SVEUČILIŠTE U SPLITU
FAKULTET ELEKTROTEHNIKE, STROJARSTVA I BRODOGRADNJE

SEMINARSKI RAD

gRPC protokol i razvoj jednostavne aplikacije za chat

IVAN LAZARUŠIĆ

Split, svibanj 2025.

IZJAVA

Ovom izjavom potvrđujem da sam seminarski rad s naslovom „**gRPC protokol i razvoj jednostavne aplikacije za chat**“ pod mentorstvom prof. dr. sc. Ljiljane Šerić pisao samostalno, primijenivši znanja i vještine stečene tijekom studiranja na Fakultetu elektrotehnike, strojarstva i brodogradnje, kao i metodologiju znanstveno-istraživačkog rada te uz korištenje literature koja je navedena u radu. Spoznaje, stavove, zaključke, teorije i zakonitosti drugih autora koje sam izravno ili parafrazirajući naveo u završnom radu citirao sam i povezao s korištenim bibliografskim jedinicama.

Student

Ivan Lazarušić

SADRŽAJ

1 UVOD

U današnje moderno doba, kada su aplikacije sve više povezane i rade na različitim uređajima i platformama, važno je imati efikasan način komunikacije među njima. Tradicionalni pristupi poput REST-a često nisu dovoljno brzi ili fleksibilni za određene moderne potrebe, pogotovo kada se radi o real-time aplikacijama. Zbog toga je razvijen **gRPC**, moćan i brz komunikacijski protokol koji omogućuje jednostavnu razmjenu podataka između klijenata i servera.

Cilj ovog rada je objasniti što je gRPC, kako funkcionira i koje su njegove prednosti u odnosu na druge tehnologije. Osim teorijskog dijela, rad uključuje i praktičnu implementaciju u obliku jednostavne chat aplikacije koja koristi gRPC za slanje i primanje poruka u stvarnom vremenu. Kroz ovaj primjer, demonstrirat će se kako se gRPC može koristiti u praksi te koliko je jednostavan za integraciju i upotrebu.

U nastavku će biti objašnjeni osnovni pojmovi gRPC-a, njegova arhitektura i način rada, nakon čega slijedi osnovni prikaz implementacije chat aplikacije i njene komunikacije između klijenta i servera.

2 TEORIJSKI DIO

2.1 Što je gRPC?

gRPC (Google Remote Procedure Call) je okvir koji je razvio Google kako bi omogućio učinkovitu komunikaciju između različitih usluga. Osnova gRPC-a je HTTP/2 protokol, na čijim je temeljima izgrađen. Korištenje HTTP/2 protokola je važno jer nam omogućuje prijenos različitih poruka preko samo jedne veze. Ako to usporedimo s korištenjem klasičnog REST API-ja, možemo vidjeti da korištenje HTTP/2 protokola značajno poboljšava performanse i smanjuje kašnjenje.

Osnovna ideja iza gRPC-a je omogućiti programerima pozivanje funkcija na udaljenom poslužitelju kao da su lokalne. Ovo načelo može pojednostaviti razvoj distribuiranih sustava sakrivanjem složenosti mrežne komunikacije.

Još jedna ključna značajka gRPC-a je njegova sposobnost rada s međuspremnicima protokola (koji se nazivaju protobuf).

Protobuf može definirati strukture podataka na dobro strukturiran način, te je vrlo učinkovit u serijalizaciji i deserijalizaciji podataka. To rezultira manjim prijenosom podataka, što omogućuje bržu komunikaciju i smanjuje potrebu za propusnošću mreže.

Zahvaljujući podršci za višestruke programske jezike, gRPC je vrlo fleksibilan i lako se integrira u različite tehnološke okoline. Osim toga, nudi različite modele komunikacije, od jednostavnih sinkronih poziva do kompleksnih streaming rješenja, što ga čini pogodnim za širok raspon primjena, od mikroservisnih arhitektura do real-time aplikacija poput chat sustava (što se i razrađuje u drugom dijelu ovog rada).

Ukratko, gRPC predstavlja moderan pristup razvoju distribuiranih sustava, pružajući brzu, učinkovitu i skalabilnu komunikaciju koja je prilagođena zahtjevima današnjih aplikacija.

Mnoge velike i poznate tvrtke koriste gRPC u svojim sustavima, iako se često radi o internim implementacijama unutar mikroservisnih arhitektura, a ne unutar krajnjih aplikacija koje su vidljive korisnicima. Primjeri uključuju:

- **Google:** Kao kreator gRPC-a, Google koristi ovaj protokol u mnogim svojim internim servisima i na Google Cloud platformi. [1]

- **Netflix:** Netflix se oslanja na gRPC za brzu i pouzdanu komunikaciju između mikroservisa. [2]
- **Lyft i Uber:** Obje ove tvrtke koriste gRPC kako bi osigurale učinkovitu komunikaciju unutar svojih sustava za naručivanje prijevoza. [3]
- **Dropbox i Square:** Također su primijenili gRPC u određenim dijelovima svojih sustava, koristeći ga za optimizaciju performansi i smanjenje latencije. [4]

Slika 1 Uber - jedna od kompanija koja koristit gRPC

2.2 Protocol Buffers

U prvom smo poglavlju predstavili osnovnu zamisao Protocol Buffers-a, no nismo detaljno razradili na koji način on funkcionira i koje su mu prednosti,a koji nedostatci.

Protobuf je format za serijalizaciju podataka koji je razvijen od strane Google-a. Razvijen je zbog potrebe za učinkovitom i brzom razmjenom podataka između različitih sustava. On je zapravo **temelj** na kojem je izgrađen cijeli gRPC.

Za razliku od XML i JSON formata, Protobuf je binarni format što mu omogućuje brži prijenos podataka jer podaci zauzimaju manje prostora. U XML-u i JSON-u, podaci se

spremaju kao čitljivi znakovi a to znači da se svaki znak treba zapisati kao ASCII ili UTF-8 vrijednost. Takav način zapisivanja znakova zauzima više prostora nego direktno spremanje podataka u binarnom obliku kako to vrši Protobuf. Zbog toga je Protobuf zapravo savršen kandidat za aplikacije koje zahtijevaju visoke performanse i nisku latenciju (kao što je aplikacija za chat).

Protobuf koristi posebne datoteke s ekstenzijom *.proto* kako bi definirao strukture podataka. Unutar tih datoteka definiraju se poruke koje sadrže polja sa određenim tipovima podataka. Sličnog su oblika kao i strukture (ili klase) u poznatim programskim jezicima. Na temelju tih poruka u *.proto* datoteci može se generirati bazni kod za razne programske jezike. Ova značajka je odlična zbog toga što omogućuje interoperabilnost među aplikacijama koje su razvijene koristeći drugačije tehnologije.

Na slici (Slika 2) može se vidjeti kako izgleda definicija poruke i kako ta definicija uistinu podsjeća na strukture ili klase u programskim jezicima. U ovom primjeru također vidimo da je svakom polju unutar poruke pridružen i broj koji označava redoslijed i jedinstvenost tih polja unutar poruke. Takvi brojevi su jako važni za proces serijalizacije.

gRPC ne koristi Protobuf samo kako bi definirao podatke, već i kako bi definirao usluge i metode koje se mogu pozivati između klijenta i servera. To znači da *.proto* datoteka opisuje kako komponente međusobno komuniciraju. U njoj možemo vidjeti **podatke** koji se razmjenjuju između komponenti, ali također su vidljive **i funkcije** koje se mogu pozvati.

primjer.proto

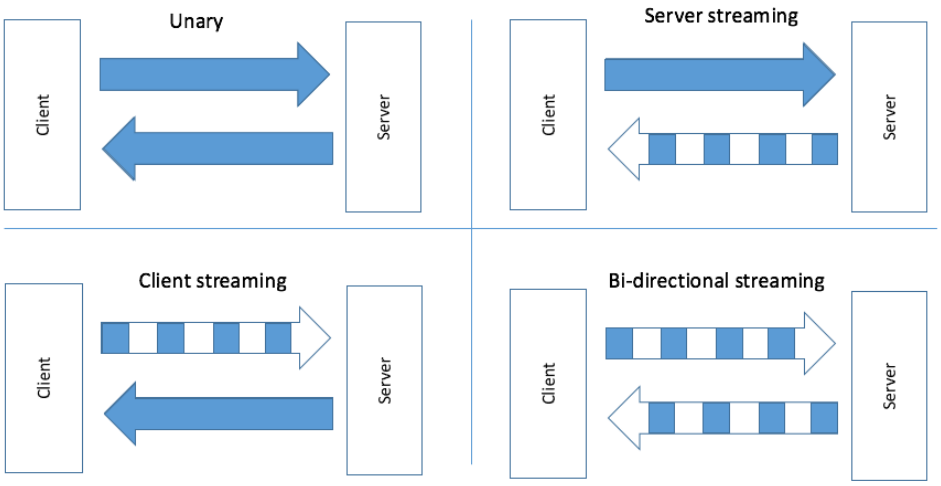
```
1
2   // primjer.proto
3   // Komentari se pisu isto kao i u C-u
4
5   syntax = "proto3";
6
7   message ChatMessage {
8     string username = 1;
9     string message = 2;
10    int64 timestamp = 3;
11  }
```

Slika 2 Izgled jednostavne proto datoteke

2.3 Tipovi komunikacije u gRPC-u

Velika prednost koju gRPC ima naspram nekih drugih često korištenih protokola je ta što gRPC nudi podršku za više tipova komunikacije između servera i klijenta. Standardni zahtjevi su najčešće sinkrone prirode, no osim što gRPC podržava takve zahtjeve, on također podržava i naprednije tipove komunikacije koji nam daju mogućnost slanja podataka u stvarnom vremenu putem streaminga. Definirana su četiri osnovna tipa komunikacije u gRPC-u [5]:

1. **Unary RPC:** Jako sličan REST pristupu. Kada klijent šalje zahtjev serveru, on mu vraća jedan odgovor. Ako nam u aplikaciji ne treba nešto kompleksno, onda je ovakav tip komunikacije idealan.
2. **Server-side streaming RPC:** Kompliciraniji od prvoga. Kada klijent pošalje zahtjev serveru, server mu vraća niz odgovora putem streama. Kako odgovori postaju dostupni, tako ih i klijent može čitati. Idealan je kad server klijentu treba postupno isporučiti više podataka.
3. **Client-side streaming RPC:** Obrnuto od drugoga. Klijent šalje niz poruka serveru, a server vraća jedan odgovor kada primi sve podatke od klijenta. Može se koristiti u slučajevima kada klijent želi poslati veliku količinu podataka, a da pritom ne mora čekati odgovor nakon svakog pojedinačnog slanja.
4. **Bidirectional straming RPC:** Najnapredniji oblik komunikacije. Klijent i server mogu istovremeno i neovisno slati poruke jedan drugome. Ovakav model je koristan za aplikacije koje zahtjevaju dvosmjernu komunikaciju u stvarnom vremenu (Chat aplikacija ili online igrica).



Slika 3 Grafički prikaz tipova komunikacije u gRPC-u

2.4 Nedostatci gRPC-a

U dosadašnjem dijelu rada navodili smo samo prednosti gRPC-a. Jako je važno znati u kojim slučajevima se gRPC isplati koristiti, a u kojima ne. Bitno poznavati i nedostatke tako da možemo zaključiti kada gRPC nije adekvatan kao tehničko rješenje nekog projekta ili zahtjeva.

gRPC koristi HTTP/2 i binarni Protobuf, što **nije nativno podržano u web preglednicima**. Ako ga hoćemo koristiti u frontend aplikacijama koje koriste web preglednike, potreban je dodatni sloj poznat pod nazivom gRPC-web. Zbog ovoga imamo više posla u izradi aplikacije nego u slučaju kada bi koristili REST pristup.

Protobuf je binarni format i nije lako čitljiv ljudima. Ručno testiranje aplikacije zbog ovoga može biti otežano. Nekad nam je primamljivije koristiti JSON koji se vrlo lagano može pregledati u tekstualnom obliku bez ikakvih dodatnih alata.

Velika većina developera upoznata je sa načinom funkcioniranja REST API-ja. S njim su se susretali u brojnim kolegijima na svojim fakultetima. To definitivno nije slučaj za gRPC. Rad sa gRPC-om iziskuje **ulaganje dodatnih napora za edukaciju** korištenja gRPC-a. Kada bi koristili neki tradicionalniji pristup, gotovo sigurno bi potrošili manje novaca i vremena.

3 IMPLEMENTACIJA CHAT APLIKACIJE

3.1 Opis aplikacije

Aplikacija razvijena u kontekstu ovog seminarskog rada je jednostavna chat aplikacija koja se temelji na gRPC tehnologiji. Jezik u kojemu je razvijena je Python. Razlog odabira aplikacije za chat je taj što svaki chat sustav mora omogućiti dvosmjernu komunikaciju između dva ili više korisnika u stvarnom vremenu, a gRPC ima idealno rješenje za ostvarivanje ovakve komunikacije koristeći **bi-directional streaming**.

Aplikacija se sastoji od idućih funkcionalnosti:

- Slanje i primanje poruka u stvarnom vremenu
- Više klijenata može komunicirati istovremeno
- Jednostavno grafičko sučelje izrađeno u Tkinteru
- Prikaz poruka s vremenskom oznakom

Aplikacijska pozadina se sastoji od nekoliko ključnih datoteka:

- **chat.proto** – Definira gRPC servis kroz ChatService i poruku ChatMessage koja sadrži korisničko ime i sadržaj poruke. Ova datoteka je važna jer se koristi za automatsko generiranje Python klasa putem protoc kompajlera.

- **chat_server.py** – Implementira gRPC server koji prihvaća veze više klijenata. Svakom klijentu se dodjeljuje vlastiti red poruka (queue), a poruke se prosljeđuju svim ostalim aktivnim klijentima.

- **chat_client.py** – Predstavlja klijentsku aplikaciju s grafičkim sučeljem (GUI) razvijene koristeći Tkinter biblioteku. Svaki korisnik ima mogućnost unošenja svog imena, slanja poruka i primanja poruka od ostalih korisnika. Komunikacija se odvija putem gRPC streama.

- **chat_pb2.py i chat_pb2_grpc.py** – Automatski generirane datoteke iz *chat.proto* datoteke. Sadrže definicije klasa i dijelova potrebnih za komunikaciju između klijenta i servera putem gRPC-a.

3.2 Implementacija servera

Opis implementacije servera se može razdijeliti u pet osnovnih dijelova.

Unutar *chat.proto* datoteke definiran je servis *ChatService* koji omogućuje stalni dvosmjerni tok poruka između klijenta i server. (Kod 1)

```
// ChatService defines a bi-directional streaming RPC.
service ChatService {
rpc ChatStream(stream ChatMessage) returns (stream ChatMessage);
}
```

Kod 1 ChatService

Nadalje, Ovo je definicija servisa u Protobuf-u. Moramo definirati servis i u samoj Python skripti za server (Kod 2). Svaki spojeni klijent dobiva vlastiti red za primanje poruka. Redovi svih klijenata pohranjuju se u globalnu listu klijenata (*klijenti*).

```
class ChatServicer(chat_pb2_grpc.ChatServiceServicer):
  def ChatStream(self, request_iterator, context):
    q = queue.Queue()
    klijenti.append(q)
```

Kod 2 ChatServicer klasa

Poruke koje klijent šalje dolaze kroz *request_iterator*. Te se poruke šalju svim ostalim klijentima (u njihov red): Ova funkcija se izvršava u zasebnoj niti kako bi server mogao paralelno slati i primati poruke (Kod 3).

```
def citaj_poruke():
  for poruka in request_iterator:
    for k in klijenti:
      if k != q:
        k.put(poruka)
```

Kod 3 Funkcija za čitanje poruka

Postoji i beskonačna petlja servera koja konstantno šalje poruke iz reda klijenata prema odgovarajućem streamu (Kod 4)

```
while True:
  msg = q.get()
  yield msg
```

Kod 4 Slanje poruka iz reda kllijenta

Server se pokreće na na portu 50051 koristeći *ThreadPoolExecuto*r za obradu više klijenata.

Povežimo i objasnimo sve ove dijelove kako bi bolje shvatili njihovo funkcioniranje. Kada se klijent poveže na server, klijent dobije svoj poseban red (queue) od servera koji se koristi za primanje poruka. Svaka primljena poruka se prosljeđuje redovima klijenata, a zatim se iz tih redova šalje natrag prema odgovarajućim klijentima (s obzirom od kojeg je klijenta taj red). Zašto se baš koristi red? Red se koristi kako bi se osigurala pravilna dostava poruka i omogućila neovisna obrada podataka za svakog pojedinačnog klijenta. Svi klijenti **nemaju zajednički red** jer bi to dovelo do problema s istovremenim pristupom i mogućim gubitkom poruka.

3.3 Implementacija klijenta

Ideja klijentske aplikacije je da svaki korisnik, nakon što pokrene aplikaciju, može unositi svoje poruke koje se onda šalju serveru i distribuiraju ostalim korisnicima. Za prikaz jednostavnog grafičkog sučelja koristi se biblioteka Tkinter.

Klasa *ChatKlijent* pokreće komunikaciju sa serverom i priprema redove koji služe za slanje i primanje poruka (Kod 5)

```
self.out_msg = queue.Queue()
self.in_msg = queue.Queue()
self.channel = grpc.insecure_channel(SERVER)
self.stub = chat_pb2_grpc.ChatServiceStub(self.channel)
```

Kod 5 Redovi

Red *in_msg* se koristi za primanje poruka od drugih korisnika koje su pristigle preko servera, a *out_msg* služi za pripremanje i slanje poruka prema serveru.

Stalnu vezu sa serverom održava pozadinska nit koja šalje poruke iz generatora i prima odgovore (Kod 6).

```
threading.Thread(target=self.chat_stream, daemon=True).start()


def chat_stream(self):
  responses = self.stub.ChatStream(self.poruka_generator())
  for resp in responses:
    self.in_msg.put(f"{resp.username}: {resp.message}")
```

Kod 6 chat_stream klasa, pozadinska nit

Kada korisnik unese svoju poruku i pritisne tipku Enter, uneseni tekst se dodaje u već spomenuti red *out_msg* kako bi se uspješno poslao ka serveru. (Kod 7). U isto se vrijeme i odmah prikazuje korisniku lokalno (korisnik ne mora čekati da mu server pošalje natrag njegovu vlastitu poruku kako bi je prikazao na ekranu).

```
def posalji(self, event):
    txt = self.unos.get().strip()
    if txt:
        self.out_msg.put(txt)
        self.prikazi_poruku("Ti: " + txt)
        self.unos.delete(0, tk.END)
```

Kod 7 funkcija za slanje poruke

Funkcija *poruka_generator* je zadužena za slanje poruke ka serveru čim red *out_msg* dobije novu poruku (Kod 8).

```
def poruka_generator(self):
    while True:
        txt = self.out_msg.get()
        yield chat_pb2.ChatMessage(username=self.ime, message=txt)
```

Kod 8 Funkcija poruka_generator

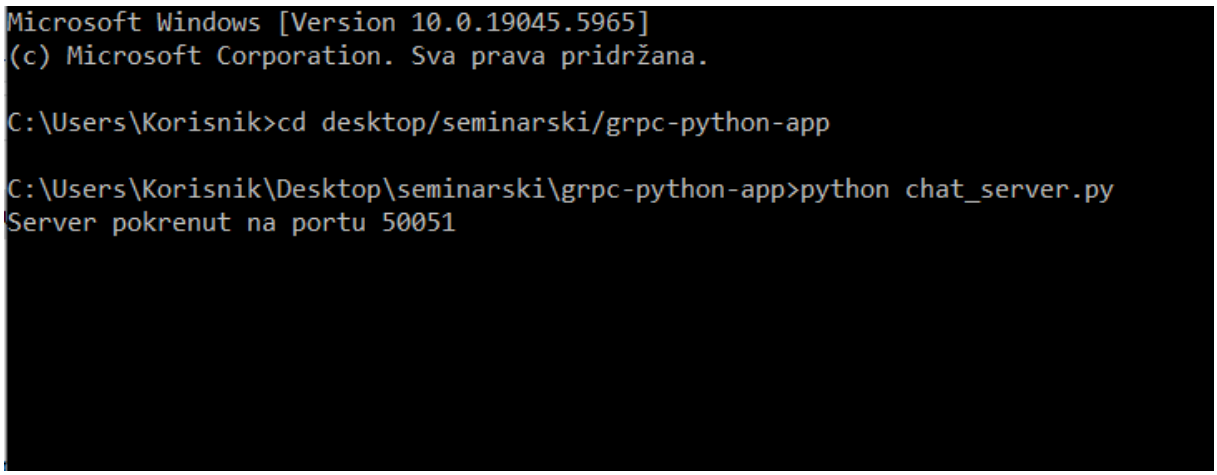
Primljene poruke iz *in_msg* prikazuju se na korisničkom sučelju i osvježavaju svakih 100 milisekundi. Funkcija *polleraj* također poziva funkciju za prikaz poruka na sučelju.

```
def polleraj(self):
    while not self.in_msg.empty():
        p = self.in_msg.get()
        self.prikazi_poruku(p)
    self.root.after(100, self.polleraj)
```

Kod 9 Funkcija za primanje poruka

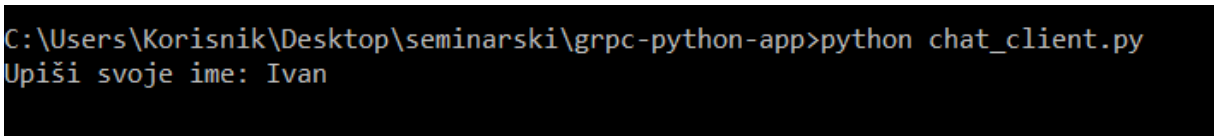
3.4 Primjer rada aplikacije

Kako bi prikazali osnovni princip rada potrebno je prvo pokreniti Python skriptu u kojoj se nalazi kod od servera (Slika 4).



Slika 4 Pokretanje servera

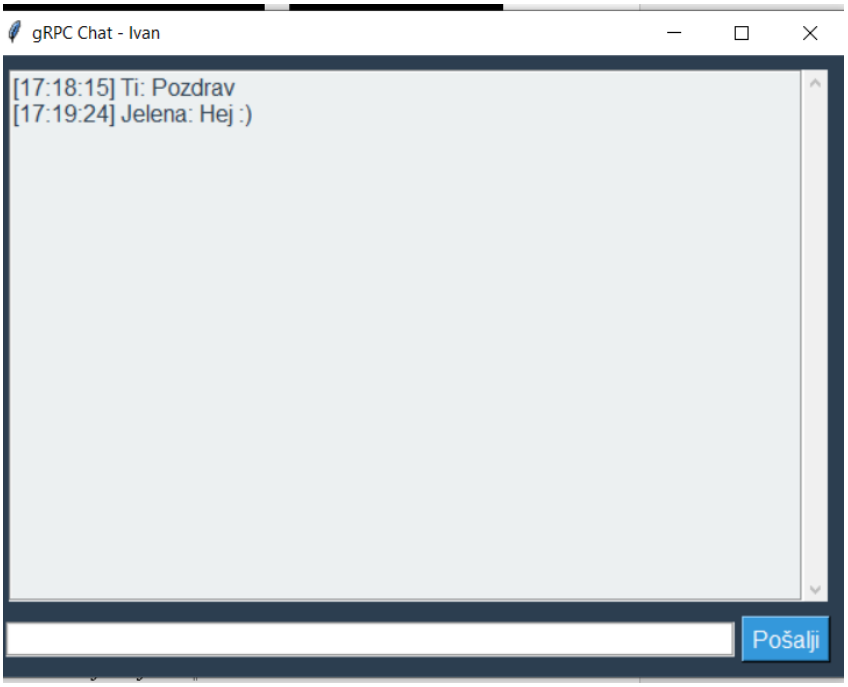
Nakon pokretanja Python skripte servera, potrebno je pokrenuti skriptu koja simulira klijente u novim prozorima naredbenog retka (Slika 5). Nužno je pokrenuti skriptu dva puta kako bi simulirali dva korisnika chat sustava.



Slika 5 Pokretanje klijenta

Dobili smo sučelje u kojem je moguće slati poruke. Ako pošaljemo poruku s jednog od klijenata, drugi klijent bi trebao primiti tu poruku i prikazati je na svojem sučelju (Slika 6).

Važno je napomenuti da je aplikacija razvijena na način da funkcionira lokalno. Kada bi htjeli omogućiti da aplikacija radi na više stvarnih i nepovezanih uređaja, bilo bi potrebno minimalno izmijeniti kod i postaviti server na neki javni servis za hosting.



Slika 6 Kratki razgovor na aplikaciji

4 ZAKLJUČAK

Kroz ovaj seminarski rad prikazano je kako gRPC uistinu može biti moćan alat za razvoj distribuiranih sustava. Teorijski dio rada dao je pregled osnovnih pojmova vezanih uz gRPC, uključujući njegove prednosti, arhitekturu, način rada i tipove komunikacije, dok je praktični dio prikazao konkretnu implementaciju jednostavne chat aplikacije koja koristi dvosmjernu komunikaciju između korisnika.

Unatoč tome što je gRPC složeniji za početnike, on nudi značajne prednosti nad tradicionalnim pristupima poput REST-a, posebno kada su u pitanju performasne i skalabilnost.

Implementirana aplikacija demonstrira moguću upotrebu gRPC-a u razvoju jednostavnih sustava koji omogućuju komunikaciju u stvarnom vremenu. Iako projekt sadrži određena ograničenja poput ograničenosti na lokalno izvođenje, on postavlja dobar temelj za buduća proširenja i dokazuje kako se gRPC može efikasno iskoristiti za razvoj ovakvog tipa sustava.

Zaključno, gRPC predstavlja suvremeni pristup razvoju mrežnih aplikacija i ima potencijal sve većeg korištenja u modernim softverskim arhitekturama.

LITERATURA

[1] Google Open Source, *gRPC*, dostupno na: https://opensource.google/projects/grpc (Zadnje pristupljeno: 19. 5. 2025.)

[2] Netflix Tech Blog, *Practical API Design at Netflix, Part 1: Using Protobuf FieldMask*, dostupno na: https://netflixtechblog.com/practical-api-design-at-netflix-part-1-using-protobuf-fieldmask-35cfdc606518 (Zadnje pristupljeno: 19. 5. 2025.)

[3] Lyft Engineering, *Scaling productivity on microservices at Lyft (Part 2)*, dostupno na: https://eng.lyft.com/scaling-productivity-on-microservices-at-lyft-part-2-optimizing-for-fast-local-development-9f27a98b47ee (Zadnje pristupljeno: 19. 5. 2025.)

[4] Dropbox Tech Blog, *Courier: Dropbox migration to gRPC*, dostupno na: https://dropbox.tech/infrastructure/courier-dropbox-migration-to-grpc (Zadnje pristupljeno: 19. 5. 2025.)

[5] GeeksforGeeks, gRPC Communication in Distributed Systems, dostupno na: https://www.geeksforgeeks.org/grpc-communication-in-distributed-systems/ (Zadnje pristupljeno: 20. 5. 2025.)

PRILOZI

Kazalo slika, tablica i kodova

Kazalo slika

Kazalo kodova

Popis oznaka i kratica

gRPC	Google Remote Procedure Call
REST	Representational State Transfer Application Programming Interface
API	Application Programming Interface
XML	Extensible Markup Language
JSON	JavaScript Object Notation

SAŽETAK I KLJUČNE RIJEČI

Sažetak[1]

gRPC (Google Remote Procedure Call) je okvir koji je razvio Google kako bi omogućio učinkovitu komunikaciju između različitih usluga. Osnova gRPC-a je HTTP/2 protokol, na čijim je temeljima izgrađen. Osnovna ideja iza gRPC-a je omogućiti programerima pozivanje funkcija na udaljenom poslužitelju kao da su lokalne. Ovo načelo može pojednostaviti razvoj distribuiranih sustava sakrivanjem složenosti mrežne komunikacije. Aplikacija razvijena u kontekstu ovog seminarskog rada je jednostavna chat aplikacija koja se temelji na gRPC tehnologiji. Razlog odabira aplikacije za chat je taj što svaki chat sustav mora omogućiti dvosmjernu komunikaciju između dva ili više korisnika u stvarnom vremenu, a gRPC ima idealno rješenje za ostvarivanje ovakve komunikacije koristeći bi-directional streaming.

Ključne riječi

gRPC, Chat aplikacija, Tkinter, streaming, istovremena komunikacija, Python