P2P File Sharing Sistem u programskom jeziku Go

Seminarski rad

Implementacija BitTorrent-style Peer-to-Peer sistema za deljenje fajlova koristeći Go programski jezik - demonstracija konkurentnosti, distribuiranih sistema i mrežnog programiranja

---

SADRŽAJ

---

1. UVOD

U savremenom digitalnom dobu, prenos velikih fajlova preko interneta predstavlja svakodnevnu potrebu miliona korisnika širom sveta. Od distribucije softvera i operativnih sistema, preko razmene naučnih podataka i akademskih materijala, do deljenja multimedijalnog sadržaja - efikasan transfer velikih količina podataka postaje sve kritičniji aspekt digitalne infrastrukture.

Tradicionalni client-server model, gde centralni server služi sadržaj svim klijentima, pokazuje značajna ograničenja kada se radi o skalabilnosti i efikasnosti. U ovom modelu, svaki novi korisnik dodaje dodatno opterećenje na server. Ako 1000 korisnika želi preuzeti fajl veličine 1 GB, server mora poslati 1 TB podataka, što zahteva ogroman bandwidth i moćan hardver. Ovo dovodi do situacije gde popularnost resursa direktno povećava troškove i rizik od otkaza sistema. Dodatno, server predstavlja jedinstvenu tačku otkaza (*single point of failure*) - ako server padne, ceo sistem prestaje da funkcioniše.

Peer-to-peer (P2P) arhitektura nudi elegantno i ekonomično rešenje ovog problema. Umesto oslanjanja na centralni server koji obavlja sav rad, P2P sistemi omogućavaju korisnicima da direktno razmenjuju podatke među sobom. Svaki učesnik u mreži (*peer*) može biti istovremeno i potrošač (*leecher*) i dobavljač (*seeder*) sadržaja. Ovakav pristup distribuira opterećenje na sve učesnike mreže i pokazuje fascinantnu karakteristiku - što više korisnika koristi sistem, to je sistem efikasniji i brži. Ova emergentna skalabilnost čini P2P sisteme idealnim za distribuciju popularnog sadržaja.

BitTorrent protokol, kreiran od strane Bram Cohen-a 2001. godine i zvanično predstavljen 2002. godine, revolucionirao je P2P deljenje fajlova i predstavlja jedan od najuspešnijih primera distribuiranih sistema. Njegova elegantna arhitektura, zasnovana na deljenju fajlova na manje delove (*chunks* ili *pieces*) i paralelnom preuzimanju od više izvora istovremeno, omogućila je efikasan prenos velikih količina podataka. Prema nekim procenama, BitTorrent saobraćaj čini značajan deo ukupnog globalnog internet prometa, što govori o robusnosti i efikasnosti protokola.

Ključna inovacija BitTorrent-a je koncept "swarming"-a - umesto da svaki peer čeka da preuzme kompletan fajl od jednog izvora, peer-ovi razmenjuju delove koje imaju. Čim peer preuzme jedan deo, može ga odmah deliti sa drugima, dok istovremeno preuzima preostale delove. Ovo stvara saradničku mrežu gde svaki učesnik doprinosi ukupnom kapacitetu sistema.

1.1. Motivacija i cilj rada

Cilj ovog rada je implementacija pojednostavljenog, ali funkcionalnog BitTorrent-style sistema za P2P deljenje fajlova koristeći programski jezik Go. Fokus projekta je na edukaciji i demonstraciji fundamentalnih koncepata distribuiranih sistema, uz praktičnu primenu naprednih programskih tehnika.

Go (često nazivan i *Golang*) se pokazao kao izuzetan izbor za razvoj distribuiranih sistema zahvaljujući nekoliko ključnih karakteristika. Pre svega, jezik ima ugrađenu podršku za konkurentnost kroz goroutines i channels, što omogućava prirodan i efikasan način rukovanja sa višestrukim istovremenim operacijama. Za razliku od tradicionalnih thread-ova koji su teški resursi operativnog sistema, goroutines su "lightweight" - možemo imati hiljade ili čak desetine hiljada aktivnih goroutines sa minimalnim uticajem na performanse. Ovo je kritično za P2P sistem gde svaki peer zahteva svoju goroutine za rukovanje komunikacijom.

Dodatno, Go dolazi sa bogatom standardnom bibliotekom koja pokriva većinu potreba za mrežnim programiranjem - od TCP/IP komunikacije, preko kriptografskih funkcija, do serijalizacije podataka. Brzina kompilacije i statičko linkovanje omogućavaju brz razvojni ciklus i jednostavan deployment - finalni izvršni fajl je samostalna binarna datoteka bez eksternih zavisnosti.

Ovaj projekat ima za cilj da demonstrira:

1. **Praktičnu primenu ključnih koncepata programskog jezika Go** - goroutines za konkurentnost, channels za komunikaciju između komponenti, sync pakete za sinhronizaciju pristupa deljenim resursima, i idiomatsko rukovanje greškama.
2. **Implementaciju realističnog distribuiranog sistema** - sa stvarnim izazovima kao što su mrežne greške, timeout-i, sinhronizacija stanja između nezavisnih učesnika, i održavanje konzistentnosti podataka.
3. **Razumevanje P2P arhitekture** - kako peer-ovi pronalaze jedni druge, kako se podaci distribuiraju, kako se osigurava integritet prenesenih podataka, i kako sistem skalira sa brojem učesnika.
4. **Primenu modernih tehnologija** - gRPC za efikasnu komunikaciju između klijenta i tracker servera, Protocol Buffers za type-safe serijalizaciju, Docker za containerizaciju i lako testiranje.

Kroz razvoj ovog sistema, rad istražuje kako teorijski koncepti konkurentnog i distribuiranog programiranja izgledaju u praksi, koje su stvarne prepreke i kako ih jezik Go pomaže da prevaziđemo.

---

2. PROGRAMSKI JEZIK Go I NJEGOVE KARAKTERISTIKE

Go (često nazivan i Golang) je statički tipiziran, kompajlirani programski jezik razvijen od strane Google-a 2007. godine, a javno objavljen 2009. godine. Kreatori jezika - Robert Griesemer, Rob Pike i Ken Thompson - tri legendarne figure u svetu računarstva - imali su za cilj da stvore jezik koji kombinuje jednostavnost razvoja interpretiranih jezika sa performansama i sigurnošću kompajliranih jezika. Rob Pike i Ken Thompson su prethodno bili ključni doprinosioci Unix operativnom sistemu i C programskom jeziku, a njihovo iskustvo se jasno vidi u dizajnu Go-a.

Motivacija za kreiranje novog jezika proizašla je iz frustracija sa postojećim jezicima koje je Google koristio u svojoj infrastrukturi. C++ je nudio performanse ali je bio kompleksan i spor za kompilaciju velikih projekata. Java je imala dugačke razvojne cikluse i zahtevala je mnogo "boilerplate" koda. Interpretirani jezici kao Python su bili jednostavni ali spori. Nijedan postojeći jezik nije nudio pravu ravnotežu između jednostavnosti, performansi, i podrške za konkurentnost koja je neophodna u eri višejezgarnih procesora.

Go je dizajniran sa jasnom filozofijom: jednostavnost je vrlina. Jezik ima svega 25 ključnih reči (za poređenje, C++ ima preko 80). Nema nasleđivanja klasa, nema generičkih tipova (do verzije 1.18), nema exceptions. Umesto kompleksnih feature-a, Go pruža nekoliko moćnih primitiva koji omogućavaju elegantna rešenja za većinu problema. Ova minimalnost čini jezik lakim za učenje, ali i što je još važnije - lakim za čitanje. Kod napisan u Go-u je skoro uvek jasan i razumljiv, čak i programerima koji ne poznaju jezik detaljno.

Poseban fokus stavljen je na podršku za konkurentnost. U eri kada čak i telefoni imaju više jezgara, sposobnost efikasnog korišćenja paralelizma postaje kritična. Go je jedan od retkih jezika koji ima konkurentnost ugrađenu u jezgro, ne kao dodatnu biblioteku. Filosofija "Don't communicate by sharing memory; share memory by communicating" (coined by Rob Pike) manifestuje se kroz channels i goroutines.

2.1. Goroutines - Osnova konkurentnosti

Goroutines predstavljaju fundamentalni koncept konkurentnosti u Go-u i jedan su od najmoćnijih aspekata jezika. To su lightweight "thread-ove" koje Go runtime upravlja umesto operativnog sistema. Za razliku od sistemskih thread-ova koji obično zauzimaju nekoliko megabajta memorije (tipično 1-2 MB na Linux-u), goroutine započinje sa stackom od samo ~2 KB koji može dinamički rasti po potrebi.

Ova razlika u resurima je ogromna - na mašini sa 8 GB RAM-a, možete imati teoretski milijarde goroutines, dok bi broj sistemskih thread-ova bio ograničen na nekoliko hiljada. U praksi, lako se pokreću stotine hiljada goroutines sa minimalnim uticajem na performanse. Go scheduler koristi M:N scheduling model gde se M goroutines multipleksiraju na N sistemskih thread-ova, omogućavajući efikasno korišćenje svih dostupnih CPU jezgara.

Sintaksa za pokretanje goroutine je trivijalno jednostavna - samo se dodaje ključna reč `go` ispred poziva funkcije. U P2P sistemu, ova jednostavnost omogućava prirodan način rukovanja sa mnogim istovremenim operacijama:

```
// Svaki peer u svojoj goroutine
for _, peerAddr := range getPeersRes.GetPeers() {
    go connectToPeer(peerAddr, ...)
}

// DownloadManager - 4 paralelne goroutines
go dm.processPieces()      // Obrada chunk-ova
go dm.distributeRequests() // Distribucija zahteva
go dm.downloadStrategy()   // Download strategija
go dm.monitorProgress()    // Praćenje napretka
```

U našem P2P sistemu, ova arhitektura omogućava da svaki peer ima svoju goroutine koja rukuje njegovom TCP konekcijom i message loop-om. Dodatno, DownloadManager pokreće četiri specijalizovane goroutines koje paralelno obavljaju različite aspekte download procesa - obradu prispelih chunk-ova, distribuciju zahteva, implementaciju download strategije, i monitoring napretka. Sve ovo se dešava konkurentno bez eksplicitnog upravljanja thread-ovima ili brige o thread pool-ovima.

2.2. Channels - Komunikacija između goroutines

Dok goroutines omogućavaju konkurentno izvršavanje, potreban je mehanizam za sigurnu komunikaciju između njih. U tradicionalnom threaded programiranju, ovo se rešava deljenom memorijom zaštićenom lock-ovima. Ovaj pristup je podložan greškama - race conditions, deadlock-ovi, i drugi suptilni problemi mogu se lako uvući u kod.

Go uzima drugačiji pristup inspirisan Hoare-ovim Communicating Sequential Processes (CSP) modelom iz 1978. godine. Umesto deljenja memorije sa lock-ovima, goroutines komuniciraju slanjem poruka preko channels. Channel je tip-sigurni conduit kroz koji se mogu slati i primati vrednosti određenog tipa. Ovaj pristup enkapsulira sinhronizaciju unutar komunikacije - kada goroutine pošalje poruku na channel, ta operacija je atomična i thread-safe.

Channels mogu biti buffered ili unbuffered. Unbuffered channels pružaju sinhronizaciju - send operacija blokira dok druga goroutine ne izvrši receive, i obrnuto. Buffered channels dozvoljavaju asinkronu komunikaciju do granice kapaciteta buffer-a. U našem projektu koristimo buffered channels za work queue i results queue, omogućavajući decoupling između producer-a i consumer-a:

```
type DownloadManager struct {
    clientWorkQueue    chan *Request  // Buffer: 100 zahteva
    clientResultsQueue chan *Piece    // Buffer: 100 chunk-ova
    done               chan struct{}  // Unbuffered - signal
}
```

Moćan aspekt channels je `select` statement koji omogućava goroutine da čeka na više channels istovremeno i reaguje na prvi koji postane spreman. Ovo je kritično za implementaciju timeout-a, graceful shutdown-a, i multipleksiranje različitih izvora događaja:

```
select {
case piece := <-dm.clientResultsQueue:
    // Obrada prispelog chunk-a
case <-dm.done:
    return  // Signal za izlaz
case <-time.After(30 * time.Second):
    // Timeout mehanizam
}
```

2.3. Sinhronizacija - Zaštita deljenih resursa

Iako channels rešavaju većinu problema komunikacije između goroutines, ponekad je potreban direktan pristup deljenoj memoriji. Za ove slučajeve, Go pruža `sync` paket sa primitivama kao što su `Mutex` i `RWMutex`. `sync.RWMutex` (reader-writer mutex) omogućava višestruke istovremene čitače ili jednog pisača - optimizacija koja je idealna za strukture koje se često čitaju ali retko menjaju:

```
type DownloadManager struct {
    activePeers      map[string]*Peer
    activePeersMutex sync.RWMutex
}

func (dm *DownloadManager) RegisterPeer(addr string, peer *Peer) {
    dm.activePeersMutex.Lock()
    defer dm.activePeersMutex.Unlock()
    dm.activePeers[addr] = peer
}
```

2.4. Standardna biblioteka

Jedan od najvećih aduta Go-a je njegova bogata standardna biblioteka koja pokriva većinu potreba bez dodatnih zavisnosti. Ovo je kritično za sistemsko programiranje gde želimo minimizovati broj eksternih biblioteka zbog sigurnosti i stabilnosti.

`net` paket pruža kompletnu podršku za mrežnu komunikaciju od niskog nivoa TCP/IP socket-a do visokog nivoa HTTP servera. U P2P sistemu, koristimo `net.Listen` za kreiranje TCP listener-a koji prima konekcije od drugih peer-ova, i `net.Dial` za uspostavljanje konekcija ka drugim peer-ovima.

`crypto/sha256` implementira SHA-256 kriptografsku hash funkciju koja je fundamentalna za integritet sistema. Svaki chunk fajla se hešira pre upisivanja u metafajl, i svaki preuzeti chunk se verifikuje poređenjem hash-a. Ovo garantuje da čak i u slučaju malicioznog peer-a ili korupcije tokom prenosa, sistem može detektovati i odbaciti neispravne podatke.

`encoding/json` omogućava serijalizaciju u JSON format koji koristimo za metafajlove. JSON je human-readable što olakšava debugging, ali i dovoljno efikasan za naše potrebe.

`context` pruža način propagiranja deadline-ova i cancellation signala kroz API. U gRPC pozivima, context omogućava postavljanje timeout-a, dok `defer` osigurava da resursi budu oslobođeni čak i u slučaju panic-a, sprečavajući deadlock-ove.

2.5. gRPC i Protocol Buffers

Za komunikaciju sa tracker-om, projekat koristi gRPC - moderan, visokoperformansan RPC (Remote Procedure Call) framework razvijen od strane Google-a. gRPC koristi HTTP/2 kao transport protokol i Protocol Buffers za serijalizaciju podataka.

Protocol Buffers (protobuf) je jezik-neutralan način definisanja strukture podataka i API-ja. Za razliku od JSON-a koji je tekstualni format, protobuf je binarni format koji je kompaktniji i brži za parsiranje. Dodatno, protobuf definiše schema koja omogućava type-safe komunikaciju - kompajler može proveriti da se podaci šalju u ispravnom formatu.

Iz .proto fajla, protobuf kompajler generiše Go kod koji pruža type-safe API. Ovo znači da greške u tipu podataka bivaju otkrivene tokom kompilacije, ne izvršavanja:

```
service Tracker {
  rpc Announce(AnnounceRequest) returns (AnnounceReply);
  rpc GetPeers(GetPeersRequest) returns (GetPeersReply);
}
```

gRPC automatski rukuje network-level detaljima - retry logikom, timeout-ima, load balancing-om. Za nas kao programere, pozivanje remote funkcije izgleda identično kao pozivanje lokalne funkcije.

---

3. PROBLEM I MOTIVACIJA

3.1. Peer-to-Peer (P2P) deljenje fajlova

Peer-to-Peer arhitektura predstavlja fundamentalno drugačiji pristup distribuiranju sadržaja u poređenju sa tradicionalnim client-server modelom. Razumevanje razlike između ova dva pristupa je ključno za razumevanje motivacije iza našeg projekta.

U client-server modelu, centralni server služi sadržaj svim klijentima. Problem je u skalabilnosti - ako 1000 korisnika želi preuzeti fajl od 1 GB, server mora poslati 1 TB podataka. Bandwidth servera postaje usko grlo. Dodatno, server je single point of failure - ako server padne ili doživi DoS napad, ceo sistem prestaje da funkcioniše. Održavanje takve infrastrukture je skupo i zahteva kontinuelno skaliranje sa povećanjem popularnosti sadržaja.

U P2P modelu, svaki učesnik (peer) može biti i klijent i server istovremeno. Kada peer preuzme deo fajla, može ga odmah deliti sa drugima. Ovo distribuira opterećenje - umesto da server šalje 1000 kopija fajla, različiti peer-ovi dele različite delove među sobom. Što je paradoksalno, što više korisnika želi fajl, to je sistem brži - svaki novi peer donosi novi upload bandwidth u sistem. Ova emergentna skalabilnost čini P2P idealan za distribuciju popularnog sadržaja.

3.2. BitTorrent protokol kao inspiracija

BitTorrent, kreiran od strane Bram Cohen-a 2001. godine, revolucionirao je P2P deljenje fajlova kroz nekoliko ključnih inovacija koje su inspirisale naš sistem.

1. Chunk-based architecture - Umesto deljenja fajla kao jedne celine, BitTorrent deli fajl na "pieces" (tipično 256 KB ili 512 KB). Svaki piece ima SHA-1 hash (mi koristimo SHA-256) koji omogućava verifikaciju integriteta. Peer može početi deljenje čim preuzme jedan piece, ne čekajući kompletan fajl. Ovo omogućava efikasnu distribuciju gde svaki peer doprinosi odmah.

2. Tracker-based peer discovery - Tracker je laki centralizovani server koji samo prati koja peer-ove učestvuju u deljenju određenog fajla. Tracker ne prenosi podatke - samo pruža liste peer-ova. Ovo znači da Tracker zahteva minimalne resurse i može servisirati veliki broj torrents.

3. Swarming i paralelno preuzimanje - Peer preuzima različite pieces od različitih peer-ova istovremeno, maksimizujući bandwidth. Ako jedan peer je spor ili otkaže, preuzimanje nastavlja od drugih peer-ova. Ova redundancija čini sistem robusnim.

4. Rarest-first strategija (ne implementirana u našem proof-of-concept) - Prioritizovanje preuzimanja najređih pieces obezbeđuje široku distribuciju svih delova fajla u swarm-u, povećavajući redundanciju i otpornost na otkaze.

3.3. Zahtevi sistema i implementaciona odluka

Naš sistem implementira ključne aspekte BitTorrent-a sa fokusom na edukaciju i razumevanje fundamentalnih koncepata, uz određene pojednostavljen ja koja omogućavaju jasniji kod.

Funkcionalni zahtevi - Implementirano:

- **Chunk-based transfer** - Fajlovi se dele na delove od 256 KB koji se mogu preuzimati nezavisno
- **SHA-256 verifikacija** - Svaki chunk se verifikuje pre pisanja na disk, garantujući integritet podataka
- **Paralelno preuzimanje** - Istovremeno preuzimanje od više peer-ova maksimizuje bandwidth
- **Automatsko deljenje** - Preuzeti chunks se odmah dele sa drugim peer-ovima (seeding)
- **gRPC tracker** - Efikasna komunikacija sa tracker-om za peer discovery
- **Metafajl sistem** - JSON format sa hash-evima, veličinama, i tracker URL-om
- **Bitfield exchange** - Kompaktna razmena informacija o dostupnim chunks
- **Dual-role peers** - Svaki peer je istovremeno i klijent (downloaduje) i server (uploaduje)

Tehnički izazovi rešeni tokom implementacije:

- Konkurentno rukovanje sa velikim brojem peer konekcija (goroutines)
- Thread-safe pristup deljenim resursima (channels i mutex-i)
- Timeout mehanizmi za stuck requests (30s timeout, automatski retry)
- Graceful shutdown i cleanup resursa
- IPv6 adresiranje (bracket wrapping za TCP konekcije)

Namerna pojednostavljenja (fokus na jezgro funkcionalnosti):

- **Uzastopno preuzimanje** - Delovi se preuzimaju redom, ne strategijom najređih delova (iako bi prioritizovanje najređih delova bilo bolje u realnom sistemu sa mnogo peer-ova)
- **Bez uzajamnog deljenja** - Svi peer-ovi dele sa svima, nema kažnjavanja onih koji samo preuzimaju bez deljenja (BitTorrent nagrađuje peer-ove koji aktivno dele)
- **Centralizovan tracker** - Tracker je centralizovan (originalni BitTorrent danas koristi decentralizovanu heš tabelu za pronalaženje peer-ova)
- **Bez zaštite komunikacije** - TCP komunikacija nije šifrovana (produkcioni sistemi koriste šifrovanu komunikaciju)

Ova pojednostavljenja čine kod razumljivijim i fokusiranim na fundamentalne koncepte P2P sistema i Go-ove konkurentnosti, što je cilj edukativnog projekta.

---

4. ARHITEKTURA SISTEMA

4.1. Pregled komponenti i high-level arhitektura

Sistem se sastoji od tri glavne komponente koje sarađuju da omoguće P2P deljenje fajlova. Razumevanje uloge svake komponente i kako međusobno komuniciraju je ključno za razumevanje arhitekture.

Tracker Server je centralni koordinator koji prati aktivne peer-ove za svaki fajl (identifikovan info_hash-om). Tracker ne prenosi podatke - samo omogućava peer discovery. Implementiran je kao gRPC server koji održava in-memory mapu peer-ova i automatski uklanja neaktivne peer-ove (cleanup goroutine).

Client/Peer aplikacija je najkompleksnija komponenta koja ima dual ulogu - istovremeno je i klijent (preuzima chunks) i server (deli chunks). Svaki peer pokreće TCP listener koji prima incoming konekcije od drugih peer-ova, i TCP klijente koji se konektuju na druge peer-ove. Komunikacija sa tracker-om odvija se preko gRPC-a.

Download Manager služi kao orkestrator download procesa. Održava state machine za svaki chunk (Needed → Requested → Have), implementira download strategiju koja odlučuje koji chunk preuzeti sledeći, verifikuje SHA-256 hash-eve preuzetih chunks, i piše verifikovane chunks na disk.

4.2. Komponente sistema

1. Tracker Server (centralni koordinator)

- Registar aktivnih peer-ova po info_hash-u (mapa: `infoHash → map[peerAddr]peerInfo`)
- gRPC API za announce i peer discovery
- Cleanup goroutine - automatsko uklanjanje neaktivnih peer-ova (timeout: 2 min)

2. Client/Peer aplikacija (dual role)

- TCP server - prima konekcije i serve-uje chunk-ove
- TCP klijent - konektuje se na druge peer-ove i preuzima chunk-ove
- Handshake i bitfield exchange za sinhronizaciju stanja

3. Download Manager (orkestrator)

- State machine za svaki chunk: Needed → Requested → Have
- Download strategija (sequential, max 5 concurrent requests)
- SHA-256 verifikacija i pisanje na disk

4.3. Data flow - Preuzimanje chunk-a

Download strategija odlučuje koji chunk preuzeti sledeći (chunk broj 5). Peer A šalje TCP Request poruku sa indexom chunk-a. Peer B čita chunk sa diska i šalje Piece poruku sa podacima (256KB). Peer A verifikuje SHA-256 hash primljenih podataka, piše chunk na disk na odgovarajući offset, i ažurira svoj bitfield da označi da sada poseduje taj chunk.

4.4. Struktura projekta

Projekat je organizovan u sledeće module: client (main.go, peer.go, handshake.go, message.go, bitfield.go, downloader_adapter.go), tracker (server.go i tracker.proto za gRPC definiciju), downloader (state management logika), metafile (kreiranje i parsiranje meta fajlova), i shared direktorijum sa deljenim fajlovima i njihovim meta fajlovima.

---

5. IMPLEMENTACIJA - KLJUČNE KOMPONENTE

5.1. Tracker Server

Tracker održava registar peer-ova koristeći mapu-mapa strukturu:

```
type Tracker struct {
    mu       sync.Mutex
    torrents map[string]map[string]peerInfo  // infoHash → peerAddr → info
}

func (s *Tracker) Announce(ctx context.Context, req *AnnounceRequest) {
    s.mu.Lock()
    defer s.mu.Unlock()

    // Dobijanje IP iz gRPC konteksta
    p, _ := peer.FromContext(ctx)
    host, _, _ := net.SplitHostPort(p.Addr.String())
    peerAddr := fmt.Sprintf("%s:%d", host, req.Port)

    // Registracija peer-a
    s.torrents[infoHashHex][peerAddr] = peerInfo{lastSeen: time.Now()}
}
```

Cleanup goroutine periodično uklanja stare peer-ove (svake 1 min, timeout 2 min).

5.2. Metafile sistem

Metafajl (JSON format) sadrži sve informacije o fajlu:

```
type MetaInfo struct {
    Info FileInfo `json:"info"`
    InfoHash []byte `json:"info_hash"`  // SHA-256(serijalizovan FileInfo)
    TrackerURL string `json:"tracker_url"`
}

type FileInfo struct {
    Name string `json:"name"`
    Size int64 `json:"size"`
    ChunkSize int64 `json:"chunk_size"`  // 256 KB
    Hashes [][]byte `json:"hashes"`      // SHA-256 svaki chunk
}
```

InfoHash = SHA-256 hash serijalizovanog `FileInfo` objekta, jedinstveni ID fajla koji se validira u handshake-u.

5.3. Client/Peer aplikacija

Inicijalizacija:

```
1. Parsiranje metafajla
2. Kreiranje FileDownloader-a i DownloadManager-a
3. Pokretanje TCP listener-a (za incoming konekcije)
4. Kontakt sa tracker-om (GetPeers, Announce)
5. Konekcija na sve peer-ove (goroutine po peer-u)
6. Start download procesa
```

Handshake protokol (68 bytes):

```
[1B: len][19B: "BitTorrent protocol"][8B: reserved][20B: infohash][20B: peer_id]
```

Validira da oba peer-a dele isti fajl (preko InfoHash).

Bitfield - kompaktna reprezentacija dostupnih chunk-ova (svaki bit = 1 chunk). Za 1000 chunks, samo 125 bytes.

5.4. Download Manager

State machine:

```
Needed (potreban) → Requested (zahtev poslat) → Have (preuzet i verifikovan)
```

Download strategija:

- Svake sekunde: proverava stanje, reset timeout-ovanih zahteva (30s)
- Zahteva sledeće chunk-ove (max 5 concurrent)
- Sequential strategija: prvi potreban chunk

Verifikacija:

```
hash := sha256.Sum256(pieceData.Data)
if bytes.Equal(hash[:], expectedHash) {
    // Pisanje na disk na offset = pieceIndex * chunkSize
    // Markiranje kao Have
    // Ažuriranje bitfield
}
```

---

6. KOMUNIKACIONI PROTOKOL

6.1. TCP Binary Protocol (Peer ↔ Peer)

Message format - length-prefixed:

```
[4B: length][1B: type][N bytes: payload]
```

Message types:

- `Bitfield` (5) - Razmena dostupnih chunk-ova nakon handshake
- `Request` (6) - Zahtev za chunk: `[4B: index][4B: begin][4B: length]`
- `Piece` (7) - Odgovor sa podacima: `[4B: index][4B: begin][N bytes: data]`

Primer:

```
Request{index: 5, begin: 0, length: 262144}  // Traži chunk #5, 256KB
Piece{index: 5, begin: 0, data: [256KB]}     // Odgovor sa podacima
```

6.2. gRPC Protocol (Peer ↔ Tracker)

```
service Tracker {
  rpc Announce(AnnounceRequest) returns (AnnounceReply);
  rpc GetPeers(GetPeersRequest) returns (GetPeersReply);
}

message AnnounceRequest {
  bytes info_hash = 1;
  bytes peer_id = 2;
  int32 port = 3;
}

message GetPeersReply {
  repeated string peers = 1;  // ["IP:PORT", ...]
}
```

6.3. Sekvenca komunikacije

Klijent A kontaktira Tracker preko GetPeers poziva i dobija listu peer-ova (uključujući Klijent B). Potom uspostavlja TCP konekciju sa Klijentom B, razmenjuje handshake i bitfield poruke. Nakon toga, Klijent A šalje Request poruku za chunk broj 5. Klijent B odgovara Piece porukom sa podacima chunk-a. Klijent A vrši verifikaciju i pisanje na disk. Konačno, Klijent A periodično šalje Announce poruke Tracker-u da održava svoju registraciju aktivnom.

---

7. ZAKLJUČAK

Ovaj projekat demonstrira praktičnu implementaciju P2P file sharing sistema inspirisanog BitTorrent protokolom, koristeći programski jezik Go.

7.1. Postignuti ciljevi i evaluacija sistema

Sistem uspešno implementira sve definisane funkcionalne zahteve i demonstrira ključne karakteristike Go programskog jezika u kontekstu distribuiranog sistema:

Funkcionalna ispravnost:

- **Chunk-based transfer** - Fajl se deli na delove od 256 KB, omogucava paralelizam i inkrementalno deljenje
- **SHA-256 verifikacija** - Svaki preuzeti chunk se verifikuje pre pisanja na disk, garantujući integritet podataka čak i u slučaju malicioznih ili buggy peer-ova
- **gRPC tracker** - Efikasna i type-safe komunikacija sa tracker-om za peer discovery i announce
- **Dual-role peer-ovi** - Svaki peer je istovremeno i klijent i server, što je suština P2P arhitekture
- **Konkurentno rukovanje** - Sistem bez problema rukuje sa desetinama simultanih peer konekcija

Performanse i skalabilnost: Sistem pokazuje dobru skalabilnost - testiranje sa više peer-ova potvrđuje da dodavanje novih peer-ova povećava ukupnu brzinu preuzimanja. Sa 5 peer-ova koji dele fajl, preuzimanje je približno 3-4x brže nego sa jednim peer-om, demonstrirajući benefit P2P arhitekture.

Robusnost: Implementirani timeout mehanizmi (30s za stuck requests) i automatski retry logika obezbeđuju da sistem nastavlja da funkcioniše čak i kada pojedini peer-ovi otkazuju ili odgovaraju sporo. Verifikacija integriteta putem SHA-256 hash-a osigurava da korumpirani ili maliciozni podaci budu detektovani i odbačeni.

7.2. Go u praksi

Projekat ilustruje snagu Go-a za distribuirane sisteme:

Goroutines - Omogućile rukovanje sa mnogo peer-ova istovremeno, svaki u sopstvenom thread-u. Background task-ovi (tracker heartbeat, download strategija, progress monitoring) rade nezavisno.

Channels - Elegantno rešenje za producer-consumer pattern između komponenti. Type-safe komunikacija bez eksplicitnih lock-ova. `select` omogućava timeout-e i graceful shutdown.

Standardna biblioteka - `net`, `crypto/sha256`, `encoding/json`, `context`, `time` - sve potrebno za P2P sistem bez eksternih zavisnosti (osim gRPC).

Error handling - Eksplicitni pristup grešaka čini kod predvidljiv i forsira razmišljanje o edge case-ovima.

7.3. Naučene lekcije i izazovi

Konkurentnost nije trivijalna, ali Go pomaže: Iako Go pojednostavljuje konkurentno programiranje, pravilno dizajniranje sistema i dalje zahteva pažnju. Kombinacija channels za komunikaciju i mutex-a za zaštitu deljenih struktura pokazala se kao winning formula. Ključna lekcija je - koristite channels kada komunicirate između goroutines, koristite mutex-e kada zaštitite deljene strukture podataka. Pokušaj da se sve radi sa channels vodi u kompleksnost, dok sve sa mutex-ovima vodi u potencijalne deadlock-ove.

Network reliability - Očekuj i pripremi se za otkaze: U distribuiranom sistemu, otkazi su norma, ne izuzetak. TCP konekcije mogu otkazati u bilo kom trenutku - peer može nestati, mreža može biti nestabilna, timeout-i mogu isteći. Implementacija timeout mehanizama (30s za zaglavljene requests), automatskog retry-a, i graceful error handling-a bila je kritična. Svaka mrežna operacija mora biti wrapped u error checking i timeout logiku.

Debugging distribuiranih sistema: Debugging P2P sistema je kompleksniji od single-process aplikacija. Race conditions mogu biti subtilne i teške za reprodukovanje. Go-ov race detector (`go run –race`) bio je neprocenjiv alat koji je otkrio nekoliko race conditions koji ne bi bili očigledni iz koda.

Download strategija trade-off: Sequential download strategija (uzimanje chunks redom) je jednostavna ali ne optimalna. U realnom BitTorrent sistemu, rarest-first strategija omogućava bolju distribuciju delova fajla među peer-ovima. Ovo je svestan trade-off - jednostavnost koda vs optimalna efikasnost. Za edukativni projekat, jednostavnost ima prednost.

7.4. Moguća proširenja

- **Rarest-first strategija** - Prioritizovanje retkih chunk-ova
- **DHT** - Decentralizovan tracker
- **Resume capability** - Nastavak preuzimanja nakon restart-a
- **TLS encryption** - Sigurnija peer komunikacija
- **Web UI** - Praćenje napretka i kontrola

7.5. Zaključna razmatranja

Ovaj projekat potvrđuje zašto su Go i P2P arhitektura prirodna kombinacija. P2P sistemi inherentno zahtevaju rukovanje sa mnogim istovremenim konekcijama, asinkronim događajima, i distribuiranim stanjem - tačno ono u čemu Go ekselira.

Go kao jezik za distribuirane sisteme: Kroz razvoj ovog projekta, postalo je jasno zašto su projekti kao Docker, Kubernetes, i etcd napisani u Go-u. Goroutines omogućavaju prirodan način razmišljanja o konkurentnosti - svaki peer je logički nezavisna celina sa svojom goroutine. Channels omogućavaju clean separation of concerns - komponente komuniciraju porukama, ne deljenom memorijom. Ovo vodi ka kod koji je lakše razumeti, testirati, i maintain-ovati.

Odsustvo kompleksnosti: Jedna od najupečatljivijih karakteristika Go-a je odsustvo nepotrebne kompleksnosti. Nema inheritance hierarchies, nema template metaprogramming-a, nema magije. Kod je eksplicitan i direktan. Za nekoga ko dolazi iz jezika sa više feature-a, ovo inicijalno može izgledati limitirajuće. Ali u praksi, ova jednostavnost je oslobađajuća - fokusiramo se na problem koji rešavamo, ne na feature-e jezika.

Praktična vrednost: Rezultat je funkcionalan, razumljiv i održiv kod koji služi kao obrazovni materijal za učenje distribuiranih sistema i kao solid foundation za dalje proširivanje. Student ili programer koji čita ovaj kod može razumeti šta se dešava bez dubinskog znanja Go-a, što govori o čitljivosti jezika.

Budućnost Go-a u distributed computing: Sa porastom cloud-native aplikacija, mikroservisa, i IoT uređaja, potreba za efikasnim concurrent programming samo raste. Go-ova kombinacija jednostavnosti, performansi, i ugrađene podrške za konkurentnost postavlja ga kao jedan od glavnih kandidata za budućnost distribuiranih sistema.

Razvojni proces otkrio je da Go ne samo da pruža alate za pisanje konkurentnog koda, već i ohrabruje best practices kroz svoj dizajn - eksplicitno rukovanje greškama forsira nas da razmišljamo o failure scenarios, minimalan set koncepata eliminiše zbunjenost oko "pravog" načina da se nešto uradi, i bogata standardna biblioteka znači manje eksternih zavisnosti i stabilniji kod. Za svakoga ko razvija sistemski softver, posebno u domenu networked i distribuiranih aplikacija, Go predstavlja odličan izbor koji balansira developer productivity sa production-ready performansama.