Pouzdana komunikacija u uvjetima loše mrežne povezanosti

Đuras, Eduard

Master's thesis / Diplomski rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Electrical Engineering and Computing / Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:168:357083

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-29**

Repository / Repozitorij:

FER Repository - University of Zagreb Faculty of Electrical Engineering and Computing repozitory

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 433

POUZDANA KOMUNIKACIJA U UVJETIMA LOŠE MREŽNE POVEZANOSTI

Eduard Đuras

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 433

POUZDANA KOMUNIKACIJA U UVJETIMA LOŠE MREŽNE POVEZANOSTI

Eduard Đuras

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

DIPLOMSKI ZADATAK br. 433

Pristupnik: **Eduard Đuras (0036515983)**
Studij: Računarstvo
Profil: Računarska znanost
Mentor: izv. prof. dr. sc. Marin Vuković

Zadatak: **Pouzdana komunikacija u uvjetima loše mrežne povezanosti**

Opis zadatka:

Loša mrežna povezanost onemogućuje pouzdan prijenos podataka unutar bilo koje telekomunikacijske mreže, neovisno o tehnologiji. Uzroka loše mrežne povezanosti može biti više; od nepristupačnih područja u kojim nema adekvatne infrastrukture, preko područja s velikom interferencijom signala ili velikim brojem korisnika, pa sve do pokretnih uređaja na kojima ponekada nije adekvatno implementirana mobilnost. Vaš je zadatak analizirati postojeća rješenja temeljena na malim uređajima Interneta stvari koji komuniciraju putem mobilnih mreža. Na temelju analize predložite i implementirajte rješenje za pouzdanu komunikaciju u uvjetima loše povezanosti te predloženo rješenje validirajte na proizvoljnom primjeru usluge.

Rok za predaju rada: 28. lipnja 2024.

Sadržaj

1. Uvod **3**

2. Problemi tradicionalnih protokola **5**

2.1. HTTP/1.1 5

2.1.1. Head-of-Line blokiranje 5

2.1.2. Trošak veze 6

2.1.3. Loše korištenje širine pojasa 6

2.1.4. Osjetljivost na gubitak paketa 7

2.2. HTTP/2 7

2.2.1. Problemi s prioritizacijom i ovisnošću tokova 7

2.2.2. Head-of-line blokiranje na *TCP* sloju 7

2.2.3. Spora uspostava veze 7

2.2.4. Migracija veze 8

2.3. Problemi s korištenjem tradicionalnih protokola na primjeru vozila u pokretu 9

2.3.1. Primjer 9

2.3.2. Uzorci problema 9

3. Rješenja novih tehnologija **11**

3.1. QUIC 11

3.1.1. Temeljen na UDP-u 11

3.1.2. Brža uspostava konekcije 12

3.1.3. Multipleksiranje bez *Head-of-line* blokiranja 13

3.1.4. Migracija konekcije 14

3.1.5. Mehanizmi kontrole zagušenja 15

3.2. HTTP/3 16

4. Razvijeno rješenje 17

4.1. Korištene tehnologije 17

4.1.1. *HTTP/3* i *QUIC* 17

4.1.2. Programski jezik Go 18

4.2. Klijent 19

4.2.1. Inicijalizacija 19

4.2.2. Naredba 22

4.3. Poslužitelj 26

4.3.1. API 26

4.3.2. Domena 27

5. Rezultati i rasprava 33

5.1. Rezultati 33

5.2. Moguća poboljšanja 36

5.2.1. Klijent 36

5.2.2. Poslužitelj 37

6. Zaključak 39

Literatura 40

Sažetak 42

Abstract 43

1. Uvod

IoT uređaji moraju raditi u različitim okruženjima, gdje je dostupnost signala bežične mreže, kao što su mobilne mreže, *LoRaWan* i *Sigfox*, često nepouzdana. Tradicionalni mrežni protokoli, poput *TCP/IP*, pokazuju ograničenja u ovakvim uvjetima zbog svoje potrebe za stabilnom i stalnom vezom te značajnim kašnjenjem u prijenosu podataka. Ovi izazovi postaju još izraženiji kada su *IoT* uređaji u pokretu. Cilj ovog rada je istražiti i testirati nove mrežne protokole koji bi mogli pokazati bolje performanse u uvjetima loše povezanosti i mobilnosti. Kao praktični primjer, odabran je uređaj instaliran u automobilu, koji periodički šalje fotografije s kamere u oblak. Testiranje će se fokusirati na procjenu pouzdanosti, učinkovitosti prijenosa podataka i otpornosti na prekide signala kod ovih novih protokola.

Loša mrežna povezanost onemogućuje pouzdan prijenos podataka unutar bilo koje telekomunikacijske mreže, neovisno o tehnologiji. Uzroka loše mrežne povezanosti može biti više; od nepristupačnih područja u kojim nema adekvatne infrastrukture, preko područja s velikom interferencijom signala ili velikim brojem korisnika, pa sve do pokretnih uređaja na kojima ponekada nije adekvatno implementirana mobilnost.

Postoje načini pomoću kojih se utjecaj loše mrežne povezanosti može smanjiti kako bi posljedice bile što manje. Standardni komunikacijski protokoli poput HTTP/1.1 i HTTP/2, iako robusni u normalnim okolnostima, teško održavaju performanse suočeni s visokom latencijom, gubitkom paketa i promjenjivom propusnosti. Kako bi se izbjegla ova ograničenja, rad ispituje potencijal novog protokola *HTTP/3*, koji ispod sebe koristi protokol *QUIC*. HTTP/3 osigurava poboljšane performanse i pouzdanost pomoću značajki kao što su multipleksiranje, migracija veze te mehanizam unaprijednog ispravljanja pogreške (engl. *forward error correction*).

Osim toga, ovaj rad istražuje tehnike manipulacije mrežom te segmentaciju veće količine podataka na manje komponente. Moguće je skenirati mrežu na koju je spojen uređaj i iz toga zaključiti parametre kojima se može manipulirati slanje podataka, poput veličine jedne komponente koja se šalje s uređaja na poslužitelj. Razbijanjem podataka na manje komponente smanjuje se utjecaj izgubljenih ili oštećenih paketa, omogućujući učinkovitije iskorištavanje mehanizama protokola HTTP/3 te fragmentacije i retransmisije.

Cilj ovog rada jest opisati realizirano rješenje koje adresira navedeni problem, detaljno prikazati korištene protokole i tehnologije te predložiti načine za unapređenje rješenja u daljnjem razvoju.

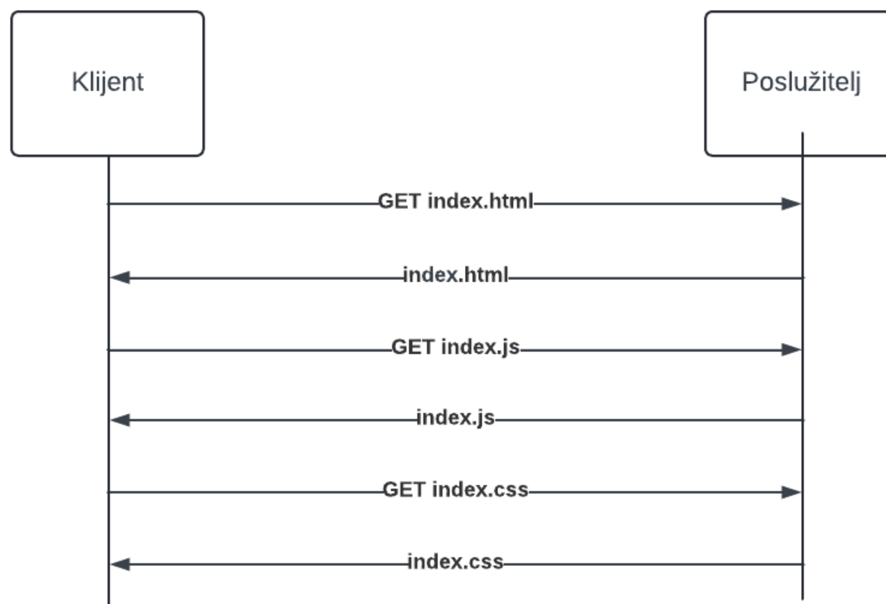
U sljedećem poglavlju detaljno su objašnjeni problemi standardnih tehnologija te zašto one nisu primjenjive u uvjetima loše povezanosti. Treće poglavlje objašnjavaja nove tehnologije te njihove mehanizme kojima se rješavaju navedeni problemi. U četvrtom poglavlju opisuje se razvijeno rješenje koje koristi nove tehnologije i iskorištava njihove prednosti. Navode se sve korištene tehnologije i način na koje je rješenje dizajnirano i strukturirano. U petom poglavlju su priloženi rezultati i zaključci dobiveni testiranjem razvijenog rješenja.

2. Problemi tradicionalnih protokola

2.1. HTTP/1.1

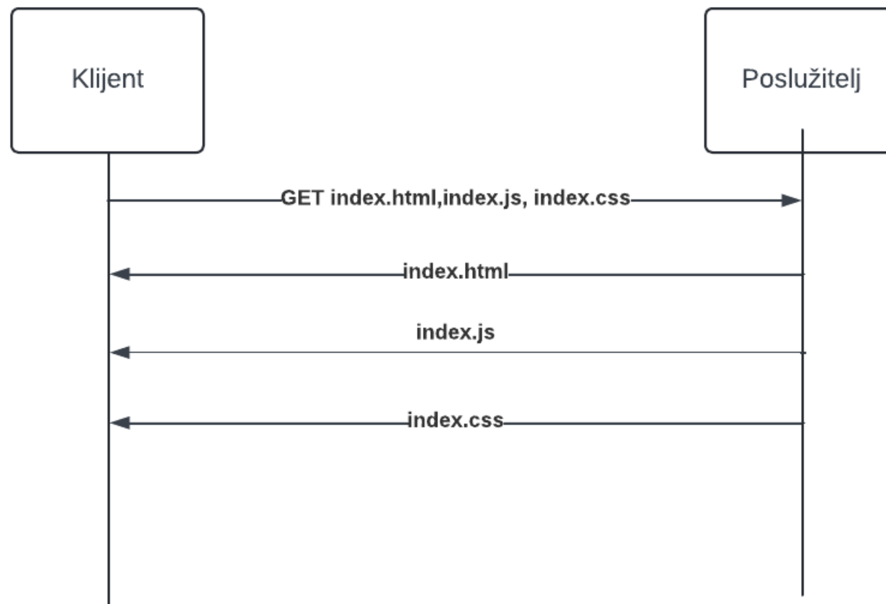
2.1.1. Head-of-Line blokiranje

Do ovog problema dolazi kada se od poslužitelja zatraži veći broj resursa. *HTTP/1.1* posjeduje mehanizam stavljanja sekvencijalnih zahtjeva u cjevovod[1]. To znači da klijent može poslati veći broj zahtjeva, bez da čeka pojedinačni odgovor od njih. Bez korištenja cjevovoda klijent pošalje zahtjev i čeka odgovor na njega prije nego pošalje drugi zahtjev. Korištenjem cjevovoda klijent šalje veći broj zahtjeva preko iste *TCP* konekcije bez čekanja odgovora na pojedinačne zahtjeve. Poslužitelj obrađuje zahtjeve redoslijedom kojim siu pristigli, a potom šalje odgovore u istom redoslijedu[2].



Slika 2.1. *HTTP/1.1* bez korištenja cjevovoda

Head-of-line blokiranje događa se u ukoliko se neki raniji zahtjev dugo obrađuje ili



Slika 2.2. *HTTP/1.1* sa korištenjem cjevovoda

dođe do gubitaka paketa prilikom slanja odgovora. Tada svi preostali zahtjevi u cjevovodu moraju čekati.

2.1.2. Trošak veze

Kako bi se izbjegao ovaj problem, klijenti najčešće otvaraju veći broj *TCP* konekcija prema poslužitelju. Svaka nova konekcija uključuje trošak TCP-ovog sporog starta, gdje se brzina prijenosa postepeno povećava kako bi se se provjerila dostupna širina transmisijskog pojasa. Ovaj proces uvodi dodatnu latenciju i neučinkovito korištenje širine pojasa, posebno u mrežama s visokom latencijom.

2.1.3. Loše korištenje širine pojasa

U situacijama visokih latencija, *round-trip time* (RTT) se značajno povećava, čineći *TCP* rukovanje i inicijalni spori start još dužima. S obzirom da konekcije ne mogu dovoljno brzo povećati brzinu prijenosa kako bi u potpunosti iskoristile kapacitet mreže, navedeno rezultira neiskoristivošću moguće širine pojasa[2].

2.1.4. Osjetljivost na gubitak paketa

Oslanjanje na *TCP* znači da gubitak paketa dovodi do retransmisije kojom upravlja on sam. Učestali gubitak paketa rezultira ponovljenim retransmisijama i kašnjenjima. *Head-of-line blokiranje* pogoršava ovaj problem, uzrokujući značajno smanjenje performansi[2].

2.2. HTTP/2

2.2.1. Problemi s prioritizacijom i ovisnošću tokova

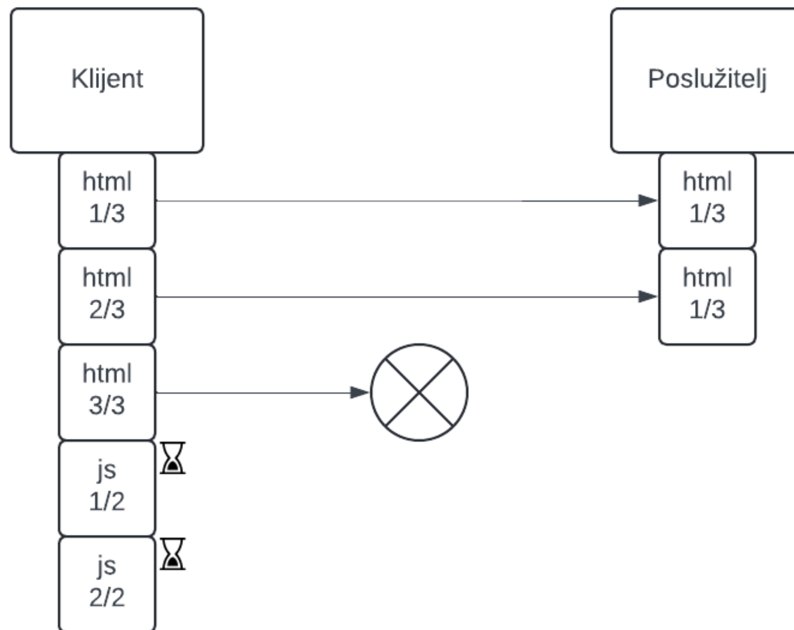
HTTP/2 uvodi multipleksiranje tokova preko jedne *TCP* veze, omogućujući međusobno ispreplitanje više tokova podataka[3]. Međutim, ako dođe do gubitka paketa, TCP-ov mehanizam retransmisije utječe na samo jedan tok unutar te veze. *HTTP/2* dijeli podatke na manje komade, tzv. okvire. Multipleksiranjem se okviri, koji pripadaju različitim tokovima podataka, međusobno miješaju te šalju preko iste veze bez međusobnog čekanja tokova podataka da završe.To znači da ako se jednom toku podataka dogodi kašnjenje ili gubitak paketa, to ne utječe na druge tokove podataka[2].

2.2.2. Head-of-line blokiranje na *TCP* sloju

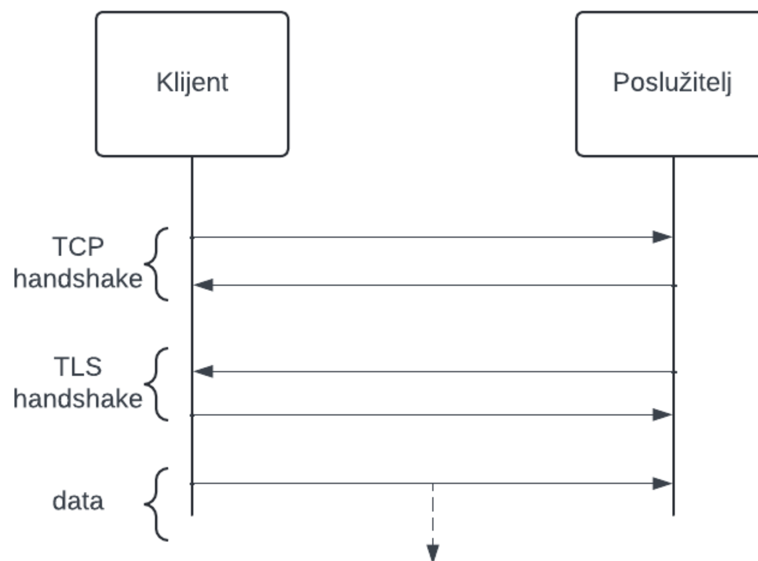
Iako *HTTP/2* rješava ovaj problem na aplikacijskom sloju, *Head-of-line blokiranje* je i dalje prisutno na *TCP* sloju[2]. U slučaju gubitka paketa, TCP-ov mehanizam retransmisije blokira sve tokove podataka na toj konekciji dok taj paket nije uspješno poslan. Za razliku od *HTTP/1.1*, *HTTP/2* koristi jednu *TCP* konekciju za rukovanje većim brojem tokova podataka. Svaki tok podataka ima pripadajući identifikator kako bi se paketi različitih tokova razlikovali. Također, *HTTP/2* također podržava prioritizaciju tokova, osiguravajući da se važniji resursi dohvate prvi[4].

2.2.3. Spora uspostava veze

Uspostava veze uključuje vrijeme koje je potrebno da se napravi *TCP handshake* i pregovaranje SSL parametara gdje je to omogućeno. Usporenje se može dogoditi iz više razloga, no sama činjenica da je potrebno napraviti dva zahtjeva-odgovora prije no što se pošalje prvi bajt podataka, postoji mogućnost velikog utroška vremena.



Slika 2.3. *HTTP/2 Head-of-line blokiranje* na razini TCP-a



Slika 2.4. Uspostavljanje *HTTP/2* konekcije

2.2.4. Migracija veze

Nemogućnost jednostavnog prijenosa postojeće *TCP* veze s jedne klijentske adrese na drugu može stvarati veliki problem prilikom prijenosa veće količine podataka. Ovaj problem najčešće se javlja zbog mobilnosti klijenta (učestale promjene mreže) te promjena na poslužiteljskoj strani (npr.*engl. load-balancing*). Razlog tomu je ovisnost protokola

HTTP/2 o protokolu *TCP* koji je po svojoj prirodi konekcijski orijentiran. *TCP* konekcija je definirana sa 4 parametra: izvorna IP adresa, izvorišni port, odredišna IP adresa i odredišni port. U slučaju promjene bilo kojeg od navedenih parametara, zahtijeva se ponovna uspostava veze.

2.3. Problemi s korištenjem tradicionalnih protokola na primjeru vozila u pokretu

2.3.1. Primjer

Kako bi se lakše vizualizirao problem koji se rješavao, u nastavku slijedi primjer gdje se taj problem javlja u stvarnome svijetu. Pretpostavlja se da postoji automobil u kojem se nalazi uređaj na koji je spojena kamera. Ta kamera periodički snima fotografije koje se naknadno procesiraju. Procesirane fotografije se potom šalju na poslužitelj koji se nalazi u oblaku.

Problem nastaje kada se automobil nađe u području slabe povezanosti na mrežu te se, zbog njene veličine, fotografija neuspješno pošalje na poslužitelj. Najčešća situacija je da dio fotografije stigne na odredište, no zbog ograničenja protokola dolazi do retransmijsije cijele fotografije prilikom ponovnog spajanja na mrežu. Ono što se u ovom slučaju može pokušati jest ponovo poslati fotografiju. No, u tom slučaju postoji mogućnost da će se ponoviti ista greška kao i prvi puta, te će samo dio fotografije stići na odredište. Želi se izbjeći situacija u kojoj se prilikom retransmisije mora ponavljati slanje cijele fotografije i, po mogućnosti, nastaviti od onog paketa pri kojem se javio problem.

Trenutno rješenje ne ponavlja slanje u slučaju pojavljivanja pogreške prilikom slanje fotografije, već sprema tu fotografiju na datotečni sustav uređaja.

2.3.2. Uzorci problema

Rješenje za komunikaciju između klijenta i poslužitelja koristi protokol *HTTP/2* bez optimiziranja veličine paketa koji se šalju te ponovne (programske) retransmisije ukoliko dođe do pogreške.

Najveći problema trenutnog rješenja jest *Head-of-line blokiranje* koje značajno uspo-

rava slanje podataka na poslužitelj. Osim što su paketi blokirani za slanje u slučaju gubitka veze, ti se paketi šalju jedan za drugim umjesto da se to radi istovremeno. Zbog te slabe iskoristivosti mrežnog pojasa, osim što je prijenos podataka značajno sporiji, zbog čestog pucanja mreže često dolazi do neuspješnog slanja cijele fotografije.

Drugi veliki problem jest *migracija veze*. Kretanje automobila uzrokuje promjenu pristupne točke na koju je on povezan. Promjenom pristupne točke dolazi do promjene klijentske mrežne adrese. Posljedica je što poslužitelj nakon toga više nema kontekst koji je imao s prijašnjom mrežnom adresom. Sva komunikacija koju je klijent obavljao u datom trenutku mora krenuti ispočetka s obzirom da standardni protokoli kao identifikator pri komunikaciji koriste upravo mrežnu adresu.

Također, problem je i spora uspostava veze. Dva poziva prema poslužitelju prije prijenosa prvog podataka stvara veliki trošak u lošim mrežnim uvjetima. Često se dogodi situacija da se niti ne uspostavi veza prije no što dođe do greške. Ovaj je problem još izraženiji u slučaju da želimo uspostaviti više konekcija za multipleksiranje podataka.

Još jedan problem je kada paketi uspješno stignu do poslužitelja, no potvrda o privitku paketa nikad ne stigne do klijenta. Tada dolazi do retransmisije jer klijent nije dobio potvrdu o uspješnosti slanja.

3. Rješenja novih tehnologija

3.1. QUIC

QUIC je protokol transportnog sloja razvijen od strane *Google*-a i standardiziran od *IETF*-a (*Internet Engineering Task Force*)[5]. Dizajniran je kako bi osigurao sigurnu, nisko latnetnu i pouzdanu komunikaciju preko interneta. Glavna značajka *QUIC*-a je što umjesto TCP-a koristi UDP i ima mogućnost da podupire svojstva koja su inače asocirana sa protkolima aplikacijskog sloja[6].

3.1.1. Temeljen na UDP-u

QUIC koristi UDP umjesto TCP-a kako bi zaobišao troškove i ograničenja koja dolaze s TCP-om, dok u isto vrijeme omogućuje brži početak i kraj komunikacije.

Za razliku od TCP-a, UDP nije konekcijski protokol, što znači da ne zahtijeva rukovanje (*engl. handshake*) kako bi uspostavio konekciju. Ovo omogućava QUIC-u da sam bude odgovoran za rukovanje kod uspostavljanja konekcije i parametara za sigurnu komunikaciju [7].

TCP je *stateful* protokol koji u sebi ima ugrađene mehanizme za pouzdanost, kontrolu toka i zastoja. Kako su ove značajke usko vezane uz TCP, teško je mijenjati ponašanje protokola na način koji nama odgovara jer bi to značilo da se te promjene moraju standardizirati nad cijelim internetom. UDP je jednostavan, *stateless* protokol koji omogućuje jednostavni prijenos paketa bez ugrađenih značajki za pouzdanost i kontrolu toka. To omogućuje protokolima koji koriste UDP da sami ugrade mehanizame koji su zaduženi za to, na način koji njima odgovara[8].

Također, UDP ne sili redoslijed i pouzdanost prijenosa paketa, što omogućava pro-

tokolima da sami implementiraju mehanizam multipleksiranja bez *head-of-line* blokiranja.

Sami razvoj protokola mnogo je lakši ukoliko se koristi UDP. Promjene nad TCP-om zahtjevaju promjene u samoj jezgri operacijskog sustava te se moraju standardizirati uz rigorozne mjere. S obzirom da je UDP vrlo jednostavan protokol, nikakve promjene nisu zahtjevane unutar samog UDP-a, već se sami razvoj događa iznad njega u aplikacijskom sloju (npr. u protokolu *QUIC*).

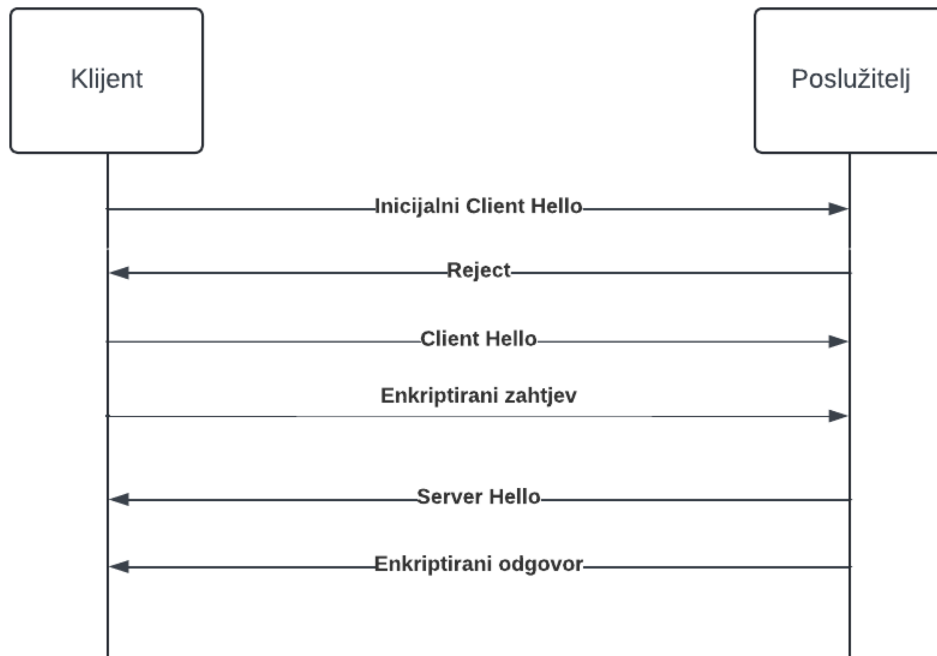
3.1.2. Brža uspostava konekcije

QUIC objedinjuje transportna i kriptografska rukovanja u jedan korak. Integrirajući *TLS 1.3* direktno u protokol može uspostaviti sigurne konekcije sa znatno manje poruka.

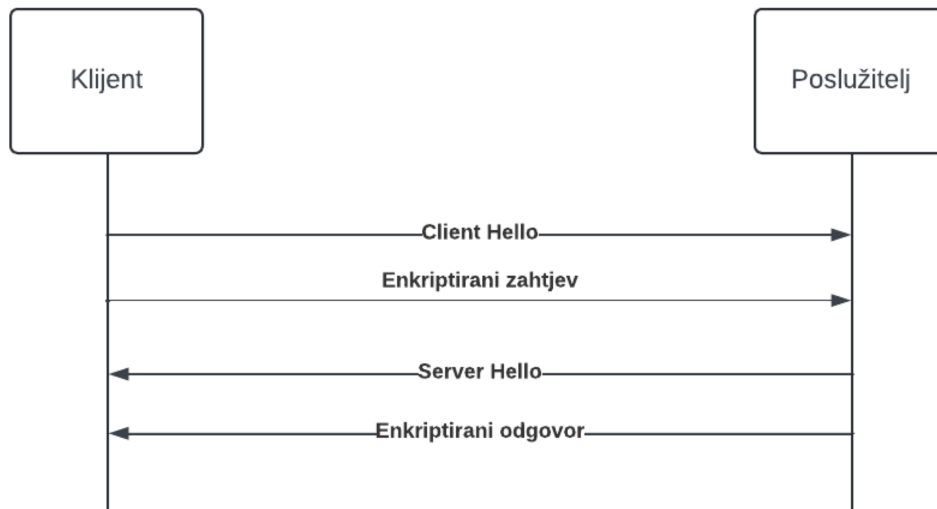
U inicijalnoj konekciji, klijent nema informacije potrebne za komunikaciju s poslužiteljom. Nakon prvog uspješnog rukovanja, klijent sprema informacije o poslužitelju na način da za svaku slijednu konekciju s tim poslužiteljem, klijent ne treba obaviti taj inicijalni poziv, nego može odmah početi slati podatke. Također, podaci se mogu slati odmah nakon slanja zahtjeva za rukovanje, bez čekanja odgovora od poslužitelja.

Kada klijent prvi puta uspostavlja konekciju sa poslužiteljem, klijent će poslati nedovršenu *CLIENT HELLO* (CHLO) poruku na koju očekuje *REJECT* (REJ) odgovor. REJ poruka sadržava poslužiteljevu konfiguraciju, certifikat za autentifikaciju, potpis poslužiteljevog certifikata i token izvorišne adrese *engl. source-address token*. Token izvorišne adrese se koristi u budućim komunikacijama kako bi se potvrdio identitet klijenta. Klijent koristi informacije koje je primio u REJ poruci kako bi poslao kompletnu CHLO poruku. Nakon toga klijent šalje potpunu CHLO poruku koja sadrži privremeni *Diffie-Hellman* ključ.

Nakon inicijalnog rukovanja, klijent posjeduje sve potrebne podatke za uspostavljanje konekcije. Jednom kada klijent pošalje kompletnu CHLO poruku poslužitelju, može početi slati podatke bez čekanja SHLO odgovora. Na ovaj način *QUIC* omogućava *0-RTT (engl. 0-round-trip)*. Nakon što klijent primi SHLO poruku može početi slati podatke koristeći krajnje izračunate ključeve za sigurnu komunikaciju.



Slika 3.1. Inicijalno uspostavljanje *QUIC* konekcije



Slika 3.2. Slijedno uspostavljanje *QUIC* konekcije

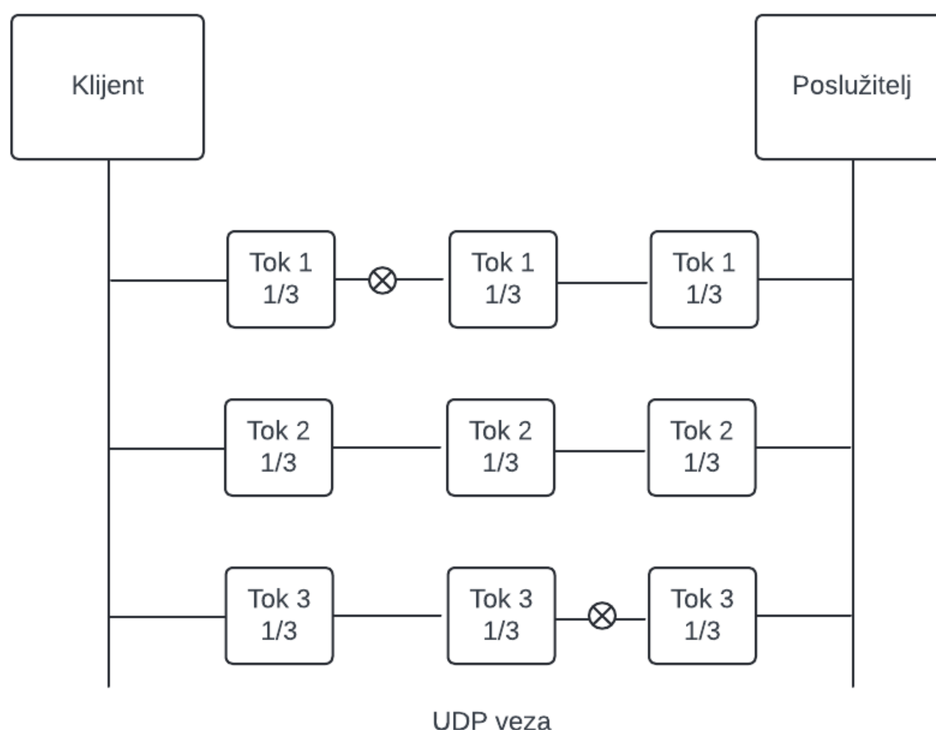
3.1.3. Multipleksiranje bez *Head-of-line* blokiranja

QUIC rješava problem *Head-of-line* blokiranja na način da je svaki tok podataka identificiran jedinstvenim identifikatorom, omogućujući klijentu i poslužitelju da razlikuju tokove podataka na istoj konekciji. Multipleksiranje podataka omogućuje kontrolu za svaki zasebni tok. To znači da slanje i primanje podataka za svaki pojedinačni tok ne

utječe na ostale tokove podataka. Razlog tomu jest što je *QUIC* izgrađen nad UDP protokolom koji ne prisiljava redoslijed slanja paketa.

QUIC dijeli podatke u okvire koji su temeljna transportna jedinica. Okviri različitih tokova podataka mogu biti spojeni u jedan QUIC paket. Ovo spajanje omogućuje *QUIC*-u da bolje iskorištava dostupni mrežni pojas i tako izbjegava *Head-of-line* blokiranje.

Rezultat ovakvog načina rukovanja tokovima podataka je značajno smanjenje latencije u komunikaciji. Također, mogućnost rukovanja gubitkom paketa na razini toka podataka osigurava sveukupnu pouzdanost i robustnost protokola.



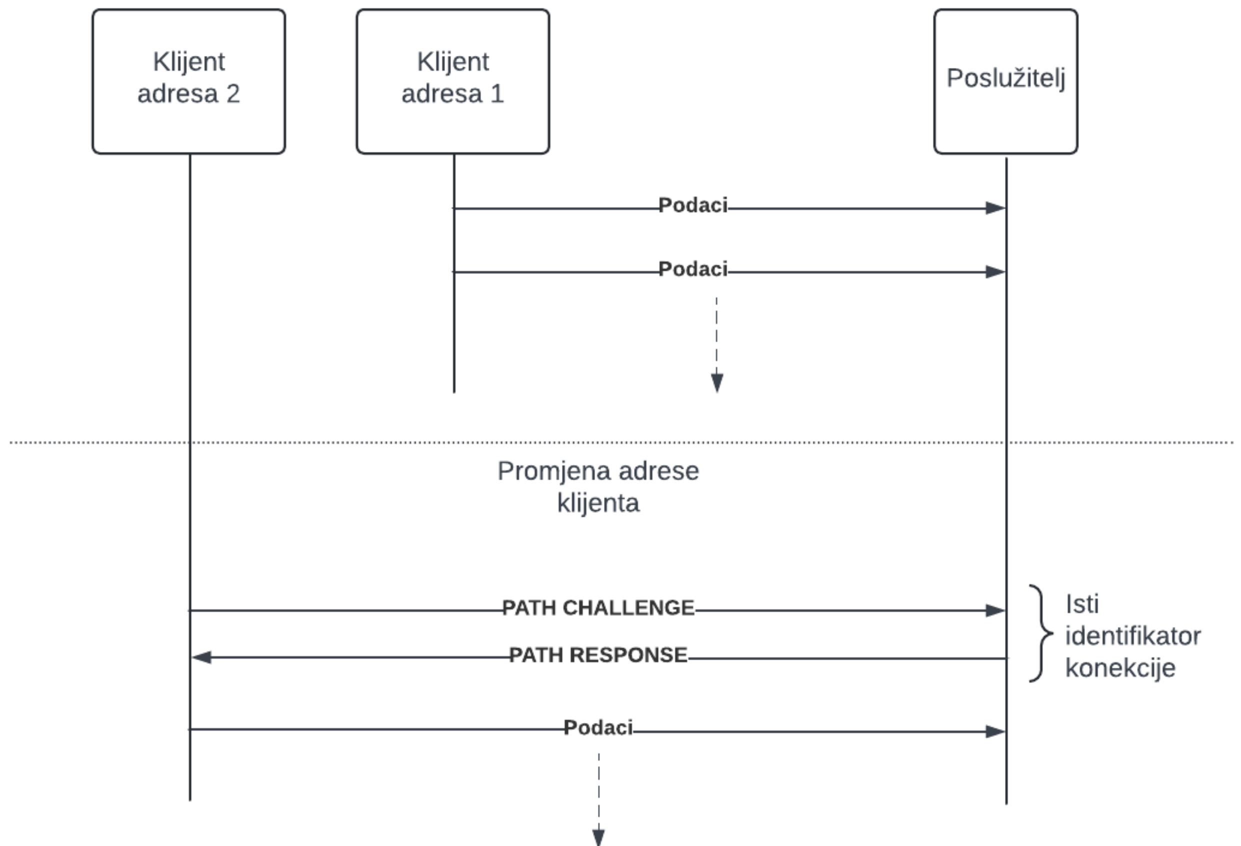
Slika 3.3. Multipleksiranje protokola QUIC

3.1.4. Migracija konekcije

Migracija konekcije jedna je od najvažnijih značajka protokola *QUIC* koja omogućava nastavak komunikacije u situacijama kada se mrežna adresa sudionika promijeni.

QUIC koristi identifikator konekcije umjesto četvorke izvorišna adresa i port te ciljna adresa i port kako bi razlikovao konekcije. Taj identifikator uvijek ostaje isti, bez obzira

na mrežne promjene. *QUIC* radi validaciju puta mrežne adrese kada se ona promijeni. To radi slanjem *PATH CHALLENGE* okvira od klijenta prema poslužitelju ili obrnuto te očekuje *PATH RESPONSE* poruku kako bi potvrdio validnost nove putanje. Oboje, klijent i poslužitelj, pamte informacije o konekciji, uključujući kriptografske parametre i detalje o sesiji. Ove informacije povezane su preko identifikatora konekcije što omogućuje jednostavnu migraciju.



Slika 3.4. Migracija konekcije protokola QUIC

3.1.5. Mehanizmi kontrole zagušenja

QUIC koristi napredne mehanizme kontrole zagušenja kako bi učikovito upravljao brzinom prijenosa podataka te minimizirao zagušenje, osiguravajući optimalne preformanse, čak i u uvjetima loše mrežne povezanosti. TCP također koristi neke slične mehanizme kojima kontrolira veličinu prozora (količinu podataka koji mogu biti poslani bez da se za njih primi ACK) s obzirom na opažene preformanse mreže.

QUIC također koristi te mehanizme u svojem protokolnom stogu, no on također može

podržavati i naprednije algoritme te kombinaciju mehanizama zbog svoje fleksibilnosti razvoja na aplikacijskom sloju (zato što ispod sebe koristi UDP koji nema nikakva ograničenja).

BBR (*engl. Bottleneck Bandwidth and Round-trip propagation time*) je dizajniran kako bi maksimirao protočnost podataka te minimizirao latenciju na način da procjenjuje propusnost uskog grla mreže te *engl. round-trip time* od izvorišta do cilja.

Cubic modificira linearnu funkciju rasta prozora u kubičnu funkciju kako bi poboljšao skalabilnost. Također, postiže pravedniju raspodjelu širine pojasa među tokovima s različitim RTT-ovima tako što čini rast prozora neovisnim o RTT-u.

NewReno uvodi pojam parcijalne potvrde (*engl. acknowledgment* koji poboljšava i ubrzava oporavak od gubitka paketa u istom prozoru. Kada se dogodi gubitak više paketa unutar istog prozora, smanji veličinu prozora 50 posto za svaki paket. Primjerice, ako su izgubljena 2 paketa, veličina prozora biti će smanjena 4 puta.

PCC (*Performance-oriented Congestion Control*) - dinamički mijenja frekvenciju slanja paketa s obzirom na kontinuiranu evaulaciju mrežnih preformansi (propusnost, latenciju i frekvenciju gubitaka).

3.2. HTTP/3

HTTP/3 će biti prva velika promjena nad protokolom od kad je *HTTP/2* standardiziran 2015. godine. Glavna značajka ovog protokola jest ta što kao transportni protokol koristi QUIC, iskorištavajući sve prednosti koji dolaze s njime, a koje su objašnjene u prošlom poglavlju.

Protokol je dizajniran za intenzivno korištenje interneta preko mobilnih uređaja koje ljudi nose sa sobom cijelo vrijeme, konstantno u pokretu te mijenjajući mrežu. S tom pretpostavkom nisu bile dizajnirane prošle vrste protokola koje su izrađene za stacionarne uređaje koji ne mijenjaju često mrežu.

Zaobilazi se pojava sporog preofrmansa kada se mobilni uređaj prebaci s *wifi*-a na mobilnu mrežu i obrnuto. Također, smanjuje se utjecaj gubitaka paketa koji je vrlo čest u mobilnim mrežama.

4. Razvijeno rješenje

4.1. Korištene tehnologije

4.1.1. *HTTP/3* i *QUIC*

Iz prednosti tehnologija objašnjenih u prošlom poglavlju vidljivo je da su one očit izbor za problem koji se rješava ovim projektom. Svojim novim mehaniznima one adresiraju sve probleme koji se nailaze u projektu te ih pokušavaju otkloniti.

Brža uspostava konekcije osigurava bržu povezanost urešaja s poslužiteljem u trenucima kada treba prenijeti fotografiju. Korištenjem protokola *HTTP/2* često se događalo da klijent ne uspije dovršiti proces rukovanja sa poslužiteljem zbog latencije mreže, te se do tada već dogodi da uređaj izgubi signal u potpunosti. Tijekom utrke treba poslati veliki broj fotografija, a konstantno uspostavljanje veze jedno je od očekivanih uskih grla prilikom komunikacije.

Multipleksiranje bez head-of-line blokiranja omogućuje upravljanje načinom slanja fotografije na poslužitelj (objašnjeno kasnije). Ukratko, ovaj mehanizam može se iskoristiti da fotografiju podijelimo na više tokova podataka te nastavi sa slanjem tamo gdje je stalo u trenutku gubitka signala.

Migracija konekcije najznačajnije je unaprjeđenja za ovaj projekt. Automobil se svojim kretanjem konstantno nalazi u situaciji da mora promjeniti mrežu, tj. pristupnu točku na koju je spojen. Mogućnost nastavka slanja paketa od trenutka promjene značajna je optimizacija s obzirom da nije potrebno početi slati cijelu fotografiju ispočetka. Također, treba uzeti u obzir da se trkaći automobil kreće velikim brzinama, stoga do migracije konekcije dolazi učestalije nego kod prosječnog korisnika.

4.1.2. Programski jezik Go

Go, ili često zvan *Golang* [9], statički je tipiziran programski koji je dizajnirao Google 2009. godine. U ovom projektu korišten je za izgradnju poslužitelja i klijenta.

Karakteristike Go-a su sljedeće:

Istodobnost izvršavanja (engl. Concurrency) je glavna prednost jezika, koja pruža veliku podršku za istodobno izvršavanje više zadataka jednog programa. Ovdje glavnu ulogu ovdje imaju *go rutine* (*engl. Goroutines*), što su zapravo jednostavne dretve koje su orkestrirane pomoću *Go runtime-a*. Također, uvode se kanali, tj. cjevovodi koji su jezgrena primitiva jezika za komunikaciju između više rutina.

Jednostavna organizacija koda je sintaksa jezika vrlo je jednostavna za pisanje i čitanje, a promovira pisanje čistog i efikasnog koda. Sintaksom je jezik sličan C-u, ali jednostavniji.

Brzina znači da je dizajniran za brzo prevođenje i izvođenje što ga čini izvrsnim za aplikacije koje su ograničene preformansama.

Sakupljanje smeća (engl. garbage collection) - automatizirana briga o memoriji što znači programer je oslobođen odgovornosti za zauzimanje i oslobađanje memorije.

Jezik dolazidolazi s bogatom *standardnom bibliotekom* koja uključuje pakete za izvršavanje raznih zadataka poput ulaza-izlaza, obrade teksta... omogućujući pisanje robusnih aplikacija bez velike ovisnosti o vanjskim bibliotekama.

Alati i ekosustav - *Go toolchain* koji zadržava alate za izgradnju, testiranje i upravljanje Go kodom. *Go modules* su alat koji upravlja ovisnostima i paketa unutar koda. Predstavlja aktivnu zajednicu, odličnu dokumentaciju te mnoštvo vanjskih biblioteka i okvira. Za izradu ovog projekta koristila se biblioteka *quic-go*[1] na klijentu i poslužitelju koja omogućuje komunikaciju koristeći protokol *QUIC*.

Pojedini veliki projekti koji su pisani u ovom jeziku su *Docker* (platforma za kontejnerizaciju), *Kubenretes* (sustav za orkestraciju kontejnera) i *etcd* (distribuirana ključ-vrijednost baza).

[1] https://github.com/quic-go/quic-go

Za bolje upoznavanje s jezikom preporucio bih knjigu od Jon Bodnera [10].

4.2. Klijent

Klijent je program napisan u Gou, a namjenjen je da se pokreće iz ljuske operacijskog sustava ili pozivanjem iz drugih programa. Njegova je zadaća poslati datoteku koju je dobio na ulazu prema poslužitelju gdje će ta datoteka biti obrađena. Također, zamišljen je da se u budućnosti, ako bude potrebe, u njega mogu implementirati i druge funkcionalnosti. Izvorni kod klijenta može se naći na *Github*-u [2].

4.2.1. Inicijalizacija

Tijekom izvršavanja klijent može generirati određene greške, a te greške vraćaju se iz programa kao izvršni kodovi. Kodovi su definirani kao enumeracija, tj. konstante.

```
package exitcodes
const (
    ExitSuccess = iota
    ExitUnkownCommand
    ExitFailedInit
    ExitBadArguments
    ExitFailedReadingFile
    ExitFailedProcessingData
)
```

Klijent prima listu ulaznih argumenata (*engl. command-line arguments*) i s obzirom na njih provodi neku akciju. Parametri su sljedeći:

1. Broj naredbe
    - Potrebno je specificirati naredbu koja će se izvršiti. Trenutno postoji samo jedna naredba, a to je slanje datoteke na poslužitelj i njen broj je 0. Ovo je napravljeno zbog vjerojatnosti da će klijenta biti potrebno proširiti u budućnosti s dodatnim naredbama koje će slati neke druge podatke.

[2]https://github.com/Edrudo/diplomski-rad-client

2. Lista argumenata

    - Svi argumenti koje odabrana naredba zahtijeva kako bi mogla biti izvršena. Naredba za slanje datoteke zahtijeva dva argumenta:

        (a) Url na koji će datoteka biti poslana

        (b) Lista puteva do datoteke na računalu koji pokreće program. Ovo je moguće poslati kao i relativni put iz direktorija.

Ulazna točka programa je *main.go* datoteka. Kada se klijent pokrene njegova prva zadaća je parsirati ulazne argumente. U slučaju da argumenti nisu ispravno predani programu vraća se greška s prikladnim ispisom.

```
args := os.Args
if len(args) < 2 {
    utils.DefaultLogger.Fatalf(
        errors.New(
            "arguments needed for the program: "+
                "\t - command number"+
                "\t - arguments needed for the command",
        ), exitcodes.ExitBadArguments,
    )
}

commandNum, err := strconv.Atoi(args[1])
if err != nil {
    utils.DefaultLogger.Fatalf(
        errors.New(
            "command number is not a number",
        ), exitcodes.ExitUnkownCommand,
    )
}
```

Nakon toga potrebno je inicijalizirati samog klijenta i sve njegove ovisnosti poput http

klijenta, *loggera* i objekta koji je zadužen za izračunavanje *SHA256* sažetka. Klijent se inicijalizira s otvorenom konekcijom prema poslužitelju te konfiguriranim *TLS*-om.

```
client, roundTripper := bootstrap.NewClient()
defer func() {
   ...
}()


func NewClient() (*application.Client, *http3.RoundTripper) {
utils.DefaultLogger.SetLogLevel(utils.LogLevelError)
...
return application.NewClient(sha256.New(), httpClient, roundTripper), roundTripper
}


func initilizeRoundTripper() *http3.RoundTripper {
    insecure := flag.Bool("insecure", false, "skip certificate verification")
    ...
    pool, err := x509.SystemCertPool()
    if err != nil {
        utils.DefaultLogger.Fatalf(err, exitcodes.ExitFailedInit)
    }
    tlsconfig.AddRootCA(pool)

    var qconf quic.Config
    ...
    return &http3.RoundTripper{
        TLSClientConfig: &tls.Config{
            RootCAs:            pool,
            InsecureSkipVerify: *insecure,
        },
        QuicConfig: &qconf,
    }
}
```

4.2.2. Naredba

Nakon prethodno opisanog, na redu je izvršavanje same naredbe. Ako je naredba uspješno izvršena iz programa se izlazi s *ExitSuccess* izvršnim kodom. Klijent je kreiran koristeći oblikovni obrazac *Command* te je za izvršavanje potrebno klijentovoj *Execute* metodi proslijediti strukturu podataka *Command* koja zadržava ime (broj) komande i argumente potrebne za njeno izvršavanje.

```
client.ExecuteCommand(
    application.Command{
        Name: application.CommandName(commandNum),
        Args: args[2:],
    },
)

os.Exit(exitcodes.ExitSuccess)

type CommandName int

const (
    SendFile CommandName = iota
)

type Command struct {
    Name CommandName
    Args []string
}

func (c *Client) ExecuteCommand(command Command) {
    switch command.Name {
    case SendFile:
        c.sendFile(command.Args)
    ...
```

```
}
```

Na početku *sendFile* naredbe ponovo se provjerava ispravnost argumenata na sličan način kao što se to radi i u funckiji *main*.

Potrebna je iteracija po svim datotekama koje su predane naredbi s obzirom da da podržavamo slanje više njih jednim pozivom programa. Ovo nije preporučljivo (iako je podržano) zato što u slučaju greške prilikom procesiranja jedne od datoteka, klijent vraća pripadajući izvršni kod programa i pokretač klijenta ne zna koja su datoteke uspješno obrađene, a koje nisu. Također, trenutno se datoteke obrađuju jedna za drugom, a ne istovremeno što je jedna od stvari koje su sklone unaprjeđenju u ovom projektu.

Prvo se te datoteka učita iz datotečnog sustava u izvršni program, a u slučaju da to ne uspije izlazi se iz programa sa *ExitFailedReadingFile* izvršnim kodom.

```
readFile, err := os.ReadFile(geoshotPath)
if err != nil {
    utils.DefaultLogger.Fatalf(err, exitcodes.ExitFailedReadingFile)
}
```

Ako čitanje datoteke prođe uspješno, radimo podjelu polja bajtova koji sadržavaju učitanu datoteku na manje dijelove. Cilj navedenoga je da se datoteka podijeli na više manjih dijelova, gdje je svaki je svaki dio manji od veličine *MTU*-a (*engl. maximum transmission unit*). Po određenom standardu *MTU* je najviše 1500 bajtova. Zbog toga je određeno da veličina jednog dijela bude 1400 bajtova, ostavljajući 100 bajtova za zaglavlja protokola. Ovo se radi kako bi se izbjegla fragmentaciju paketa na prvom skoku u mreži i smanjila mogućnost gubitka paketa na putu prema poslužitelju. Također, ovo omogućuje da se zapamti koju su dijelovi fotografije uspješno poslani te se nastavi sa slanjem tamo gdje se stalo u slučaju gubitka veze. Prvo je potrebno odrediti na koliko će se manjih paketa podijeliti datoteka.

```
dataPartSize := 1400
...
numDataParts := len(readFile) / dataPartSize
if len(readFile)%dataPartSize > 0 {
```

```
    numDataParts++
}
```

Zatim, kako bi se znalo spojiti dijelove datoteke na poslužitelju, potrebno je svaki dio iste datoteke označiti nekim identifikatorom. Prilikom izrade projekta, zaključeno je da je za to najbolje koristiti izračunati *SHA-256* sažetak datoteke što može poslužiti i kao provjera ispravnosti podatka na poslužitelju.

```
c.HashGenerator.Write(readFile)
calculatedHash := base64.URLEncoding.EncodeToString(c.HashGenerator.Sum(nil))
```

S ovime su dostupni svi potrebni podaci te se može ići korak dalje sa slanjem podatka.

Struktura podataka koja se koristi za komunikaciju između klijenta i poslužitelja zove se *DataPart* i serijalizira se u *JSON* format te stavlja u tijelo zahtjeva koje zatim poslužitelj čita.

- DataHash - identifikator paketa kojoj datoteci pripada (*SHA-256* sažetak)
- PartNumber - redni broj paketa, služi kako bi na odredištu znali sastaviti datoteku
- TotalParts - ukupan broj dijelova koji čine datoteku
- PartData - sami podaci toga dijela

```
type DataPart struct {
    DataHash   string `json:"dataHash"`
    PartNumber int    `json:"partNumber"`
    TotalParts int    `json:"totalParts"`
    PartData   []byte `json:"partData"`
}
```

Kako bi se poslali svi dijelovi datoteke potrebno je iterirati po varijabli *numDataParts*. Svaka iteracija uzima sljedećih *dataPartSize* bajtova i uz pomoć navedene strukture šalje na poslužitelj.

```
var bdy []byte
if partNumber == numDataParts-1 {
```

```
    bdy, err = json.Marshal(
        DataPart{
            DataHash:   fmt.Sprintf("%v", calculatedHash),
            PartNumber: partNumber + 1,
            TotalParts: numDataParts,
            PartData:   readFile[partNumber*dataPartSize:],
        },
    )
    if err != nil {
        utils.DefaultLogger.Fatalf(err, exitcodes.ExitFailedProcessingData)
    }
} else {
    bdy, err = json.Marshal(
       ...
    )
    ...
}
```

Za slanje svih dijelova datoteke koristi se ista konekcija kako bi se iskoristilo novo multipleksiranje *QUIC* protokola. Slanje se odvija u beskonačnoj petlji dok se dio uspješno ne pošalje na poslužitelja.

```
for true {
    ...
    rsp, err := c.HttpClient.Post(addr, "application/json", bytes.NewBuffer(bdy))
    ...
}
```

Za slanje svakog dijela stvara se nova Go rutina (dretva) te se cijeli proces sinkronizira sa *WaitGroup*-om iz paketa *sync* standardne biblioteke.

```
var wg sync.WaitGroup
wg.Add(numDataParts)
...
```

```
wg.Wait()
```

Ako je sve prošlo kako treba, pokretaču programa se vraća izvršni kod *ExitSuccess*.

4.3. Poslužitelj

4.3.1. API

Poslužitelj je također program napisan u Go-u i izvorni kod se također može naći na *Github*-u [3]. Pozlužitelj trenutno ima jednu zadaću, a to je obrađivanje zahtjeva koje klijent šalje naredbom *sendFile*. Pokreće se ovisno o okolini u kojoj se treba vrtiti, a omogućena je izgradnja *Docker image*-a, no može se pokrenuti i direktno iz ljuske operacijskog sustava.

Poslužitelj ima jednu otvorenu krajnju točku servera (*engl. endpoint*) */part.* što je definirano u *server/internal/api/router/router.go* datoteci.

```
...
// PART PROCESSING ENDPOINT
{
    router.POST(PartRoute, controller.ProcessPart)
}
...
```

Kontroler koji je zadužen za obradu zahtjeva tog endpoint-a nalazi se u

server/internal/api/controller/controller.go. Endpoint od parametara očekuje *Part* strukturu kao JSON unutar tijela zahtjeva.

```
type Part struct {
    DataHash   string ‘json:"dataHash"‘
    PartNumber int    ‘json:"partNumber"‘
    TotalParts int    ‘json:"totalParts"‘
    PartData   []byte ‘json:"partData"‘
}
```

[3] https://github.com/Edrudo/diplomski-rad

Ako dođe do neke greške prilikom čitanja tijela zahtjeva vraća se odgovor sa statusom *StatusInternalServerError* (500). Greška prilikom serijalizacije JSON-a u Go strukturu, vraća odgovor sa statusom *StatusBadRequest* (400).

Zatim se *DTO (engl. data transfer object)* model mapira u domenski objekt. Razlog je što su DTO ili domenski model na ovaj način podložni promjenama u budućnosti te sama domenska logika ne ovisi o komunikaciji između klijenta i poslužitelja.

```
...
part := c.serverRequestMapper.MapPartDtoToPartDomainModel(partDto)
...
```

Mapirani domenski model šalje se domenskoj komponenti *partsStoringService* pozivom metode *StorePart*. Ako poziv te metode vrati neku grešku, klijentu se vraća odgovor sa statusom *StatusInternalServerError* (500).

```
// store the part
err = c.partsStoringService.StorePart(part)
if err != nil {
    ...
    httpStatusCode = http.StatusInternalServerError
    requestContext.Writer.WriteHeader(httpStatusCode)
}
```

4.3.2. Domena

Domenska komponenta *partsStoringService* nalazi se u

server/internal/domain/services/partsStoringService.go datoteci. Njene ovisnosti su *PartsRepository* koji je zadužen spremanje i čuvanje dijelova koji su primljeni u memoriji poslužitelja i kanal prema drugom domenskom servisu koji obrađuje dijelove jedne datoteke na način da ih spaja u jednu.

```
type PartsStoringService struct {
    partsRepository            PartsRepository
    partsProcessingEngineChan chan string
```

```
}

type PartsRepository interface {
    DoesPartListExist(string) (bool, error)
    DeletePartList(string) error
    StorePart(models.Part) error
    GetNumberOfPartsInStorage(string) (int, error)
    GetPartsList(string) ([]models.Part, bool, error)
}
```

Domenska metoda *StorePart* prvo provjerava jesmo li taj dio već zaprimili i spremili u memoriju poslužitelja.

```
// check if this part already exists in stoage
parts, ok, err := i.partsRepository.GetPartsList(part.DataHash)
...
if ok {
    for _, p := range parts {
        if p.PartNumber == part.PartNumber {
            return nil
        }
    }
}
```

Ako taj dio ne postoji u memoriji, spremamo ga.

```
// otherwise store the part
err = i.partsRepository.StorePart(part)
if err != nil {
    return errctx(err)
}
```

Ako smo zaprimili sve dijelove, trebamo ih poslati na obradu *partsProcessingEngine*-u (+1 se dodaje zbog dijela kojeg smo upravo spremili u memoriju).

```
if len(parts)+1 == part.TotalParts {
    i.partsProcessingEngineChan <- part.DataHash
}
```

Nakon upisa identifikatora datoteke u kanal izlazi se uspješno iz metode, šaljući odgovor klijentu o uspješnoj obradi.

Druga domenska komponenta zove se *PartsProcessingEngine* te se nalazi u

server/internal/domain/services/partsProcessingEngine.go datoteci. Ova komponenta izvršava se paralelno u zasebnoj Go rutini. Navedeno omogućuje da se dijelovi datoteke obrađuju bez da klijent mora čekati završetak nakon što pošalje zadnji dio. Klijent dobiva odgovor da je dio uspješno stigao na odredište, a obrada svih paketa stavljena je na odgovornost poslužitelja.

Njegove ovisnosti variraju ovisno o tome što je potrebno poduzeti sa spojenom datotekom. S obzirom na to da mora moći izvući sve spremljene dijelove iz memorije, jedina konstantna ovisnost je *PartsRepository* kao i kod *partsStoringService* komponente.

Komponenta se pokreće sa *StartProcessing* metodom koja osluškuje kanal te pokreće metodu za obradu nakon što je zaprimila neku poruku u njega.

```
func (e *PartsProcessingEngine) StartProcessing() {
    for {
        select {
        case dataHash := <-e.dataHashChan:
            go e.ProcessParts(dataHash)
        }
    }
}
```

Nakon zaprimanja poruke poziva se *ProcessParts* sa identifikatorom datoteke. Prva stvar koja je potrebna jest izvući sve dijelove iz memorije. Ako se prilikom toga dogodi greška ili dijelovi s tim identifikatorom jednostavno nisu nađeni, radi se povrat iz funkcije i završava se obrada.

```
}

parts, ok, err := e.partsRepository.GetPartsList(dataHash)
if err != nil || !ok {
    return
}
```

Zatim se dijelovi stavljaju u mapu gdje je ključ redni broj dijela, a vrijednost cijeli model.

```
partsListLen := len(parts)
partNumberPartMap := getPartsMapFromList(parts)

func getPartsMapFromList(parts []models.Part) map[int]models.Part {
    partNumberPartMap := make(map[int]models.Part)
    for _, part := range parts {
        partNumberPartMap[part.PartNumber] = part
    }

    return partNumberPartMap
}
```

Finalni korak jest spajanje svih bajtova u izvornu datoteku i brisanje dijelova iz memorije.

```
func getPartsMapFromList(parts []models.Part) map[int]models.Part {
    partNumberPartMap := make(map[int]models.Part)
    for _, part := range parts {
        partNumberPartMap[part.PartNumber] = part
    }

    return partNumberPartMap
}

// delete parts from memory
```

```
err = e.partsRepository.DeletePartList(dataHash)
```

Nakon ovoga moguće je definirati što treba napraviti sa datotekom koja je generirana (spojena). Za potrebe ovog projekta zahtjev je bio da se fotografija spremi na datotečni sustav uređaja, no to je sklono promjenama u budućnosti.

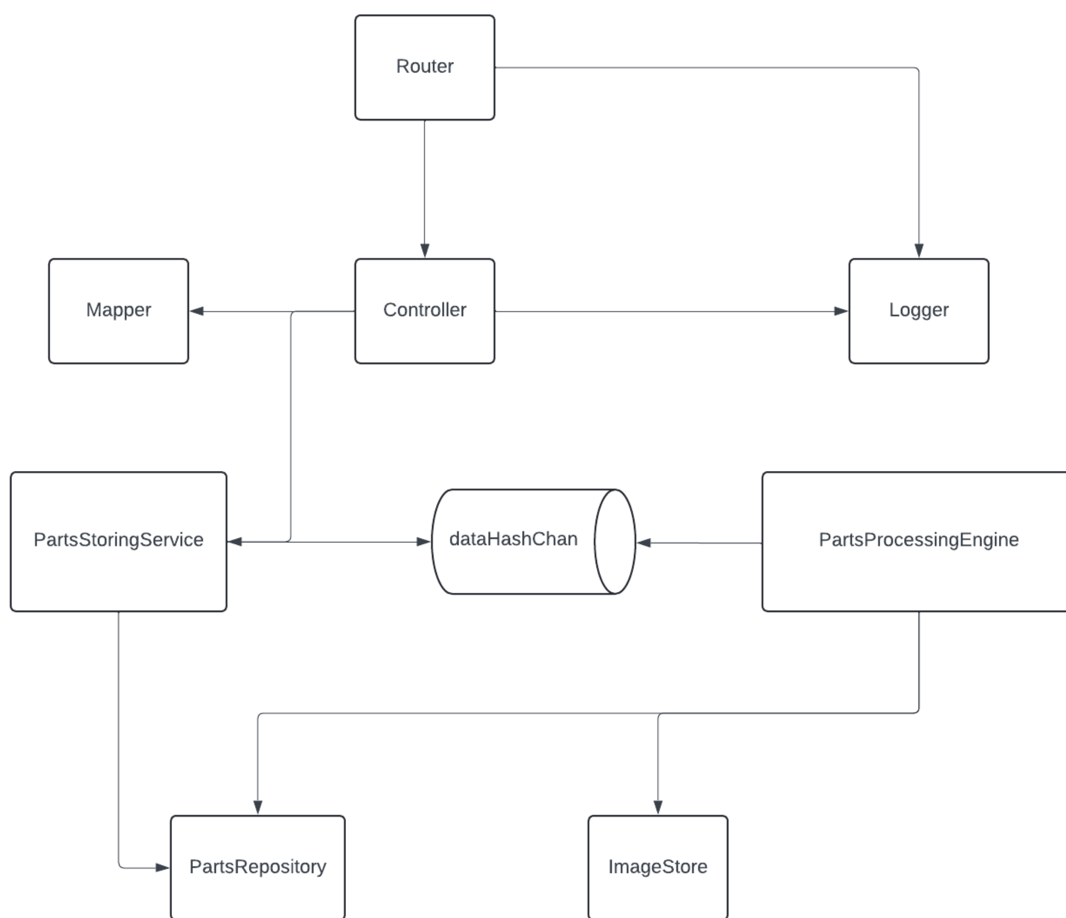
```
err = e.imageStore.StoreImage(dataHash, dataBytes)
```

Servis koji je zadužen za spremanje fotografija nalazi se u

server/internal/infrastructure/filesystem/imageStore.go datoteci.

```
filePath := fmt.Sprintf(
    "%s/%s.%s",
    config.Cfg.ImageConfig.ImageDirectory,
    imageName,
    config.Cfg.ImageConfig.ImageExtension,
)
out, err := os.Create(filePath)
...
if _, err := out.Write(imgBytes); err != nil {
...
```

S obzirom poslužitelj ima mnogo komponenti koje ovise jedne o drugoj, u nastavku se nalazi njegov dijagram ovisnosti.

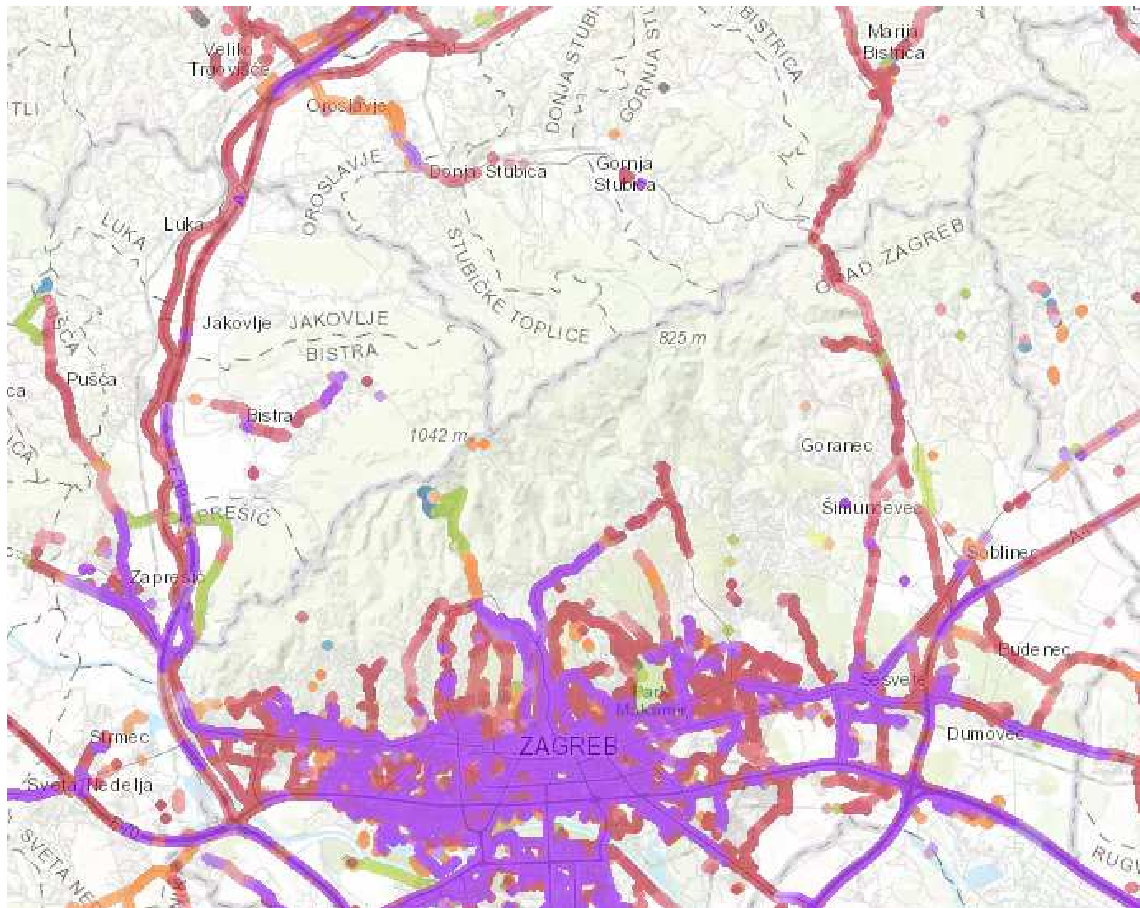


Slika 4.1. Dijagram ovisnosti poslužitelja

5. Rezultati i rasprava

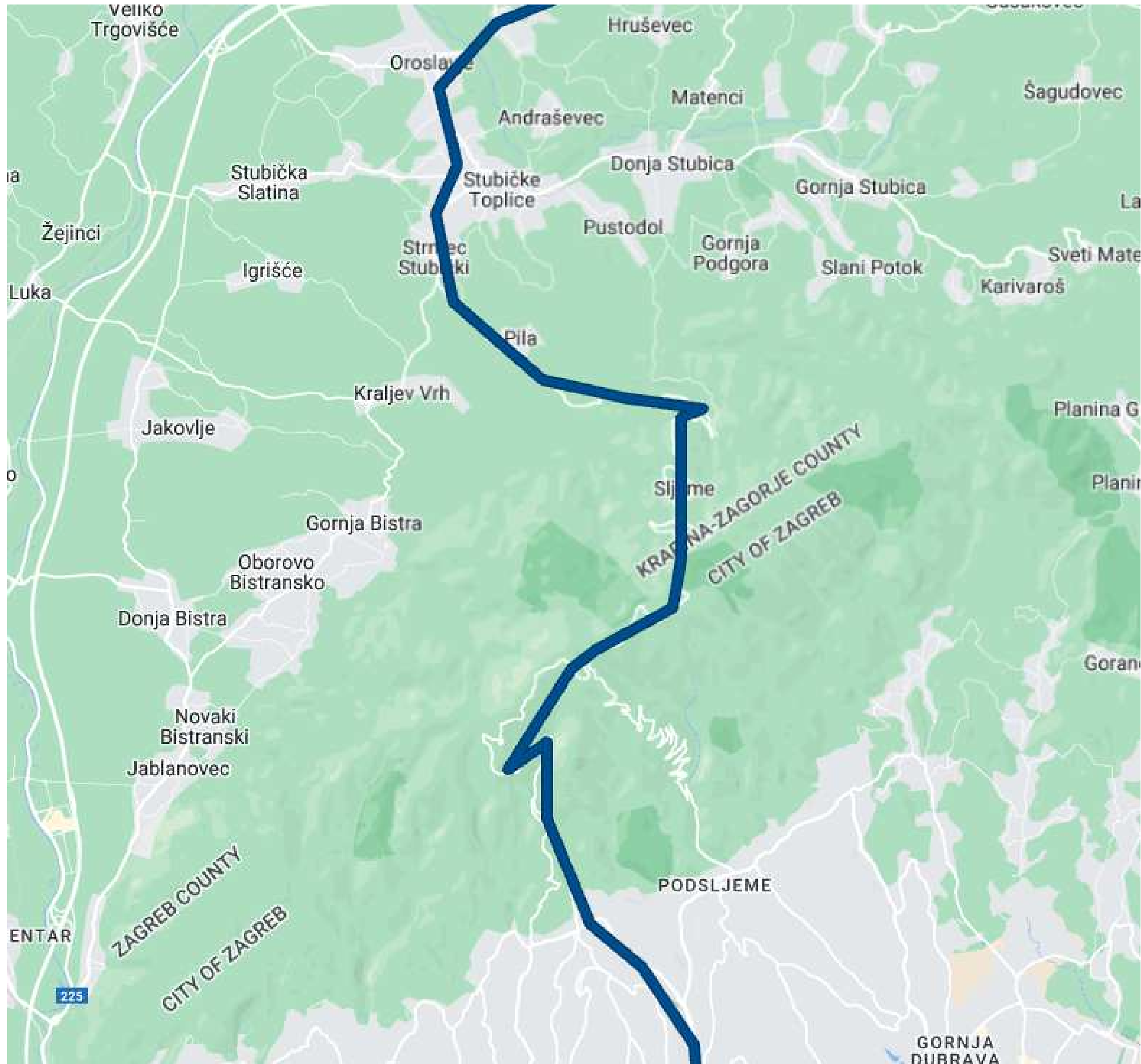
5.1. Rezultati

Projekt je bilo potrebno testirati u okolini gdje su uvjeti slični onima za koje je sustav i napravljen. Zbog toga smo testiranje odlučili napraviti vožnjom po Medvednici (planini u blizini Zagreba). Na vrhu Medvednice (Sljeme) nalazi se odašiljač mobilne mreže, no zbog konfiguracije tla dolazi do gušenja signala te zbog toga dijelovi planine nisu pokriveni mobilnom mrežmom.



Slika 5.1. Karta mrežne pokrivenosti Medvednice za operatera A1[11]

Vidimo da je pokrivenost na većini područja veoma slaba, a tamo gdje je ima prolazi cesta. Ovo su idealni uvjeti za testiranje projekta, jer ćemo vozeći auto na trenutke gubiti signal pa ga zatim dobiti na neko kratko vrijeme.



Slika 5.2. Put automobila tijekom testiranja

Konfigurirali smo klijenta da inicira slanje podataka svakih 15 sekundi te logira početak i kraj slanja. Podaci su strukturirani u *JSON* formatu gdje se u jednoj vrijednosti nalazi polje bajtova same fotografije (fotografija je veličine 238 kB), a u drugoj vrijednosti vrijeme početka slanja fotografije. Kako se vrijeme za svako slanje podataka razlikuje, kriptografski sažetak će nam uvijek biti drugačiji što nam osigurava da na poslužitelju razlikujemo različite skupove podataka te ih možemo ispravno procesirati.

Poslužitelja smo pokrenuli na AWS-ovom (*engl. Amazon Web Services*) računalu u Dublinu, on također logira svaki pristigli zahtjev te rezultat njegove obrade. Samu foto-

grafiju i cijeli JSON sprema na datotečni sustav gdje je ime datoteke vrijednost kriptografskog sažetka pripadajućeg formata.

Testiranje smo započeli krenuvši iz Donje Stubice te vozeći se do samog vrha. Na tom dijelu pokrivenosti uopće nema te su i rezultati bili takvi. Tijekom cijele vožnje koja je trajala 15-ak minuta na poslužitelj je uspješno stiglo 8 fotografija (od započetih 60). Nažalost konfiguracija samog klijenta bila je kriva te nema mogućnosti doći do njegovih logova rada. Ova vožnja do vrha poslužila je kao inicijalni test da utvrdimo što bi mogli napraviti bolje prilikom spuštanja. Zanimljiva stvar je što se u poslužiteljevim logovima vidi da neki zahtjevi nisu uspjeli u cijelosti stići do njega, javlja se greška prilikom čitanja tijela zahtjeva.

```
{"level":"error",
"ts":1719073828.7774086,
"caller":"controller/controller.go:38",
"msg":"processing part, error while reading request body",
...
[GIN] 2024/06/22 - 16:30:28 | 500 | 11.876133915s |  212.15.176.185 |
POST "/part"
```

Na južnoj strani planine, put za spuštanje s vrha ima bolju pokrivenost te su samim time i rezultati bolji. Od 50 fotografija koje je klijent krenuo slati, sve su završile na datotečnom sustavu poslužitelja.

Iz logova klijenta vidi se da započne slati nekoliko fotografija prije nego one stignu do poslužitelja, u tim trenucima je klijent izgubio vezu s mrežom. Prosječno vrijeme slanja fotografije u oblak je 39.82 sekundi.

```
2024/06/22 18:50:36 Sending image:  GmAdVWXrNmqF8Pt_un5bbds9ffMmKVY2eyQ9...
2024/06/22 18:50:41 Image sent:  GmAdVWXrNmqF8Pt_un5bbds9ffMmKVY2eyQ9ZWM...
2024/06/22 18:50:51 Sending image:  NAPQvEbSH6KVInmV64nSKaLSpNRYmzJ-4DQY...
2024/06/22 18:50:53 Image sent:  NAPQvEbSH6KVInmV64nSKaLSpNRYmzJ-4DQYBtp...
2024/06/22 18:51:06 Sending image:  _WaGa_57yZv6ug96nNUbfOBVg4crIL2ko1kG...
```

Slika 5.3. Graf trajanja slanja fotografija u oblak tokom vremena

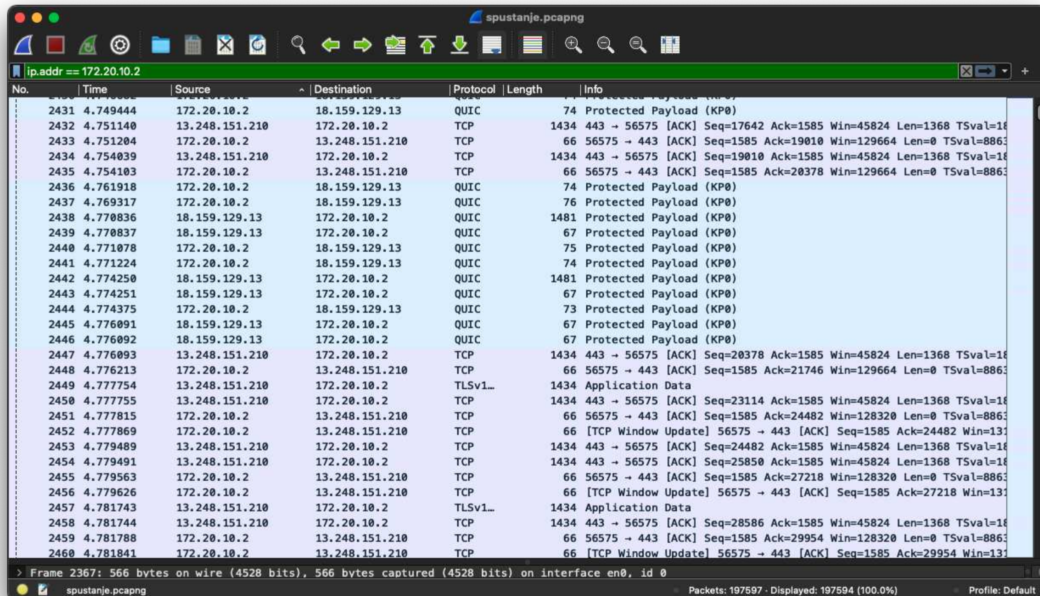
```
2024/06/22 18:51:21 Sending image:  0sip4umsPLnmlg-sV_8thzz8IdpZZ2U4uvf7...
2024/06/22 18:51:37 Sending image:  SIztlg_Z5obl0ctUDxuqD62K04rpVvBihkRy...
2024/06/22 18:51:52 Sending image:  0Rvpbe4wxyguq1g_wlU9lYn68r2Rr0PKm1rC...
2024/06/22 18:51:53 Image sent:  0Rvpbe4wxyguq1g_wlU9lYn68r2Rr0PKm1rCTEq...
2024/06/22 18:51:55 Image sent:  _WaGa_57yZv6ug96nNUbf0BVg4crIL2ko1kGuaH...
2024/06/22 18:52:00 Image sent:  0sip4umsPLnmlg-sV_8thzz8IdpZZ2U4uvf7fbs...
2024/06/22 18:52:02 Image sent:  SIztlg_Z5obl0ctUDxuqD62K04rpVvBihkRyIwX...
```

U trenutku kada je izgubljen signal, pokrenuto je snimanje mreže alatom *Wireshark*. Iz snimke može se iščitati da klijent i poslužitelj nastavljaju komunikaciju u području slabijeg signala (5.4.).

5.2. Moguća poboljšanja

5.2.1. Klijent

Trenutno klijent ima eksplicitno definiranu veličinu paketa na koju dijeli datoteku. Ovo bi se moglo optimizirati na način da se skenira mreža do prvog mrežnog čvora te se vidi koliki je njen MTU [12]. Veličina paketa bi se tada podijelila na tu vrijednost manju za veličinu zaglavlja [13].



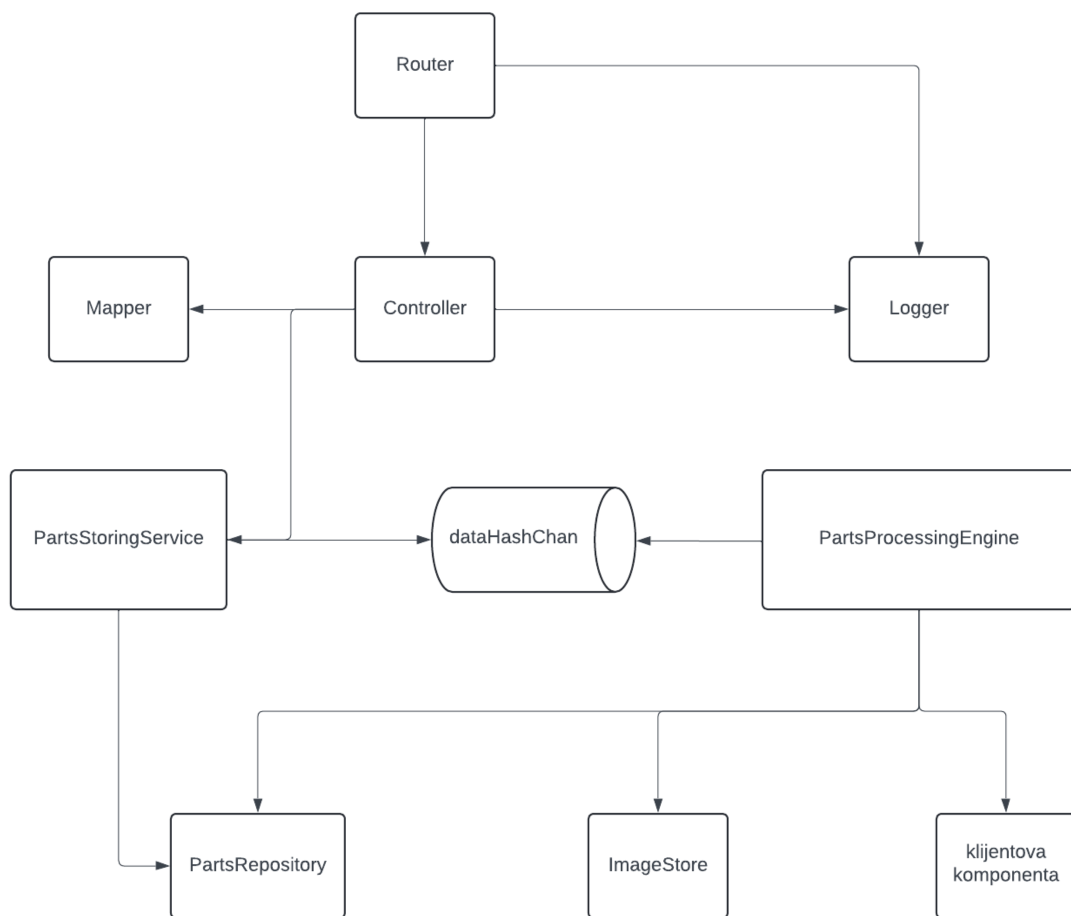
Slika 5.4. Snimka mrežnog prometa na klijentu nakon gubitka signala

Dijelovi datoteke šalju se u beskonačnoj petlji dok prijenos podataka ne uspije. Potencijalno poboljšanje jest da se u toj petlji skenira dostupnost mreže te se krene sa slanjem tek kada je uspostavljena veza.

5.2.2. Poslužitelj

Kako bi se nešto poduzelo s datotekom koja je složena na poslužitelju potrebno je to definirati unutar *processPart* metode *PartsProcessingEngine*-a. Moguće je definirati novo sučelje o kojem bi ta komponenta ovisila s metodom *ProcessFile* koja bi tu datoteku obradila. Navedena bi metoda tu datoteku obradila te bi na taj način korisnik ovog poslužitelja mogao kod sebe definirati svoju komponentu i proslijediti ju u *bootstrapu* poslužitelja..

Poslužitlelj trenutno nije skalabilan. Ako dignemo više instanci, postoji velika vjerojatnost da poslužitelj prilikom spajanja dijelova neće imati sve dijelove kod sebe. Krivac za to je što pristigle dijelove čuvamo u radnoj memoriji, a ne u nekoj bazi podataka kojoj bi svaka instanca mogla pristupiti. Rješenje je upravo to, da se spremanje pristiglih dijelova prebaci iz radne memorije u neku *key-value* bazu podataka poput *Redis*-a.



Slika 5.5. Dijagram ovisnosti poslužitelja sa klijentskom komponentom

6. Zaključak

IoT uređaji moraju raditi u različitim okruženjima, gdje je dostupnost signala bežične mreže često nepouzdana. Tradicionalni mrežni protokoli, poput TCP/IP, pokazuju ograničenja u ovakvim uvjetima zbog svoje potrebe za stabilnom i stalnom vezom te značajnim kašnjenjem u prijenosu podataka. Ovi izazovi postaju još izraženiji kada su navedeni uređaji u pokretu.

Ovaj rad izlaže važnost razvitka novih i naprednijih komunikcijskih protokola poput *QUIC*-a kako bi se postigla pouzdana komunikacija u nepovoljnim uvjetima. Koristeći *HTTP/3* sa *QUIC*-om opaženo je značajno unaprjeđenje u preformansama, pouzdanosti i sigurnosti. Ovaj rad doprinosi širokom području mrežne komunikacije, nudeći uvid i praktična rješenja nekih problema s kojima se to područje danas nosi.

Razvijeno rješenje na optimalan način iskorištava prednosti novog protokola, no ostavlja se mjesta za unaprjeđenja koje bi potencijalno još više ubrzalo preformanse. Jedna od mogućnosti je uvođenje skeniranja mreže na klijentu kako bi osigurali da ne dođe do fragmentacije paketa koji se šalju.Također, moguće je dizajnirati poslužitelja na način da on bude generalniji, tj. da se iskoristi kao komponenta u rješenju nekog drugog problema koja je zadužena za komunikaciju. Tada bi klijenti trebali misliti samo na domensku logiku svojeg sustava, dok im je sama komunikacija apstrahirana.

Testiranje pokazuje da je i s praktične strane rad zadovoljavajući. Krenuli smo s problemom da veliki udio fotografija koje klijent šalje ne stignu na odredište, no staveći rješenje u iste uvjete, sve fotografije su uspješno prenesene u oblak.

Literatura

[1] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, i T. Berners-Lee, "Hypertext transfer protocol – http/1.1", World Wide Web Consortium (W3C), 1999., accessed: 2024-06-30. [Mrežno]. Adresa: https://www.w3.org/Protocols/rfc2616/rfc2616.html

[2] J. F. Kurose i K. W. Ross, *Computer Networking: A Top-Down Approach*, 7. izd. Pearson, 2017.

[3] M. Belshe, R. Peon, i M. Thomson, "Hypertext transfer protocol version 2 (http/2)", Internet Engineering Task Force (IETF), 2015., accessed: 2024-06-30. [Mrežno]. Adresa: https://tools.ietf.org/html/rfc7540

[4] J. Postel, "Internet protocol, version 4 (ipv4) specification", RFC 791, Internet Engineering Task Force (IETF), 1981., https://tools.ietf.org/html/rfc791.

[5] M. Bishop, "Hypertext transfer protocol version 3 (http/3)", Internet Engineering Task Force (IETF), 2022., accessed: 2024-06-30. [Mrežno]. Adresa: https://tools.ietf.org/html/rfc9114

[6] D. Stenberg, *HTTP/3 Explained*. Leanpub, 2020.

[7] J. Postel, "User datagram protocol (udp)", RFC 768, Internet Engineering Task Force (IETF), 1980., https://tools.ietf.org/html/rfc768.

[8] W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[9] The Go Programming Language, "Go", 2024., accessed: 2024-06-30. [Mrežno]. Adresa: https://go.dev/

[10] J. Bodner, *Learning Go: An Idiomatic Approach to Real-World Go Programming*, 1. izd. O'Reilly Media, 2021.

[11] nPerf, "A1 mobile signal map", https://www.nperf.com/hr/map/HR/-/161490.A1-Mobile/signal?ll=45.88522863923791&lg=15.969314575195314&zoom=11, accessed: 2024-06-23.

[12] E. Blanton, M. Allman, i K. Fall, "A study of internet path mtu discovery behavior", *ACM SIGCOMM Computer Communication Review*, sv. 34, br. 4, str. 312–323, 2004.

[13] A. Woodbeck, *Network Programming with Go: Code Secure and Reliable Network Services from Scratch*. No Starch Press, 2021.

Sažetak

Pouzdana komunikacija u uvjetima loše mrežne povezanosti

Eduard Duras

U današnjem svijetu svakim danom raste količina komunikacije među uređajima koji ljudi koriste u razne svrhe. Od velike je važnosti pouzdana komunikacija, pogotovo u uvjetima gdje mrežni uvjeti nisu idealni. Ovaj rad adresirao je izazove s kojima se pritom možemo susreti te načine na koji oni mogu biti riješeni.

Uspoređivanje protokola *HTTP/2* i *HTTP/3* naglašava superiornost *HTTP/3*-a okolinama koje su karakterizirane visokim latencijama i gubitcima paketa zahvaljujući *QUIC*-ovim značajkama poput multipleksiranja, brzog oporavka i migracije konekcije. Također se naglašava važnost naprednih protokola poput *QUIC*-a za uspostavljanje pouzdane komunikacije u uvjetima loše povezanosti.

Osim korištenja novih protokola, tijekom razvoja aplikacije važno je razmišljati o načinu na koji je te protokole najbolje iskoristiti. Podjela podatka koji se šalju prema poslužitelju na više tokova i omogućujući nastavak komunikacije od točke gdje se dogodila pogreška bilo je od velike važnosti za ovaj projekt.

Ključne riječi: HTTP/3, QUIC, loša mrežna povezanost, Golang

Abstract

Reliable communication in poor network connectivity conditions

Eduard Duras

In today's world, the amount of communication between devices that people use for various purposes is growing daily. Reliable communication is of great importance, especially in conditions where network conditions are not ideal. This work addressed the challenges that can be encountered and the ways in which they can be resolved.

Comparing the HTTP/2 and HTTP/3 protocols highlights the superiority of HTTP/3 in environments characterized by high latencies and packet losses, thanks to QUIC's features such as multiplexing, fast recovery, and connection migration. This work underscores the importance of advanced protocols like QUIC in establishing reliable communication under poor connectivity conditions.

In addition to using new protocols, it is important to consider how best to utilize these protocols during the development of an application. Splitting the data sent to the server into multiple streams and allowing communication to resume from the point where an error occurred was of great importance for this project.

Keywords: HTTP/3, QUIC, bad network conditions, Golang