



UNIVERZITET U NOVOM SADU
FAKULTET TEHNIČKIH NAUKA
U NOVOM SADU



Matija Maksimović E2 33/2024

Rast i Pajton na primeru implementacije asinhronog koda

SEMINARSKI RAD

- Master akademske studije -

Novi Sad, 2024.



UNIVERZITET U NOVOM SADU o FAKULTET
TEHNIČKIH NAUKA
21000 NOVI SAD, Trg Dositeja Obradovića 6

**Predmet: Paralelne i distribuirane
arhitekture i jezici**
Predmetni profesor: prof. dr Dinu Dragan

SADRŽAJ

1. Uvod	2
2. Osnovni pojmovi	4
3. Implementacija asinhronog koda	6
4. Pajton i Rast rešenja	7
5. Zaključak	9
6. Literatura	10

1.Uvod

Razvoj savremenih softverskih rešenja suočava se sa sve većim zahtevima za brzinom, skalabilnošću i efikasnošću. Za ovakve inženjerske izazove, asinhrono programiranje je postalo ključna paradigma. Ono omogućava optimizaciju korišćenja resursa i bolje korišćenje računarskog vremena.

Cilj ovog rada je da pruži uvid u osnovne principe asinhronog programiranja, njegovu implementaciju u različitim programskim jezicima, i da na praktičnom primeru pokaže kako se ova tehnika može primeniti. Pažnja je posvećena analizi rešenja u jezicima Pajton i Rast, koji svojim jedinstvenim pristupima nude različite prednosti u asinhronom kodu.

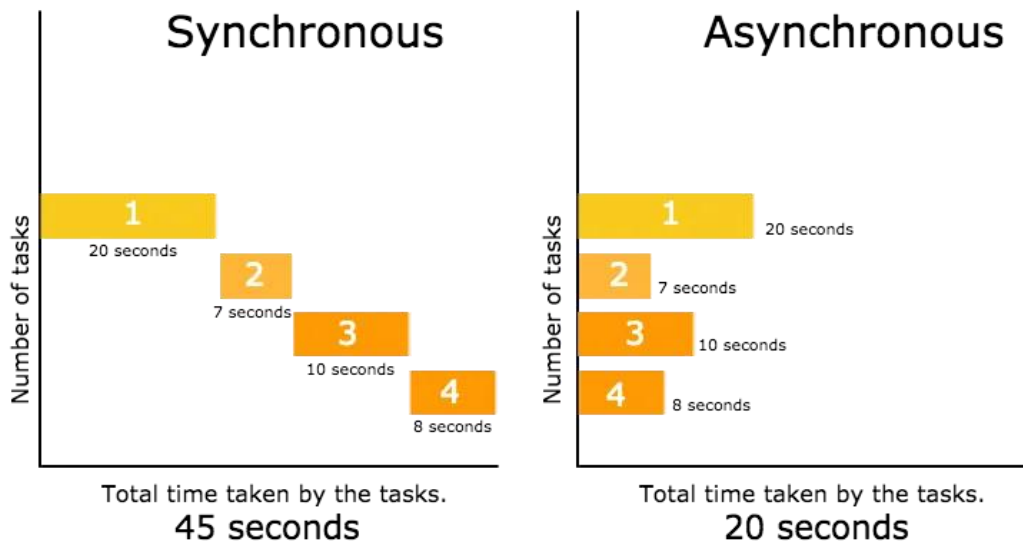
Predstaviću osnovne pojmove i principe asinhronog programiranja, detalje implementacije našeg reprezentativnog primera, i analizirati razlike između rešenja u pomenutim jezicima. Cilj je da se stekne ideja o tome kako izbor alata i jezika utiče na performanse i sam proces pisanja koda, kao i na ukupno iskustvo razvoja aplikacija. Dobar inženjer od gomile alata zna da izabere one koji optimalno rešavaju njegov problem.

2.Osnovni pojmovi

Asinhrono programiranje predstavlja tehniku razvoja softvera koja omogućava izvršavanje zadataka bez blokiranja glavnog toka programa. Ova paradigma je nastala kao odgovor na potrebe za brzim i efikasnim sistemima koji mogu obraditi veliki broj zadataka u isto vreme, poput mrežnih poziva, obrade podataka ili pristupa eksternim resursima.

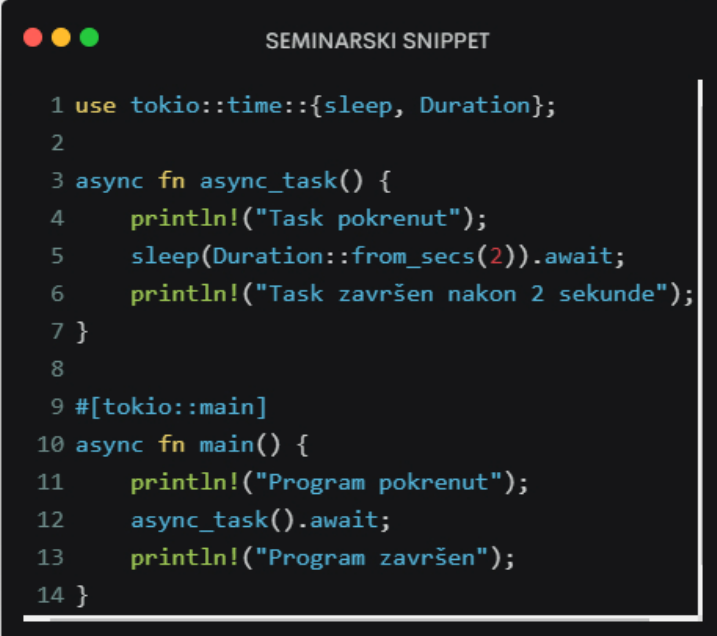
Tradicionalni sinhroni programski model često blokira izvršavanje programa dok se određeni zadatak ne završi. Na primer, dok se čeka odgovor sa mreže, program pauzira izvršavanje, što dovodi do neefikasnog korišćenja resursa. Asinhrono programiranje rešava ovaj problem tako što dozvoli programu da nastavi sa izvršavanjem drugih zadataka dok čeka kraj sporijih operacija. Glavni ciljevi asinhronog programiranja uključuju:

- Optimizaciju vremena obrade zadataka.
- Smanjenje kašnjenja (**engl.** latency) u aplikacijama koje koriste spore ili nepredvidive resurse.
- Povećanje skalabilnosti sistema bez potrebe za kreiranjem dodatnih niti.



Slika 1: Poređenje sinhronog i asinhronog koda (preuzeto sa <https://blog.devgenius.io/multi-threading-vs-asynchronous-programming-what-is-the-difference-3ebfe1179a5>)

Savremeni programski jezici, poput Pajtona i Rasta, uvode specifične ključne reči *async* i *await* kako bi pojednostavili rad sa asinhronim operacijama. Ključna reč *async* označava funkciju kao asinhronu, čime omogućava njeno pozivanje pomoću *await*. Ključna reč *await* pauzira izvršavanje trenutne asinhrone funkcije dok se zadatak ne završi, nakon čega se rezultat vraća i nastavlja dalje izvršavanje.



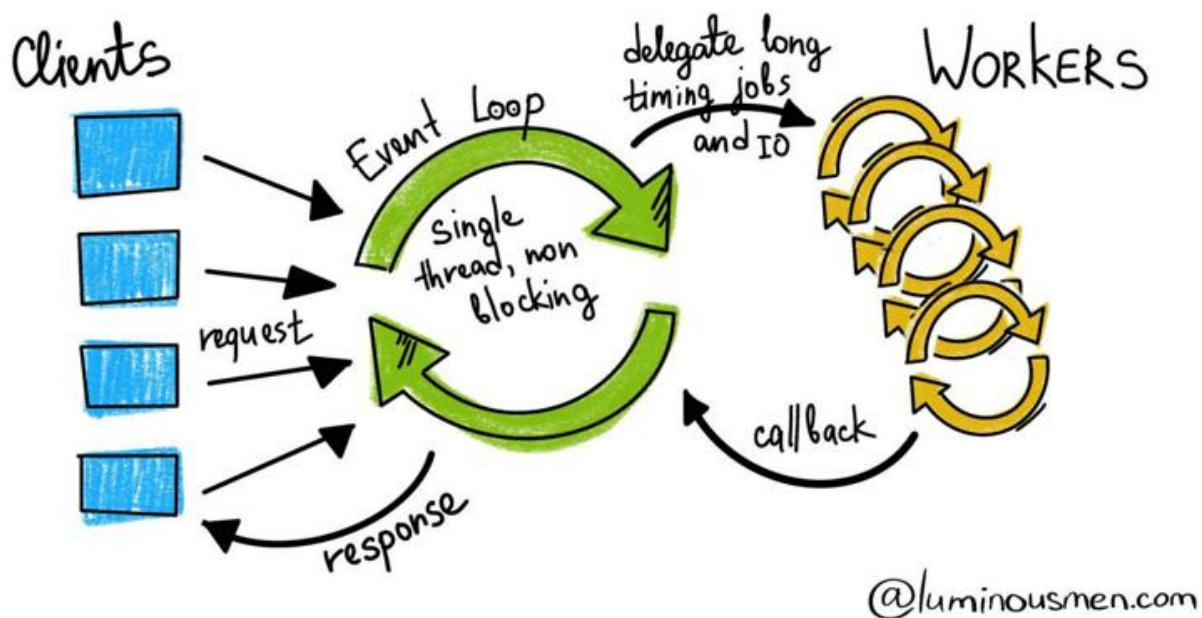
```
1 use tokio::time::{sleep, Duration};
2
3 async fn async_task() {
4     println!("Task pokrenut");
5     sleep(Duration::from_secs(2)).await;
6     println!("Task završen nakon 2 sekunde");
7 }
8
9 #[tokio::main]
10 async fn main() {
11     println!("Program pokrenut");
12     async_task().await;
13     println!("Program završen");
14 }
```

Slika 2: Asinhrona funkcija (preuzeto sa <https://codeimg.io/>)

Problemi koji se rešavaju asinhronim programiranjem mogu se rešavati i pristupima konkurentnog programiranja. Konkurenti pristup nove resurse, programske niti, tretira kao radnike, dok asinhroni pristup problem rešava uvođenjem više novih zadataka (**engl.** task). U realnom svetu, više radnika će uraditi neki posao brže, ali će to biti i znatno skuplje za poslodavca. Ako je naš poslodavac operativni sistem jasno je da previše niti nije uvek u našem interesu. Asinhroni pristup neće sekvencijalni kod ubrzati kao klasični konkurentni pristup, ali će biti štedljiviji sa resursima kojima raspolazemo.

Petlja događaja (**engl.** event loop) je centralni mehanizam asinhronog programiranja. Ona upravlja redosledom izvršavanja zadataka, čime omogućava glavnoj niti da efikasno prelazi sa jednog zadatka na

drugi. Kada zadatak čeka na resurs, petlja događaja ga privremeno zaustavlja i prelazi na sledeći zadatak iz reda. **Jednostavan prikaz rada petlje događaja** - U red čekanja se dodaju svi događaji (**engl.** event). Petlja događaja bira prvi zadatak i šalje ga na izvršenje. Ako zadatak zahteva čekanje (npr. mrežni poziv), vraća se u red i prelazi na sledeći zadatak. Ovaj model omogućava konkurentno izvršavanje zadataka unutar jedne niti, bez potrebe za višenitnim (**engl.** multithreading) pristupom.



Slika 3: Petlja događaja (preuzeto sa <https://luminousmen.com/>)

3.Implementacija asinhronog koda

Konekcija sa krajnjom tačkom programskom interfejsa aplikacije (**engl.** application programming interface) je odličan posao za asinhrono programiranje. Ovaj pristup nudi efikasno upravljanje resursima i paralelizaciju zahteva. Cilj rada je da se uz reprezentativni primer stekne svest o asinhronom izvršavanju programa, kao i o alatima, programskim jezicima, koji podržavaju asinhronu paradigmu.

Scenario:

- Imamo krajnu tačku (**engl.** endpoint) koju treba kontaktirati više puta. Sistem može istovremeno podneti do pet paralelnih konekcija. Svaki poziv na *endpoint* ima 20% šanse za neuspeh, pa je neophodno implementirati ponovno slanje zahteva u slučaju greške.

Konceptualna implementacija:

- Upravljanje paralelnim konekcijama - Ograničavanje broja istovremenih konekcija postiže se korišćenjem semafora. Semafor upravlja brojem dozvoljenih paralelnih zahteva, i tako osigurava da u svakom trenutku ne bude više od pet aktivnih konekcija.
- Ponovno slanje zahteva - Svaki neuspešan zahtev se ponavlja dok ne uspe. To podrazumeva i eksponencijalno odstupanje (**engl.** exponential backoff), tj. povećanje vremena čekanja između ponovljenih pokušaja.
- Asinhrona funkcija – Svaki poziv se obavlja unutar asinhrone funkcije koja koristi *await* za čekanje odgovora bez blokiranja ostatka programa. Ovo omogućava da se drugi zadaci izvršavaju u međuvremenu.

Proces rada:

- Glavna funkcija kreira više asinhronih zadataka (taskova) koji istovremeno šalju zahteve API-ju. Svaki zadatak koristi semafor za kontrolu pristupa i *await* za asinhrono čekanje. Nakon završetka svih zahteva, rezultati se sakupljaju i obrađuju.

4. Pajton i Rast rešenja

Implementacije asinhronih biblioteka za ova dva jezika oslanjaju se na *async/await* sintaksu, koja je ranije spomenuta u teorijskom delu rada. Pajton koristi biblioteku *asyncio* za asinhrono programiranje. Pajton rešenje prati strukturu navedenu ranije, asinhroni API pozivi su ograničeni semaforom kako bi se istovremeno omogućilo do pet paralelnih konekcija. Program svakom pozivu daje šansu od 20% za neuspeh, i u tom slučaju se pokušava ponovo uz eksponencijalno povećanje vremena čekanja.

```
^ Request 2 succeeded: Response for request 2
Request 3 succeeded: Response for request 3
Request 7 succeeded: Response for request 7
Request 6 succeeded: Response for request 6
Request 4 succeeded: Response for request 4
Request 5 succeeded: Response for request 5
Request 1 retry(1) failed: Request 1 failed
Request 9 succeeded: Response for request 9
Request 11 succeeded: Response for request 11
Request 12 retry(1) failed: Request 12 failed
Request 10 succeeded: Response for request 10
Request 8 succeeded: Response for request 8
Request 13 succeeded: Response for request 13
Request 15 retry(1) failed: Request 15 failed
Request 14 succeeded: Response for request 14
Request 17 retry(1) failed: Request 17 failed
Request 16 succeeded: Response for request 16
Request 18 succeeded: Response for request 18
Request 19 retry(1) failed: Request 19 failed
Request 20 succeeded: Response for request 20
Request 12 succeeded: Response for request 12
Request 1 succeeded: Response for request 1
Request 15 succeeded: Response for request 15
Request 17 succeeded: Response for request 17
Request 19 succeeded: Response for request 19
All requests completed.
```

Slika 4: Konzolni ispis (slika ekrana)

Rast upotrebljava *tokio*, moćnu asinhronu *runtime* biblioteku koja omogućava konkurentno izvršavanje koristeći *future* i *async/await* sintaksu. Apstrakcija *future* predstavlja vrednost koja će biti

dostupna u nekom trenutku u budućnosti. Rast rešenje je slično Pajton rešenju, ali podržava moćne koncepte vlasništva i sigurnosti tipova.

Ključna poređenja:

- Performanse - Rast, uz *tokio*, omogućava performantno asinhrono programiranje sa efikasnim korišćenjem resursa. Pajton sa *asyncio* pruža jednostavnu implementaciju, ali je znatno sporiji.
- Zajednica i ekosistem - Pajton ima veliku zajednicu i razvijen ekosistem za asinhrono programiranje, što olakšava pronalaženje alata i biblioteka. Rastova zajednica je manja, ali brzo raste, posebno u domenima računarstva visokih performansi.
- Težina učenja - Pajtonov *asyncio* je jednostavan za razumevanje i upotrebu, što ga čini pogodnim za početnike u asinhronom programiranju. Rast, iako moćniji, znatno je teži za učenje zbog kompleksnosti vlasništva i tipova.
- Sigurnost tipova - Rast osigurava sigurnost tipova i detektuje greške tokom kompajliranja, što smanjuje rizik od *runtime* grešaka u asinhronom kodu. Pajton je dinamički tipiziran, što može povećati šanse za greške u izvršavanju.
- Skalabilnost - Rast koristi minimalne resurse, i to ga čini idealnim kandidatom za implementaciju servera sa rastućim brojem istovremenih konekcija. Pajton je dobro rešenje, ali manje efikasno zbog ograničenja samog jezika.

5. Zaključak

Asinhrono programiranje predstavlja moćan pristup za upravljanje konkurentnim zadacima bez potrebe za stvaranjem velikog broja niti, što dovodi do efikasnijeg korišćenja resursa, posebno u I/O-intenzivnim aplikacijama. Pajton je izuzetno jednostavan za upotrebu zahvaljujući *asyncio* biblioteci, što ga čini idealnim kandidatom za brzi razvoj aplikacija manje složenosti, iako mu primitiva GIL (Global Interpreter Lock) ograničava mogućnost paralelnog razvoja. S druge strane, Rast omogućava potpunu paralelizaciju, što daje prostora za proširenje reprezentativnog primera i čini ga izuzetnim izborom za aplikacije koje zahtevaju visoke performanse. On se oslanja na sistem tipova i vlasništvo kako bi garantovao sigurnost kod konkurentnih operacija.

Izbor između Pajtona i Rasta zavisi od specifičnih zahteva aplikacije. Pajton je odličan za brzi razvoj I/O-intenzivnih sistema, dok Rust, iako teži za učenje, pruža značajnu prednost u skalabilnosti i sigurnosti, što ga čini idealnim za aplikacije u računarstvu visokih performansi.

6. Literatura

- [1] "Asynchronous Programming. Threads and Processes",
<https://luminousmen.com/post/asynchronous-programming-threads-and-processes>,
poslednji pristup 1.1.2025.

- [2] "Multi-Threading vs Asynchronous programming. What is the difference?",
<https://blog.devgenius.io/multi-threading-vs-asynchronous-programming-what-is-the-difference-3ebfe1179a5>, poslednji pristup 1.1.2025.