UNIVERZITET
U NOVOM SADU
**FAKULTET TEHNIČKIH
NAUKA U NOVOM SADU**

Nikola Tomašević
Vladislav Petković

Load Balancer

Projekat

Osnovne akademske studije

Novi Sad

Sadržaj

1. Uvod

U svetu modernih softverskih sistema, zahtevi za skalabilnošću, pouzdanošću i visokom dostupnošću su postali imperativ. Aplikacije koje opslužuju veliki broj korisnika moraju biti u stanju da efikasno obrade i skladište ogromne količine podataka bez gubitka performansi. Jedan od ključnih izazova u dizajnu takvih sistema je izbegavanje centralizovanih tačaka otkaza i ravnomerno raspoređivanje opterećenja na sve raspoložive resurse.

Ovaj projekat se bavi razvojem servisa za skladištenje podataka koji je dizajniran upravo sa ovim principima na umu. Cilj projekta je bio kreirati distribuirani sistem koji se sastoji od jedne centralne komponente za upravljanje, **Load Balancer** (LB), i proizvoljnog broja radnih komponenti, **Worker** (WR). Load Balancer ima zadatak da prima zahteve od klijenata i inteligentno ih prosleđuje Workeru koji je u datom trenutku najmanje opterećen. Sa druge strane, Workeri su zaduženi za skladištenje podataka i, što je najvažnije, za međusobnu sinhronizaciju kako bi se osigurala konzistentnost i redudantnost podataka unutar celog sistema.

Ovaj dokument detaljno opisuje arhitekturu razvijenog rešenja, korišćene tehnologije i alate, kao i konkretne mehanizme koji omogućavaju balansiranje opterećenja, dinamičko dodavanje novih radnih komponenti i peer-to-peer replikaciju podataka. Kroz analizu ključnih delova koda i praktičan primer rada sistema, biće prikazano kako implementirano rešenje ispunjava sve postavljene zahteve, kreirajući robustan i efikasan servis za skladištenje podataka.

2. Opis korišćenih tehnologija i alata

Za realizaciju ovog projekta izabrane su fundamentalne tehnologije koje omogućavaju direktnu kontrolu nad mrežnom komunikacijom, radom sa memorijom i višenitnim izvršavanjem. Ovakav pristup, iako zahtevniji, pruža dublji uvid u mehanizme koji pokreću složene distribuirane sisteme i omogućava optimizaciju performansi na niskom nivou.

2.1. Programski jezik C

Programski jezik C je izabran kao osnova za implementaciju celog sistema zbog svojih karakteristika koje su idealne za sistemsko programiranje. Glavne prednosti koje su bile presudne za ovaj projekat su:

- **Performanse:** C nudi izuzetno visoke performanse jer se direktno kompajlira u mašinski kod, bez dodatnog sloja apstrakcije kao što je virtuelna mašina. U sistemu gde su brzina odziva i efikasna obrada mrežnih paketa od ključnog značaja, ovo je velika prednost.
- **Upravljanje memorijom:** Mogućnost manuelnog upravljanja memorijom (`malloc`, `free`) omogućava preciznu kontrolu nad resursima. U ovom projektu, to je iskorišćeno za dinamičku alokaciju memorije za strukture podataka kao što su lista radnika (Workera) i *hash* mapa.
- **Prenosivost i blizina hardveru:** C omogućava pisanje koda koji je blisko povezan sa operativnim sistemom, što je neophodno za direktan rad sa soketima i nitima (eng. *threads*).

2.2. Winsock 2 (Windows Sockets API)

Za realizaciju mrežne komunikacije između klijenta, Load Balancera i Workera, korišćen je **Winsock 2** API. Winsock je standardni interfejs za mrežno programiranje na Windows platformi i predstavlja implementaciju Berklijevih soketa.

Ključne funkcionalnosti koje su korišćene:

- **TCP/IP Soketi:** Projekat koristi TCP (*Transmission Control Protocol*) sokete (SOCK_STREAM) kako bi se osigurala pouzdana i uspostavljena veza za prenos podataka. TCP garantuje da će podaci stići u onom redosledu u kom su poslati i bez grešaka, što je fundamentalno za integritet podataka koji se skladište.
- **Blokirajuće i neblokirajuće operacije:** Korišćene su standardne blokirajuće operacije poput accept() i recv(). U naprednijoj komponenti, Workeru, korišćena je funkcija select() koja omogućava neblokirajuće praćenje više soketa istovremeno. Ovo je bilo presudno za implementaciju peer-to-peer komunikacije, gde Worker mora istovremeno da sluša i poruke od Load Balancera i od drugih Workera.
- **Funkcije za rad sa soketima:** Standardne funkcije kao što su socket(), bind(), listen(), accept(), connect(), send() i recv() čine osnovu mrežne logike u svim komponentama sistema.

2.3. Windows API za višenitno programiranje

Distribuirana priroda sistema zahteva istovremeno izvršavanje više operacija. Load Balancer mora biti u stanju da opsluži više klijenata istovremeno, dok svaki Worker radi kao nezavisna, autonomna jedinica. Ovo je postignuto korišćenjem Windows API-ja za rad sa nitima i sinhronizaciju.

- **Kreiranje niti (CreateThread):** Svaki Worker u sistemu se izvršava u svojoj zasebnoj niti, koju kreira i kojom upravlja Load Balancer. Takođe, za svakog klijenta koji se poveže na Load Balancer, kreira se posebna nit za obradu njegovih zahteva. Ovo omogućava da sistem ostane responzivan i da opslužuje više zadataka paralelno.
- **Mehanizmi za sinhronizaciju (CRITICAL_SECTION):** Pošto više niti (npr. niti koje opslužuju klijente i nit koja dodaje novog Workera) pristupa deljenim resursima, kao što je globalni niz workers, neophodno je osigurati bezbedan pristup tim resursima. Za to su korišćene kritične sekcije. InitializeCriticalSection, EnterCriticalSection i LeaveCriticalSection fun kcije se koriste da bi se sprečilo stanje trke (*race condition*) i osigurala konzistentnost deljenih podataka.

- **Atomičke operacije (InterlockedExchange):** Za specifične, jednostavne operacije kao što je postavljanje flegova koji se dele između niti, korišćene su atomičke funkcije. U ovom projektu, InterlockedExchange se koristi tokom procesa sinhronizacije novog Workera kako bi se privremeno "zaključao" soket postojećeg Workera i sprečilo da ga druge niti koriste u isto vreme.

2.4. Razvojno okruženje i alati

- **Microsoft Visual Studio:** Kompletan projekat je razvijen i kompajliran korišćenjem Visual Studio IDE. Njegov integrisani dibager je bio od neprocenjive važnosti za praćenje izvršavanja višenitnog koda, analizu mrežnog saobraćaja i otklanjanje kompleksnih grešaka vezanih za sinhronizaciju.
- **Git i GitHub:** Za kontrolu verzija i praćenje promena u kodu korišćen je Git sistem. Razvojni proces, uključujući i drastične promene u arhitekturi, je dokumentovan kroz commit poruke na GitHub platformi, što je omogućilo jasan pregled evolucije projekta.

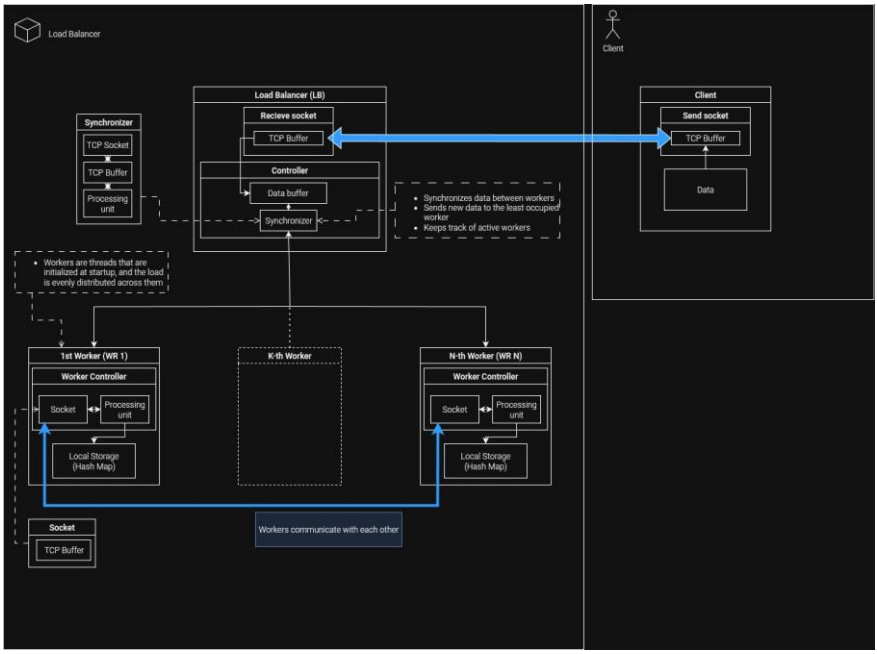
3. Opis rešenja problema

Problem razvoja distribuiranog servisa za skladištenje podataka rešen je kroz modularnu arhitekturu koja se sastoji od dve ključne komponente: **Load Balancera (LB)** i **Workera (WR)**. Ove dve komponente međusobno komuniciraju preko definisanog mrežnog protokola kako bi ispunile sve zahteve projekta, uključujući balansiranje opterećenja, dinamičku skalabilnost i peer-to-peer replikaciju podataka.

3.1. Arhitektura sistema

Sistem je dizajniran kao klijent-server model sa više nivoa. Na najvišem nivou su klijenti koji šalju zahteve. U centru sistema je Load Balancer, koji deluje kao jedinstvena ulazna tačka (*single point of entry*) i "dirigent" celog sistema. Ispod njega se nalazi sloj sa proizvoljnim brojem (N) Worker komponenti koje vrše stvarni posao skladištenja podataka.

Ključna karakteristika arhitekture je **decentralizovana replikacija podataka**. Iako LB inicira zadatke, on ne učestvuje u samom procesu replikacije. Ta odgovornost je prebačena na Workere, što smanjuje opterećenje na LB i čini sistem skalabilnijim.



3.2. Komponenta Load Balancer (LB)

Load Balancer je centralna komponenta sistema, implementirana kao višenitni TCP server. Njegove glavne funkcije su:

1. **Prihvatanje klijentskih konekcija:** LB sluša na predefinisanom portu (5059). Kada se novi klijent poveže, LB kreira posebnu nit (`handle_client`) koja je zadužena isključivo za komunikaciju sa tim klijentom. Ovo omogućava da LB istovremeno opslužuje više klijenata.
2. **Balansiranje opterećenja:** Kada klijent pošalje zahtev za skladištenje, LB poziva funkciju `distribute_task_to_worker`. Ova funkcija implementira "Least Load" algoritam. Ona prolazi kroz listu svih aktivnih Workera, pronalazi onog sa najmanjom vrednošću u polju `load` (koje predstavlja kumulativnu veličinu podataka koje je Worker obradio) i njemu prosleđuje zadatak.

3. **Upravljanje životnim ciklusom Workera:**
    - **Registracija:** Prilikom pokretanja sistema (initialize_workers) ili na zahtev klijenta (ADD_WORKER), LB kreira novi Worker thread i odmah uspostavlja TCP konekciju sa njim. Ta konekcija se čuva i koristi za slanje zadataka.
    - **Održavanje liste kolega (Peer List):** Ovo je ključna uloga LB-a u peer-to-peer modelu. Svaki put kada se novi Worker uspešno doda u sistem, LB poziva funkciju broadcast_peer_list. Ova funkcija kreira poruku sa listom portova svih trenutno aktivnih Workera i šalje je svakom od njih. Na ovaj način, LB osigurava da svaki Worker u svakom trenutku ima ažuran "telefonski imenik" svojih kolega, što je preduslov za direktnu replikaciju.
4. **Sinhronizacija novih Workera:** Kada se doda novi Worker, on je u početku prazan. Da bi postao koristan član sistema, mora dobiti sve podatke koji su već sačuvani. Funkcija sync_new_worker obavlja ovaj proces. LB privremeno "rezerviše" jednog od postojećih Workera, šalje mu GET_ALL komandu, prima kompletnu listu podataka, i zatim te podatke prosleđuje novom Workeru u vidu REPL komandi.
5. **Primanje potvrda (ACK):** Nakon što pošalje zadatak Workeru, LB ne radi replikaciju. Umesto toga, njegova worker_response_handler nit samo pasivno sluša za ACK:OK poruku od Workera, što je potvrda da je podatak uspešno sačuvan na primarnoj lokaciji.

3.3. Komponenta Worker (WR)

Worker je implementiran kao serverska komponenta koja se izvršava u sopstvenoj niti. Svaki Worker je autonoman i ima svoje lokalno skladište podataka, implementirano preko jednostavne *hash* mape. Njegova funkcionalnost je kompleksna jer mora da komunicira i sa LB-om i sa drugim Workerima.

Ovo je postignuto upotrebom select() funkcije unutar glavne petlje worker_function. select() omogućava Workeru da istovremeno prati dva soketa:

- lb_connection_socket: Stalna konekcija sa Load Balancerom.

- listen_socket: Soket na kom sluša za dolazne konekcije od drugih Workera.

Glavne funkcije Workera su:

1. **Obrada zadataka od LB-a:** Kada preko lb_connection_socket stigne poruka TASK:STORE:<podatak>, Worker radi sledeće:
   a. Skladišti podatak u svoju *hash* mapu.
   b. Šalje ACK:OK poruku nazad Load Balanceru da potvrdi uspešno skladištenje.
   c. **Inicira peer-to-peer replikaciju:** Prolazi kroz svoju internu peer_ports listu. Za svaki port sa liste, otvara novu, kratkotrajnu konekciju ka tom Workeru i šalje mu REPL:STORE:<podatak> poruku.
2. **Obrada replikacija od drugih Workera:** Kada preko listen_socket stigne nova konekcija, Worker je prihvata. Ako je poruka REPL:STORE:<podatak>, on jednostavno skladišti podatak u svoju *hash* mapu i odmah zatvara konekciju. Ključno je da na REPL poruke **ne odgovara**, čime se sprečava "echo" efekat i beskonačne petlje.
3. **Ažuriranje liste kolega:** Kada od LB-a stigne UPDATE_PEERS poruka, Worker parsira listu portova i ažurira svoj peer_ports niz, osiguravajući da uvek ima tačne informacije o svojim kolegama.

3.4. Komunikacioni protokol

Da bi se omogućila jasna i nedvosmislena komunikacija između komponenti, definisan je jednostavan tekstualni protokol zasnovan na prefiksima. Svaka poruka se sastoji od delova razdvojenih dvotačkom (:).

- TASK:STORE:<podatak>
  - **Smer:** LB -> WR
  - **Značenje:** Ovo je novi zadatak od klijenta. Worker mora da ga sačuva, pošalje ACK nazad LB-u, i pokrene replikaciju ka svojim kolegama.
- REPL:STORE:<podatak>
  - **Smer:** WR -> drugi WR (ili LB -> novi WR tokom sinhronizacije)

    - **Značenje:** Ovo je poruka za replikaciju. Worker mora da sačuva podatak, ali **ne sme** da šalje nikakav odgovor.
- UPDATE_PEERS:<port1,port2,...>
    - **Smer:** LB -> svi WR
    - **Značenje:** Nalaže Workeru da ažurira svoju internu listu portova aktivnih kolega.
- GET_ALL:
    - **Smer:** LB -> WR
    - **Značenje:** Nalaže Workeru da vrati listu svih podataka koje čuva. Koristi se isključivo za sinhronizaciju novih Workera.
- ACK:OK
    - **Smer:** WR -> LB
    - **Značenje:** Potvrda da je TASK uspešno primljen i sačuvan.

3.5. Praktičan primer rada sistema

Da bismo ilustrovali kako sistem funkcioniše u praksi, proći ćemo kroz scenario opisan u logovima generisanim tokom testiranja.

Scenario: Sistem se pokreće sa 5 Workera. Klijent šalje dva podatka ("Test1", "Test2"), zatim dodaje novog, šestog Workera.

Korak 1: Inicijalizacija
Sistem se pokreće i kreira 5 Worker niti. Za svakog novog Workera, LB poziva broadcast_peer_list.

```
C:\Users\Tomas\Desktop\LoadBalancer\LoadBalancer-IKP\LoadBalancer\x64\Debug\LoadBalancer.exe
Worker 1: Assigned to port 60167
Server connected to Worker 1 on port 60167.
Worker 1: Accepted connection from Load Balancer.
Broadcasting new peer list to all workers...
Worker 1 received updated peer list.
Worker 2: Assigned to port 60169
Server connected to Worker 2 on port 60169.
Worker 2: Accepted connection from Load Balancer.
Broadcasting new peer list to all workers...
Worker 1 received updated peer list.
Worker 2 received updated peer list.
Initialized 2 workers.
Server listening on port 5059... Ready to accept multiple clients.
```

Na kraju, svih 5 Workera imaju listu koja sadrži 5 portova.

Korak 2: Prvi zahtev za skladištenje ("Test1")

Klijent šalje "Test1".

```
Handling connection for client 127.0.0.1:60934
Received task from client 127.0.0.1:60934: Test1
Assigning task to Worker 1 with load 0.
Successfully sent task to Worker 1: TASK:STORE:Test1
Worker 1 stored data from TASK: 237528342 -> Test1
LB received acknowledgment from Worker 1: ACK:OK
Worker 1 starting replication to 1 peers...
  -> Worker 1 sent REPL to port 60864
Worker 2 stored data from REPL: 237528342 -> Test1
```

LB: Bira Workera sa najmanjim opterećenjem

Worker 1: Prima zadatak, skladišti ga i javlja LB-u

Worker 1: Inicira peer-to-peer replikaciju

Worker 2: Prima REPL poruku od Workera 1 i skladišti podatak

U ovom trenutku, svih 5 Workera poseduju podatak "Test1".

Korak 3: Drugi zahtev za skladištenje ("Test2")

Klijent šalje "Test2". LB sada vidi da Worker 1 ima veće opterećenje, pa bira sledećeg slobodnog.

```
Assigning task to Worker 2 with load 0.
Successfully sent task to Worker 2: TASK:STORE:Test2
Worker 2 stored data from TASK: 237528343 -> Test2
Worker 2 starting replication to 1 peers...
LB received acknowledgment from Worker 2: ACK:OK
  -> Worker 2 sent REPL to port 60862
Worker 1 stored data from REPL: 237528343 -> Test2
```

// LB:

Bira Workera 2

Worker 2: Sada on preuzima ulogu inicijatora replikacije

Worker 1: Prima REPL poruku od Workera 2

Sada oba Workera imaju i "Test1" i "Test2". Sistem je konzistentan.

Korak 4: Dodavanje novog Workera (Worker 6)

Klijent šalje komandu ADD_WORKER.

```
Received 'ADD_WORKER' command from client 127.0.0.1:65370
Worker 3: Assigned to port 65443
Server connected to Worker 3 on port 65443.
Worker 3: Accepted connection from Load Balancer.
Broadcasting new peer list to all workers...
Worker 1 received updated peer list.
Syncing new worker with existing data...
Syncing new worker 3 with existing worker 1 (socket claimed)...
Worker 2 received updated peer list.
Worker 3 received updated peer list.
Received 32 bytes of sync data from Worker 1. Applying to new worker...
  -> Forwarding item 'Test1' to new worker 3
  -> Forwarding item 'Test2' to new worker 3
Sync for worker 3 complete (socket released).
```

LB: Kreira novog Workera i ažurira sve o tome

LB: Pokreće sinhronizaciju za Worker 6

Nakon ovog koraka, Worker 6 je u potpunosti sinhronizovan i spreman da učestvuje u radu. Sistem je uspešno i dinamički skaliran.

4. Zaključak

Projekat je uspešno realizovan, rezultirajući funkcionalnim distribuiranim servisom za skladištenje podataka koji ispunjava sve postavljene zahteve. Implementirano rešenje demonstrira ključne koncepte modernih skalabilnih sistema, uključujući balansiranje opterećenja, dinamičko dodavanje resursa i mehanizme za osiguravanje konzistentnosti podataka.

Najveći izazov u projektu bio je dizajn i implementacija pouzdanog mehanizma za replikaciju podataka. Kroz evoluciju rešenja, došlo se do peer-to-peer modela koji doslovno prati specifikaciju projekta. Ovaj model, iako kompleksniji za implementaciju jer zahteva da svaki radnik (Worker) bude svestan svojih kolega i da aktivno učestvuje u replikaciji, uspešno je realizovan korišćenjem `select()` funkcije za asinhrono praćenje više mrežnih događaja.

Razvijeni sistem je robustan i efikasan. Load Balancer uspešno raspoređuje opterećenje, a mehanizam za dinamičko dodavanje i sinhronizaciju novih radnika omogućava laku skalabilnost.

Predlozi za dalja usavršavanja:

- **Pouzdanija inicijalizacija Workera:** Trenutno rešenje koristi `Sleep(100)` da bi se sačekalo da Worker thread započne sa radom. Ovo se može unaprediti korišćenjem `Event` objekata iz Windows API-ja, gde bi Worker eksplicitno signalizirao Load Balanceru kada je spreman za prihvatanje konekcija.
- **Perzistentnost podataka:** Trenutno se podaci čuvaju samo u memoriji, što znači da se gube gašenjem servisa. Sistem bi se mogao unaprediti tako da svaki Worker upisuje podatke u fajl na disku.
- **Oporavak od otkaza:** Implementirati mehanizam "otkucaja srca" (*heartbeat*) gde bi Workeri periodično slali poruku Load Balanceru da potvrde da su aktivni. Ukoliko "otkucaj" izostane, LB bi mogao da smatra da je Worker otkazao i da ga ukloni iz liste aktivnih kolega.

Sve u svemu, ovaj projekat pruža solidnu osnovu i dubok uvid u fundamentalne principe i izazove projektovanja i implementacije distribuiranih sistema.