UNIVERZITET U NOVOM SADU

FAKULTET TEHNIČKIH NAUKA U NOVOM SADU

SEMINARSKI RAD

- Master akademske studije -

PARALELNE DISTRIBUIRANE ARHTEKTURE I JEZICI

Poređenje *Rust* i *Scala* programskog jezika na primjeru implementacije asinhronog koda

Student

Bojan Radović,
e2-121-2023

Mentor

Dr Dinu Dragan

Novi Sad, 2024

Sadržaj:

1.Motivacija

Razumijevanje performansi različitih programskih jezika u asinhronom okruženju postaje sve značajnije sa porastom popularnosti distribuiranih sistema i visokopropusnih aplikacija. Poređenje programskih jezika *Rust* i *Scala* na praktičnom primjeru omogućava analizu njihove efikasnosti u stvarnim scenarijima i pomaže u donošenju informisanih odluka pri izboru tehnologija za implementaciju sličnih sistema.

1.1 Uvod

Ovaj seminarski rad izrađen je u okviru predmeta *Paralelne i distribuirane arhitekture i jezici*. Cilj rada je uporediti performanse i brzinu jezika *Rust* i *Scala* na primjeru implementacije asinhronog koda. Fokus je na konkurentnom slanju HTTP zahtjeva ka *Mock serveru*, koji je takođe implementiran kako bi se omogućilo testiranje performansi. Pored implementacije *Rust* i *Scala* konkurentnih klijenata, analiziraju se rezultati i vrši poređenje u pogledu brzine izvršavanja i korišćenja resursa. Cjelokupan izvorni kod, uključujući implementacije i testove, dostupan je na sledećem *GitHub* repozitorijumu: https://github.com/rbojan2000/multithreaded-server-rust-vs-scala.

1.2 Opis slučaja korišćenja

Zamislimo sistem u kome je potrebno obraditi veliki broj HTTP zahtjeva prema jednom API endpointu.
Sistem ima sledeće specifične zahtjeve:

1. Istovremeno može da obradi maksimalno 5 zahtjeva i
2. Svaki zahtjev ima šansu od 20% da ne uspije, pa je potrebno omogućiti ponovno slanje zahtjeva dok ne uspije ili dok se ne iscrpi maksimalni broj pokušaja.

2. Implementacija

U ovoj sekciji biće predstavljena implementacija svih ključnih komponenti sistema, koji je razvijen kako bi se uporedile performanse *Scala* i *Rust* programskog jezika u kontekstu konkurentnog slanja *HTTP* zahtjeva. Sistem se sastoji od tri glavne komponente:

1. ***Mock server*** - server koji simulira API endpoint za testiranje performansi klijenata,
2. ***Scala konkurentni klijent*** - implementacija klijenta u *Scala* programskom jeziku sa korišćenjem *STTP biblioteke* za paralelno slanje zahtjeva i
3. ***Rust* konkurentni klijent** - implementacija klijenta u *Rust* programskom jeziku sa korišćenjem *tokio* biblioteke za paralelno slanje zahtjeva.

Svaka od ovih komponenti biće detaljno objašnjena kako bi se prikazao način na koji su riješeni tehnički izazovi u oba jezika.

2.1 *Mock* server

Za potrebe testiranja performansi i simulacije API endpointa, razvijen je *Mock server* koristeći *Flask* framework u *Python-u*. Server ima jedan *API* endpoint (/api) koji imitira rad stvarnog servera i omogućava testiranje konkurentnog slanja *HTTP* zahtjeva. Server je dizajniran da simulira greške i kašnjenje u odgovoru, kako bi se stvorili uslovi za testiranje mehanizama ponovnog slanja zahtjeva u slučaju neuspijeha.

Specifikacije *Mock servera***:**

- **Stopa grešaka(*Fail rate*)**: Da bi se testirala robusnost sistema, postoji šansa od 20% (0.2) da svaki zahtjev neće biti uspješan i da će server vratiti HTTP status *503 Service Unavailable*.
- **Kašnjenje u odgovoru(*Sleep time*)**: Da bi se simulirao rad servera, svaki zahjtev traje 0.1 sekundi prije nego što server vrati odgovor.
- **Multithreaded server**: *wsgi server* koji omogućava pokretanje servera sa višestrukim radnicima (*workers*). Ovaj pristup omogućava da server obradi više zahtjeva paralelno, što odgovara slučaju korišćenja.

Kod 2.0 ispod prikazuje implementaciju komponente 1: *Mock server.*

```
from flask import Flask, jsonify
from http import HTTPStatus
import random
import time

app = Flask(__name__)

PORT = 5000
```

```
HOST = "0.0.0.0"
FAIL_RATE = 0.2
SLEEP_TIME = 0.1

@app.route("/api", methods=["GET"])
def api_endpoint():
    time.sleep(SLEEP_TIME)
    if random.random() < FAIL_RATE:
        return "Request failed", HTTPStatus.SERVICE_UNAVAILABLE
    return "Success", HTTPStatus.OK

if __name__ == "__main__":
    app.run(host=HOST, port=PORT)
```

Kod 2.0: Implementacija Mock servera

Kod 2.1 prikazuje konfiguraciju *Mock servera*, koji nakon pokretanja postaje multithreaded server sa 5 *worker*-a i jednim *master*-om(*Kod 2.1*). Svaka komponenta servera, uključujući *worker*-e i *master*, predstavlja posebnu nit u procesu(*Kod 2.2*). Ovaj pristup omogućava serveru da efikasno upravlja velikim brojem zahtjeva i da održava visoku dostupnost.

```
authors = [
    "Radovic, Bojan | rbojan2000@gmail.com",
]

bind = "0.0.0.0:8000"
workers = 5
```

Kod 2.1: Konfiguracija Mock servera

```
(venv)------------------------------------------------------- bojanradovic@fedora
~/multithreaded-server-rust-vs-scala/mock_server (main*) » gunicorn mock_server:app

[2024-12-05 13:48:08 +0100] [134270] [INFO] Starting gunicorn 23.0.0
[2024-12-05 13:48:08 +0100] [134270] [INFO] Listening at: http://0.0.0.0:8000
[2024-12-05 13:48:08 +0100] [134270] [INFO] Using worker: sync
[2024-12-05 13:48:08 +0100] [134278] [INFO] Booting worker with pid: 134278
```

```
[2024-12-05 13:48:08 +0100] [134279] [INFO] Booting worker with pid: 134279
[2024-12-05 13:48:08 +0100] [134280] [INFO] Booting worker with pid: 134280
[2024-12-05 13:48:08 +0100] [134281] [INFO] Booting worker with pid: 134281
[2024-12-05 13:48:08 +0100] [134282] [INFO] Booting worker with pid: 134282
```

Kod 2.2: Ispis u konzoli nakon pokretanja servera

2.3 *Rust* konkurentni klijent

U nastavku je opisana implementacija konkurentnog klijenta u *Rust* programskom jeziku. Klijent šalje zahtjeve ka *Mock serveru* koristeći *tokio* i *reqwest* biblioteke. Klijent će slati više zahtjeva u paraleli, a implementacija obuhvata i kontrolu broja istovremenih zahtjeva pomoću semafora i mogućnost ponovnog pokušaja (*retries*) u slučaju grešaka.

Ključne komponente koda:

1. **Konkurentno slanje zahtjeva**: Funkcija *send_requests_concurrently*(*Kod 2.3*) pokreće konkurentno slanje zahtjeva, koristeći ***Tokio*** taskove. Za svaki zahtjev, funkcija koristi semafor (*semaphore*), koji omogućava da samo *MAX_CONCURRENT_REQUESTS* zahtjeva bude aktivno u isto vrijeme. Kada je kapacitet semafora dostignut, sledeći zahtjev čeka dok se jedan od radnika ne oslobodi. Kada se jedan zahtjev završi, semafor se oslobađa(*drop(permit)*), omogućavajući novom zahtjevu da preuzme dozvolu za izvršenje. Svaki zahjtev se procesuira unutar taska (koji se pokreće korišćenjem *tokio::spawn*), a kada zahtjev završi, broj uspješnih ili neuspješnih zahtjeva se inkrementira. Ovaj broj se čuva unutar ***tokio Mutex*** promjenljivih, kako bi se omogućilo sigurno ažuriranje broja zahtjeva u više niti. Funkcija *send_requests_concurrently* vraća broj uspiješnih i neuspiješnih zahtijeva ka *Mock serveru*.

```rust
async fn send_requests_concurrently() -> (usize, usize) {
    let semaphore = Arc::new(Semaphore::new(MAX_CONCURRENT_REQUESTS));
    let mut tasks = Vec::new();

    let successful_requests = Arc::new(tokio::sync::Mutex::new(0));
    let failed_requests = Arc::new(tokio::sync::Mutex::new(0));

    for _ in 0..NUM_REQUESTS {
        let permit = semaphore.clone().acquire_owned().await.unwrap();
        let successful_requests = successful_requests.clone();
        let failed_requests = failed_requests.clone();

        let task = tokio::spawn(async move {
            match send_request(MOCK_SERVER_URL).await {
```

```
                Ok(_) => {
                    let mut count = successful_requests.lock().await;
                    *count += 1;
                }
                Err(_) => {
                    let mut count = failed_requests.lock().await;
                    *count += 1;
                }
            }
            drop(permit);
        });

        tasks.push(task);
    }

    for task in tasks {
        task.await.unwrap();
    }

    let successful = *successful_requests.lock().await;
    let failed = *failed_requests.lock().await;

    (successful, failed)
}
```

Kod 2.3: Konkurentno slanje zahtjeva

2. **Slanje HTTP zahtjeva**: Funkcija *send_request*(*Kod 2.4*) pokušava da pošalje *HTTP GET* zahtjev ka **Mock serveru**. Ako je zahtjev uspiješan (status je *200 OK*), funkcija vraća *Ok()*. Ako zahtjev nije uspiješan, funkcija će pokušavati ponovno slanje zahtjeva sve do maksimalnog broja pokušaja (*MAX_RETRIES*), koji u ovom slučaju koršćenja ima vrijednost 3. Ako sva tri pokušaja budu neuspiješna, funkcija vraća grešku sa porukom "*Max retries reached*".

```
async fn send_request(url: &str) -> Result<(), String> {
   let mut attempts = 0;

   while attempts < MAX_RETRIES {
       let response = reqwest::get(url).await;

       match response {
           Ok(res) if res.status().is_success() => {
               println!("Success: {}", res.status());
               return Ok(());
           }
           Ok(res) => {
```

```
                println!("Failed (attempt {}): {}", attempts + 1, res.status());
            }
            Err(_) => {
                println!("Request failed (attempt {}): Network error or timeout",
attempts + 1);
            }
        }
        attempts += 1;
    }
    Err("Max retries reached".to_string())
}
```

Kod 2.4: Slanje HTTP zahtjeva

2.4 *Scala* konkurentni klijent

U implementaciji konkurentnog klijenta u *Scala* programskom jeziku, koristi se *STTP* biblioteka za slanje *HTTP* zahtjeva ka *Mock serveru*, a konkurentnost se postiže korišćenjem *Scala Futures* i semafora za kontrolu broja istovremenih zahtjeva.
Ključne komponente koda:

1. **Konkurentno slanje zahtjeva**: Funkcija *sendConcurrentRequests*(*Kod 2.5*) kreira listu *Future* objekata, od kojih svaki predstavlja jedan *HTTP* zahtjev. Ovi zahtjevi se šalju paralelno pomoću *Future.sequence*. *Semaphore* (*semaphore.acquire()* i *semaphore.release()*) je ključni element za kontrolu broja istovremenih zahtjeva. Semafor omogućava da maksimalan broj istovremenih zahtjeva bude ograničen na *maxConcurrentRequests*. Kada se neki zahtjev završi, semafor se "oslobađa", omogućavajući slanje novog zahtjeva. Funkcija *sendConcurrentRequests* broji uspiješne i neuspiješne zahtjeve ka serveru i ispisuje ih u konzoli.

```
def sendConcurrentRequests(): Future[Unit] = {
  val requestFutures = (1 to numRequests).map { _ =>
    sendRequest(maxRetries).map {
      case response if response.code.isSuccess =>
        synchronized {
          successCount += 1
        }
        response.body match {
          case Right(body) => println(s"Success: $body")
          case Left(error) => println(s"Error: $error")
        }
      case response =>
        synchronized {
          failureCount += 1
        }
        println(s"Request failed with status ${response.code}:
```

```
${response.statusText}")
      }
    }

    Future.sequence(requestFutures).map(_ => ()).andThen {
      case Success(_) =>
        println(s"All requests completed. Success: $successCount, Failure:
$failureCount")
      case Failure(exception) =>
        println(s"Error occurred: $exception")
    }
  }
```

Kod 2.5: Konkurentno slanje zahtjeva

2. **Slanje HTTP zahtjeva**: Korišćenjem *basicRequest*(*Kod 2.6*) inicijalizuje se *HTTP* GET zahtjev koji se šalje serveru. Metoda *send(backend)* koristi asinhroni *backend* za slanje zahtjeva, omogućavajući paralelno izvršavanje. *AsyncHttpClientFutureBackend* je backend koji koristi *Future* za izvršavanje *HTTP* zahtjeva, omogućavajući asinhronu obradu odgovora bez blokiranja glavne niti. U funkciji *sendRequest* koristi se recoverWith blok kako bi se omogućio ponovni pokušaj u slučaju greške. Ako zahtjev ne uspije, broj preostalih pokušaja (*retriesLeft*) se smanjuje, a funkcija pokušava ponovo. Ako svih *maxRetries*(u ovom slučaju korišćenja ima vrijednost 3) pokušaja ne uspije, vraća se greška.

```
val semaphore = new Semaphore(maxConcurrentRequests)
def sendRequest(retriesLeft: Int): Future[Response[Either[String, String]]] = {
  basicRequest
    .get(uri"$mockServerUrl")
    .send(backend)
    .recoverWith {
      case _ if retriesLeft > 0 =>
        println(s"Retrying request, remaining retries: $retriesLeft")
        sendRequest(retriesLeft - 1)
    }
}
```

Kod 2.6: Slanje HTTP zahtjeva

3. Poređenje

U ovoj sekciji su rezultati poređenja implementacije konkurentnog klijenta implementiranog u *Rust* i *Scala* programskom jeziku. Eksperimenti su izvedeni na istom računaru, kako bi se obezbijedila konzistentnost rezultata, a testiranje je obuhvatilo podešavanja koja uključuju broj niti, broj zahtjeva ka *Mock serveru* i vrijeme izvršenja.
Specifikacije Računara na kojem je izvedeno testiranje prikazano je na *Kod 3.1*.

```
------------------------------------------------------------  bojanradovic@fedora
~/multithreaded-server-rust-vs-python/scala_client (main*) » free -h & lscpu
               total        used        free      shared  buff/cache   available
Mem:            30Gi        12Gi       578Mi       2.5Gi        20Gi        18Gi
Swap:          8.0Gi       3.2Mi       8.0Gi

Architecture:             x86_64
 CPU op-mode(s):         32-bit, 64-bit
 Address sizes:          46 bits physical, 48 bits virtual
 Byte Order:             Little Endian
CPU(s):                   22
 On-line CPU(s) list:    0-21
Vendor ID:                GenuineIntel
 Model name:             Intel(R) Core(TM) Ultra 7 155H
   CPU family:           6
   Model:                170
   Thread(s) per core:   2
   Core(s) per socket:   16
   Socket(s):            1
   Stepping:             4
   CPU(s) scaling MHz:   23%
   CPU max MHz:          4800.0000
   CPU min MHz:          400.0000
   BogoMIPS:             5990.40
```

Kod 3.1: Dostupna RAM memorija i specifikacija procesora

3.1 Rezultati

Tabela 1 prikazuje rezultate izvršavanja na konkurentnim klijentima opisanih u prethodnoj sekciji.

Broj niti	Broj uspiješnih zahtjeva	Broj neuspiješnih zahtjeva	Konkurentni klijent(*Rust*/*Scala*)	Vrijeme izvršavanja u sekundama

10	97	19	Scala	19.12
10	95	20	Rust	17.1
10	991	33	Scala	177.62
10	992	36	Rust	171.3
15	996	21	Scala	181.5
15	995	23	Rust	176.2
5	98	15	Rust	16.3
5	96	19	Scala	16.9

Tabela 1: Rezultati izvršavanja konkurentnih klijenata

4. Zaključak

U ovom radu opisane su implementacije konkurentnih klijenata u *Rust* i *Scala* programskim jezicima, koji se koriste za slanje *HTTP* zahteva ka *Mock serveru*. Implementacije se baziraju na asinhronom slanju zahteva, korišćenju semafora za kontrolu broja istovremenih zahteva i ponovnim pokušajem (*retry*) u slučaju grešaka. Ove strategije omogućavaju efikasno upravljanje velikim brojem zahteva, minimizirajući opterećenje servera i povećavajući uspješnost izvršenja. Pored same implementacije, fokusirali smo se i na poređenje performansi između *Rust* i *Scala* klijenata. Poređenje se baziralo na broju niti, broju zahteva i vremenu izvršenja. Za svaku implementaciju postavljen je broj niti i zahtjeva, a mjerenje vremena omogućilo je analizu performansi svakog jezika u kontekstu konkurentnog izvršenja. Kroz ovaj eksperiment, *Rust* se pokazao kao izuzetno efikasan za konkurentne zadatke, zahvaljujući svom niskom nivou i kontroli resursa. S druge strane, Scala, sa svojom robusnom podrškom za funkcionalno programiranje i alatima poput *Future*, takođe nudi solidne performanse u konkurentnim okruženjima, iako sa nešto većim zahtjevima u odnosu na *Rust*.