Implementacija konkurentnog chat servera u programskom jeziku Go

Istraživački projekat iz predmeta Dizajn programskih jezika

Jelisaveta Gavrilović 1028/2024

21. oktobar 2025.

Sadržaj

1 Uvod

Razmena poruka u realnom vremenu predstavlja ključnu funkcionalnost savremenih softverskih sistema, od društvenih mreža do poslovnih alata. Sa razvojem mrežnih tehnologija, pojavila se potreba za pouzdanim i efikasnim sistemima koji omogućavaju korisnicima da komuniciraju međusobno, kako u grupama, tako i direktno. Chat serveri pružaju takvu funkcionalnost koristeći mrežne protokole poput TCP-a, koji obezbeđuje pouzdanu i efikasnu komunikaciju između korisnika.

Razvoj chat servera omogućava praktičan uvid u osnovne principe konkurentnog programiranja, sinhronizacije i upravljanja mrežnim konekcijama. Konkurentnost je ključna, jer server mora istovremeno da obrađuje više korisnika bez blokiranja i gubitka poruka. Sinhronizacija podataka, kao što su liste aktivnih korisnika i statusi konekcija, zahteva pažljivo korišćenje mehanizama za zaštitu zajedničkih resursa, poput `mutex`-a, kako bi se sprečile trke za resursima i nepredviđeni konflikti.

U ovom radu implementiran je terminalski chat server korišćenjem programskog jezika Go. Go je posebno pogodan za mrežne aplikacije koje zahtevaju konkurentnost, jer nudi jednostavnu i efikasnu podršku kroz `gorutine` (*goroutines*) i `kanale` (*channels*). Gorutine omogućavaju istovremeno izvršavanje funkcija, dok kanali služe za bezbednu komunikaciju između gorutina. Ovi mehanizmi su direktno primenjeni u implementaciji servera za upravljanje aktivnim korisnicima i distribuciji poruka.

Glavne karakteristike implementiranog sistema uključuju:

- **Terminalska komunikacija:** korisnici komuniciraju direktno kroz terminalski interfejs.
- **Konkurentno programiranje:** `gorutine` omogućavaju istovremenu obradu poruka za svakog korisnika.
- **Sinhronizacija i upravljanje korisnicima:** mapa aktivnih korisnika i kanali se koriste za sigurno deljenje podataka i izbegavanje kolizija.
- **Distribucija poruka:** podržano je slanje poruka svim korisnicima (*broadcast*) ili pojedincima (*privatne poruke*).
- **Mrežna komunikacija:** TCP protokol omogućava sigurno prenošenje podataka između servera i klijenata.

2 Teorijski koncepti

Razumevanje teorijskih osnova predstavlja ključni korak u izgradnji pouzdanih i efikasnih sistema. U ovom poglavlju obrađeni su osnovni koncepti koji čine temelj implementacije chat servera, uključujući konkurentnost, sinhronizaciju i mrežnu komunikaciju. Poseban akcenat stavljen je na način na koji programski jezik Go omogućava realizaciju ovih principa kroz gorutine, kanale i rad sa mrežnim konekcijama putem TCP protokola.

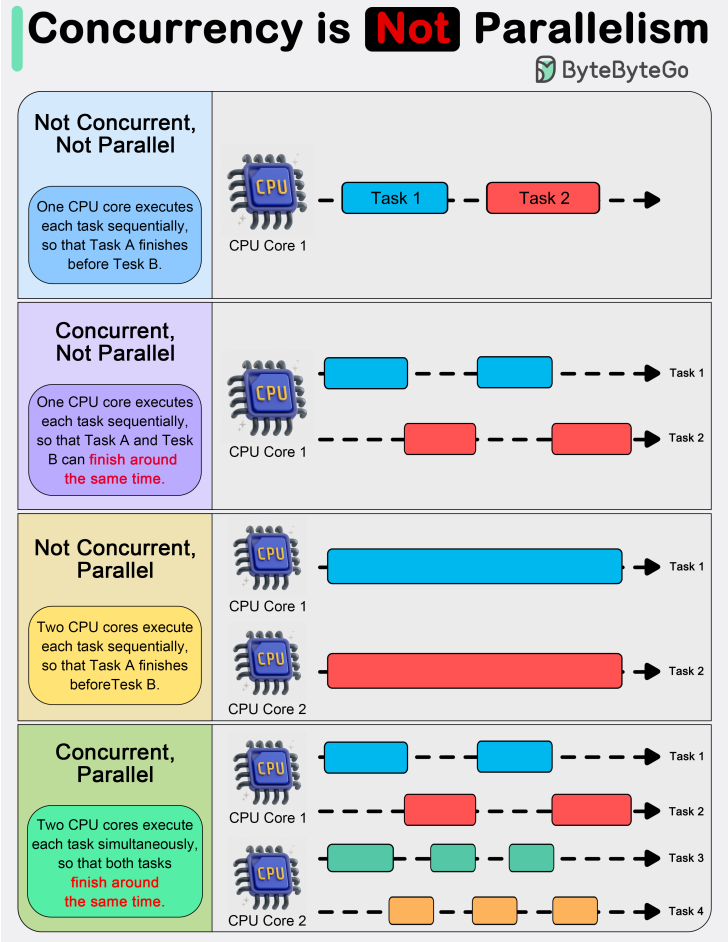
Ovi koncepti čine osnovu savremenih konkurentnih i distribuiranih sistema, omogućavajući istovremeno izvršavanje više zadataka, bezbednu razmenu podataka i stabilnu komunikaciju između klijenata i servera u realnom vremenu.

2.1 Konkurentnost

Konkurentnost (*concurrency*) predstavlja sposobnost programa ili sistema da istovremeno upravlja izvršavanjem više zadataka, čime se postiže efikasnije korišćenje resursa i brža reakcija aplikacije. Umesto da čeka da se jedan zadatak u potpunosti završi pre nego što započne sledeći, konkurentni sistem omogućava da više zadataka napreduje nezavisno, raspoređujući procesorsko vreme između njih. Na taj način, program može ostati odzivan čak i dok pojedini procesi čekaju na spore operacije ulaza/izlaza.

U suštini, konkurentnost je sposobnost programa da „misli“ o više stvari odjednom, dok paralelizam predstavlja sposobnost da „radi“ više stvari u isto vreme. Iako se ovi pojmovi često koriste kao sinonimi, postoji jasna razlika između njih:

- **Konkurentnost (*concurrency*)** omogućava da više zadataka napreduje „istovremeno“, čak i kada se izvršavaju na jednom procesorskom jezgru. Cilj je da sistem ostane odzivan i funkcionalan.
- **Paralelizam (*parallelism*)** označava fizičko istovremeno izvršavanje više zadataka na različitim procesorskim jezgrima. Paralelizam je jedan od načina realizacije konkurentnosti, ali konkurentnost se može postići i bez stvarnog paralelizma.



Slika 1: Razlika između konkurentnosti i paralelizma

2.1.1 Sinhronizacija i komunikacija

U konkurentnim programima, više tokova izvršavanja često mora da sarađuje i razmenjuje informacije. Da bi se izbegle greške i obezbedilo pravilno ponašanje sistema, neophodno je kontrolisati redosled izvršavanja i razmenu podataka. Ova kontrola se ostvaruje kroz:

- **Sinhronizaciju:** koordinacija više tokova izvršavanja kako bi se sprečio neispravan pristup zajedničkim resursima. Bez odgovarajuće sinhronizacije može doći do *race condition* situacija, gde istovremeni pristup menja podatke ili ih čita u nesigurnom trenutku. Tradicionalni mehanizmi uključuju zaključavanja (`mutex`-i), semafore i monitore, koji zahtevaju pažljivo upravljanje resursima i mogu izazvati *deadlocks* ili *livelocks*.

- **Komunikaciju:** razmena informacija između tokova izvršavanja. Model *message passing* podrazumeva eksplicitno slanje podataka između nezavisnih procesa, čime se smanjuje rizik od grešaka i olakšava održavanje konzistentnog stanja.

2.1.2 Konkurentnost u programskom jeziku Go

Go jezik je dizajniran sa posebnim fokusom na jednostavno i efikasno konkurentno programiranje. Dve osnovne konstrukcije koje to omogućavaju su **gorutine** i **kanali**, što čini osnovu modela *Communicating Sequential Processes (CSP)*, gde nezavisni procesi komuniciraju razmenom poruka umesto deljenja memorije.

Gorutine

Gorutine (*goroutines*) su lagane jedinice izvršavanja koje omogućavaju konkurentno izvršavanje funkcija. Pokreću se pomoću ključne reči `go`, a Go runtime automatski upravlja njihovim raspoređivanjem. Za razliku od standardnih OS niti, svaka gorutina zahteva minimalno memorije, što omogućava kreiranje hiljada gorutina u isto vreme.

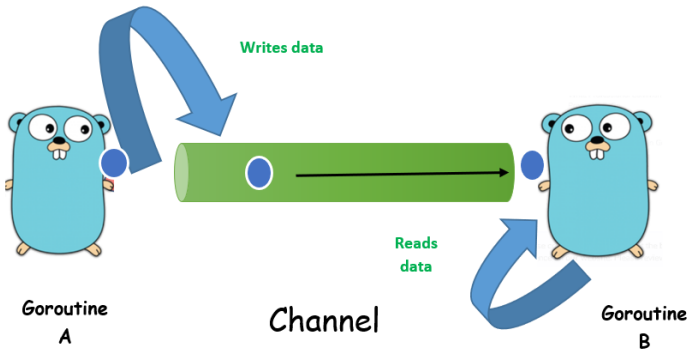
```
go funkcija()
```

Prednosti gorutina:

- Minimalno opterećenje memorije.
- Automatsko planiranje izvršavanja od strane Go runtime-a.
- Jednostavno i čitljivo konkurentno programiranje.

Kanali

Kanali (*channels*) omogućavaju komunikaciju i sinhronizaciju između gorutina. Umesto da gorutine direktno pristupaju deljenim strukturama podataka, informacije se razmenjuju kroz kanale, što eliminiše trke za resursima i složenu ručnu sinhronizaciju.



Slika 2: Komunikacija između gorutina putem kanala

Kanali se definišu pomoću operatora `chan`, a komunikacija se obavlja pomoću operatora strelice (`<-`). Kroz kanal se može slati samo jedan tip podataka.

```
messages := make(chan string)

go func() {
    messages <- "Pozdrav iz gorutine!"
}()

msg := <-messages
fmt.Println(msg)
```

- Slanje poruke u kanal zaustavlja gorutinu koja šalje dok druga gorutina ne primi poruku, dok gorutina koja prima čeka dok poruka ne postane dostupna. Na ovaj način kanali automatski omogućavaju sinhronizaciju gorutina.
- Kanali omogućavaju bezbednu komunikaciju i koordinaciju između gorutina bez eksplicitnog zaključavanja.

Iako kanali pokrivaju većinu potreba za komunikacijom i sinhronizacijom, u praksi postoji potreba za zaštitom deljenih struktura podataka (npr. mapa ili liste). U tim slučajevima koristi se zaključavanje (`mutex`), ali princip ostaje isti: maksimalno se koristi komunikacija putem kanala, dok se zaključavanja primenjuju samo kada je neophodno.

2.2 Mrežna komunikacija

Mrežna komunikacija predstavlja osnovu svih sistema koji omogućavaju razmenu podataka između više uređaja ili korisnika u realnom vremenu. Kod aplikacija kao što su chat serveri, server i klijenti moraju moći da šalju i primaju poruke bez gubitka podataka i u pravilnom redosledu, kako bi korisničko iskustvo bilo konzistentno i pouzdano.

Osnovni koncepti mrežne komunikacije uključuju:

- **klijent-server model:** Server je centralna tačka koja prima konekcije od više klijenata. Klijenti iniciraju konekciju i šalju zahteve serveru. Server obrađuje dolazne poruke i prosleđuje ih drugim korisnicima. Ovaj model omogućava centralizovano upravljanje podacima, kontrolu pristupa i jednostavno skaliranje sistema.
- **socket konekcije:** Socket je apstrakcija koja omogućava komunikaciju između programa preko mreže. Svaka konekcija se identifikuje jedinstvenom kombinacijom IP adrese i porta, što omogućava istovremene višestruke konekcije između servera i različitih klijenata. Socketi predstavljaju osnovni mehanizam za slanje i prijem podataka u mrežnim aplikacijama.
- **protokoli:** Protokoli definišu pravila za razmenu podataka i određuju način na koji uređaji komuniciraju. Najčešće se koriste TCP i UDP transportni protokoli.

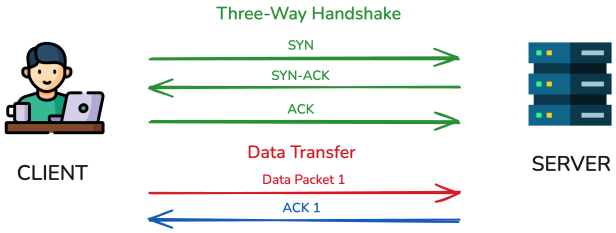
2.2.1 Transportni protokoli

TCP protokol

TCP (*Transmission Control Protocol*) je transportni protokol koji omogućava pouzdanu i kontrolisanu komunikaciju između servera i klijenata. TCP je zasnovan na konekciji, što znači da pre slanja podataka mora biti uspostavljena veza između pošiljaoca i primaoca, čime se omogućava dvosmerna komunikacija u realnom vremenu. Ključne karakteristike TCP-a uključuju:

- **pouzdanost i integritet podataka:** TCP garantuje da će poruke stići na odredište u celosti i u ispravnom redosledu. Ako neki segment poruke ne stigne ili se izgubi, TCP automatski inicira ponovno slanje.
- **veza (*connection-oriented*):** TCP uspostavlja sesiju između klijenta i servera pre prenosa podataka. U okviru ove veze, podaci mogu da teku u oba smera, što omogućava serveru da šalje poruke klijentu i obrnuto.

- **kontrola toka i zagušenja:** TCP dinamički reguliše brzinu prenosa podataka kako bi se izbeglo preopterećenje mreže ili gubitak poruka.
- **segmentacija podataka (*packetization*):** Velike poruke se dele na manje segmente koji se zasebno šalju. Na odredištu, TCP rekonstruiše originalnu poruku u ispravnom redosledu.
- **detekcija grešaka:** TCP koristi kontrolne sume za detekciju oštećenih paketa. Ako je paket oštećen tokom prenosa, on se odbacuje, a protokol inicira ponovno slanje.

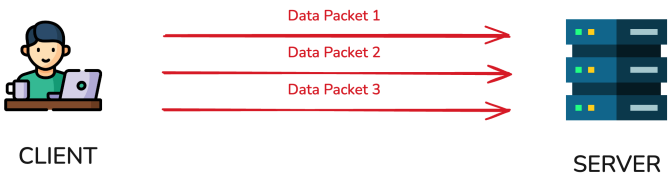


Slika 3: Tok TCP komunikacije između klijenta i servera

UDP protokol

UDP (*User Datagram Protocol*) je alternativni transportni protokol koji, za razliku od TCP-a, ne uspostavlja vezu i ne garantuje pouzdanost prenosa podataka. Ključne karakteristike UDP-a su:

- **bez veze (*connectionless*):** Ne postoji uspostavljena sesija između pošiljaoca i primaoca, što smanjuje kašnjenje i pojednostavljuje protokol.
- **nepouzdanost:** Poruke (*datagrami*) mogu stići izvan redosleda ili biti izgubljene, a protokol ne inicira ponovno slanje.
- **jednostavnost i brzina:** Zbog odsustva kontrole toka i potvrda, UDP je brži i manje zahteva resursa od TCP-a.



Slika 4: Tok UDP komunikacije između klijenta i servera

UDP je koristan za aplikacije gde je brzina bitnija od pouzdanosti, kao što su strimovanje video zapisa ili online igre. Za chat servere, gde je bitno da poruke stignu u ispravnom redosledu i bez gubitaka, TCP je pogodniji izbor.

2.2.2 TCP komunikacija u programskom jeziku Go

Go jezik ima ugrađenu podršku za mrežnu komunikaciju kroz standardnu biblioteku `net`, koja omogućava jednostavno kreiranje TCP servera i klijenata. Go apstrahuje detalje sa niskih nivoa mrežne komunikacije i omogućava programeru da se fokusira na logiku aplikacije, a ne na implementaciju protokola. Zahvaljujući svojoj konkurentnoj prirodi, Go omogućava da svaka mrežna konekcija bude obrađena u zasebnoj gorutini, čime se ostvaruje visoka efikasnost i skalabilnost.

TCP server

Server u Go-u koristi funkciju `net.Listen` za otvaranje porta i čekanje dolaznih konekcija. Kada se klijent poveže, server prihvata konekciju i prosleđuje je funkciji koja obrađuje komunikaciju sa tim klijentom. Svaka nova konekcija se obrađuje u zasebnoj gorutini, čime se omogućava istovremena komunikacija sa više klijenata bez blokiranja glavnog toka programa.

```
// server kreira listener koji "osluškuje"
// TCP konekcije na portu 8080
listener, err := net.Listen("tcp", ":8080")
if err != nil {
    log.Fatal(err)
}
// osigurava da se port zatvori nakon završetka programa
defer listener.Close()

for {
    // prihvata nove konekcije
    conn, _ := listener.Accept()

    // svaka konekcija se obrađuje u posebnoj gorutini
    go handleConnection(conn)
}
```

Ovaj pristup koristi konkurentni model Go-a — svaka konekcija ima svoju gorutinu, što omogućava paralelno procesiranje poruka bez blokiranja glavnog toka programa. U praksi, server može istovremeno rukovati sa puno klijenata

uz minimalno korišćenje resursa.

TCP klijent

Klijent koristi funkciju `net.Dial` za povezivanje sa serverom. Nakon uspostavljanja veze, podaci se mogu slati i primati putem `Read` i `Write` metoda nad konekcijom. Ova konekcija funkcioniše kao dvosmerni tok bajtova (*bidirectional stream*).

```
// klijent se povezuje na server na adresi localhost:8080
conn, err := net.Dial("tcp", "localhost:8080")
if err != nil {
    log.Fatal(err)
}
// zatvara konekciju kada se program završi
defer conn.Close()

// slanje poruke serveru
conn.Write([]byte("Pozdrav serveru!"))

// čitanje odgovora sa servera
message := make([]byte, 1024)
n, _ := conn.Read(message)
```

Ovaj model omogućava dvosmernu komunikaciju: klijent može da šalje poruke serveru, a istovremeno da prima podatke u posebnoj gorutini, bez čekanja da se prethodne operacije završe.

Prednosti Go implementacije TCP komunikacije

- **Jednostavna i čitljiva implementacija konkurentnog mrežnog koda:** osnovne operacije (slušanje, prihvatanje konekcija, slanje i prijem podataka) implementiraju se sa svega nekoliko linija koda.
- **Automatsko upravljanje gorutinama i sinhronizacija putem kanala:** Go runtime samostalno upravlja raspoređivanjem gorutina, što pojednostavljuje implementaciju.
- **Efikasno rukovanje velikim brojem istovremenih konekcija:** zbog malog memorijskog zauzeća gorutina, Go serveri mogu lako opslužiti hiljade klijenata istovremeno.
- **Ugrađena podrška u standardnoj biblioteci:** sve funkcionalnosti TCP protokola dostupne su bez potrebe za eksternim paketima.

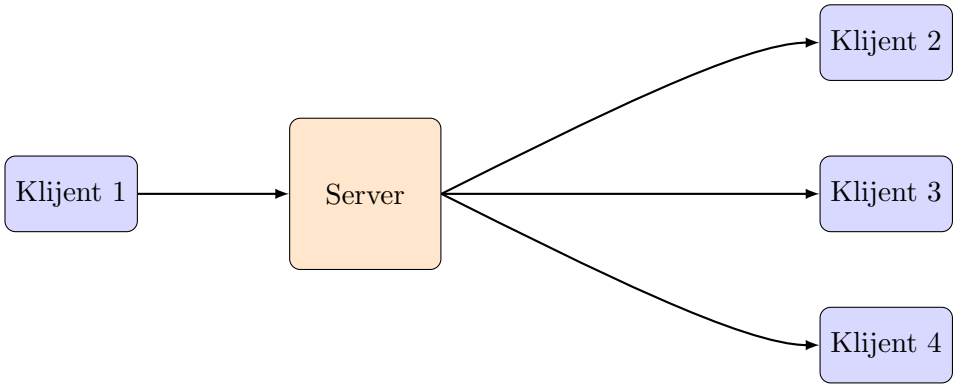
Kombinacija `gorutina`, `kanala` i `TCP protokola` čini Go pogodnim za razvoj mrežnih aplikacija kao što su chat serveri, HTTP servisi, distribuirani sistemi i real-time aplikacije. Go-ov pristup mrežnoj komunikaciji predstavlja balans između jednostavnosti, efikasnosti i konkurentnosti, što ga čini savršenim izborom za praktične primene poput ove.

3 Implementacija chat servera

U ovom poglavlju biće predstavljena praktična realizacija chat sistema razvijenog u programskom jeziku Go. Sistem se sastoji iz dva osnovna dela: **servera** i **klijenta**. Server je zadužen za prihvatanje konekcija i distribuciju poruka, dok klijentska aplikacija omogućava korisniku da šalje i prima poruke putem jednostavnog tekstualnog interfejsa.

3.1 Arhitektura sistema

Aplikacija je zasnovana na arhitekturi **klijent–server**. Server otvara TCP port i osluškuje dolazne konekcije, dok se svaki klijent povezuje na taj port i registruje se unosom jedinstvenog korisničkog imena. Nakon uspostavljene veze, komunikacija između klijenta i servera odvija se pomoću jednostavnih tekstualnih poruka.



Slika 5: Arhitektura sistema

Na slici 5 prikazan je logički tok komunikacije između servera i više klijenata. Svaki klijent komunicira isključivo sa serverom, koji obavlja ulogu posrednika i prosleđuje poruke dalje odgovarajućim korisnicima.

3.2 Implementacija servera

Server je implementiran koristeći gorutine za paralelnu obradu svakog klijenta i kanale za centralizovano prosleđivanje poruka, čime se omogućava da veliki broj korisnika istovremeno komunicira bez blokiranja glavnog toka servera.

Svaki klijent koji se poveže na server obrađuje se u posebnoj gorutini, što znači da server može istovremeno prihvatati nove konekcije i obrađivati postojeće korisnike bez čekanja. Ovaj pristup direktno primenjuje teorijski princip konkurentnosti, omogućavajući višestruke istovremene tokove izvršavanja unutar jednog procesa.

```go
func main() {
    // start listening on TCP port 8080
    listener, err := net.Listen("tcp", ":8080")
    if err != nil {
        fmt.Println("Error starting server:", err)
        return
    }
    // ensure listener is closed when main exits
    defer listener.Close()

    fmt.Println("Server started on port 8080")

    // starts a goroutine that continuously listens on the 'messages' channel
    // and forwards messages to the appropriate clients
    go dispatcher()

    // accept incoming connections
    for {
        conn, err := listener.Accept()
        if err != nil {
            fmt.Println("Error accepting connection:", err)
            continue
        }
        // handle each client in a separate goroutine
        go handleClient(conn)
    }
}
```

Slika 6: Server prihvata nove TCP konekcije i pokreće gorutinu za svakog klijenta

Funkcija `handleClient` čita poruke koje dolaze od klijenta i prosleđuje ih centralnom kanalu poruka `messages`. Kanal koristi `dispatcher` gorutina, koja kontinuirano osluškuje kanal i prosleđuje sve pristigle poruke ostalim korisnicima — svima ukoliko se radi o broadcast porukama ili pojedincima ukoliko su privatne poruke.

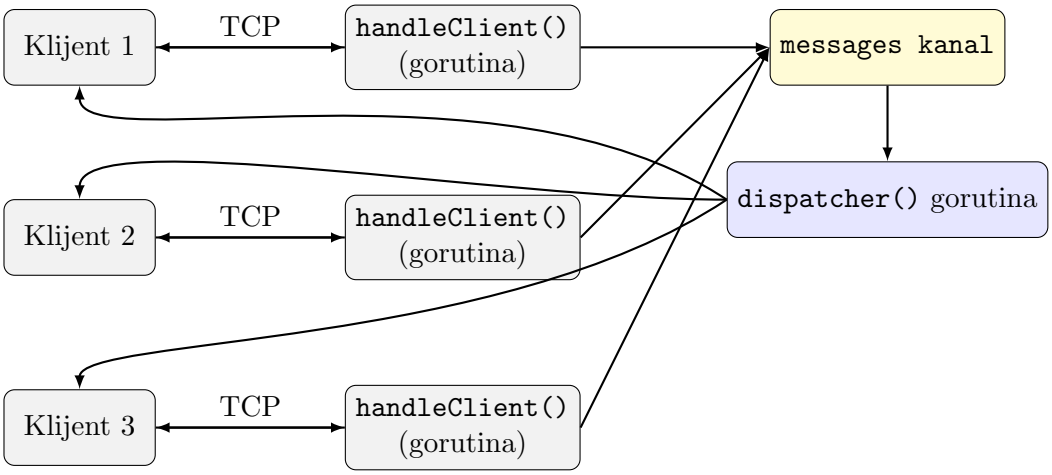
```go
// dispatcher listens on messages channel and sends message
func dispatcher() {
    for {
        msg := <- messages
        clientsMux.Lock()
        if msg.To == "" {
            // broadcast
            for _, c := range clients {
                c.Conn.Write([]byte(fmt.Sprintf("%s: %s\n", msg.From, msg.Content)))
            }
        } else {
            // private or system message
            if target, ok := clients[msg.To]; ok {
                target.Conn.Write([]byte(fmt.Sprintf("%s: %s\n", msg.From, msg.Content)))
            }
        }
        clientsMux.Unlock()
    }
}

// helper function for sending message in channel
func SendMessage(from, to, content string) {
    messages <- Message{
        From:    from,
        To:      to,
        Content: content,
    }
}
```

Slika 7: Dispatcher gorutina koja osluškuje kanal 'messages' i prosleđuje poruke odgovarajućim klijentima

Ovim pristupom se postiže da samo jedna gorutina upravlja slanjem poruka, čime se izbegavaju konflikti i trke u pristupu mrežnim resursima.

Iako Go pruža napredne mehanizme konkurentnosti putem kanala, u ovoj implementaciji je dodatno korišćen mutex za zaštitu zajedničke strukture podataka — mape `clients`. Kanali omogućavaju sigurno prosleđivanje poruka, ali ne garantuju bezbedan pristup deljenim strukturama; mutex se koristi da bi se obezbedio ekskluzivan pristup mapi prilikom dodavanja ili uklanjanja korisnika.

Ovaj pristup je praktičniji od čistog Go-ovog modela u kojem bi samo jedna gorutina pristupala mapi klijenata i primala zahteve kroz kanal. Takva implementacija bi zahtevala dodatnu logiku za proveru registracije korisnika, dodavanje i uklanjanje iz mape, dok bi paralelna obrada poruka mogla značajno povećati složenost i vreme čekanja, smanjujući odziv sistema i kvalitet korisničkog iskustva. Korišćenje mutex-a omogućava jednostavniju, bržu i sigurnu implementaciju, bez kompromisa u konkurentnosti i skalabilnosti servera.



Slika 8: Arhitektura sistema sa tri klijenta

3.3 Implementacija klijenta

Klijent je dizajniran da istovremeno šalje poruke korisnika i prima dolazne poruke od servera. Ovo se postiže korišćenjem gorutina, što omogućava da obe funkcionalnosti rade istovremeno bez blokiranja, čime se postiže glatka i brza komunikacija u realnom vremenu. Ako bi klijent radio samo jednu od ovih stvari u datom trenutku (npr. prvo čitao poruke, a tek onda slao), korisničko iskustvo bi bilo loše – poruke bi stizale sa zakašnjenjem ili bi se aplikacija "zamrzla" dok korisnik ne unese poruku.

Ovaj princip je posebno važan u real-time aplikacijama poput chat sistema, jer kašnjenje u prijemu ili slanju poruka direktno utiče na interakciju između korisnika, što može dovesti do konfuzije ili frustracije. Paralelna obrada omogućava klijentu da ostane odzivan, čime se obezbeđuje tečna i prirodna komunikacija, što je ključni zahtev modernih mrežnih aplikacija.

Po pokretanju, klijent se povezuje na server koristeći funkciju `net.Dial`, koja uspostavlja TCP konekciju na definisanoj adresi i portu. Nakon uspostavljanja veze, korisnik bira svoje ime.

```go
func main() {
    // connect to the server
    conn, err := net.Dial("tcp", "localhost:8080")
    if err != nil {
        log.Fatal("Error connecting to server:", err)
    }
    // ensure connection closes when program exits
    defer conn.Close()

    name := askName(conn)
    setMyName(name)

    activeUsers := []string{} // local list of active users
    app := tview.NewApplication() // create a new TUI application

    // create message view and input field
    messageView, input := NewChatUI(app, conn, &activeUsers)
    // layout
    flex := tview.NewFlex().
        SetDirection(tview.FlexRow).
        AddItem(messageView, 0, 1, false).
        AddItem(input, 1, 0, true)

    // start goroutine to handle incoming messages
    go clientReader(conn, app, messageView, input, &activeUsers)

    // run the TUI application
    if err := app.SetRoot(flex, true).Run(); err != nil {
        log.Fatal(err)
    }
}
```

Slika 9: Inicijalizacija klijenta

Funkcija `askName` omogućava unos korisničkog imena i proveru da li je već zauzeto. Provera se vrši kroz dvosmernu komunikaciju sa serverom – ukoliko je ime zauzeto, korisnik dobija povratnu informaciju i može uneti novo.

```go
func askName(conn net.Conn) string {
    var name string
    reader = bufio.NewReader(conn)

    for {
        name = ""
        fmt.Print("Enter your username: ")
        fmt.Scanln(&name)
        conn.Write([]byte(name + "\n"))

        // read server response
        response, _ := reader.ReadString('\n')
        response = strings.TrimSpace(response)

        if response == "NAME_ACCEPTED" {
            return name
        } else if response == "NAME_TAKEN" {
            fmt.Println("Username is already taken, please choose another one.")
        } else if response == "NOT_NAME" {
            fmt.Println("Please choose a username.")
        } else {
            fmt.Println("Unexpected server response.")
        }
    }
}
```

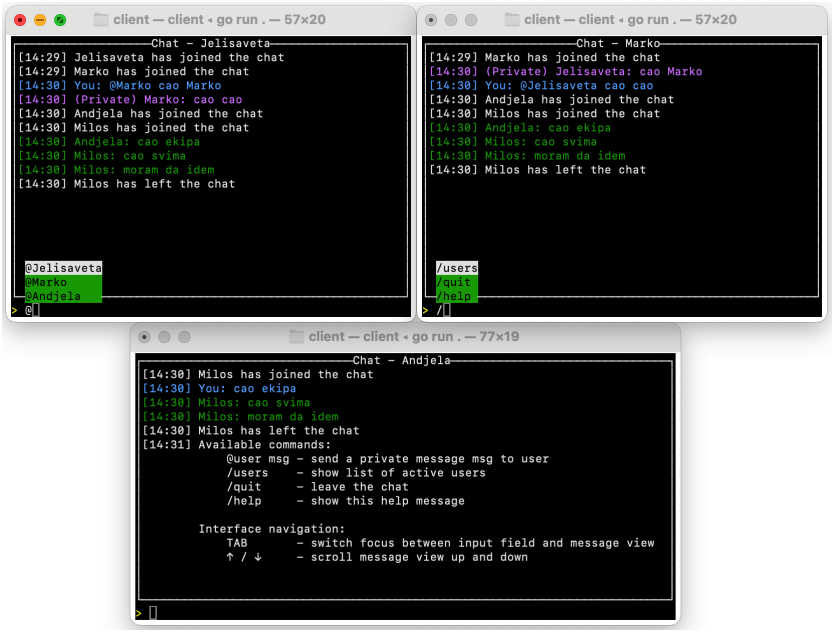
Slika 10: Provera i izbor korisničkog imena kroz TCP komunikaciju sa serverom

Za čitanje poruka sa servera koristi se konstrukcija:

`reader := bufio.NewReader(conn)`

`bufio.NewReader` kreira buffered reader koji obavija TCP konekciju (conn). Za razliku od osnovnog `Read` poziva, koji čita bajtove direktno sa konekcije, buffered reader omogućava efikasnije čitanje po linijama ili blokovima podataka. Konkretno, metoda `reader.ReadString('\n')` čita celu poruku do prvog newline karaktera, što olakšava parsiranje tekstualnih protokola kao što je chat, i smanjuje broj poziva ka TCP sloju, poboljšavajući performanse pri većem broju poruka.

Terminalski interfejs klijenta je kreiran korišćenjem biblioteke `tview`, koja omogućava dinamički prikaz poruka i interaktivni unos. Funkcija `NewChatUI` kombinuje `TextView` za prikaz poruka i `InputField` za unos novih.



Slika 11: Interfejs klijenata

Klijent omogućava korisniku da unosi i obične poruke i posebne komande (kao što su `/quit`, `/users`, `/help`) direktno u polje za unos teksta (InputField). Privatne poruke šalju se u obliku `@user msg`.

Da bi klijent mogao istovremeno da prima poruke i šalje svoje, koristi se gorutina `clientReader`, koja u pozadini osluškuje dolazne poruke sa servera i ažurira terminalski interfejs u realnom vremenu. Glavni tok klijenta ostaje slobodan za unos novih poruka, čime se omogućava neprekidna i neometana interakcija.

```go
// clientReader reads messages from the server and updates the UI
func clientReader(conn net.Conn, app *tview.Application, view *tview.TextView, input *tview.InputField, activeUsers *[]string) {
    for {
        msg, err := reader.ReadString('\n')
        if err != nil {
            app.QueueUpdateDraw(func() {
                fmt.Fprintf(view, "[red]Disconnected from server.[-]\n")
                input.SetDisabled(true)
            })
            return
        }

        msg = strings.TrimSpace(msg)
        if msg == "" {
            continue
        }

        app.QueueUpdateDraw(func() {
            colonIdx := strings.Index(msg, ":")
            if colonIdx == -1 {
                AppendSystemMessage(view, "Unexpected server response.")
                return
            }

            from := strings.TrimSpace(msg[:colonIdx])
            content := strings.TrimSpace(msg[colonIdx+1:])

            // system messages
            if from == "System" {
                // existing users - list of users already logged in
                if strings.HasPrefix(content, "Active users: ") {
                    users := strings.TrimPrefix(content, "Active users: ")
                    *activeUsers = strings.Split(users, ", ")
                    return
                }

                // user joined
                if strings.HasSuffix(content, "has joined the chat") {
                    username := strings.TrimSuffix(content, " has joined the chat")
                    *activeUsers = appendUser(*activeUsers, username)
                    AppendSystemMessage(view, content)
                    return
                }

                // user left
                if strings.HasSuffix(content, "has left the chat") {
                    username := strings.TrimSuffix(content, " has left the chat")
                    *activeUsers = removeUser(*activeUsers, username)
                    AppendSystemMessage(view, content)
                    return
                }

                // user not found
                if strings.HasSuffix(content, "User not found") {
                    AppendSystemMessage(view, content)
                    return
                }

            }

            if from == myName {
                return
            }

            // display messages
            if strings.HasPrefix(content, "[Private]") {
                AppendMessage(view, content, false, true)
            } else {
                AppendMessage(view, content, false, false)
            }
        })
    }
}
```

Slika 12: Gorutina clientReader kontinuirano osluškuje dolazne poruke i ažurira interfejs

Pošto je terminalski interfejs deljeni resurs, direktno ažuriranje iz `clientReader` gorutine može dovesti do trka i nepredvidivog ponašanja. Da bi se to izbeglo, koristi se:

```
app.QueueUpdateDraw(func() {
// kod koji ažurira interfejs
})
```

Funkcija `QueueUpdateDraw` stavlja funkciju u red za ažuriranje interfejsa, koji `tview` izvršava u glavnoj gorutini. Na ovaj način se postiže:

- sigurno ažuriranje interfejsa iz više gorutina,
- izbegavanje race condition-a između gorutina koje čitaju poruke i one koja crta UI,
- glatko osvežavanje ekrana bez zamrzavanja ili preskakanja poruka.

Slanje poruka realizuje se funkcijom `SendMessage`, koja jednostavno šalje tekstualnu poruku serveru preko TCP konekcije.

Kombinacija gorutina, kanala i TCP konekcije u klijentu omogućava efikasnu konkurentnu komunikaciju, sinhronizovano primanje i slanje poruka, i stabilan rad čak i kada veliki broj korisnika istovremeno šalje podatke serveru.

3.4 Instalacija i pokretanje

Za pokretanje ovog projekta potrebno je da korisnik ima instaliran Go kompajler (verzija 1.20 ili novija) i internet konekciju za preuzimanje zavisnosti.

Instalacija Go okruženja

Go se preuzima sa zvanične stranice: https://go.dev/dl/ ili pokretanjem komande u terminalu:

```
sudo apt install golang-go
```

za Linux operativni sistem, odnosno

```
brew install go
```

za macOS.

Nakon instalacije, proveriti verziju naredbom:

```
go version
```

Preuzimanje projekta

Ceo projekat se nalazi na GitHub-u na linku.

Projekat se može preuzeti, pokretanjem komande iz terminala:

```
git clone https://github.com/jelisavetagavrilovic/mini-chat-server
```

Instalacija zavisnosti

Sve zavisnosti su definisane u `go.mod` i `go.sum` fajlovima. Prilikom prvog pokretanja Go automatski preuzima sve potrebne biblioteke. Alternativno, moguće je ručno instalirati:

```
go get github.com/rivo/tview
go get github.com/gdamore/tcell/v2
```

Go-ov `go.mod` i `go.sum` osiguravaju da svi korisnici projekta koriste iste verzije biblioteka, čime se izbegavaju problemi sa kompatibilnošću. Celokupna instalacija i pokretanje projekta mogu se izvršiti kroz terminal, što olakšava korišćenje na različitim operativnim sistemima.

Pokretanje servera i klijenta

U terminalu, iz foldera projekta, pokreću se server i klijenti:

pokretanje servera

```
go run server/main.go
```

pokretanje klijenta

```
go run client/main.go
```

Server mora biti pokrenut pre klijenata, jer se klijenti povezuju na TCP port definisan u serveru. Nakon pokretanja, klijent bira korisničko ime i može komunicirati sa ostalim korisnicima u realnom vremenu.