Mrežno programiranje u Asio biblioteci

Krcivoj, Ivan

Master's thesis / Diplomski rad

2022

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:217:665025

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Ivan Krcivoj

MREŽNO PROGRAMIRANJE U ASIO BIBLIOTECI

Diplomski rad

Voditelj rada:
prof. dr. sc. Mladen Jurak

Zagreb, rujan 2022.

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. _____

Zahvaljujem roditeljima i sestri na bezuvjetnoj podršci i razumijevanju u svakom trenutku.
Hvala prijateljima na zajedničkim učenjima i nezaboravnim druženjima.
Hvala mentoru prof. dr. sc. Mladenu Juraku na pomoći i vremenu uloženom u ovaj rad.

Sadržaj

Uvod

Mrežno programiranje poprima sve veću važnost. Zbog velike raširenosti i pristupačnosti interneta, računala nikada nije bilo jednostavnije međusobno povezati. Uzmemo li u obzir da je jeftinije kupiti više „prosječnih" računala i povezati ih u distribuirani sustav nego kupiti jedno superračunalo istih performansi ne čudi da je većina današnjih sustava distribuirana. Kod distribuiranih sustava neophodno je omogućiti komunikaciju između računala preko mreže. Osim u distribuiranim sustavima mrežna komunikacija sastavni je dio svih igrica koje omogućuju zajedničko igranje više igrača na različitim računalima. C++ je uz C najpopularniji jezik za izradu video igara pa je potreba za bibliotekom koja će omogućiti mrežno programiranje velika. Jedna takva biblioteka u C++-u je ASIO. Ime biblioteke je kratica za *asynchronous input and output*, odnosno asinkroni ulaz i izlaz. Kasnije će biti vidljivo da je asinkronost u mrežnom programiranju veoma važna. U slijednim (sinkronim) programima instrukcije se izvršavaju jedna iza druge pa ukoliko nekoj instrukciji treba više vremena za izvršavanje cijeli program čeka. Na primjer ukoliko program treba primiti neku poruku, instrukcija za primanje poruke može se dugo izvršavati jer ne znamo točno kada će poruka stići. U takvim situacijama asinkrono izvršavanje je korisno jer možemo izvršavati dijelove programa za koje nam poruka nije potrebna dok u „pozadini" čekamo poruku.

ASIO omogućuje mrežno programiranje na dvije razine. Niža razina bazirana je na *BSD socket API*-ju. On omogućuje bolju učinkovitost jer se može povećati efikasnost programa. Iako je *BSD socket API* često korišten, sklon je greškama jer nije pisan u objektno orijentiranoj paradigmi. Primjerice, utičnica je reprezentirana kao broj (`int`) što može dovesti do greške prilikom izvođenja jer nema provjere ima li podatak „smisla". ASIO u svojoj implementaciji rješava taj problem jer koristi zasebne tipove podataka. Viša razina koristi ulazno/izlazni tok. Na taj način pojednostavljuje se pisanje aplikacije. Briga oko protokola i uspostavljanja konekcije prepušta se ASIO-u.

U ovom radu bit će govora o obje razine. Ponekad ćemo htjeti veću kontrolu nad komunikacijom između računala. Veća kontrola će nam omogućiti bolje razumijevanje mrežnog programiranja s ASIO bibliotekom. Također, vidjet ćemo kako u nekim situacijama ne moramo toliko detaljno brinuti o aspektima mrežnog programiranja pa će u tim situacijama apstraktnija razina biti sasvim zadovoljavajuća. Na početku će biti riječi o protokolima koje ćemo koristiti i uvest ćemo pojmove koje ćemo koristiti kasnije u ovom radu. Svaki

od protokola ćemo opisati i navesti njegove prednosti i mane. Vidjet ćemo kako se računala mogu povezati te kako mogu slati i primati poruke. Objasnit ćemo razliku između sinkronih i asinkronih funkcija. Naposljetku ćemo na studijskom primjeru pokazati kako se svi dijelovi mogu povezati u cjelinu.

Poglavlje 1

Mrežno programiranje

U ovom poglavlju precizno će biti definirani osnovni pojmovi vezani uz mrežno programiranje. Neke od njih, poput utičnice već smo spomenuli u Uvodu, ali ovdje će biti detaljnije objašnjeni.

U Uvodu je rečeno da mrežno programiranje služi za omogućavanje komunikacije između dva računala. To nije sasvim točno, mrežno programiranje omogućuje procesima komunikaciju preko računalne mreže. **Proces** nastaje pokretanjem računalnog programa. Isti program možemo pokrenuti više puta na istom računalu i svakim pokretanjem nastat će novi proces, također na računalu se može izvršavati više procesa istovremeno. O tome kako će se provoditi paralelni rad više procesa ovisi o svojstvima sklopovlja te o operacijskom sustavu. Više o tome možete pronaći u [3].

Da bi jedan proces mogao komunicirati s drugim mora znati gdje se drugi proces nalazi. Za početak potrebno je znati na kojem se računalu proces izvršava. Svako računalo koje je spojeno na mrežu ima jedinstveni broj koji nazivamo **IP adresa**. Kako je brojčane IP adrese teško pamtiti svako računalo ima i svoje **simboličko ime** (eng. *hostname*). **Sustav domenskih imena ili DNS** (eng. *Domain Name System*) je sustav koji omogućava pretvorbu simboličkog imena u IP adresu i obratno. U točki 2.2 bit će objašnjeno kako možemo koristiti DNS unutar ASIO biblioteke. Budući da na jednom računalu može biti više procesa svakom od njih pridružen je jedinstven broj kojeg nazivamo **priključak** (eng. *port*) pomoću kojeg ih možemo razlikovati. Uređeni par koji se sastoji od IP adrese i priključka nazivamo **krajnja točka** (eng. *endpoint*). Udaljeni proces s kojim želimo razmjenjivati poruke jedinstveno je određen svojom krajnjom točkom. U točki 2.3 bit će vidljivo kako ASIO omogućava povezivanje udaljenih procesa. Objekt koji omogućava slanje i primanje poruka na nekoj krajnjoj točki nazivamo **utičnica** (eng. *socket*). U točki 2.1 bit će detaljnije objašnjeno koje sve vrste utičnica postoje u ASIO biblioteci i za što se koriste.

Poruke koje putuju između dva procesa mogu prolaziti različitim mrežama. Osim toga računala na kojima se procesi izvršavaju mogu koristiti različito sklopovlje. Da bi se osigurala neovisnost aplikacije o sklopovlju i fizičkim svojstvima mreže koristi se **slojevita struktura**. Sloj prima podatke od sloja koji se nalazi u lancu prije njega, obrađuje ih, te ih prosljeđuje sloju koji je sljedeći u lancu. Postoji više modela mrežnih protokola. Prvi među njima bio je **OSI model** (eng. *Open Systems Interconnection Model*) koji se sastoji od 7 slojeva. Iako nikada nije u potpunosti implementiran često se koristi kao primjer apstraktnog višeslojnog mrežnog protokola. Drugi često korišteni model je **TCP/IP** model. On se sastoji od 5 slojeva.

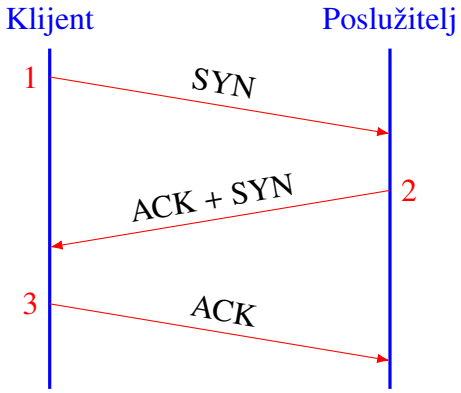
OSI referentni model	**TCP/IP model**
aplikacijski sloj	aplikacijski sloj
prezentacijski sloj	
sloj sesije	
transportni sloj	transportni sloj
mrežni sloj	internet sloj
sloj podatkovnog linka	sloj mrežnog sučelja
fizički sloj	fizički sloj

Tablica 1.1: Usporedba OSI i TCP/IP modela.

Slojeve u OSI i TCP/IP modelu možete vidjeti u tablici 1.1. Kada pošiljatelj šalje poruku ona prolazi kroz slojeve odozgo prema dolje, dok kada primatelj primi poruku ona prolazi kroz slojeve odozdo prema gore. U ostatku ovog rada bavimo se aplikacijskim slojem. Više o ostalim slojevima možete pročitati u [11]. Spomenimo ukratko transportni sloj. Postoji više protokola koji implementiraju funkcionalnost transportnog sloja, dva najpopularnija su TCP i UDP.

TCP (eng. *Transmission Control Protocol*) omogućuje spojnu komunikaciju između dva procesa. Procesi se povezuju algoritmom **trostrukog rukovanja** (eng. *3-way handshake*). Poruke koje se razmjenjuju možete vidjeti na slici 1.1. Nakon što se procesi povežu mogu razmjenjivati poruke u oba smjera. Protokol je pouzdan jer nema gubitka podataka i poruke dolaze u istom redoslijedu u kojem su poslane.

UDP (eng. *User datagram Protocol*) omogućuje bespojnu komunikaciju između procesa. Nema garancije na redoslijed u kojem će poruke stići niti hoće li uopće stići. To se može činiti kao veliki nedostatak u usporedbi s TCP protokolom, ali gubitak poruka je u praksi rijedak, a mreža nije opterećena gomilom potvrdnih poruka o primitku poruke pa



Slika 1.1: Trostruko rukovanje.

je protokol brži i efikasniji. Kako nema uspostave veze između procesa UDP je pogodan za slanje iste poruke na više odredišta (eng. *multicast*). Odabir protokola ovisi o potrebi aplikacije.

Prije početka razvoja same aplikacije potrebno je odabrati transportni protokol kojeg će aplikacija koristiti. Kako se aplikacijski sloj nalazi neposredno iznad transportnog sloja, izlaz aplikacijskog sloja mora biti kompatibilan s ulazom transportnog sloja pa aplikacija ovisi o odabranom transportnom protokolu. Osim transportnog protokola potrebno je odabrati arhitekturu sustava koju će aplikacija koristiti. Postoje razni modeli arhitekture distribuiranih procesa poput modela gospodara i roba, modela ravnopravnih partnera, arhitekture s klijentom i poslužiteljem o kojima možete pročitati u [7]. U ovom radu bit će korištena arhitektura s klijentom i poslužiteljem. Procesi u toj arhitekturi mogu se podijeliti u dvije kategorije: klijente i poslužitelje. **Klijent** je proces koji zahtjeva uslugu, dok je **poslužitelj** proces koji pruža usluge. Poslužitelj može zahtijevati uslugu od nekog drugog poslužitelja, i tada je u ulozi klijenta, ali ne može tražiti uslugu od klijenta. Klijent treba znati informacije o poslužiteljima jer on inicira komunikaciju s njima, ali ne mora znati nikakve informacije o ostalim klijentima. Poslužitelj pasivno čeka zahtjev od klijenta te mu pruža traženu uslugu.

Poglavlje 2

Povezivanje

Nakon uvodnih napomena o mrežnom programiranju dolazimo do glavne teme ovog rada, a to je mrežno programiranje s ASIO bibliotekom. Informacije o preuzimanju biblioteke i njenom pripremanju za rad možete pronaći u [2, str. 141-162].

ASIO prostor imena sastoji se od brojnih funkcija i klasa. Najvažnije među njima bit će obrađene u ovome radu. Najvažniji objekt je zasigurno objekt klase `asio::io_service`. On omogućuje pristup resursima operacijskog sustava i uspostavlja vezu između našeg programa i operacijskog sustava. Svi zahtjevi za pisanjem ili čitanjem koje omogućuju utičnice moraju biti proslijeđeni operacijskom sustavu pa je `asio::io_service` objekt neophodan. Popis svih metoda možete pronaći u dokumentaciji [6], a ovdje ćemo izdvojiti dvije najvažnije: `run`() i `poll`(). Obje metode pokreću petlju za obradu događaja. Metode se razlikuju u tome što je metoda `run`() blokirajuća, izvršavanje programa se zaustavlja dok se ne obrade svi događaji, a metoda `poll`() nije blokirajuća. Kada se svi događaji obrade (npr. pošalju se sve poruke) metode `run`() i `poll`() završavaju svoje izvršavanje. U većini slučajeva to će nam biti neželjeno svojstvo jer ako naknadno želimo poslati neku novu poruku potrebno je ponovno pozvati metode `run`() ili `poll`(), također takav pristup je sklon greškama. Ukoliko se nađemo u situaciji da želimo poslati poruku, zatim nešto izračunati pa ponovno poslati poruku, može se dogoditi situacija da smo mi gotovi s računanjem i želimo slati drugu poruku dok još prva nije poslana i tada će program raditi bez problema. No, moguća je situacija u kojoj se prva poruka pošalje dok se obavljaju izračuni te metoda `run`()\`poll`() završi svoje izvršavanje. Kad nakon završetka izračuna želimo poslati poruku ona se neće poslati i program će raditi neispravno. Problem je moguće riješiti pomoću `asio::io_service::work` klase čiji konstruktor prima `asio::io_service` objekt. Pozovemo li sada metodu `run`() ili `poll`() ona neće stati s izvršavanjem kada su svi događaji obrađeni jer će ju `asio::worker` držati „zaposlenom”. Kao što smo rekli metoda `run`() je blokirajuća pa ako je koristimo s klasom `work` daljnje izvršavanje programa će biti zauvijek blokirano. Upravo se zbog toga metoda `run`() često izvršava u zasebnoj dretvi. Kod

metode `poll`() nema tih problema jer ona nije blokirajuća. Ukoliko želimo ukloniti objekt klase `work` trebamo na njemu pozvati njegov destruktor kako bi ga uništili.

Još jedna važna klasa koju ćemo često koristiti je klasa `asio::error_code`. Objekti te klase sadrže informaciju o grešci ukoliko neka metoda završi na nestandardni način. Promatrajući definicije funkcija u dokumentaciji [6] često se pojavljuju preopterećenja od kojih jedno ne koristi `asio::error_code`, a drugo ga koristi. Razlika je u tome što ako dođe do izuzetka prva varijanta funkcije će podići izuzetak te je zadatak programera uhvatiti taj izuzetak i obraditi ga. U drugoj varijanti funkcije neće doći do izbacivanja izuzetka nego će funkcija završiti naizgled normalno, no informacije o izuzetku bit će pohranjene u objektu klase `asio::error_code`. Varijanta funkcije koja izbacuje izuzetak se smatra zastarjelom i preporučuje se korištenje varijante s objektom klase `asio::error_code`.

2.1 Utičnice

Utičnice smo već spomenuli u poglavlju 1, no ovdje ćemo reći više o njima. Utičnice se mogu podijeliti u dvije kategorije: **aktivne** i **pasivne**. Aktivne utičnice se koriste za slanje i primanje poruka te za iniciranje uspostave veze s udaljenim procesom. Pasivne utičnice čekaju zahtjev od udaljenog procesa. Kao što je rečeno ranije utičnice omogućuju sučelje za slanje i primanje poruka. Način na koji se šalju i primaju poruke ovisi o transportnom protokolu koji se koristi. Ovdje ćemo spomenuti kako se koriste TCP i UDP transportni protokoli jer su oni najčešće korišteni.

TCP protokol u ASIO-u ima tri utičnice: `acceptor`, `resolver` i `socket`. Utičnice tipa `acceptor` služe za osluškivanje novih zahtjeva za povezivanjem i za obradu tih zahtjeva. Detaljnije o njihovom korištenju bit će riječi u poglavlju 2.3 Povezivanje utičnica. Utičnice tipa `resolver` koristi se za pretraživanje krajnjih točaka u sustavu domenskih imena (DNS). U poglavlju 2.2 bit će detaljno opisano kako koristiti DNS sustava unutar ASIO biblioteke. Utičnica koja se najviše koristi je `socket`. Njen konstruktor kao i konstruktor ostalih utičnica prima objekt klase `io_service`. Utičnicu povezujemo s udaljenim procesom tako što nad njom pozovemo metodu `open` koja prima objekt krajnje točke procesa s kojim utičnicu želimo povezati. Ako metoda završi uspješno utičnicu smatramo otvorenom te ju je moguće koristiti za slanje i primanje poruka.

UDP protokol sastoji se od dvije vrste utičnica: `resolver` i `socket`. Njihovi konstruktori također primaju objekt klase `io_service` i utičnice imaju ulogu analognu istoimenoj varijanti za TCP protokol, ali koriste UDP kao transportni protokol. UDP nema utičnicu tipa `acceptor` jer UDP koristi bespojnu komunikaciju pa nema potrebe za utičnicom koja bi uspostavljala vezu.

2.2 Pretraživanje imena računala

Kao što je već rečeno, puno je jednostavnije pamtiti simboličko ime računala, nego njegovu brojevnu adresu. Međutim, da bismo računala mogli povezati potrebna nam je brojčana IP adresa. Uz pomoć DNS sustava koji između ostalog omogućava pretvaranje simboličkog imena u IP adresu možemo pretraživati računala po simboličkim imenima. Za tu svrhu u ASIO biblioteci postoji posebna klasa utičnica po imenu `resolver`. Klasa `resolver` sadrži metodu `resolve`(). Varijantu koju ćemo koristiti prima simboličko ime i priključak te objekt klase `asio::error_code`, a vraća kolekciju objekata dobivenih upitom. Kolekcija se sastoji od objekata klase `asio::ip::basic_resolver_entry`<> koja sadrži podatke o krajnjoj točki. Samu krajnju točku moguće je dobiti pozivom metode `endpoint`().

```
#include <iostream>
#include <asio.hpp>

int main(void) {
    std::string hostName = "www.math.hr";
    std::string portNumber = "80";

    asio::io_service ios;
    asio::error_code ec;

    asio::ip::tcp::resolver resolver(ios);
    asio::ip::tcp::resolver::results_type result =
        resolver.resolve(hostName, portNumber, ec);

    if(ec.value() != 0){
        std::cout << "Nije uspjelo pronalazenje imena
        DNS sustavom." << " Kod greske = " << ec.value()
        << ". Poruka = " << ec.message() << ". \n";
        return ec.value();
    }

    for(; result != result.end(); ++result){
        std::cout << result->endpoint() << std::endl;
    }
    return 0;
}
```

Kod 2.1: Pretraživanje imena TCP protokolom.

Pretpostavljamo da znamo simboličko ime i priključak računala kojeg tražimo. Ti podaci su spremljeni redom u varijablama `hostName` i `portNumber`. Primijetimo da `portNumber` nije broj nego string. Razlog je taj što funkcija `resolve`() očekuje da identifikator usluge (u našem slučaju priključak) bude string. Zatim stvaramo instancu objekta klase `asio::io_service` jer je on neophodan za sve vrste utičnica. Kako je preporučljivo koristiti preopterećenje funkcije koje ne izbacuje izuzetke potreban nam je objekt klase `asio::error_code` u kojem će biti spremljeni podaci o grešci, ako do nje dođe. Stvaramo objekt utičnice `resolver` te na njemu pozivamo metodu `resolve`() koja provodi željeni upit u DNS sustavu. Zatim provjeravamo je li došlo do greške i ako se pojavila greška ispisujemo podatke o grešci i zaustavljamo izvršavanje programa. U suprotnom imamo popis krajnjih točaka koje zadovoljavaju naš upit. U ovom primjeru sve prikupljene krajnje točke ispisujemo u konzoli.

Sljedeći primjer pokazuje kako se koristi pretraživanje imena s UDP transportnim protokolom. Programski kod je gotovo identičan, jedina razlika je ta što svugdje klasu `tcp` zamijenimo klasom `udp`.

```
#include <iostream>
#include <asio.hpp>
int main(void) {
    std::string hostName = "www.math.hr";
    std::string portNumber = "80";
    asio::io_service ios;
    asio::error_code ec;
    asio::ip::udp::resolver resolver(ios);
    asio::ip::udp::resolver::results_type result =
        resolver.resolve(hostName, portNumber, ec);
    if(ec.value() != 0){
        std::cout<<"Nije uspjelo pronalazenje imena
        DNS sustavom." << " Kod greske = " << ec.value()
        << ". Poruka = " << ec.message() << ". \n";
        return ec.value();
    }
    for(; result != result.end(); ++result){
        std::cout << result->endpoint() << std::endl;
    }
    return 0;
}
```

Kod 2.2: Pretraživanje imena UDP protokolom.

U literaturi (primjerice [10]) navodi se primjer gdje se koristi preopterećenje funkcije `resolve` koje prima objekt klase `resolver::query` i vraća iterator na prvi objekt kolekcije pronađenih krajnjih točaka. Taj se pristup smatra zastarjelim i trebalo bi ga izbjegavati.

2.3 Povezivanje utičnica

Nakon što smo vidjeli kako pronaći krajnju točku udaljenog računala pomoću njegovog simboličkog imena možemo vidjeti kako povezati dva procesa. Jedan od tih procesa bit će poslužitelj, a drugi klijent.

Promotrimo situaciju kada koristimo TCP protokol. Za početak poslužitelj treba početi „slušati" klijentske zahtjeve za povezivanje. Za slušanje nadolazećih zahtjeva koristimo posebnu klasu utičnica `asio::ip::tcp::acceptor`. Objekt te klase metodom `bind`() vežemo za krajnju točku na kojoj će poslužitelj primati zahtjeve za spajanje od klijenata. Time je poslužitelj spreman za uspostavu veze s klijentom.

S klijentske strane stvorimo objekt klase `asio::ip::tcp::socket` koji će nam služiti za razmjenu poruka jednom kada se povežemo s poslužiteljem. Kao u poglavlju 2.3 pronađemo na kojim krajnjim točkama poslužitelj čeka povezivanje s klijentom. Zatim uz pomoć statičke metode `connect`(), koja prima `socket` objekt i kolekciju krajnjih točaka koju je vratio `resolver`, povezuje `socket` objekt s poslužiteljem.

```
#include <iostream>
#include <asio.hpp>

int main(void) {
    const int SIZE = 10;
    int port_num = 8080;

    asio::ip::tcp::endpoint ep(asio::ip::address_v4::any(),
        port_num);

    asio::io_service ios;
    asio::error_code ec;

    asio::ip::tcp::acceptor acceptor(ios, ep.protocol());
    acceptor.bind(ep, ec);

    if (ec.value() != 0) {
       std::cout << "Neuspjelo vezanje acceptora s krajnjom
       tockom." << " Kod greske: " << ec.value()
```

```
        << ". Poruka: " << ec.message();
        return ec.value();
    }

    acceptor.listen(SIZE, ec);

    if (ec.value() != 0) {
        std::cout << "Neuspjeli pocetak slusanja."
          << " Kod greske: " << ec.value() << ". Poruka: "
          << ec.message();
        return ec.value();
    }

    asio::ip::tcp::socket socket(ios);
    acceptor.accept(socket, ec);

    if (ec.value() != 0) {
        std::cout << "Neuspjelo prihvacanje veze."
          << " Kod greske: " << ec.value() << ". Poruka: "
          << ec.message();
        return ec.value();
    }

    // Veza je uspostavljena. Utičnica socket se može
    //koristiti za primanje i slanje poruka
    std::cout << "Uspostavljena veza sa: "
    << socket.remote_endpoint();
    return 0;
}
```

Kod 2.3: Primjer povezivanja — TCP poslužitelj.

Za početak odredimo na kojem će priključku poslužitelj osluškivati i spremimo taj podatak u varijablu `portNumber`. Primijetimo ovdje je varijabla broj, a ne string. Da bi stvorili krajnju točku potrebna nam je IP adresa. Koristimo IP adresu verzije 4. Više o verzijama IP adresa i zašto je uvedena verzija 6 možete pronaći u [9]. Računalo može imati više IP adresa, a kako nama nije bitno koju točno koristimo, funkcijom `asio::ip::address_v4::any()` dobit ćemo neku od njih. Sada možemo stvoriti objekt koji će predstavljati krajnju točku. Stvaramo i objekt klase `asio::io_service` koji je neophodan za stvaranje utičnica. Kod stvaranja `asio::ip::tcp::acceptor` utičnice ne ko-

ristimo konstruktor koji prima samo `asio::io_service` objekt jer on samo stvara utičnicu te ju je zatim potrebno otvoriti. Korištenjem konstruktora koji osim `asio::io_service` objekta prima i protokol (u naše slučaju isti kao i protokol krajnje točke). Tada utičnica nije samo stvorena nego i otvorena. Zatim povezujemo utičnicu s krajnjom točkom koju smo stvorili za slušanje zahtjeva za spajanje. Ukoliko kod povezivanja dođe do greške prekidamo izvođenje i ispisujemo podatke o grešci. Ako se nije pojavila greška kod povezivanja, utičnicu za slušanje treba staviti u način rada u kojem sluša. U suprotnom operacijski sustav odbija sve zahtjeve koji pristignu. Utičnica se stavlja u način rada u kojem sluša metodom `listen`() koja prima broj koji određuje veličinu reda čekanja. Također, koristimo varijantu koja ne izbacuje izuzetke, nego eventualne greške sprema u poseban objekt. Kada pristigne zahtjev za povezivanje on se stavlja u red čekanja. Ako je red čekanja pun, operacijski sustav će odbijati sve nove zahtjeve dok se ne oslobodi mjesto u redu. Stvaramo utičnicu koju ćemo koristiti za komunikaciju s klijentom. Pozivom metode `accept`() poslužitelj se povezuje s klijentom koji je prvi u redu čekanja. Ukoliko je povezivanje završilo bez greške, klijent i poslužitelj su povezani te mogu razmjenjivati poruke. U našem primjeru ispisuje se poruka o uspješnom povezivanju te krajnja točka klijenta s kojim smo povezani u konzolu.

```
#include <iostream>
#include <asio.hpp>

int main(void) {
    std::string hostName = "www.google.com";
    std::string portNumber = "80";

    asio::io_service ios;
    asio::error_code ec;

    asio::ip::tcp::resolver resolver(ios);
    asio::ip::tcp::resolver::results_type endpoints =
        resolver.resolve(hostName, portNumber, ec);

    if (ec.value() != 0) {
       std::cout << "Neuspjelo pronalazenje posluzitelja."
         << " Kod greske: " << ec.value() << ". Poruka: "
         << ec.message();
       return ec.value();
     }

    asio::ip::tcp::socket socket(ios);
```

```
    asio::connect(socket, endpoints, ec);

    if (ec.value() != 0) {
        std::cout<<"Neuspjelo povezivanje sa posluziteljem."
          << " Kod greske: " << ec.value() << ". Poruka: "
          << ec.message();
        return ec.value();
     }

    std::cout << "Uspostavljena veza sa: "
    << socket.remote_endpoint() << std::endl;
    return 0;
}
```

Kod 2.4: Primjer povezivanja — TCP klijent.

U primjeru 2.4 vidimo kako povezati klijenta s poslužiteljem koristeći TCP transportni protokol. Varijable `hostName` i `portNumber` sadržavaju simboličko ime i broj priključka od poslužitelja s kojim se želimo povezati. Primijetimo da je ovdje `portNumber` string, a ne broj. Kao u poglavlju 2.2 pronađemo krajnje točke na kojima se nalazi poslužitelj. Ako je pretraga završila bez grešaka varijabla `endpoints` sadrži kolekciju pronađenih točaka. Pozivamo funkciju `asio::connect()` koja prima objekt utičnice za komunikaciju, zatim kolekciju krajnjih točaka i objekt u kojeg će biti spremljene informacije o grešci. Funkcija prolazi po kolekciji krajnjih točki te redom pokušava povezati našu utičnicu s krajnjom točkom. Ukoliko je povezivanje završilo bez grešaka utičnica `socket` je povezana s poslužiteljem i spremna za razmjenu poruka.

Nakon što smo obradili povezivanje korištenjem TCP transportnog protokola slijedi primjer povezivanja koji koristi UDP protokol. Za razliku od pretraživanja imena gdje su razlike bile minimalne, trebalo je samo zamijeniti klasu `tcp` s klasom `udp`, ovdje su razlike veće. Kako UDP koristi bespojnu komunikaciju nema potrebe za posebnom klasom utičnica koja bi prihvaćala nove klijente. Povezivanje se sastoji od toga da poslužitelj zauzme neku krajnju točku te da se klijentske utičnice povežu s njom bez potrebe da poslužitelj odobri povezivanje.

```
#include <iostream>
#include <asio.hpp>

int main(void){
    int portNumber = 8080;
    asio::ip::udp::endpoint ep(asio::ip::address_v4::any(),
        portNumber);
```

```
    asio::io_service ios;
    asio::error_code ec;

    asio::ip::udp::socket socket(ios, ep.protocol());
    socket.bind(ep, ec);

    if (ec.value() != 0) {
       std::cout << "Neuspjelo vezanje uticnice s
       krajnjom tockom." << " Kod greske: " << ec.value()
       << ". Poruka: " << ec.message();
       return ec.value();
     }

    //Utičnica socket je spremna za slanje i primanje poruka
    std::cout << "Posluzitelj se nalazi na: "
    << socket.local_endpoint();
    return 0;
}
```

Kod 2.5: Primjer povezivanja — UDP poslužitelj.

Stvorimo krajnju točku preko koje će poslužitelj komunicirati s klijentima. IP adresa je bilo koja adresa koju računalo posjeduje, a vrijednost priključka je zadana varijablom `portNumber`. Stvorimo utičnicu za komunikaciju. Ponovno koristimo verziju konstruktora koja stvara i otvara utičnicu. Potrebno je još utičnicu vezati s ranije stvorenom krajnjom točkom i ako se ne pojave greške poslužitelj je spreman za razmjenu poruka.

Klijent koji koristi UDP transportni protokol veoma je sličan klijentu koji koristi TCP protokol. Potrebno je samo klasu `tcp` zamijeniti s klasom `udp` kao što možemo vidjeti u sljedećem primjeru.

```
#include <iostream>
#include <asio.hpp>

int main(void) {
    std::string hostName = "www.google.com";
    std::string portNumber = "80";

    asio::io_service ios;
    asio::error_code ec;
```

```
    asio::ip::udp::resolver resolver(ios);
    asio::ip::udp::resolver::results_type endpoints =
        resolver.resolve(hostName, portNumber, ec);

    if (ec.value() != 0) {
       std::cout << "Neuspjelo pronalazenje posluzitelja."
         << " Kod greske: " << ec.value() << ". Poruka: "
         << ec.message();
       return ec.value();
     }

    asio::ip::udp::socket socket(ios);
    asio::connect(socket, endpoints, ec);

    if (ec.value() != 0) {
       std::cout<<"Neuspjelo povezivanje sa posluziteljem."
         << " Kod greske: " << ec.value() << ". Poruka: "
         << ec.message();
       return ec.value();
     }

    // Utičnica socket je povezana sa udaljenim procesom
    //i spremna za slanje i primanje poruka
    std::cout << "Uspostavljena veza sa: "
    << socket.remote_endpoint() << std::endl;
    return 0;
}
```

Kod 2.6: Primjer povezivanja — UDP klijent.

Kao i ranije, koristeći primjer pretraživanja imena pronađemo krajnje točke na kojima se nalazi poslužitelj. Zatim stvorimo i otvorimo utičnicu koju ćemo koristiti za komunikaciju. Funkcijom `asio::connect`() povežemo utičnicu za komunikacijom s nekom krajnjom točkom na kojoj se nalazi poslužitelj. Ako povezivanje završi uspješno možemo razmjenjivati poruke s poslužiteljem.

Utičnica uvijek mora biti vezana za krajnju točku preko koje šalje i prima poruke. Vezivanje možemo napraviti eksplicitno pozivom metode `bind`(). Taj način smo koristili na strani poslužitelja jer nam je bilo bitno da znamo na kojoj se krajnjoj točki nalazi poslužitelj kako bi se klijenti mogli povezati. Ako ne povežemo utičnicu s krajnjom točkom eksplicitno, operacijski sustav će sam vezati utičnicu za neku krajnju točku na računalu na kojem

se nalazi. Implicitni način smo koristili na strani klijenta jer nam tamo nije važno na kojoj se krajnjoj točki klijent nalazi. Za jednu krajnju točku može biti vezana samo jedna utičnica zato kod korištenja funkcije `bind`() treba paziti da se krajnja točka već ne koristi jer će inače nastupiti greška. Zato je dobra praksa uvijek kada je moguće koristiti implicitni način vezivanja jer će se operacijski sustav pobrinuti da se utičnica veže na slobodnu krajnju točku.

Poglavlje 3

Slanje poruka

Nakon što smo vidjeli kako se procesi mogu međusobno povezati vrijeme je da nešto kažemo o slanju i primanju poruka. U ovom ćemo se poglavlju baviti slanjem poruka, a u sljedećem njihovim primanjem.

Vidjet ćemo razliku između slijednih i asinkronih funkcija. Kao što smo već rekli slijedne funkcije blokiraju izvršavanje programa dok se funkcija ne završi dok to nije slučaj kod asinkronih funkcija. Asinkrone funkcije se izvode u posebnoj dretvi u kojoj se izvršava petlja događaja (eng. *event loop*). Kod poziva asinkrone funkcije kao parametar se navodi tako zvana **povratna funkcija** (eng. *call back function*). Povratna funkcija kao argumente prima izlazne podatke asinkrone funkcije i obavještava glavnu dretvu da je asinkroni poziv završio.

Razmjenom poruka šaljemo podatke iz jednog procesa u drugi. Podaci koje šaljemo i primamo spremljeni su u **međuspremnik** (eng. *buffer*). Međuspremnici mogu biti promjenjivi ili nepromjenjivi. Kako im samo ime kaže u promjenjivim spremnicima se može mijenjati sadržaj, a u nepromjenjivima ne može. Također, međuspremnike možemo podijeliti po tome je li im duljina zadana ili promjenjiva. Kod slanja poruke češće se koriste nepromjenjivi međuspremnici zadane duljine jer znamo koje podatke šaljemo. Kod primanja poruke koriste se promjenjivi međuspremnici jer u njih trebaju biti upisani podaci koje šaljemo. Također, kod primanja poruke mogu se koristiti međuspremnici promjenjive i zadane duljine. Ovisno o potrebama aplikacije, no o tome više u poglavlju 4 koje je posvećeno primanju poruka.

Promotrimo klase međuspremnika koje su zadane duljine. Ako želimo nepromjenjivi međuspremnik zadane duljine koristit ćemo klasu `asio::const_buffer`. Međuspremnik je nepromjenjiv što znači da se iz njega mogu samo čitati podaci, ali ne i pisati. Ako želimo pisati u međuspremnik onda trebamo koristiti promjenjivi međuspremnik. Klasa u ASIO-u kojom je implementiran promjenjivi međuspremnik je `asio::mutable_buffer`. Funkcije za primanje i slanje poruka koje ćemo vidjeti u nastavku ne primaju objekte klasa

asio::const_buffer ili asio::mutable_buffer nego kolekciju tih objekata. Preciznije rečeno primaju objekte koje zadovoljavaju ConstBufferSequence odnosno MutableBufferSequence ovisno radi li se o slanju ili primanju poruke. Da neki objekt zadovoljava ConstBufferSequence svojstva znači da se radi o kolekciji asio::const_buffer objekata. Na primjer, tražena svojstva zadovoljava objekt klase std::vector<asio::const_buffer>. Za objekt ćemo reći da je zadovoljio svojstva MutableBufferSequence ako se radi o skupu objekata klase asio::mutable_buffer. U tu svrhu ponovno možemo koristiti std::vector<asio::mutable_buffer> klasu. Iako nam ASIO omogućava da se poruka pohrani u više jednostavnih međuspremnika, u pravilu ćemo koristiti samo jedan. Kako ne bi stalno trebali stvarati vektor koji će imati samo jedan element možemo koristiti funkciju asio::buffer(). Funkcija vraća objekt klase asio::const_buffer_1 koja zadovoljava ConstBufferSequence svojstva ili objekt klase asio::mutable_buffer_1 koja zadovoljava MutableBufferSequence svojstva. Klasa izlaznog objekta ovisi je li ulazni spremnik konstantan ili ne.

Kada želimo koristiti međuspremnik promjenjive duljine trebamo koristiti klasu asio::streambf. Radi se o međuspremniku koji je baziran na toku podataka. Možemo ga predati konstruktoru za objekt klase std::iostream, pa ga koristiti isto kao da pišemo ili čitamo iz konzole.

3.1 Sinkrono slanje poruka

U ovoj točki bit će riječi o sinkronom slanju poruka. To znači da će metode i funkcije za slanje poruka biti sinkrone, odnosno zaustavit će izvršavanje programa dok ne pošalju barem dio podataka ili dok se ne pojavi greška.

Kod slanja poruka važno je odrediti koji ćemo transportni protokol koristiti. U daljnjem tekstu koristit ćemo TCP kao transportni protokol. Na kraju ove točke navest ćemo razlike koje bi uslijedile da smo koristili UDP protokol. Pretpostavljamo da u programu imamo utičnicu koja je otvorena i povezana s udaljenim procesom kojem želimo slati poruke te da imamo međuspremnik u kojem se nalaze podaci koje želimo poslati.

Prvi način slanje poruke je korištenjem metode write_some() koju nam pruža utičnica. Metoda prima međuspremnik i objekt za pohranu informacija o eventualnoj grešci i vraća broj bajtova koji je poslan. Metoda ne šalje cijelu poruku odjednom. Nemamo garanciju koliko će bajtova biti poslano pri pozivu metode. Znamo jedino da ako je metoda uspješno završila barem jedan bajt će biti poslan. Da bi poslali cijelu poruku metodu write_some() je potrebno pozvati nekoliko puta sve dok ukupan broj poslanih bajtova nije jednak veličini međuspremnika. Primjer koda 3.1 pokazuje kako možemo poslati cijelu poruku koristeći metodu write_some().

```
int myWrite(asio::ip::tcp::socket &socket,
        const std::string &buffer, asio::error_code &ec) {
    int byteSent = 0;
    while(byteSent < buffer.size()){
        byteSent += socket.write_some(
            asio::buffer(buffer.c_str() + byteSent,
            buffer.size() - byteSent), ec);
        if(ec.value() != 0)
            return 0;
    }
    return byteSent;
}
```

Kod 3.1: Funkcija `myWrite`().

Funkcija `myWrite`() prima referencu na TCP utičnicu, referencu na string u kojem se nalazi poruka koju želimo poslati i referencu na `asio::error_code` objekt u kojem će biti spremljena greška, ako do nje dođe. Pretpostavljamo da je utičnica povezana s udaljenim procesom i spremna za slanje poruka. U varijablu `byteSent` bit će spremljen broj bajtova koji je poslan, a na početku je taj broj 0. Petlju ponavljamo sve dok broj poslanih bajtova nije jednak veličini poruke, odnosno dok nismo poslali cijelu poruku. U svakoj iteraciji petlje šaljemo dio poruke pomoću metode `write_some`(). Prvi parametar metode treba biti međuspremnik koji zadovoljava `ConstBufferSequence` svojstva. Klasa `std::string` ne zadovoljava ta svojstva pa moramo koristiti funkciju `asio::buffer`() koja prima pokazivač na prvi znak u polju znakova te duljinu polja. Metoda `c_str`() pretvara `std::string` u polje znakova i vraća pokazivač na prvi znak. Kako će svakim prolaskom kroz petlju biti poslan neki broj znakova s početka niza, pokazivač moramo pomaknuti na prvi znak koji nije poslan, isto tako potrebno je odrediti duljinu neposlanog djela polja. Ako je prilikom slanja došlo do greške, prekidamo izvršavanje petlje i informacije o grešci šaljemo kroz objekt klase `asio::error_code`.

Druga metoda unutar klase `asio::ip::tcp::socket` koja omogućuje slijedno (sinkrono) slanje poruka je `send`(). Njena funkcionalnost je vrlo slična metodi `write_some`() jer poput nje, ako završi bez greške, šalje barem jedan bajt podataka. Koliko će bajtova točno biti poslano nije poznato unaprijed, nego ćemo taj podatak dobiti kao povratnu vrijednost. Postoje preopterećenja metode `send`() koja primaju objekt klase `asio::socket base::message_flages` koji sadrži zastavice. Pomoću tih zastavica moguće je odrediti dodatna svojstva vezana uz slanje poruke. Klasičan način slanja poruka većinom je sasvim dovoljan, pa se zastavice rijetko koriste.

Ovakav način slanja poruke može se činiti kompleksan jer je potrebno puno linija koda za vrlo čestu operaciju. Funkcija `asio::write`() radi isto što i funkcija `myWrite`() iz

primjera 3.1. Prima tri parametra i vraća broj poslanih bajtova. Prvi parametar treba zadovoljavati `SyncWriteStream` svojstva. Detalje o svojstvima možete pronaći u dokumentaciji [6]. Klasa `asio::ip::tcp::socket` zadovoljava ta svojstva. Drugi parametar je međuspremnik koji zadovoljava `ConstBufferSequence` svojstva. Prisjetimo se funkcija `myWrite`() je kao drugi parametar primala referencu na objekt klase `std::string` koji ne zadovoljava tražena svojstva, no u primjeru 3.1 smo vidjeli kako možemo riješiti taj problem. Treći i posljednji parametar je referenca na objekt klase `asio::error_code` iz kojeg možemo pročitati podatke o grešci, ako do nje dođe. Razlika između funkcije `asio::write`() i metode `write_some`(), osim toga što je `asio::write`() slobodna funkcija, je ta da `asio::write`() šalje cijeli sadržaj međuspremnika.

Ukoliko želimo koristiti UDP transportni protokol za sinkrono slanje poruka potrebno je napraviti nekoliko manjih izmjena. Umjesto utičnice klase `asio::ip::tcp::socket` koristimo klasnu `asio::ip::udp::socket`. Klasa `asio::ip::udp::socket` ima metode `write_some`(), `send`() koje imaju isto ponašanje kao i istoimena metoda unutar klase `asio::ip::tcp::socket`. Osim toga objekt klase `asio::ip::udp::socket` zadovoljava `SyncWriteStream` svojstva pa se može koristiti kao prvi parametar pri pozivu funkcije `asio::write`().

3.2 Asinkrono slanje poruka

U ovoj ćemo točki navesti funkcije i metode koje omogućavaju asinkrono slanje poruka. Primarno ćemo koristiti TCP protokol, no ne bi bilo značajnijih promjena da smo koristili UDP kao transportni protokol. Asinkrone funkcije nisu blokirajuće. Pozivom asinkrone funkcije za slanje poruke funkcija započne operaciju slanja te odmah vraća kontrolu pozivatelju. Samo slanje poruke odvija se u posebnoj dretvi.

Kako bi znali kada je slanje završilo potrebno je navesti povratnu funkciju. Argument koji označava povratnu funkciju može primiti pokazivač na funkciju, lambda izraz ili bilo koji drugi objekt koji zadovoljava `WriteHandler` uvjete koje je moguće naći u dokumentaciji [6]. Povratnu funkciju najjednostavnije je proslijediti kao pokazivač, ali to nije uvijek moguće. Ponekad želimo povratnoj funkciji proslijediti neke dodatne parametre. To možemo tako da koristimo lambda izraz pa ti dodatni parametri mogu činiti njegovu okolinu. Drugi način je korištenjem funkcije `std::bind`(). Funkcija `std::bind`() kao prvi parametar prima funkciju koju želimo koristiti. Ostali parametri se koriste za specijalizaciju funkcije koja je proslijeđena kao prvi parametar. Iz teorije izračunljivosti znamo da za svaku izračunljivu funkciju koja prima barem jedan parametar postoji izračunljiva funkcija koja je jednaka početnoj kada fiksiramo zadnjih nekoliko parametara. Više o tome možete pronaći u skripti [12]. Specijalizaciju funkcije možemo promatrati na način da se zadnjih nekoliko ulaznih podataka fiksira na neke zadane vrijednosti. Na taj način dobijemo funkciju koja prima manje parametara. Upravo to radi funkcija `std::bind`(). Ona stvara novu

funkciju koja se dobije tako da se proslijeđenoj funkciji fiksiraju ulazni parametri. Ulazni parametri će biti fiksirani na vrijednosti koje su proslijeđene funkciji `std::bind`().

Kod slanja poruka povratna funkcija prima dva parametra. Prvi je objekt koji ima informacije o eventualnim greškama, a drugi broj bajtova koji je poslan.

```
void callback_handler( const asio::error_code &ec,
                                              std::size_t bytes);
```

Kod 3.2: Prototip povratne funkcije.

Prva metoda koju ćemo spomenuti je `async_write_some`(). Ona poput ranije spomenute metode `write_some`() pošalje neki broj bajtova, ali ovoga puta dretva nije blokirana dok se odvija slanje. Kako se radi o asinkronoj metodi potrebno je u pozivu metode navesti povratnu funkciju. Ukoliko želimo poslati cijelu poruku ne možemo kao ranije metodu pozvati u petlji nekoliko puta jer operacija nije slijedna (sinkrona). U ovom slučaju za višestruke pozive moramo koristiti povratnu funkciju. Da bismo to mogli učiniti povratnoj funkciji trebamo proslijediti dodatne parametre poput utičnice na koju šaljemo poruku, međuspremnika u kojem se nalazi poruka i broj bajtova koji je već poslan. Bitno je voditi računa o tome da svi objekti koje koristimo u povratnoj funkciji ne budu uništeni prije završetka asinkrone operacije.

```
template<typename WriteHandler>
void callback(asio::error_code &ec, std::size_t bytes,
        std::shared_ptr<asio::ip::tcp::socket> socket,
        std::string buffer, std::size_t totalBytes,
        WriteHandler handler){
    totalBytes += bytes;
    if(ec.value() != 0 || totalBytes == buffer.size()){
        handler(ec, totalBytes);
        return;
    }
    socket->async_write_some(asio::buffer(buffer.c_str() +
        totalBytes, buffer.size()- totalBytes), std::bind(
        callback, std::placeholders::_1,
        std::placeholders::_2, socket, buffer, totalBytes,
        handler));
}
```

Kod 3.3: Povratna funkcija.

Primjer koda 3.3 pokazuje jednu moguću povratnu funkciju koja omogućava slanje cijele poruke. Funkcija, kao i svaka povratna funkcija, prima objekt greške i broj poslanih bajtova. Osim toga, povratna funkcija mora imati pristup utičnici, međuspremniku i broju bajtova koji je ranije poslan pa i te podatke šaljemo kao parametre. Da bi osigurali da se utičnica ne uništi prije nego što završi izvršavanje asinkrone operacije pakiramo ju u klasu `std::shared_ptr`. Zadnji argument prima funkciju koja će biti pozvana kada je cijela poruka poslana ili je došlo do greške. Prvo provjeravamo je li došlo do greške ili jesmo li poslali cijelu poruku. U tom slučaju nema potrebe za ponovnim pozivom metode `async_write_some`() i pozivamo funkciju koja je proslijeđena kao zadnji argument kako bi označili kraj rada asinkrone operacije. Ako se nije pojavila nikakva greška i nismo poslali cijeli sadržaj međuspremnika ponovno pozivamo metodu `async_write_some`() s istom povratnom funkcijom, ali s novim parametrima. Nakon nekoliko uzastopnih poziva cijela poruka će biti poslana ili će se pojaviti greška. U svakom slučaju izvršavanje funkcije će završiti. Početni poziv metode `async_write_some`() koji koristi ranije opisanu povratnu funkciju možete pronaći u primjeru koda 3.4.

```
template<typename WriteHandler>
void myAsyncWrite(std::shared_ptr<asio::ip::tcp::socket>
        socket, std::string buffer, WriteHandler handler){
    socket->async_write_some(asio::buffer(buffer),
    std::bind(callback, std::placeholders::_1,
        std::placeholders::_2, socket, buffer, 0, handler));
}
```

Kod 3.4: Funkcija `myAsyncWrite`().

Drugi način je korištenjem metode `async_send`(). Ona je također slična metodi `async_write_some`() jer šalje neku količinu bajtova koja se ne može unaprijed odrediti, no ova metoda prima i dodatni parametar sa zastavicama koje daju dodatne informacije o slanju. Te zastavice se nalaze unutar objekta klase `asio::socket_base::message_flages`. Budući da se zastavice rijetko koriste nećemo ih detaljnije opisivati.

Slanje čitave poruke metodom `async_write_some`() čini se poprilično kompleksno za operaciju koju često koristimo u mrežnom programiranju. Kako bi slanje čitave poruke bilo jednostavnije možemo koristiti funkciju `asio::async_write`() koja poput funkcije `myAsyncWrite`() poziva nekoliko puta metodu `async_write_some`() sve dok se ne pošalje cijela poruka ili ne dođe do greške. Ulazni parametri su malo općenitiji nego u našem primjeru. Prvi parametar mora zadovoljavati `AsyncWriteStream` svojstva koja klasa `asio::ip::tcp::socket` zadovoljava. Drugi i treći parametar jednaki su kao prva dva parametra metode `async_write_some`(). Ti parametri predstavljaju međuspremnik i povratnu funkciju. Povratna funkcija koja se koristi u funkciji `asio::async_write`()

bit će pozvana ako je nastupila greška ili ako je cijela poruka poslana. Prototip povratne funkcije isti je kao i ranije. Prima objekt greške i broj poslanih bajtova (primjer 3.2).

Ako želimo koristiti UDP transportni protokol programski kod se neće značajno razlikovati. U imenima klasa riječ `tcp` je potrebno zamijeniti s riječju `udp`. Utičnice TCP i UDP protokola imaju iste metode za slanje. Osim toga klasa `asio::ip::udp::socket`, koja predstavlja UDP utičnicu, zadovoljava `AsyncWriteStream` svojstva. Zahvaljujući tome UDP utičnicu možemo proslijediti kao prvi parametar funkciji `asio::async_write`().

Poglavlje 4

Primanje poruka

U ovom poglavlju govorit ćemo o primanju poruka. ASIO nam omogućava da na više načina primimo poruku. Za početak trebamo odrediti koji ćemo transportni protokol koristiti: TCP ili UDP. Kada smo odabrali željeni protokol stvaramo odgovarajuću utičnicu preko koje ćemo vršiti komunikaciju. Pretpostavljamo da je utičnica povezana s procesom od kojeg želimo primiti poruku. Pokazat ćemo načine kako poruke primati na sinkron i asinkron način. Kod primanja poruka asinkronost je veoma korisna jer nemamo gornju ogradu na vrijeme kašnjenja poruke. To znači čak i kada bismo znali kada je poruka poslana ne znamo koliko ćemo ju dugo čekati.

Kod slanja poruke koristili smo statične međuspremnike jer smo unaprijed znali sadržaj i duljinu poruke koju želimo poslati. Kod primanja poruke to nije slučaj. Međuspremnici trebaju biti promjenjivi kako bi u njih mogli zapisati sadržaj poruke. Postoje situacije kada zbog svojstva problema unaprijed znamo koliko će poruka biti dugačka. U nastavku će biti navedeni primjeri kako postupati u takvim situacijama. U većini slučajeva ne znamo unaprijed koliko će biti velika poruka koju primamo. Potreban nam je način pomoću kojeg ćemo znati da smo pročitali cijelu poruku. Najjednostavniji način bi bio taj da znamo koliko će poruka biti velika, pa provjerimo veličinu pročitane poruke. Kao što smo rekli to nije uvijek slučaj pa moramo koristiti neke druge načine provjere. Jedan mogući način je da pri razvoju aplikacije postignemo dogovor kako će izgledati kraj poruke. To može biti znak, string, regularni izraz ili neka funkcija. Nažalost niti ovaj način nije uvijek moguće koristiti u praksi. Ponekad želimo primati poruke od procesa koje nismo mi razvijali i nemamo nikakva saznanja o tome postoji li neka oznaka u poruci koja označava kraj.

Već smo rekli da utičnica mora biti povezana ako želimo kroz nju slati i primati podatke. Podaci se šalju i primaju kroz dva toka podataka. Postoji ulazni tok pomoću kojeg se primaju poruke i izlazni tok pomoću kojeg se šalju poruke. Kada se utičnica zatvori oba toka se prekidaju. Međutim, ASIO nam omogućava da svaki tok ugasimo zasebno. Tako primjerice nakon što pošaljemo poruku možemo ugasiti izlazni tok. Kod primanja

te poruke doći će do greške koja će signalizirati da je izlazni tok zatvoren. No, do greške će doći tek nakon što pročitamo cijelu poruku i to će nam biti oznaka da je cijela poruka pročitana. Potrebno je samo provjeriti je li greška tipa `asio::error::eof` jer tada znamo da to nije greška i da trebamo nastaviti izvršavanje na uobičajen način. Problem kod ovog pristupa je taj što ovako možemo poslati samo jednu poruku, jer kada se izlazni tok jednom zatvori nije ga više moguće otvoriti. Ako želimo ponovno poslati poruku moramo stvoriti novu utičnicu.

4.1 Sinkrono primanje poruka

U ovoj točki obradit ćemo slijedno (sinkrono) primanje poruke. Budući da govorimo o sinkronom načinu rada, metode i funkcije će blokirati izvršavanje programa dok se poruka ne pročita ili dok ne dođe do greške. Potrebno je odrediti transportni protokol kojeg ćemo koristiti za slanje poruka. U daljnjem tekstu veći naglasak ćemo staviti na TCP protokol, ali isti zaključci vrijede i da smo koristili UDP protokol.

Ponovno pretpostavljamo da imamo objekt klase `asio::ip::tcp::socket` koji je otvoren i povezan s procesom od kojega želimo primiti poruku. Za povezivanje utičnice može se koristiti postupak opisan u poglavlju 2.3.

Klasa `asio::ip::tcp::socket` ima nekoliko metoda za sinkrono čitanje poruka. Prva koju ćemo opisati je metoda `read_some()`. Metoda prima dva argumenta. Prvi je promjenjivi međuspremnik u kojeg će biti upisana primljena poruka. Međuspremnik treba biti dovoljno velik da stane sadržaj cijele poruke. Drugi argument metode je objekt klase `asio::error_code` u kojem će biti spremljeni podaci o grešci ako do nje dođe. Metoda vraća broj bajtova koji je pročitan. Metoda ne čita cijelu poruku. Nema nikakve garancije na količinu bajtova koja će biti pročitana. Poznato je samo da ako metoda završi bez pojave greške barem jedan bajt će biti pročitan. Da bismo pročitali cijelu poruku potrebno je u petlji nekoliko puta pozvati metodu `read_some()` dok cijela poruka nije pročitana.

```
int myRead(asio::ip::tcp::socket &socket,
    asio::mutable_buffers_1 &buffer,
    asio::error_code &ec){
    int byteRead = 0;
    while(byteRead < buffer.size()){
        byteRead += socket.read_some(buffer, ec);
        if(ec.value() != 0)
            return 0;
    }return byteRead;
}
```

Kod 4.1: Funkcija `myRead()`.

U primjeru koda 4.1 možemo vidjeti jedan način kako pročitati cijelu poruku. Funkcija myRead() prima tri parametra i vraća broj pročitanih znakova. Prvi parametar je referenca na objekt klase asio::ip::tcp::socket. Utičnica koja je proslijeđena kao prvi argument funkcije mora biti otvorena i povezana s procesom. Drugi parametar je referenca na objekt klase asio::mutable_buffers_1 koji služi kao međuspremnik u kojem će biti zapisana poruka koju smo primili. Pretpostavljamo da je međuspremnik jednake veličine kao i poruka koju primamo. Zadnji parametar je objekt klase asio::error_code u kojem će biti spremljeni podaci o grešci ako do nje dođe. Varijabla byteRead bilježi koliko smo bajtova do sada primili. Na početku nismo primili niti jedan bajt pa je početna vrijednost nula. U svakom koraku petlje pročitamo dio poruke pomoću metode read_some() i ažuriramo broj pročitanih znakova. Ako se prilikom čitanja pojavila greška zaustavljamo izvršavanje s povratnom vrijednosti nula. Kada smo pročitali cijelu poruku, odnosno kada je broj pročitanih znakova jednak veličini međuspremnika izlazimo iz petlje i funkcija završava svoje izvođenje.

Druga metoda koju možemo koristiti je recive(). Ona poput metode read_some() čita neki broj bajtova koji su poslani. Metoda ima nekoliko preopterećenja od kojih neka primaju objekt klase asio::socket_base::message_flages koji predstavlja zastavice. Zastavice služe za dodatna svojstva poruke, ali rijetko se koriste pa ih nećemo detaljno obrađivati. Metoda recive() bez zastavica radi potpuno jednako kao metoda read_some().

Za primanje poruka možemo koristiti funkciju asio::read() koja kao povratnu vrijednost vraća broj pročitanih bajtova. Prvi argument funkcije mora zadovoljavati SyncReadStream uvjete koje zadovoljava klasa asio::ip::tcp::socket. Drugi parametar je međuspremnik koji treba biti tipa MutableBufferSequence. U njemu će se nalaziti pročitani sadržaj poruke. Treći i zadnji parametar ponovno je objekt za dojavu greške. Funkcija čita podatke dok se cijeli međuspremnik ne napuni ili dok se ne pojavi greška. Ukoliko zanemarimo činjenicu da su u funkciji asio:read() parametri općenitiji, funkcija radi isto što i funkcija myRead() iz primjera 4.1.

U praksi često nije poznato koliko će biti velika poruka koju želimo primiti. Međutim, mogu biti poznata neka druga svojstva poruke, poput toga s čime poruka završava. Takav način čitanja nam omogućava funkcija asio::read_until(). Postoji više varijanti funkcije asio::read_until() ovisno o uvjetu zaustavljanja. Uvjet zaustavljanja može biti znak, string, regularni izraz ili funkcija. Prvi argument funkcije treba zadovoljavati SyncReadStream uvjete. Primjer klase koja zadovoljava te uvjete je ranije spomenuta klasa asio::ip::tcp::socket. Drugi parametar predstavlja međuspremnik u koji će biti zapisan sadržaj poruke. Treći parametar je jedan od navedenih uvjeta zaustavljanja. Četvrti parametar je objekt za pohranu informacija o eventualnoj grešci. Funkcija u pozadini koristi metodu read_some(), pa nije nužno da će se uvjet zaustavljanja nalaziti točno na kraju pročitane poruke. Pretpostavimo da je uvjet zaustavljanja znak '#'. Nakon uspješnog završetka funkcije asio::read_until() u međuspremniku se mogu nalaziti znakovi

nakon prve pojave znaka '#'. Funkcija `asio::read_until`() nam garantira da će se u međuspremniku nalaziti barem jedan znak '#'. U općenitom slučaju imamo garanciju da postoji barem jedan znak (ili niz znakova) koji zadovoljava uvjet zaustavljanja.

Zadnja funkcija koja omogućava čitanje s utičnice rijetko se koristi pa ćemo ju ovdje samo spomenuti. Riječ je o funkciji `asio::read_at`() koja omogućava čitanje s pomakom. Funkcija radi poput funkcije `asio::read`(), čita znakove dok se međuspremnik ne napuni ili ne dođe do greške. Razlika je u tome što funkcija `asio::read_at`() prima broj koji označava pomak, odnosno koliko početnih znakova će biti preskočeno.

Da smo se odlučili za UDP transportni protokol kod ne bi bio znatno drugačiji. Utičnica bi bila klase `asio::ip::udp::socket`. UDP utičnica ima iste metode kao i TCP utičnica tako da ostatak koda ne trebamo mijenjati. S UDP utičnicom možemo koristiti funkcije za čitanje poruke poput `asio::read_until`(), `asio::read_at`() i `asio::read`() jer ona zadovoljava `SyncReadStream` svojstva.

4.2 Asinkrono primanje poruka

Sve metode i funkcije za primanje poruke koje ćemo obraditi u ovoj točki neće blokirati glavnu dretvu. Pozivom tih funkcija započet će operacija primanja poruka i funkcija će potom završiti svoje izvršavanje. Da bismo znali kada je operacija primanja poruke završila trebamo navesti povratnu funkciju koja će tada biti pozvana. Povratna funkcija prima dva parametra. Prisjetimo se primjera 3.2. Prvi je objekt u kojem će se nalaziti informacije o eventualnoj grešci, a drugi je broj bajtova koji je primljen.

Prva metoda klase `asio::ip::tcp::socket` koju ćemo spomenuti u ovoj točki je `async_read_some`(). Kao prvi parametar metoda prima promjenjivi međuspremnik koji zadovoljava `MutableBufferedSequence` svojstva. Drugi parametar je povratna funkcija koja može biti proslijeđena kao pokazivač na funkciju, lambda izraz ili bilo koji objekt koji zadovoljava `ReadHandler` svojstva. Metoda `async_read_some`() ne mora pročitati cijelu poruku čak i ako završi bez greške. Ne znamo unaprijed koliko će maksimalno bajtova biti pročitano. Zbog toga što je funkcija asinkrona ne možemo ju pozvati nekoliko puta u petlji da bismo pročitali cijelu poruku. Eventualne dodatne pozive metode `async_read_some`() moramo obaviti u povratnoj funkciji. U sljedećem primjeru vidjet ćemo kako možemo primiti cijelu poruku ako unaprijed znamo koliko će poruka biti dugačka. Trebamo paziti da utičnica s koje primamo poruku i međuspremnik u kojeg ju spremamo ne budu uništeni prije završetka asinkrone operacije.

```
template<typename WriteHandler>
void callbackRead( asio::error_code &ec, std::size_t bytes,
        std::shared_ptr<asio::ip::tcp::socket> socket,
        std::shared_ptr<asio::mutable_buffers_1> buffer,
```

```
        std::size_t totalBytes, WriteHandler handler){
    totalBytes += bytes;
    if(ec.value() != 0 || totalBytes == buffer->size()){
        handler(ec, totalBytes);
        return;
    }
    socket->async_read_some(buffer, std::bind(callbackRead,
        std::placeholders::_1, std::placeholders::_2,
        socket, buffer, totalBytes, handler));
}
```

Kod 4.2: Povratna funkcija `callbackRead`().

U primjeru 4.2 vidimo kod koji se može koristiti kao povratna funkcija koja prima sadržaj cijele poruke koristeći metodu `async_read_some`(). Funkcija `callbackRead`() ne prima samo dva parametra koji su obavezni jer su nam potrebne neke dodatne informacije. Osim objekta greške i broja bajtova koji su primljeni u zadnjem pozivu metode `async_read_some`() potrebna nam je utičnica preko koje vršimo komunikaciju, zatim međuspremnik u kojem će biti spremljena poruka. Osim toga trebamo znati koliko je ukupno do sada bajtova primljeno te pozivom koje funkcije ćemo označiti da je cijela poruka primljena. Ako je cijela poruka primljena, odnosno broj primljenih bajtova je jednak veličini međuspremnika ili je došlo do greške označavamo kraj asinkrone operacije pozivom funkcije koja je proslijeđena kao zadnji parametar. Ako nismo došli do kraja izvođenja asinkrone operacije ponovno pozivamo metodu `async_read_some`() na zadanoj utičnici s istom povratnom funkcijom. U našem slučaju to je funkcija `callbackRead`().

```
template<typename WriteHandler>
void myAsyncRead(std::shared_ptr<asio::ip::tcp::socket>
        socket, std::shared_ptr<asio::mutable_buffers_1>
        buffer, WriteHandler handler){
    socket->async_read_some(buffer, std::bind(callbackRead,
        std::placeholders::_1, std::placeholders::_2,
        socket, buffer, 0, handler));
}
```

Kod 4.3: Funkcija `myAsyncRead`().

Asinkronu operaciju započinjemo funkcijom `myAsyncRead`() iz primjera 4.3. Funkcija prima dijeljeni pokazivač na utičnicu koja je povezana s procesom od kojeg želimo primiti poruku. Drugi parametar je dijeljeni pokazivač na međuspremnik koji je jednake veličine kao i poruka koju primamo. Zadnji parametar označava funkciju ili lambda izraz kojim će biti označen kraj izvršavanja asinkrone operacije. U metodi `async_read_some`()

za povratnu funkciju koristimo ranije spomenutu funkciju `callbackRead`(). Koristimo funkciju `std::bind`() kako bi prenijeli dodatne parametre koji su nam potrebni u funkciji `callbackRead`().

Umjesto metode `async_read_some`() možemo koristiti metodu `async_recive`() koja ima istu funkcionalnost. Funkcija `async_recive`() može primiti dodatni objekt klase `asio::socket_base::message_flages` kojim je moguće odrediti dodatne opcije vezane uz način primanja poruka.

Korištenjem programskog koda iz primjera 4.2 i 4.3 možemo primiti sadržaj cijele poruke i navesti funkciju koja će označiti kraj primanja poruke. Istu funkcionalnost nam omogućava i funkcija `asio::async_read`(). Prvi parametar koji proslijedimo mora zadovoljavati uvjete `AsyncReadStream`. Objekti klase `asio::ip::tcp::socket` zadovoljavaju tražene `AsyncReadStream` uvjete stoga njih možemo koristiti kao prvi parametar funkcije. Drugi parametar je međuspremnik u kojem će biti spremljen sadržaj poruke. Međuspremnik mora biti tipa `MutableBufferSequence` da bi se u njega mogla spremiti poruka. Zadnji parametar je povratna funkcija. Ona će biti pozvana kada cijela poruka bude pročitana ili dođe do greške. Pretpostavljamo da je veličina međuspremnika jednaka veličini poruke koju primamo. Kao i u našem primjeru određujemo je li cijela poruka učitana tako da provjerimo je li međuspremnik popunjen.

Pretpostavka da unaprijed znamo koliko je poruka velika je dosta optimistična. U praksi je to rijetko slučaj. Puno realističnija pretpostavka je da znamo s čime poruka završava. U tom slučaju možemo koristiti funkciju `async_read_until`() gdje možemo navesti znak, string, regularni izraz ili funkciju koja određuje kraj poruke. Trebamo pripaziti da međuspremnik bude dovoljno velik kako bi u njega stala cijela poruka. Ako je nakon oznake za kraj poruke poslana još jedna poruka moguće je da se dogodi situacija u kojoj pri čitanju prve poruke nakon znaka zaustavljanja bude pročitan početak druge poruke. Razlog tome je taj što se u pozadini koristi metoda `asio::async_read_some`() i moguće je da prilikom zadnjeg poziva bude pročitano previše bajtova. Funkcija `async_read_until`() nam osigurava postojanje barem jedne oznake za kraj poruke u pročitanoj poruci, a na nama je da se pobrinemo o točnom razdvajanju poruka.

Ako se pojavi situacija u kojoj želimo preskočiti nekoliko prvih bajtova poruke to možemo učiniti funkcijom `async_read_at`(). U pozivu funkcije navedemo broj koji označava koliko bajtova želimo preskočiti. U praksi nemamo često potrebu za preskakanjem početka poruke tako da se ova funkcija rijetko koristi.

Kao što smo do sada već više puta vidjeli i ovdje je razlika između TCP i UDP protokola minimalna. Potrebno je samo koristiti odgovarajuću klasu utičnica. Sve navedene metode i funkcije koje smo spomenuli u ovoj točki rade jednako bez obzira na tip transportnog protokola kojeg koristimo.

Poglavlje 5

Projekt

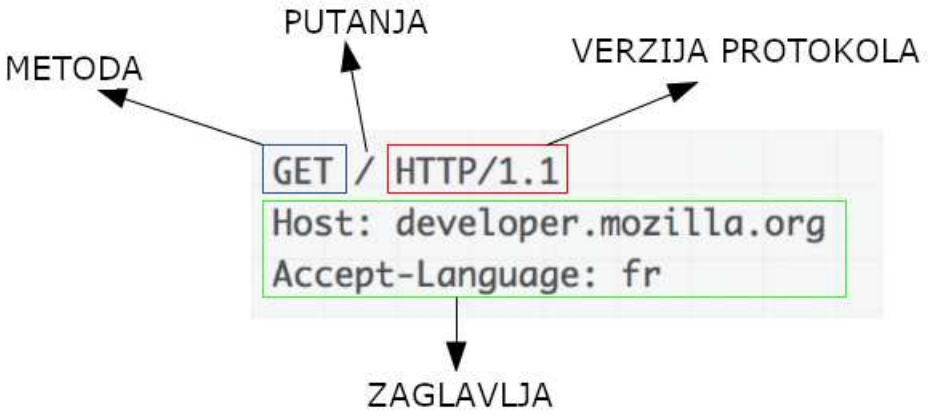
Posljednje poglavlje ovog rada je projektni zadatak. Cilj projektnog zadatka je objediniti sve stvari koje su navedene u ranijim poglavljima u smislenu cjelinu. U tu svrhu bit će implementirani HTTP klijent i poslužitelj.

5.1 HTTP protokol

HTTP (eng. *HyperText Transfer Protocol*) je protokol aplikacijskog sloja (vidi tablicu 1.1) koji koristi arhitekturu s klijentom i poslužiteljem. Klijent započinje komunikaciju slanjem zahtjeva (eng. *request*) poslužitelju. Poslužitelj obrađuje zahtjev i na njega odgovara porukom (eng. *response*). Zahtjev i odgovor istog su oblika. Oboje poruke se sastoje od dva dijela: **zaglavlja** (eng. *headers*) i **tijela** (eng. *body*) koji su odvojeni praznom linijom. U zaglavlju se nalaze dodatne informacije o poruci. Svaki element zaglavlja nalazi se u zasebnoj liniji. U tijelu poruke se nalazi sama poruka koju smo poslali.

Internetski preglednik koristi HTTP protokol za dohvat internetske stranice. On ima ulogu klijenta koji šalje poruku tipa zahtjev. Prva linija zaglavlja kod zahtjeva sastoji se od tri dijela. Prvi dio označava radnju koju zahtijevamo od poslužitelja. Popis mogućih radnji nalazi se u tablici 5.1. Zatim slijedi putanja do željene datoteke i na posljetku verzija protokola koju koristimo. Najčešće korištene verzije su HTTP/1.0 i HTTP/1.1 iako verzija HTTP/2 poprima sve veću važnost. Zatim slijede ostali dijelovi zaglavlja koji su oblika `ključ: vrijednost`. Završetak zaglavlja se označava praznom linijom nakon koje slijedi tijelo poruke. Primjer poruke zahtjeva možete vidjeti na slici 5.1.

Ulogu poslužitelja najčešće obavlja poslužitelj internetskih stranica. Poslužitelj primi zahtjev, obradi ga, i šalje odgovor. Odgovor se također sastoji od zaglavlja i tijela. Prva linija zaglavlja ponovno je posebnog oblika. Sastoji se od verzije HTTP-a koja se koristi zatim od statusnog koda i statusne poruke. Nakon toga slijede ostali elementi zaglavlja koji pružaju dodatne informacije o poruci. Primjerice element zaglavlja može biti vrijeme



Slika 5.1: Primjer HTTP zahtjeva.

zadnje promjene datoteke. U tijelu poruke se nalazi sama poruka koja primjerice kod GET zahtjeva sadrži traženu datoteku. Statusni kodovi podijeljeni su u kategorije koja su vidljive u tablici 5.2. Neke česte statusne kodove moguće je pronaći u tablici 5.3.

Prva verzija HTTP protokola, kasnije nazvana HTTP/0.9, imala je vrlo oskudnu funkcionalnost. Bila je podržana samo GET metoda. Zahtjev se sastojao od ključne riječi GET i putanje do željene datoteke, a odgovor se sastojao od tijela u kojem je bio sadržaj tražene datoteke. U verziji HTTP/1.0 dodane su metode HEAD i POST. Osim toga poruke su poprimili ranije opisani oblik sa zaglavljem i tijelom. U verziji HTTP/1.1 dodane su preostale metode iz tablice 5.1.

Rekli smo da je HTTP protokol aplikacijskog sloja koji se nalazi iznad transportnog sloja. U poglavlju 1 Mrežno programiranje bilo je riječi kako se svaki sloj mora adekvatno povezati s okolnim slojevima (onima iznad i ispod njega). HTTP protokol od transportnog protokola zahtjeva da bude pouzdan. TCP protokol zadovoljava uvjet pouzdanosti te se on često koristi kao transportni protokol unutar HTTP protokola. U komunikaciji HTTP protokolom klijent i server prvo uspostave TCP vezu te zatim razmjenjuju poruke zahtjeva i odgovora. U prvim verzijama HTTP protokola za svaki zahtjev otvarala se nova TCP veza. Međutim, razvojem internet stranica koje su postajale sve bogatije raznim elementima otvaranje zasebnih TCP veza usporavalo je njihovo učitavanje. Primjerice za svaku sliku koja se nalazi na internetskoj stranici bilo je potrebno otvoriti novu TCP vezu. Verzijom HTTP/1.1 napravljena je optimizacija koja dopušta korištenje jedne TCP veze za učitavanje svih elemenata stranice. Verzija HTTP/2 nastavila je napredak HTTP protokola

Metoda
GET
HEAD
POST
PUT
DELETE
CONNECT
OPTIONS
TRACE
PATCH

Tablica 5.1: HTTP metode.

Statusni kod	Uloga
100 – 199	Informacije
200 – 299	Uspješna obrada
300 – 399	Preusmjeravanje
400 – 499	Greška na strani klijenta
500 – 599	Greška na strani poslužitelja

Tablica 5.2: Kategorije statusnih kodova.

Statusni kod	Opis
200	OK
404	Not Found
413	Request Entity Too Large
500	Server Error
501	Not Implemented
505	HTTP Version Not Supported

Tablica 5.3: Često korišteni statusni kodovi.

u pogledu optimizacije. Podaci slani prijašnjim verzijama protokola bili su u tekstualnom obliku te su bili lako čitljivi i razumljivi ljudima. Od verzije HTTP/2 podaci se šalju u binarnom obliku te više nisu čitljivi ljudima. Osim promjene oblika podataka u kojem se poruke šalju omogućeno je paralelno slanje više zahtjeva i kompresija zaglavlja jer su podaci u zaglavlju slični u više zahtjeva.

5.2 Klijent

U ovoj točki govorit ćemo o klijentskoj strani HTTP protokola. Bit će opisan postupak kako korištenjem funkcija i metoda koje nam pruža ASIO biblioteka možemo implementirati HTTP protokol. Koristit ćemo asinkrone funkcije kako u slučaju kvara na poslužiteljskoj strani ne bi došlo do zastoja u programu, no o tome više kasnije.

TCP protokol ćemo koristiti za slanje HTTP poruka. HTTP protokol od transportnog protokola očekuje samo da bude pouzdan, a to svojstvo zadovoljava TCP protokol. Prije rada s HTTP protokolom potrebno je uspostaviti TCP vezu. Pretpostavljamo da nam je poznata krajnja točka željenog poslužitelja. Odnosno da znamo IP adresu računala na kojoj se nalazi poslužitelj te priključak na kojem poslužitelj osluškuje nadolazeće zahtjeve za povezivanjem. Stvaramo utičnicu klase `asio::ip::tcp::socket` i pozivamo metodu `connect`() kojoj kao parametar prosljeđujemo krajnju točku na kojoj se nalazi poslužitelj. Provjeravamo je li povezivanje uspjelo provjerom vrijednosti objekta `asio::error_code`, koji je također bio parametar metode `connect`(). Ukoliko je povezivanje uspjelo, otvorili smo TCP vezu i možemo prijeći na dio vezan u HTTP protokol.

U dodatku priloženom uz ovaj rad nalazi se programski kod koji implementira HTTP klijenta. Isječke tog koda možete vidjeti u nastavku. HTTP klijent je implementiran kao klasa pod nazivom `HttpClient` koju možete vidjeti u isječku koda 5.1. Kako je za

korištenje HTTP protokola nužan transportni protokol, koji će u našem slučaju biti TCP protokol, konstruktor prima `asio::ip::tcp::socket` utičnicu preko koje će biti slane i primane poruke. U daljnjem tekstu pretpostavljamo da je utičnica otvorena i povezana s HTTP poslužiteljem.

```
class HttpClient {
public:
    HttpClient(std::shared_ptr<asio::ip::tcp::socket> sock);
    void sendRequest(std::string header, std::string body);
    void stop();
private:
    std::shared_ptr<asio::ip::tcp::socket> mSocket;
    asio::streambuf mResponse;
    unsigned int mResponseStatusCode;
    std::size_t mResourceSize;
    std::map<std::string, std::string> mResponseHeaders;
    std::string mBody;
    std::unique_ptr<char[]> mResourceBuffer;
    std::string mResponseStatusLine;
    std::thread mTimeThread;
    std::atomic<bool> mDone{false};

    void startHandling();
    void onResponseReceived(
          const asio::error_code& ec,
                            std::size_t bytesTransferred);
    void onHeadersReceived(const asio::error_code& ec,
                           std::size_t bytesTransferred);

    void processResponse();
    void onRequestSent(const asio::error_code& ec,
                            std::size_t bytesTransferred);
    void onFinish();
    void onBodyReceived(const asio::error_code& ec,
                            std::size_t bytesTransferred);
};
```

Kod 5.1: Klasa HttpClient.

U HTTP protokolu klijent inicira komunikaciju slanjem *request* poruke. U tu svrhu koristimo metodu `sendRequest`() koja prima dva parametra tipa `std::string`. Prvi pa-

rametar predstavlja zaglavlje poruke, a drugi tijelo koji trebaju biti oblika kako je opisano u točki 5.1 HTTP protokol. Funkcija ulazne parametre spaja u jedan string konkatenacijom uz korištenje prazne linije kao separator. Zatim je potrebno poslati novonastali string. To činimo pozivom metode `asio::async_write` koja šalje cijelu poruku. Kako je riječ o asinkronoj funkciji potrebno joj je prenijeti povratnu funkciju koja će u ovom slučaju biti privatna metoda `HttpClient::onRequestSent`().

Koristimo asinkronu funkciju kako bi po potrebi mogli zaustaviti obradu zahtjeva. Razlozi zašto bi htjeli prekinuti izvođenje mogu biti razni. Korisnik aplikacije može odlučiti da ga više ne zanima rezultat traženog upita ili zbog kvara na poslužiteljskoj strani dođemo u stanje vječnog čekanja. Kod 5.2 prikazuje implementaciju slanja zahtjeva poslužitelju koja prestaje s obradom zahtjeva nakon 10 sekundi. Varijabla `mTimeThread` označava dretvu u kojoj se izvršava čekanje, a varijabla `mSocket` utičnicu preko koje se vrši komunikacija. Varijabla `mDone` je objekt klase `std::atomic<bool>`. Ona postane istinita kada obrada zahtjeva završi jer tada više nije potrebno mjeriti vrijeme.

```
void HttpClient::sendRequest(std::string header,
std::string body) {
    mTimeThread = std::thread([this](){
        for(int i= 0; i < 1000; i++){
            if(mDone)return;
            std::this_thread::sleep_for(
                        std::chrono::milliseconds(10));
        }mSocket->cancel();
    });
    std::vector<asio::const_buffer> responseBuffers;
    responseBuffers.push_back(
        asio::buffer(header + "\r\n\r\n" + body));
    asio::async_write(*mSocket.get(),responseBuffers,
    [this]( const asio::error_code& ec,
    std::size_t bytesTransferred) {
    onRequestSent(ec, bytesTransferred);
    });
}
void HttpClient::onRequestSent(const asio::error_code& ec,
    std::size_t bytesTransferred){
    mSocket->shutdown(asio::ip::tcp::socket::shutdown_send);
    startHandling();
}
```

Kod 5.2: Slanje zahtjeva poslužitelju.

U metodi `HttpClient::onRequestSent`() provjeravamo je li slanje poruke bilo uspješno. Ako je poruka uspješno poslana znamo da smo sa slanjem podataka gotovi, pa na utičnici možemo zatvoriti tok za slanje. Kako HTTP protokol ne definira duljinu niti oznaku za kraj tijela poruke, poslužitelja ćemo obavijestiti da je cijela poruka poslana tako što ćemo zatvoriti izlazni tok utičnice. Bitno je napomenuti kako zatvaramo samo izlazni tok, ulazni tok mora ostati otvoren kako bi mogli primiti odgovor poslužitelja. Kada smo poslali cijelu poruku i obavijestili poslužitelja da je cijela poruka poslana, klijentska strana čeka da poslužitelj pošalje odgovor na zatraženi upit.

Metodom `HttpClient::startHandling`() započinje primanje i obrada odgovora kojeg je poslao poslužitelj. Poruku počinjemo čitati funkcijom `asio::async_read_until`(). Želimo pročitati samo prvu liniju zaglavlja jer znamo da je ona posebnog oblika, pa čitamo do prelaska u novi red koji je označen nizom znakova '`\r\n`'. Kako poruku čitamo asinkronom funkcijom moramo joj predati povratnu funkciju koja je u ovom slučaju metoda `HttpClient::onResponseReceived`().

U toj metodi rastavljamo prvu liniju zaglavlja, koju smo upravo pročitali, na tri dijela. Prvi dio označava koju verziju protokola je poslužitelj koristio. Drugi dio je statusni kod poruke i na kraju je opis tog statusnog koda. Preostaje nam pročitati ostale elemente zaglavlja. Kraj zaglavlja ćemo prepoznati po praznoj liniji odnosno po nizu znakova '`\r\n\r\n`'.

Povratna funkcija koja označava kraj čitanja zaglavlja prikazana je kodom 5.5. Riječ je o metodi `HttpClient::onHeadersReceived`(). U njoj primljena zaglavlja obradimo tako da ih pospremimo u mapu. Razlog zašto smo se odlučili za mapu kao strukturu podataka za ostala zaglavlja je taj što su zaglavlja oblika `ključ: vrijednost`. Nakon obrade zaglavlja prelazimo na čitanje tijela poruke. Sada koristimo funkciju `asio::async_read`() jer ovoga puta ne navodimo oznaku za kraj čitanja budući da ona nije određena protokolom.

```
void HttpClient::startHandling()
{
    asio::async_read_until(*mSocket.get(), mResponse, "\r\n",
     [this]( const asio::error_code& ec,
                 std::size_t bytesTransferred)
    {
        onResponseReceived(ec, bytesTransferred);
    });
}
```

Kod 5.3: Čitanje prve linije zaglavlja.

```
void HttpClient::onResponseReceived(
const asio::error_code& ec, std::size_t bytes_transferred)
{
      std::string responseLine;
      std::istream responseStream(&mResponse);
      std::getline(responseStream, responseLine, '\r');
      // Uklanjanje znak '\n' iz međuspremnika.
      responseStream.get();
      // Rastavljanje prve linije zaglavlja.
      std::istringstream requestLineStream(responseLine);
      std::string responseHttpVersion;
      requestLineStream >> responseHttpVersion;
     requestLineStream >> mResponseStatusCode;
     requestLineStream >> mResponseStatusLine;
      // Prva linija zaglavlja je obrađena.
      // Pročitaj ostatak zaglavlja.
      asio::async_read_until(*mSocket.get(),
        mResponse,
        "\r\n\r\n",
        [this](const asio::error_code& ec,
                        std::size_t bytesTransferred)
         {
           onHeadersReceived(ec, bytesTransferred);
          });
      return;
}
```

Kod 5.4: Čitanje ostatka zaglavlja.

Metoda `HttpClient::onBodyReceived()` poziva se na kraju čitanja tijela poruke. Do sada smo uvijek očekivali da će čitanje proći bez izuzetaka, no sada to nije slučaj. Poslužitelj će nas obavijestiti da je poslana cijela poruka tako što će zatvoriti izlazni tok utičnice koju koristi za komunikaciju. Ta radnja će na klijentskoj strani, kod čitanja poruke, izazvati `asio::error::eof` izuzetak. Pojavom tog izuzetka znamo da smo pročitali cijelu poruku te prelazimo u obrađivanje primljene poruke.

Za obradu poruke koristimo metodu `HttpClient::processResponse()` koja ispisuje odgovor poslužitelja u konzolu.

Kraj obrade zahtjeva označavamo metodom `HttpClient::onFinish()`. U njoj postavljamo varijablu `mDone` na istinu. Time prestaje izvršavanje dretve koja mjeri vrijeme.

```
void HttpClient::onHeadersReceived(const asio::error_code&
                    ec,std::size_t bytesTransferred){
    std::istream requestStream(&mResponse);
    std::string headerName, headerValue;
    while (!requestStream.eof()) {
       if(requestStream.peek() == '\r')
           break;
     std::getline(requestStream, headerName, ':');
     if (!requestStream.eof()) {
         std::getline(requestStream, headerValue,'\r');
          requestStream.get(); // znak '\n'
          mResponseHeaders[headerName] = headerValue;
    } }
    // Uklanjanje praznog reda
    requestStream.get(); requestStream.get();
     // Čitanje tijela poruke.
    asio::async_read(*mSocket.get(), mResponse,
      [this]( const asio::error_code& ec,
                std::size_t bytesTransferred)
       {
         onBodyReceived(ec, bytesTransferred);
        });
    return;
}
```

Kod 5.5: Metoda onHeadersReceived().

```
void HttpClient::onBodyReceived(const asio::error_code& ec,
                          std::size_t bytesTransferred){
    if (ec != asio::error::eof)
        onFinish();
    mResourceSize = bytesTransferred;
    std::istream is(&mResponse);
    std::getline(is, mBody, {});
    processResponse();
    onFinish();
    return;
}
```

Kod 5.6: Metoda onBodyReceived().

5.3 Poslužitelj

Metode i funkcije koje pruža ASIO biblioteka možemo iskoristiti za implementaciju HTTP poslužitelj. Ponovno ćemo koristiti TCP transportni protokol i asinkrone funkcije.

Na početku je potrebno odrediti krajnju točku na kojoj će poslužitelj slušati. Obično se za HTTP poslužitelja koristi priključak (eng. *port*) 80, ali može se koristiti i bilo koji drugi. Kada smo stvorili krajnju točku na kojoj će naš poslužitelj slušati možemo stvoriti utičnicu za slušanje. Utičnica za slušanje je objekt klase `asio::ip::tcp::acceptor` koju je potrebno staviti u stanje slušanja kao u poglavlju 2.3 Povezivanje. Želimo omogućiti da se više zahtjeva može paralelno izvršavati. Za tu svrhu stvorit ćemo nekoliko (broj ovisi o uređaju) dretvi u kojima će se izvršavati `asio::io_service::run`() metoda. Kako će petlja događaja (eng. *event loop*) koja se izvršava u metodi `run`() imati povremeno praznog hoda trebamo spriječiti njen prestanak rada. Klijentski zahtjevi koji pristignu nakon završetka rada petlje događaja neće biti obrađeni. Klasa `asio::io_service::work` osigurava da petlja događaja ne prekine svoje izvršavanje.

```
asio::io_service::work worker(ios);
std::vector<std::unique_ptr<std::thread>> workerThreads;
for (unsigned int i = 0; i < threadNumber; i++) {
    std::unique_ptr<std::thread> th(
        new std::thread([&ios](){
            ios.run();
        }));
    workerThreads.push_back(std::move(th));
}
```

Kod 5.7: Paralelno izvođenje petlje događaja.

Kao i kod klijenta koristit ćemo asinkrone operacije kako bi ih po potrebi mogli prekinuti. Tako ćemo ovdje koristiti asinkronu operaciju za prihvat nove konekcije. U povratnoj funkciji navodimo dio koda kojim započinjemo obradu HTTP zahtjeva te započinjemo proces prihvata novog klijenta. Primijetimo da se klijenti ne prihvaćaju istovremeno, nego jedan za drugim. No, obrada njihovih zahtjeva može teći paralelno.

```
std::shared_ptr<asio::ip::tcp::socket> socket(new
                asio::ip::tcp::socket(ios));
acceptor->async_accept(*socket.get(),[socket, acceptor]
    (const asio::error_code& ec){
        start(socket, acceptor, ec);
});
```

Kod 5.8: Uspostavljanje veze.

```
void start(std::shared_ptr<asio::ip::tcp::socket>
    socket, std::shared_ptr<asio::ip::tcp::acceptor>
    acceptor, const asio::error_code& ec)
{
    if (ec.value() != 0)
        return; // Greška
    HttpServer *server = new HttpServer(socket);
    server->startHandling();
    std::shared_ptr<asio::ip::tcp::socket>
        newSocket(new asio::ip::tcp::socket(
            acceptor->get_executor()));
    acceptor->async_accept(*newSocket.get(),
        [newSocket, acceptor](const asio::error_code& ec){
        start(newSocket, acceptor, ec);
    });
}
```

Kod 5.9: Početak obrade zahtjeva.

Kada imamo TCP utičnicu za komunikaciju, dolazimo do dijela koji je vezan za HTTP protokol. Klasa `HttpServer` koju je moguće pronaći u primjeru 5.10 implementira funkcionalnost HTTP poslužitelja. Konstruktor prima utičnicu preko koje će se vršiti komunikacija. Bitno je osigurati da utičnica ne bude uništena prije nego li završe sve asinkrone operacije koje su uz nju vezane. To možemo osigurati korištenjem dijeljenog pokazivača na objekt odnosno uz korištenje klase `std::shared_ptr<asio::ip::tcp::socket>`.

```
class HttpServer {
    static const std::map<unsigned int, std::string>
                    http_status_table;
public:
    HttpServer(std::shared_ptr<asio::ip::tcp::socket> sock);
    void startHandling();
private:
    std::shared_ptr<asio::ip::tcp::socket> mSocket;
    asio::streambuf mRequest;
    unsigned int mResponseStatusCode;
    std::size_t mResourceSize;
    std::map<std::string, std::string> mRequestHeaders;
    std::string mRequestedResource;
    std::unique_ptr<char[]> mResourceBuffer;
```

```
    std::string mResponseHeaders;
    std::string mResponseStatusLine;
    std::string mBody;
    std::string mRequestMethod;
    std::thread mTimeThread;
    std::atomic<bool> mDone{false};

    void onRequestReceived(
          const asio::error_code& ec,
                           std::size_t bytesTransferred);
    void onHeadersReceived(const asio::error_code& ec,
                           std::size_t bytesTransferred);

    void processRequest();
    void sendResponse();
    void onResponseSent(const asio::error_code& ec,
                        std::size_t bytesTransferred);
    void onFinish();
    void onBodyReceived(const asio::error_code& ec,
                        std::size_t bytesTransferred);
    void get();
    void head();
    void put();
    void post();
    void del();
};
```

Kod 5.10: Klasa HttpServer.

Pozivom javne metode `Httpserver::startHandeling`() započinje obrada zahtjeva. Kao i kod klijenta pokrećemo posebnu dretvu u kojoj odbrojavamo maksimalno vrijeme za obradu zahtjeva. Ako se klijent poveže, a nikada ne pošalje zahtjev zbog greške u klijentskom kodu ili iz nekih zlonamjernih razloga resursi na poslužitelju nikada neće biti oslobođeni. Da bismo spriječili takvo ponašanje kada istekne maksimalno vrijeme predviđeno za obradu zahtjeva prekidamo obradu i oslobađamo zauzete resurse. Po propisima HTTP protokola poruke zahtjeva od klijenta se sastoje od zaglavlja i tijela poruke. Prvo se šalje zaglavlje pa zatim tijelo poruke odvojeno praznom linijom. Kako je prva linija zaglavlja posebnog oblika prvo nju čitamo s funkcijom `asio::async_read_until`() gdje za razdjelnik koristimo prelazak u novi red '`\r\n`'. Povratna funkcija u ovoj asinkronoj funkciji je privatna metoda `HttpClient::onRequestReceived`().

```
void HttpServer::startHandling() {
    mTimeThread = std::thread([this](){
        for(int i= 0; i < 1000; i++){
            if(mDone)return;
            std::this_thread::sleep_for(
                    std::chrono::milliseconds(10));
        }
        mSocket->cancel();
    });
    asio::async_read_until(*mSocket.get(),
            mRequest, "\r\n",
            [this](
            const asio::error_code& ec,
            std::size_t bytesTransferred)
          {
            onRequestReceived(ec, bytesTransferred);
          });
}
```

Kod 5.11: Čitanje prve linije zaglavlja.

Nakon što smo pročitali prvu liniju trebamo ju rastaviti na dijelove. Prvi dio označava metodu koju klijent zahtjeva od poslužitelja. Drugi dio se sastoji od putanje do resursa na kojem treba primijeniti metodu i naposljetku je verzija HTTP protokola koja se koristi. Ponovno koristimo funkciju `asio::async_read_until`(), ali sada s razdjelnikom '\r\n\r\n' kako bi pročitali ostatak zaglavlja.

```
void HttpServer::onRequestReceived( const asio::error_code&
                ec, std::size_t bytes_transferred)
{
      std::string requestLine;
      std::istream requestStream(&mRequest);
      std::getline(requestStream, requestLine, '\r');
      requestStream.get(); // Zbog znaka \n
      // Obrada prve linje zaglavlja.
      std::istringstream requestLineStream(requestLine);
      requestLineStream >> mRequestMethod;
      requestLineStream >> mRequestedResource;
      std::string requestHttpVersion;
      requestLineStream >> requestHttpVersion;
```

```
if (requestHttpVersion.compare("HTTP/1.1") != 0) {
  // Verzija HTTP protokola nije podržana.
  mResponseStatusCode = 505;
  sendResponse();
return; }
// Čitanje ostatka zaglavlja.
asio::async_read_until(*mSocket.get(),
  mRequest,
  "\r\n\r\n",
  [this](
  const asio::error_code& ec, std::size_t
                          bytesTransferred)
   {
     onHeadersReceived(ec, bytesTransferred);
    });
return; }
```

Kod 5.12: Obrada prve linije zaglavlja.

Pročitana zaglavlja potrebno je obraditi. To činimo metodom `onHeadersReceived()`. Sva pristigla zaglavlja spremamo u mapu tipa `std::map<std::string, std::string>` kako bi bilo lako kasnije preko ključa pristupiti vrijednosti zaglavlja. Time završavamo obradu zaglavlja i prelazimo na čitanje tijela poruke. Sav preostali sadržaj poruke pripada tijelu poruke pa ga primamo funkcijom `asio::async_read()`.

```
void HttpServer::onHeadersReceived(const asio::error_code&
                    ec, std::size_t bytesTransferred){
    std::istream requestStream(&mRequest);
    std::string headerName, headerValue;
    while (!requestStream.eof()) {
       if(requestStream.peek() == '\r')
          break;
     std::getline(requestStream, headerName, ':');
     if (!requestStream.eof()) {
        std::getline(requestStream, headerValue,'\r');
         requestStream.get(); // Mičemo \n
         mRequestHeaders[headerName] = headerValue;
    } }
    // Potrebno je ukloniti praznu liniju.
    requestStream.get();
    requestStream.get();
```

```
    asio::async_read(*mSocket.get(),
      mRequest,
      [this](
      const asio::error_code& ec,
                 std::size_t bytesTransferred)
       {
         onBodyReceived(ec, bytesTransferred);
        });
    return;
}
```

Kod 5.13: Obrada ostatka zaglavlja.

Po završetku čitanja tijela poruke poziva se metoda `onBodyReceived`(). Kako ne znamo unaprijed koliko će tijelo poruke biti dugačko niti nije definirana oznaka za kraj poruke, klijent će poslužitelja obavijestiti da je poslao cijelu poruku tako što će zatvoriti izlazni tok svoje utičnice. Zatvaranje izlaznog toka će kod poslužitelja izazvati izuzetak `asio::error::eof` kojeg je potrebno obraditi. Obrada se svodi na to da nastavljamo s normalnim izvršavanjem programa jer je riječ o željenom ponašanju.

```
void HttpServer::onBodyReceived(const asio::error_code& ec,
                                std::size_t bytesTransferred){
    if (ec != asio::error::eof) {
        onFinish(); // Greška.
    }
    mResourceSize = bytesTransferred;
    std::istream is(&mRequest);
    std::getline(is, mBody, {});
    processRequest();
    sendResponse();
    return;
}
```

Kod 5.14: Metoda onBodyReceived().

Sad smo preuzeli cijelu poruku koju nam je poslao klijent. Obradu poruke započinjemo u metodi `processRequest`(). Provjeravamo koju je metodu korisnik zatražio. Ako se radi o `GET` metodi tada korisnik traži da mu se pošalje sadržaj datoteke u tijelu odgovora. Metoda `HEAD` je slična kao `GET` metoda, ali se u odgovoru ne šalje tijelo poruke. Metodu `HEAD` koristimo kada želimo primiti samo zaglavlje odgovora. Ta nam metoda može biti korisna kada imamo spremljenu lokalnu kopiju nekog resursa pa želimo provjeriti je li se resurs mijenjao od zadnjeg preuzimanja. Element zaglavlja može biti vrijeme zadnje

modifikacije traženog resursa, pa je moguće pristiglo vrijeme modifikacije usporediti s vremenom modifikacije lokalnog resursa. Tako je moguće napraviti optimizaciju u kojoj se ne šalje ponovno resurs kojeg već posjedujemo. Metoda `PUT` zahtjeva od poslužitelja da stvori datoteku sa sadržajem koji se nalazio u tijelu zahtjeva. Metoda `POST` je slična kao metoda `PUT` samo što se ovdje sadržaj tijela poruke dodaje na kraj datoteke. Posljednja metoda koju ćemo podržati u ovoj implementaciji je metoda `DELETE` koja briše datoteku. Ovom implementacijom nisu podržane sve metode koje propisuje protokol HTTP/1.1. koje se nalaze u tablici 1.1. Ukoliko klijent zatraži metodu koja nije podržana dobit će poruku o grešci sa statusom '501 Not Implemented'.

Nakon što smo obradili zahtjev klijenta potrebno mu je poslati odgovor. To radimo metodom `sendResponse`(). Na početku spajamo dijelove poruke u jednu cjelinu kako je propisano HTTP protokolom. Poruku šaljemo funkcijom `async_write`().

```
void HttpServer::sendResponse()  {
    auto statusLine =
      http_status_table.at(mResponseStatusCode);
    mResponseStatusLine = std::string("HTTP/1.1 ") +
      statusLine + "\r\n";
    mResponseHeaders += "\r\n";
    std::vector<asio::const_buffer> responseBuffers;
    responseBuffers.push_back(
          asio::buffer(mResponseStatusLine));
if (mResponseHeaders.length() > 0) {
     responseBuffers.push_back(
              asio::buffer(mResponseHeaders));
}
  if (mResourceSize > 0)
    responseBuffers.push_back( asio::buffer(
              mResourceBuffer.get(), mResourceSize));
  // Započinjemo asinkrono slanje poruke.
  asio::async_write(*mSocket.get(),responseBuffers,
    [this](const asio::error_code& ec,
              std::size_t bytesTransferred)
  {
    onResponseSent(ec,bytesTransferred);
  });
}
```

Kod 5.15: Metoda sendResponse().

Metoda koja označava završetak slanja poruke zove se `onResponseSent`(). U toj metodi provjeravamo je li slanje poruke završilo uspješno. Obavještavamo primatelja poruke da je cijela poruka poslana tako što zatvorimo izlazni tok utičnice preko koje se vrši komunikacija. Time utičnica postaje neupotrebljiva za daljnje operacije jer su joj oba toka zatvorena.

```
void HttpServer::onResponseSent(const asio::error_code& ec,
    std::size_t bytesTransferred)
{
    if (ec.value() != 0) ;// Greška.
    mSocket->shutdown(asio::ip::tcp::socket::shutdown_send);
    onFinish();
}
```

Kod 5.16: Metoda onResponseSent().

Metoda `onFinish`() provodi završno čišćenje kako bi objekt klase `HttpServer` bio spreman za uništenje.

```
void HttpServer::onFinish() {
    mDone = true;
    mTimeThread.join();
}
```

Kod 5.17: Metoda onFinish().

Ako želimo zaustaviti rad poslužitelja potrebno je zaustaviti sve asinkrone operacije. Jedan pokušaj za zaustavljanje asinkronih operacija bi bio uništavanje `worker` objekta. Nakon uništenja tog objekta, kada petlja događaja obradi sve događaje u redu čekanja, novopristigli događaji neće biti obrađeni. U našoj implementaciji poslužitelja takav pristup nećemo moći koristiti jer prihvatom novog klijenta započinjemo novu asinkronu operaciju koja traje sve dok se ne pojavi novi klijent. I tako neprestano imamo operacije koje je potrebno izvršavati. Pozivom metode `stop`() na `asio::io_service` objektu prekidaju se sve asinkrone operacije vezane uz taj objekt. Iz toga slijedi da možemo koristiti metodu `stop`() za zaustavljanje rada poslužitelja.

ASIO biblioteka pruža nam dovoljnu funkcionalnost za implementaciju HTTP klijenta i poslužitelja. Implementirali smo samo dio funkcionalnosti HTTP protokola, ali i ostale dijelove je moguće napraviti koristeći ASIO biblioteku. U zadnjem poglavlju vidjeli smo kako povezati isječke kodova koji su bili prezentirani u ranijim poglavljima. Biblioteka se na prvu može činiti kompleksna za korištenje, no uz malo truda brzo se nauči. Velika prednost ASIO biblioteke je ta što je dobro dokumentirana pa je lagano naći potrebnu funkciju ili klasu.

Bibliografija

[1] *An overview of HTTP*, 2022, https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview.

[2] W. Anggoto i J. Torjo, *Boost.Asio C++ Network Programming - Second Edition*, Packt, 2015.

[3] L. Budin, M. Golub, D. Jakobović i L. Jelenković, *Operacijski sustavi*, Element, 2018.

[4] Z. Bujanović, L. Grubišić i V. Petričević, *Mreže računala vježbe 01*, 2015, https://web.math.pmf.unizg.hr/nastava/mreze/slideovi/2015/MR%20-%20Vjezbe%20-%2001.pdf, str. 11–17.

[5] L. Grubišić i R. Manger, *Mreže računala*, 2009–2013, https://web.math.pmf.unizg.hr/nastava/mreze/predavanja/MR-skripta.pdf, str. 57–60.

[6] C. M. Kohlhoff, *Boost.Asio*, 2019, https://www.boost.org/doc/libs/1_70_0/doc/html/boost_asio.html.

[7] R. Manger, *Softversko inženjerstvo*, Element, 2016.

[8] ———, *Distribuirani procesi*, 2017–2022, http://web.studenti.math.pmf.unizg.hr/~manger/protect/DP-Skripta.pdf, str. 19–36.

[9] J. Pyles, J. L. Carrell i E. Tittel, *Guide to TCP/IP: IPv6 and IPv4 5th edition*, Cengage Learning, 2016.

[10] D. Radchuk, *Boost.Asio C++ Network Programming Cookbook*, Packt, 2016.

[11] W. Stallings, *Data and Computer Communications, Eight Edition*, Pearson Prentice Hall, 2007.

[12] V. Čačić, *Komputonomikon*, 2021, https://web.math.pmf.unizg.hr/~veky/izr/Komputonomikon.pdf, str. 143–146.

Sažetak

U ovom radu proučavamo kako možemo koristiti ASIO biblioteku za mrežno programiranje. U prvom poglavlju ovog rada govorimo općenito o mrežnom programiranju. Uvodimo terminologiju koju ćemo koristiti u radu. Osim toga opisana je slojevita struktura koja je nezaobilazni dio aplikacije koja koristi mrežno programiranje. U preostalim poglavljima proučavamo klase i funkcije koje nam pruža ASIO biblioteka. Proučavanje započinjemo utičnicama i proučavanjem kako one omogućuju razmjenu podataka. Da bismo mogli razmjenjivati podatke prvo moramo povezati procese. Nakon što smo naučili glavne načine za povezivanje utičnica, obrađujemo metode i funkcije za slanje i primanje poruka. Metode i funkcije dijelimo na sinkrone i asinkrone te objašnjavamo prednosti i mane svakog od tih pristupa. Rad je većinom baziran na TCP transportnom protokolu jer ćemo njega koristiti u završnom poglavlju, tj. u izradi projektnog zadatka. Kako se i UDP protokol često koristi navodimo razlike u kodu s obzirom na to koji se protokol koristi. Za kraj prikazujemo kako sve segmente možemo spojiti u jednu cjelinu. Kao projektni zadatak implementiramo HTTP protokol. Za početak kažemo nešto općenito o protokolu, a zatim implementiramo klijenta i poslužitelja koji komuniciraju koristeći HTTP protokol.

Summary

In this paper we study how we can use the ASIO library for network programming. In the first chapter of this paper, we talk about network programming in general. We introduce the terminology that we will use in the paper. In addition, a layered structure is described, which is an indispensable part of an application that uses network programming. In the remaining chapters, we study the classes and functions provided by the ASIO library. We begin the study with sockets and studying how they enable data exchange. To be able to exchange data, we first need to connect the processes. After learning the main ways to connect sockets, we cover the methods and functions for sending and receiving messages. We divide methods and functions into synchronous and asynchronous and explain the advantages and disadvantages of each of these approaches. The work is mostly based on the TCP transport protocol because we will use it in the final chapter, i.e. in the creation of the project assignment. As the UDP protocol is also often used, we list the differences in the code with regard to which protocol is used. Finally, we show how all segments can be combined into one whole. As a project task, we implement the HTTP protocol. First, we say something general about the protocol, and then we implement a client and server that communicate using the HTTP protocol.

Životopis

Rođen sam 26. lipnja 1997. godine u Zagrebu. Svoje obrazovanje započeo sam u Osnovnoj školi Ksaver Šandor Gjalski u Zagrebu. Osnovnu školu završavam 2012. godine kada upisujem Privatnu klasičnu gimnaziju. U srednjoj školi sudjelovao sam na brojnim natjecanjima iz matematike i fizike. 2016. godine upisujem preddiplomski sveučilišni studij Matematike na Prirodoslovno–matematičkom fakultetu u Zagrebu. Na istom fakultetu 2020. godine upisujem diplomski sveučilišni studij Računarstva i matematike. Za vrijeme diplomskog studija bio sam suautor članka na temu Predviđanje tipa osobnosti koji je objavljen u 40. broju Hrvatskog matematičkog elektroničkog časopisa math.e.