javna IP adresa repr	Javni broj porta reprod.
Tip adrese	Javna IP adresa izvora	Javni broj porta izvora	Tip adrese	Lokalna IP adresa izvora	Lokalni broj porta izvora	

Slika 12: Struktura odlazne poruke Stream_source_data

Poruku Forward_player_ready (četvrta poruka na slici 10) poslužitelj može primiti samo od reproduktora multimedije, kreira poruku odgovor Player_ready (slika 13) i šalje ju izvoru multimedijskog strujanja. Adrese i brojevi porta i u ulaznoj poruci i u odlaznoj odnose se na reproduktor koji je poslao ulaznu poruku.

MSG_PLAYER_READY (poruka br. 13)	Identifikator strujanja	Tip adrese	Javna IP adresa	Javni broj porta	Tip adrese	Lokalna IP adresa	Lokalni broj porta

Slika 13: Struktura odlazne poruke Player_ready

Na poruku Req_relay_list (peta poruka na slici 10) poslužitelj odgovara sa porukom Relay_list (slika 14) u kojoj se nalazi lista poslužitelja posrednika. To je lista koja je potrebna izvoru i reproduktoru multimedije da mogu pronaći najbližeg poslužitelja posrednika jer ne mogu neposredno razmjenjivati pakete multimedije.

MSG_RELAY_LIST (poruka br. 16)	Broj zapisa (4 bajta)	Tip adrese 1. poslužit.	Javna IP adresa 1. poslužit.	Javni broj porta 1. poslužitelja	. . .

. . .	Tip adrese zadnje posluž.	Adresa ili naziv zadnjeg posluž.	Broj porta

Slika 14: Struktura odlazne poruke Relay_list

Ostale poruke aplikacijskog protokola poslužitelj za pronalaženje ne koristi pa ih neću ni obrađivati. To su poruke koje se koriste u direktnoj komunikaciji izvora i reproduktora ili, u njihovoj komunikaciji sa poslužiteljem posrednikom.

3.2.5 Primjer obrade poruke u klasi PorukaMajstor

Klasa PorukaMajstor se sastoji u public dijelu od konstruktora i destruktora i dva funkcijska člana, string Ping() i void obradaPoruke(PrijemnaPoruka2 poruka). Funkcijski član Ping() služi da za vrijeme inicijalizacije poslužitelja šalje ping poruke poslužiteljima posrednicima i na osnovu njihovih odgovora se slaže lista aktivnih poslužitelja posrednika. Funkcijski član obradaPoruke kao argument dobiva adresu na kojoj se nalazi objekt strukture

PrijemnaPoruka2 što su zapravo podaci dobiveni iz ulazne poruke koji se u obradaPoruke obrađuju.

```
class PorukaMajstor
{
private:
        string pingPoruka;
        PrijemnaPoruka2 porukaZaObradu;
+       struct StreamRegistredStruktura {. . .
        } streamRegistred;
+       struct IdentifierNotUsableStruktura {. . .
        } identifierNotUsable;
+       struct StreamSourceData {. . .
        } streamSourceData;
+       struct PlayerReady {. . .
        } playerReady;
+       struct RelayList {. . .
        } relayList;
+       StreamRegistredStruktura* PorukaStreamRegistred(){. . .}
+       IdentifierNotUsableStruktura* PorukaIdentifierNotUsable(){. . .}
+       StreamSourceData* PorukaStreamSourceData(){. . .}
+       PlayerReady* PorukaPlayerReady(){. . .}
+       RelayList* PorukaRelayList(){. . .}
public:
        PorukaMajstor() { }
        ~PorukaMajstor() { }
+       string Ping(){. . .}
+       void obradaPoruke(PrijemnaPoruka2 poruka){. . .}
};
```

U private dijelu klase PorukaMajstor se nalaze strukture potrebne za rad funkcijskih članova i jedan string i pet funkcijskih članova potrebnih za obradu podataka u strukturama koje za slanje poruka koristi javni funkcijski član obradaPoruke().

U slijedećem kodu je primjer strukture korištene u klasi PorukaMajstor. Podaci moraju biti složeni kako je propisano aplikacijskim protokolom za svaku pojedinu poruku koja se šalje. Ova struktura služi za kreiranje poruke Msg_stream_registred, a značenje pojedinih polja je opisano prilikom opisa poruka protokola.

```
struct StreamRegistredStruktura {
        u_int8_t tipPoruke;
        u_int64_t identifikatorStrujanja;
        u_int16_t TTL_u_sekundama;
        uint8_t tipJavneAdrese;
        uint32_t javnaIPAdresa;
        uint16_t javniBrojPorta;
} streamRegistred;
```

Za upisivanje podataka u istu strukturu koristim privatni funkcijski član PorukaStreamRegistred(). Tip poruke upisujem pomoću globalnog pobrojenja enum imePoruke u kojem se nalaze sve poruke.

```
StreamRegistredStruktura* PorukaStreamRegistred(){
        streamRegistred.tipPoruke = MSG_STREAM_REGISTERED;
        streamRegistred.identifikatorStrujanja =
        byteOrderMoj.toNetwork(porukaZaObradu.identifikatorStrujanja);
        streamRegistred.TTL_u_sekundama = byteOrderMoj.toNetwork(120);
        streamRegistred.tipJavneAdrese = 1;
        streamRegistred.javnaIPAdresa =
                                        porukaZaObradu.javnaAdresa.IPAdresa;
        streamRegistred.javniBrojPorta = porukaZaObradu.javnaAdresa.port;
        return &streamRegistred;
}
```

Identifikator strujanja čitam iz poruke koja se obrađuje i pretvaram ga u mrežni oblik pomoću objekta byteOrderMoj.toNetwork() jer svi brojevi koji se šalju u porukama moraju biti zapisani u mrežnom obliku. TTL u sekundama je 120, tip javne adrese je IPv4, a IP adresu i port prepisujem iz ulazne poruke.

U javnom funkcijskom članu obradaPoruke() se obrađuju ulazne poruke.

```
void obradaPoruke(PrijemnaPoruka2 poruka){
        porukaZaObradu = poruka;
        char polje[1024];
        string string3;
        std::pair<std::_Rb_tree_iterator<std::pair<const long unsigned int,
        RegistracijskeAdrese > >, bool> rez;
        UcitavanjeKonfiguracije citac;
        SocketAddress saMojaAdresa(citac.DajParametar(IPadresa),
        citac.DajParametar(port));
        DatagramSocket dsPorukaMaster(saMojaAdresa);
        string3 = inet_ntop(AF_INET, &porukaZaObradu.javnaAdresa.IPAdresa ,
polje, INET_ADDRSTRLEN);
        SocketAddress
saZaOdgovor(porukaZaObradu.adresaIzDatagramSocketaSaKojeJeDoslaPoruka);
        int n;
        u_char* A;
        RegistracijskeAdrese registracijskeAdrese;
        switch (porukaZaObradu.tipPoruke){
                case imePoruke::MSG_STREAM_ADVERTISEMENT:
                case imePoruke::MSG_STREAM_REMOVE:
                case imePoruke::MSG_REQ_RELAY_LIST:
                case imePoruke::MSG_FIND_STREAM_SOURCE:
                case imePoruke::MSG_FORWARD_PLAYER_READY:
                default:
        }
}
```

Nakon deklariranja i inicijaliziranja potrebnih varijabli i objekata u switch dijelu koda u odnosu na tip ulazne poruke izvršava se pojedini kod.

3.2.6 Baza podataka implementirana pomoću asocijativnog spremnika

Za pamćenje registriranih tokova multimedije (prijava izvora multimedije) koristio sam asocijativni spremnik map iz Biblioteke standardnih predložaka.

```
map<u_int64_t, RegistracijskeAdrese> registracija;
```

U strukturi RegistracijskeAdrese se nalaze svi potrebni podaci o izvoru multimedije koji traži registraciju identifikatora strujanja. Registracija se vrši sa naredbom kao što je pokazano u slijedećem kodu, a sama registracija se izvršava prilikom obrade ulazne poruke u switch dijelu koda funkcijskog člana obradaPoruke klase PorukaMajstor, case imePoruke::MSG_STREAM_ADVERTISEMENT:

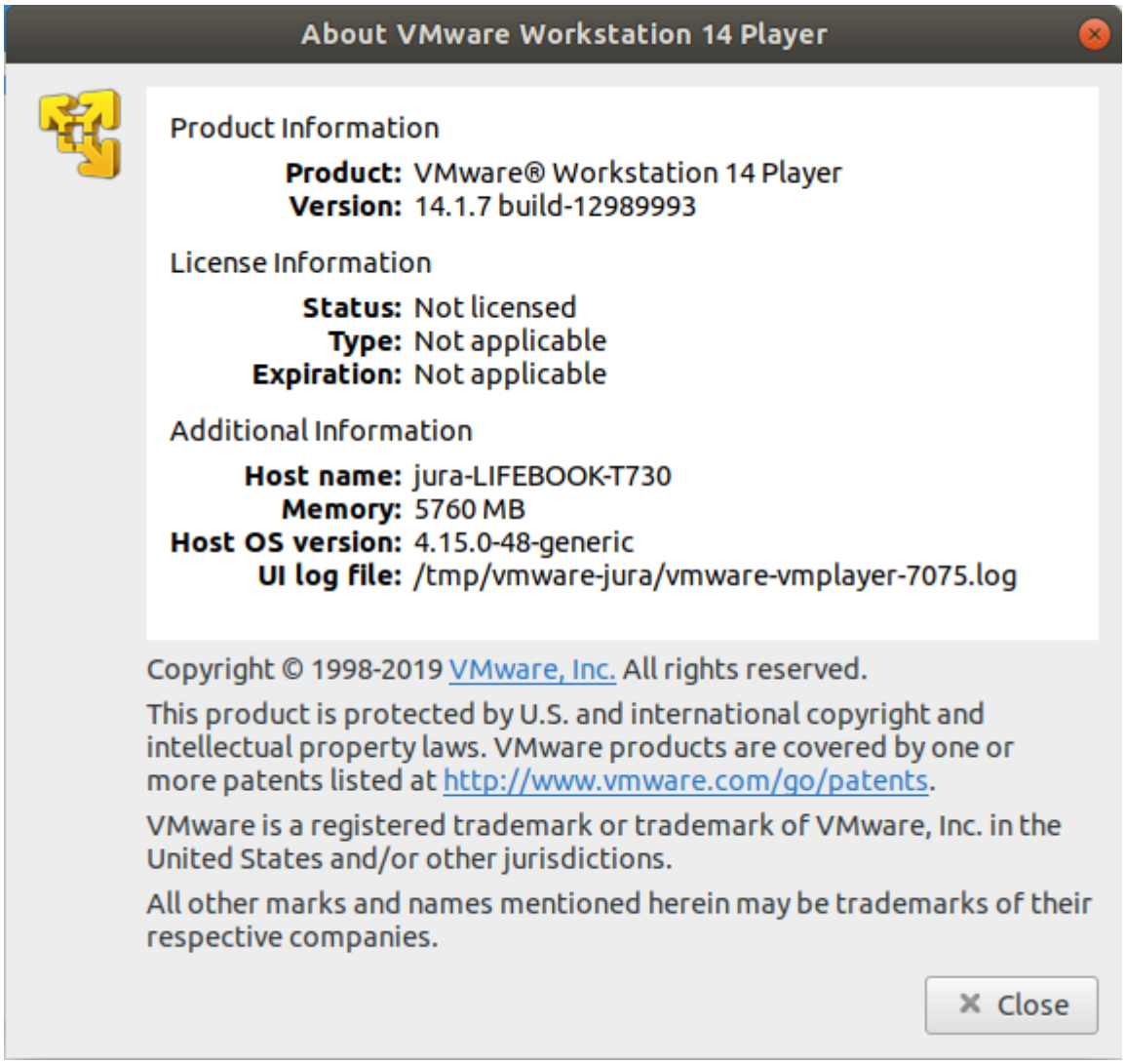
```
rez = registracija.insert({porukaZaObradu.identifikatorStrujanja,
registracijskeAdrese});
```

Na sličan način se vrše i sve ostale potrebne operacije nad objektom registracija. Odgovori se šalju pomoću objekta klase DatagramSocket dsPorukaMaster(saMojaAdresa).

```
A = (u_char*)PorukaStreamRegistred();
n = dsPorukaMaster.sendTo(A, 1024, saZaOdgovor);
```

4 Pomoćni program za testiranje poslužitelja

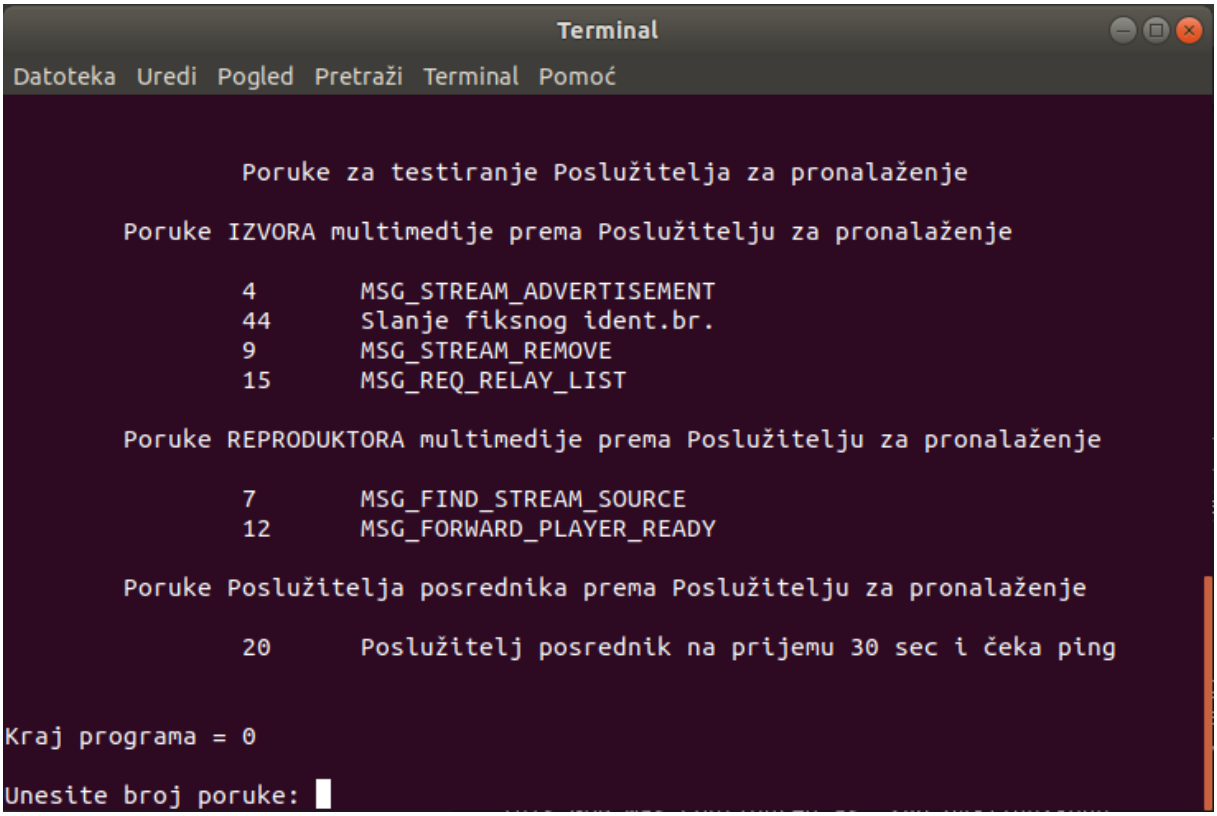
Za testiranje svih funkcionalnosti poslužitelja napravio sam pomoćni program za testiranje koji se je izvršavao na različitim računalima ili virtualnim strojevima sa operativnim sustavom Linux. Za izvršavanje virtualnih strojeva koristio sam Vmware Workstation 14 Player kao što se vidi na slici 15.



Slika 15: Vmware Workstation 14 Player

Za izradu pomoćnog programa sam koristio rješenja iz Poslužitelja za pronalaženje tamo gdje ih se je moglo primijeniti.

Kao što se vidi sa slike početnog ekrana pomoćnog programa za testiranje (slika 16) moguće je generirati sve vrste ulaznih poruka za slanje prema Poslužitelju za posredovanje. U ostalim izbornicima moguće je birati i različite količine istih poruka kao i različite kombinacije slanja istih. Nakon svih testiranja poslužitelj je korektno odradio sve zahtijevane obrade poruka.



Slika 16: Izgled početnog ekrana pomoćnog programa za testiranje

Kontrolu poslanih poruka sam vršio i programom Wireshark. Poslužitelj sam postavio na fiksnu javnu IP adresu, a poruke za testiranje sam slao iz svoje kućne mreže sa računala koje se nalazi iza mehanizma za prevođenje lokalnih adresa (NAT-a), ili sa računala koje je na internet bilo spojeno putem pristupne točke mobitela.

5 Zaključak

U ovom radu zadatak je bio izrada poslužitelja koji će biti dio raspodijeljene aplikacije, a koja će se izvršavati na različitim računalima i komunicirati putem interneta. Poslužitelj sam izradio u programskom jeziku C++ za operativni sustav Linux i tom prilikom sam morao pokazati osim znanja samog programiranja da razumijem i komunikaciju na internetu koja se ostvaruje putem različitih protokola i socket programiranja. Da bi se dobile dovoljno dobre performanse morao sam koristiti višedretveno programiranje i zaštitu kritičnih djelova koda primjenjujući mutexe i uvjetne varijable. Unutar samog aplikacijskog protokola je riješen problem komunikacije iza mehanizma za prevođenje lokalnih adresa (NAT-a) što sam testiranjem pomoću pomoćnog programa za testiranje i potvrdio da korektno radi.

6 Popis literature

[1] N.Ivković, I.Magdalenić, L.Milić, „An Ad-Hoc Smartphone-to-Smartphone Live Multimedia Streaming Application with Real-Time Constraints”, *Journal of Advances in Computer Networks,* vol. 4, no. 1, March 2016.

[2] N.Ivković, „Sustav za strujanje multimedije uživo s strogim zahtjevima za rad u realnom vremenu”, [dokument sa detaljnim specifikacijama sustava], kreirano 12.11.2014.

[3] G. Obiltschnig, A. Fabijanic, The POCO C++ Libraries project, 2006 – 2019. *Technical Steering Committee (TSC),* [Na internetu]. Dostupno: https://pocoproject.org/documentation.html [Pristupano 23.5.2019.].

[4] J. F. Kurose i K. W. Ross, *Computer Networking: A Top-Down Approach*, 6th ed. Pearson, 2012.

[5] J. Štribar i B. Motik, *Demistificirani C++,* 4. dopunjeno izdanje usklađeno sa standardom C++11/C++14, Element d.o.o., Zagreb, 2014.

7 Popis slika

Popis slika

Slika 1: Izvor i reproduktor....2

Slika 2: Obavijest o kašnjenju vizualno i kratki zvučni signal....2

Slika 3: Arhitektura sustava [1, str.7]....4

Slika 4: Visual Studio Code....5

Slika 5: Početak main() funkcije....6

Slika 6: Poco C++ Libraries Overview [3]....8

Slika 7: Izgled ekrana nakon što je server izvršio inicijalizaciju i ušao u radnu fazu....10

Slika 8: Cirkularni spremnik....25

Slika 9: Tri polja za identifikaciju socketa....27

Slika 10: Pet vrsta poruka aplikacijskog protokola koje server mora moći obraditi....28

Slika 11: Dvije odlazne poruke kao odgovori na Stream_advertisement....29

Slika 12: Struktura odlazne poruke Stream_source_data....29

Slika 13: Struktura odlazne poruke Player_ready....30

Slika 14: Struktura odlazne poruke Relay_list....30

Slika 15: Vmware Workstation 14 Player....35

Slika 16: Izgled početnog ekrana pomoćnog programa za testiranje....36

8 Popis tablica

Popis tablica

Table 1: Pet slojni internet protokol model.....11

Table 2: Sedam slojni ISO OSI referentni model.....12

Table 3: Struktura UDP segmenta.....13

Table 4: Poruke aplikacijskog protokola (sloja).....26

Table 5: Vrste adresa, vrijednosti u polju za tip adrese i duljina polja adrese.....27