
Факултет техничких наука

Универзитет у Новом Саду

Паралелне и дистрибуиране архитектуре и језици

---

# Поређење имплементације API решења у Rust и Java програмским језицима

---

*Аутор:*
Лука Ћирић

*Индекс:*
E2 18/2023

23. децембар 2023.

# Садржај

# Списак изворних кодова

# Списак слика

# Списак табела

# 1 Увод

Ефикасна и економична комуникација између различитих система и сервиса игра кључну улогу у иновацијама, а развој и анализа апликационих програмских интерфејса (API) представљају срж овог напретка. У овом раду фокус ће бити на поређење техничких аспеката приликом имплементације REST API-ја у програмским језицима Rust и Java. Ове две технологије су изабране због својих различитих карактеристика, а анализа ће бити изведена од самог почетка - од имплементације API-ја преко TCP утичница (socket).

У истраживању се врши пролазак кроз различите аспекте обе апликације, са фокусом на обраду GET и PUT захтева, примењујући приступе који имају своја посебна тежишта. Обе апликације ће бити изграђене користећи TCP утичнице, омогућавајући анализу операција на ниском нивоу и њихове релевантности у контексту протокола HTTP.

Даље, истраживање ће се ширити на њихове техничке разлике, узимајући у обзир аспекте као што су брзина извршавања, ефикасност генерисаног кода, комплексност имплементације кода и начин обраде REST захтева. Посебно, разгледа се како обе апликације управљају грешкама у овом контексту.

У овај рад се улази са циљем пружања дубоког увида у различите аспекте имплементације REST API-ја у Rust и Java програмским језицима, кроз поглед на код и перформансе ових апликација. Кроз ова истраживања добија се увид у разумевање тога како избор програмског језика и технологије може утицати на дизајн и функционалности REST API-ја.

Подаци који се користе представљају корисника (User) који поседује својства корисничко име типа String, пин типа Int и зарада типа Double.

# 2 Rust имплементација

Rust је програмски језик пројектован са фокусом на безбедност и ефикасност. Обезбеђује механизме контроле за време извршавања и систем за управљање меморијом који спречавају бројне типичне грешке у програмима. Такође, Rust компајлер покушава да реши све проблеме кроз компајлирање добрим праксама и предвиђањем понашања кода.

Rust нуди мултипарадигматски приступ програмирању, обухваћајући функционално, објектно-оријентисано и конкурентно програмирање. Ова својстава допуштају разноврсне примене, од управљања системима до веб развоја.

Rust програмски језик је константно у развоју, где заједница доприноси унапређењу језика и технологије. Идеја иза сваке наредне верзије је да и претходне верзије буду у потпуности компатибилне, за разлику од других језика као што је Python. Такође, занимљивост је да се Rust компајлер компајлира у својој претходној верзији чиме је овај језик тако уникатан.

Rust је погодан за системско програмирање и развој оперативних система, као и креирање брзих и безбедних системских апликација.

Решење је покретано коришћењем cargo окружења.

## 2.1 Пакети коришћени у решењу и оптимизација

Cargo, систем за управљање зависностима и компајлирање у Rust-у, пружа могућност детаљног контролисања нивоа оптимизације током компајлирања. Корисници могу лако поставити различите опције оптимизације кроз Cargo.toml 1 фајл користећи opt-level поље, што укључује оптимизацију за брзину или величину извршног кода. Опције:

1. **0** - Минимизира време компајлирања и обезбеђује лакше дебаговање. Међутим, извршни код може бити спорија, а размера извршног фајла већа.

2. **1** - Пружа умерене оптимизације за брзину извршавања, са компромисом у вези с величином извршног кода.

3. **2** - Средњи ниво оптимизације балансира између брзине и величине извршног кода, представљајући компромис између ниског и високог нивоа.

4. **3** - Фокусиран на постизање максималне брзине, обично на штету размере извршног фајла.

5. **s** - Специфична за Rust и поставља компајлер да се фокусира на минимизацију величине извршног кода.

6. **z** - Комбинује минимизацију величине са задржавањем Debug информација, што може бити корисно за профилисање и дебаговање.

Коришћен је максимални ниво оптимизације.

Коришћени су пакети *serde* и *serde_json* за десеријализацију доспелих података из JSON формата у објекат и за серијализацију података за слање из објекта у JSON формат.

```
[package]
name = "rust-solution"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at
  ↪ https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
serde = { version = "1.0", features = ["derive"]}
serde_json = "1.0"

[profile.dev]
opt-level = 3
```

Изворни код 1: Пакети и ниво оптимизације

Такође, потребно је било наследити својства serde пакета у за структуру *User* кроз линију `#[derive(Serialize, Deserialize, Debug)]`.

```
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize, Debug)]
pub struct User {
    pub username: String,
    pub pin: i64,
    pub zarada: f64,

}
```

Изворни код 2: User структура

Унутар решења коришћена је *HashMap* структура као *Key-Value* складиште за податке. *Read* и *Write* су коришћени за читање из *TcpStream*-а у бафер. *TcpListener* ослушкује и прихвата захтеве од клијената и прихвата *TcpStream*-ове од њих. *Arc* и *RwLock* слуше за синхронизацију унутар нити, док *thread* ствара нове нити на захтев.

```
use std::collections::HashMap;
use std::io::{Read, Write};
use std::net::{TcpListener, TcpStream};
use std::sync::{Arc, RwLock};
use std::thread;
```

Изворни код 3: Пакети коришћени унутар самог решења која потичу из Rust програмског језика

## 2.2 Код имплементације

У *main* функцији се иницијализује *TcpListener*, након тога ствара *Key-Value* складиште за податке као *HashMap*-у. Да би складиште било дељено обмотано је *Arc* (Atomic Reference Counter) који дозвољава копије за вишенитни приступ. Да би приступ складишту био синхрон, складиште је обмотано *RwLock* којим ствара опције за закључавање складишта за читање и за писање посебно.

Линијом `for stream in listener.incoming()` се прикупљају конекције у виду *Result* енумерације након чега се обрађује да ли је *Result* вредност *Ok* или *Err* за грешку. Уколико је *Ok* клонира се *Arc* и просеђује нити која ће обрадити захтев.

```rust
fn main() {
    let listener =
     ↪  TcpListener::bind("0.0.0.0:8080").unwrap();
    println!("Server listening on port 8080...");

    let map = Arc::new(RwLock::new(HashMap::<String,
     ↪  User>::new()));

    for stream in listener.incoming() {
        match stream {
            Ok(stream) => {
                let cloned_map = map.clone();
                thread::spawn(|| handle_client(stream,
                 ↪  cloned_map));
            }
            Err(e) => {
                println!("Error: {}", e);
            }
        }
    }
}
```

Изворни код 4: *main* функција

У *handle_client* функцији се прихвата податак, чита из *TcpStream* и након тога се обрађује ког је типа захтев. Ако је захтев *GET* или *PUT* онда се обрађује, ако није клијенту се враћа одговор да захтев није подржан.

```rust
fn handle_client(mut stream: TcpStream, map:
 ↪  Arc<RwLock<HashMap<String, User>>>) {
    let mut buffer = [0; 2048];
    stream.read(&mut buffer).unwrap();

    let request = String::from_utf8_lossy(&buffer[..]);
    let response = if request.starts_with("GET") {
        handle_get(request.to_string(), map)
    } else if request.starts_with("PUT") {
        handle_put(request.to_string(), map)
    } else {
        "HTTP/1.1 400 Bad Request\r\n\r\nInvalid
         ↪  request".to_string()
    };

    stream.write_all(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

Изворни код 5: *handle_client* функција

Приликом обраде *GET* захтева, проналази се да ли постоји унутар захтева "?username=". У случају да не постоји, повратна вредност ће бити све редности из мапе. Уколико се пронађе извлачи се вредност "username", мапа се закључава за читање, након чега се претражује да ли постоји у мапи (кључ мапе представља "username" поље). У случају проналаска таквог корисника враћа се његова вредност унутар одговора у JSON формату, а у суротном се шаље одговор да корисник није нађен.

```rust
fn handle_get(request: String, map:
    Arc<RwLock<HashMap<String, User>>>) -> String {
    let username_index =
        request.find("?username=").unwrap_or(0);
    let return_data: String;
    if username_index != 0 {
        let username_index = username_index + 10;
        let username_end_index =
            &request[username_index..].find('
            ').unwrap_or(0);
        let username =
            &request[username_index..=username_index +
            *username_end_index - 1];
        let map = map.read().unwrap();
        if let Some(user) = map.get(username) {
            return_data = serde_json::to_string_pret-
                ty(&user).unwrap();
        } else {
            return format!("HTTP/1.1 404 Not
                Found\r\n\r\nUser '{}' not found",
                username);
        }
    } else {
        let map = map.read().unwrap();
        let values: Vec<_> = map.values().collect();
        return_data =
            serde_json::to_string_pretty(&values).unwrap();
    }

    format!(
        "HTTP/1.1 200 OK\r\n\r\n{}", return_data
    )
}
```

Изворни код 6: *handle_get* функција

Приликом обраде *ПУТ* захтева, претражује се захтев за почетак и крај JSON тела захтева. У случају да није пронађено, враћа се одговор да захтев није исправан. Ако се пронађе JSON, преводи се у објекат уз помоћ *serde_json* пакета и ту се проверава

да ли је JSON добро десеријализован. Уколико јесте, мапа се закључава за уписивање и уписује се у мапу. Мапа аутоматски прегази податке ако у мапи постоји кључ "username" или додаје ако не постоји.

```rust
fn handle_put(request: String, map:
    Arc<RwLock<HashMap<String, User>>>) -> String {
    let data_start = request.find('{').unwrap_or(0);
    let data_end = request.rfind('}').unwrap_or(0);
    if data_start == 0 || data_end == 0 {
        return format!("HTTP/1.1 400 Bad
            Request\r\n\r\nInvalid request");
    }
    let data = &request[data_start..=data_end];
    match serde_json::from_str::<User>(data) {
        Ok(new_data) => {
            {
                let mut map = map.write().unwrap();
                map.insert(new_data.username.clone(),
                    new_data);
            }

            format!(
                "HTTP/1.1 200 OK\r\n\r\n",
            )
        }
        Err(e) => {
            eprintln!("Error deserializing JSON: {}, {}",
                e, request);
            format!("HTTP/1.1 400 Bad Request\r\n\r\nError
                deserializing JSON.")
        }
    }
}
```

Изворни код 7: *handle_put* функција

# 3 Java имплементација

Java је објектно-оријентисани програмски језик који се истиче својом портабилношћу, што значи да програми написани у Java језику могу радити на различитим оперативним системима без потребе за изменама. Ова језичка особина чини Java језик изузетно погодном за развој cross-platform апликација. JVM је заслужан за портабилност.

Java виртуелна машина (JVM) је компонента Јава програмског језика која омогућава извршавање Java кода на рачунару. JVM интерпретира бајткод (бајтове кода) који је генерисан компајлирањем Java изворног кода.

Један од јаких аспеката Java језика је и сигурност. Примењује механизме безбедности као што су sandboxing и контрола приступа, што га чини погодним за развој апликација на отвореним мрежама, као што је интернет.

Java има богат стандардни библиотеку класа и API-ја која обухвата широк спектар функционалности. Ово у значајној мери олакшава развој апликација, пошто пружа многе унапред дефинисане функције и методе.

## 3.1 Коришћени пакети и окружење

Решење је писано у Java 21 верзији Java језика, покетано Maven окружењем које покреће компајлирање пројекта и омогућава дебаговање Java решења. Компајлер који се користи је **javac**. Поред стандарних пакета за обраду улаза и излаза, коришћен је пакет `java.net.ServerSocket` представља утичницу за прихватање конекција од клијената, а `java.net.Socket` представља клијентску утичницу. Такође, користе се пакети из *concurrent* библиотеке, где `java.util.concurrent.ExecutorService` представља сервис за покретање нити из базена нити (*thread pool*), `java.util.concurrent.Thre` креира сервис са базеном нити, а *TimeUnit* и *LinkedBlockingQueue* су помоћне класе.

```
1  import java.io.BufferedReader;
2  import java.io.BufferedWriter;
3  import java.io.IOException;
4  import java.io.InputStreamReader;
5  import java.io.OutputStreamWriter;
6  import java.net.ServerSocket;
7  import java.net.Socket;
8  import java.util.concurrent.ExecutorService;
9  import java.util.concurrent.LinkedBlockingQueue;
10 import java.util.concurrent.ThreadPoolExecutor;
11 import java.util.concurrent.TimeUnit;
```

Изворни код 8: Пакети коришћени унутар Java решења

## 3.2 Код имплементације

*User* класа садржи иста поља као и код Rust структуре. Због објектно-оријентисаних принципа, за приступ пољима су нам потребна својства која су јавна.

```java
public class User {
    private String username;
    private long pin;
    private double zarada;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public long getPin() {
        return pin;
    }

    public void setPin(long pin) {
        this.pin = pin;
    }

    public double getZarada() {
        return zarada;
    }

    public void setZarada(double zarada) {
        this.zarada = zarada;
    }

    public User() {
        super();
    }

    public User(String username, long pin, double zarada) {
        this.username = username;
        this.pin = pin;
        this.zarada = zarada;
    }
}
```

Изворни код 9: *User* класа

За *Key-Value* складиште користи се такође *HashMap*-а која је брза. У овом случају због лакшег приступа и могућности синхронизације направљена је класа чијим методама се приступа синхронизовано. Такође, кључ мапе је *username*. За додавање или измену података позива се *addUser* статичка метода, за добављање одређеног корисника *getUser* статичка метода, а за добављање свих корисника *getAllUsers* метода.

```
1  import java.util.HashMap;
2  import java.util.List;
3
4  public class Map {
5      private static final HashMap<String, User> users = new
       ↪  HashMap<>();
6      public static synchronized void addUser(User user) {
7          users.put(user.getUsername(), user);
8      }
9
10     public static synchronized User getUser(String
       ↪  username) {
11         return users.get(username);
12     }
13
14     public static synchronized List<User> getAllUsers() {
15         return users.values().stream().toList();
16     }
17 }
```

Изворни код 10: *Map* класа

*Main* метода на самом почетку креира *ServerSocket* за примање клијентских конекција и у петљи за сваку конекцију коју прихвати (`Socket clientSocket = serverSocket.accept();`) позива нову нит да обради њен захтев.

```
private static final int PORT = 8080;
private static final ExecutorService executorService = new
 ↪  ThreadPoolExecutor(50, 1000, 100,
 ↪  TimeUnit.MILLISECONDS, new LinkedBlockingQueue<>());
public static void main(String[] args) {
    try (ServerSocket serverSocket = new
     ↪  ServerSocket(PORT)) {
        System.out.println("Server is listening on port " +
         ↪  PORT);
        while (true) {
            Socket clientSocket = serverSocket.accept();
            executorService.submit(() ->
             ↪  handleRequest(clientSocket));
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Изворни код 11: *main* метода

Приликом позивања методе *handleRequest* креира се *BufferedReader* и *BufferedWriter* за читање из *stream*-а захтева и за уписивање у одговор. Линија `while (!reader.ready()) continue;` је потребна да бафер буде спреман за читање. Након тога, учитава се прва линија захтева и проверава који је тип захтева. У случају *GET* захтева, позива се метода *handleGet*, у случају *PUT* позива се *handlePut*, а у другим случајевима се одговара да метода није подржана. Обе методе прихватају и *ObjectMapper* који серијализује и десеријализује податке између објекта и JSON формата. На крају захтева је обавезно затворити утичницу.

```java
private static void handleRequest(Socket clientSocket) {
    try (BufferedReader reader = new BufferedReader(new
     ↪  InputStreamReader(clientSocket.getInputStream()));
        BufferedWriter writer = new BufferedWriter(new
         ↪  OutputStreamWriter(clientSocket.getOutputStre-
         ↪  am())))
        while (!reader.ready()) continue;

        String request = reader.readLine();
        ObjectMapper objectMapper = new ObjectMapper();
        if (request.startsWith("GET")) {
            writer.write(handleGet(request, objectMapper));
        } else if (request.startsWith("PUT")) {
            writer.write(handlePut(request, objectMapper,
             ↪  reader));
        } else {
            writer.write("HTTP/1.1 405 Method Not
             ↪  Allowed\r\n\r\nUnsupported method!");
        }
        writer.flush();
        clientSocket.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Изворни код 12: *handleRequest* метода

Приликом обраде *GET* захтева, проверава се да ли у захтеву постоји "?username=", ако не постоји враћа све податке из мапе у JSON формату. У случају да постоји, извлачи вредност "username" и претражује је у мапи, ако не постоји враћа одговор о томе, ако постоји враћа вредност у JSON формату.

```
1  private static String handleGet(String request,
    ↪  ObjectMapper objectMapper) {
2      try {
3          String response_data = "";
4          int username_index = request.indexOf("?username=");
5          if(username_index >= 0) {
6              username_index += 10;
7              int username_end = request.indexOf(" ",
                ↪  username_index);
8              String username =
                ↪  request.substring(username_index,
                ↪  username_end);
9              User user = Map.getUser(username);
10             response_data =
                ↪  objectMapper.writeValueAsString(user);
11         } else {
12             response_data = objectMapper.writeValueAs-
                ↪  String(Map.getAllUsers());
13         }
14         if(response_data == null)
15                 return "HTTP/1.1 404 Not Found\r\n\r\nUser
                    ↪  not found!";
16         return "HTTP/1.1 200 OK\r\n\r\n" + response_data;
17     } catch (Exception e) {
18         e.printStackTrace();
19         return "HTTP/1.1 500 Internal Server Error\r\n\r\n"
            ↪  + e.getMessage();
20     }
21 }
```

Изворни код 13: *handleGet* метода

Приликом обраде *PUT* захтева, прослеђује се и *reader* да би се прочитали подаци из тела захтева. Након читања података се претражује тело да ли садржи JSON формат. У случају да не садржи, враћа се грешка о лошем захтеву. Ако садржи, извлачи се JSON објекат и преводи из JSON формата у објекат.

```java
private static String handlePut(String request,
    ObjectMapper objectMapper, BufferedReader reader) {
    try {
        StringBuilder requestBody = new StringBuilder();
        while (reader.ready()) {
            requestBody.append((char) reader.read());
        }
        Integer bodyStart = requestBody.indexOf("{"),
            bodyEnd = requestBody.lastIndexOf("}") + 1;
        if(bodyStart < 0 || bodyEnd < 0) {
            return "HTTP/1.1 400 Bad Request\r\n\r\nInvalid
                request body!";
        } else {
            String body = requestBody.substring(bodyStart,
                bodyEnd);
            User user = objectMapper.readValue(body,
                User.class);
            Map.addUser(user);
            return "HTTP/1.1 200 OK\r\n\r\nReceived PUT
                request!";
        }
    } catch(Exception e){
        e.printStackTrace();
        return "HTTP/1.1 500 Internal Server Error\r\n\r\n"
            + e.getMessage();
    }
}
```

Изворни код 14: *handlePut* метода

# 4 Поређење решења

## 4.1 Комплексност кода

Комплексност кода је врло слична у оба случаја, приступи су приближно исти. Поређењем главних функија 4 и 11, постоји разлика у приступу мапи и у начину производње нити. У Java језику, потребно је створити базен нити одакле се те нити повлаче, док код Rust језика је довољно само позвати производњу нове нити. Није немогуће у Java језику уради исти приступ, али би перформансе знатно опале. Такође, мапа се инстанцира унутар *main* функције у Rust решењу, а у Java решењу се прави класа са статичким пољима. Комплексност приступа и синхронизовања мапе је мало већа код Rust-а, али принцип је јако сличан.

Метода *handleRequest* у Java решењу 12 у поређењу са функцијом *handle_client* из Rust решења је јако слична. Приступи су слични, разлика је читање и уписивање у *stream*. У Rust језику је то обезбеђено директно над *stream*-ом, док у Java језику нису обавезни додатни бафери, али знатно доприносе перформансама брзине.

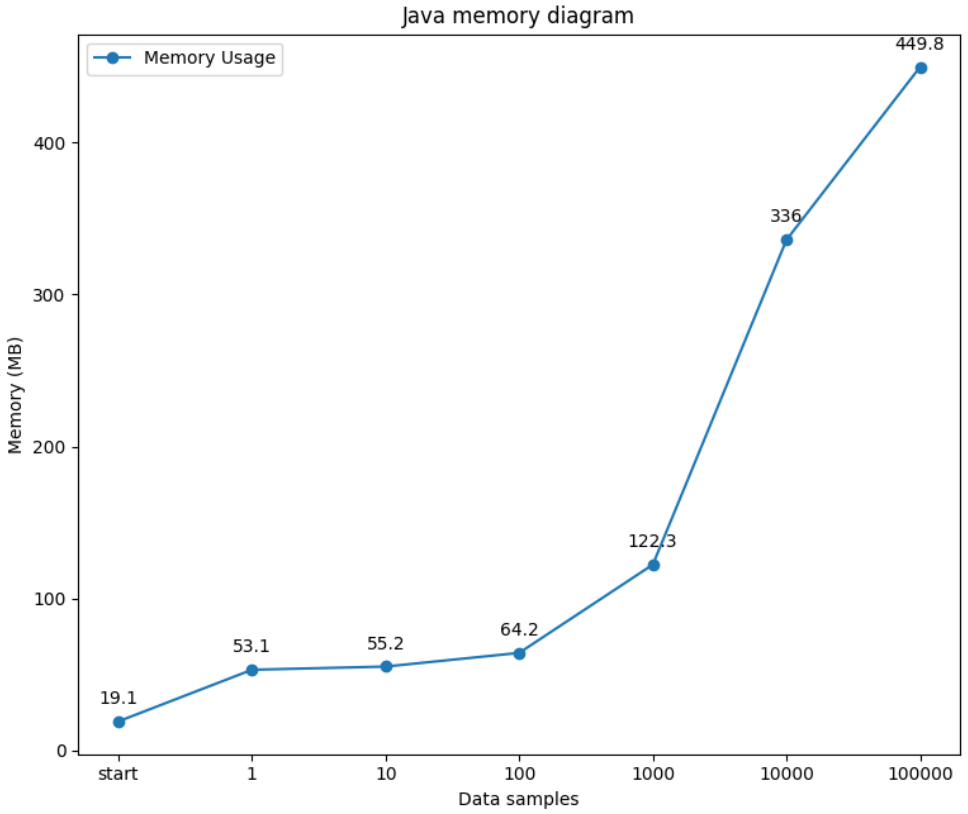
Метода *handleGet* у Java решењу 13 је слична са функцијом *handle_get* Rust решењем 6, с том разликом да је потребно направити *try-catch* унутар методе који прави додатни стек у случају изузетка.

Метода *handlePut* у Java решењу 14 у поређењу са функцијом *handle_put* у Rust решењу 7 се мало више разликује. Потребно је направити *try-catch* у Java решењу и такође исчитати део захтева. У Rust решењу, цео захтев се прочитао раније. У случају да JSON формат не може да се преведе у Java језику се добија изузетак, а у Rust језику се проверава вредност *Result* енумерације.

## 4.2 Перформансе

Решења су стављана под различита оптерећења како би се тачно утврдиле разлике и када долази до њих.

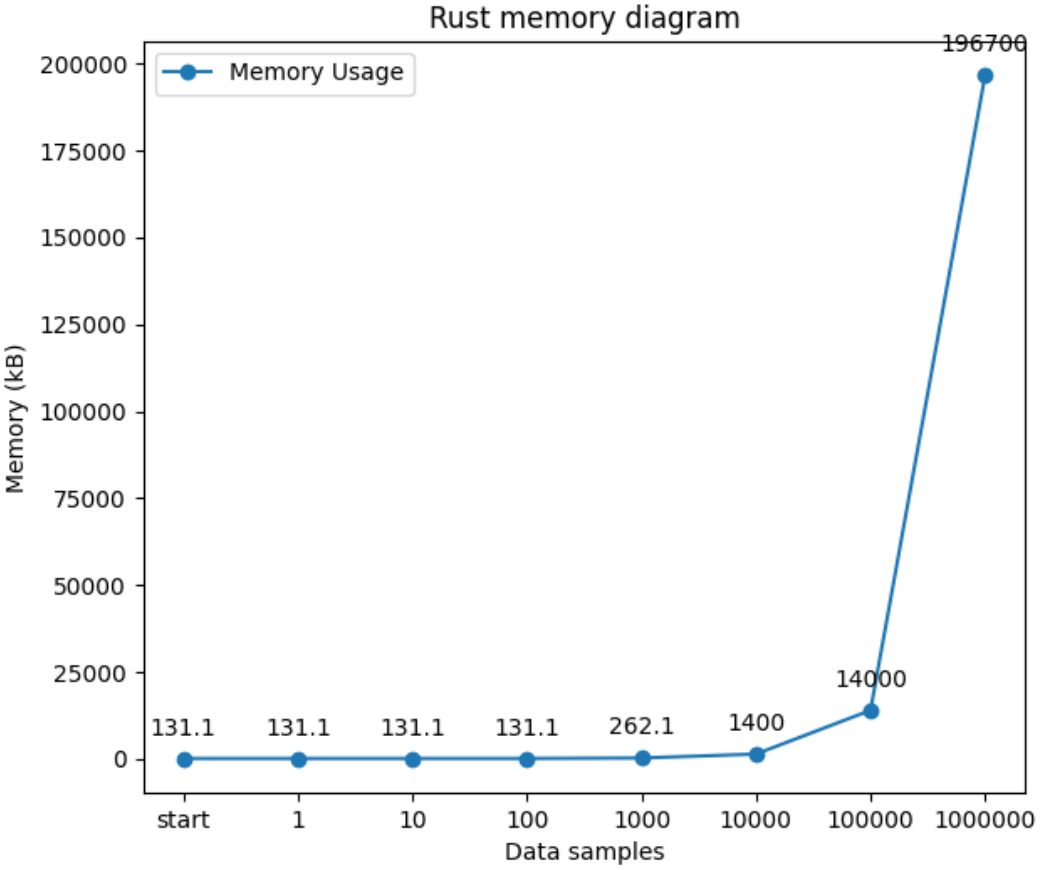
Посматрањем дијаграма 1 апликација се покреће са меморијом од **19.1MB** где приликом активације првог захтева за конекцију са утичницом меморија апликације порасте на **53.1MB**. Велики раст меморије код апликације се јавља заузимањем ресурса које утичнице користе. Тиме, почетак је спорији од остатка извршавања. Такође, скалирање меморије приликом уписа је доста варијабилно, тестирањем у пар наврата се добију приближне вредности меморије, али не исте.




Слика 1: Меморија Java решења

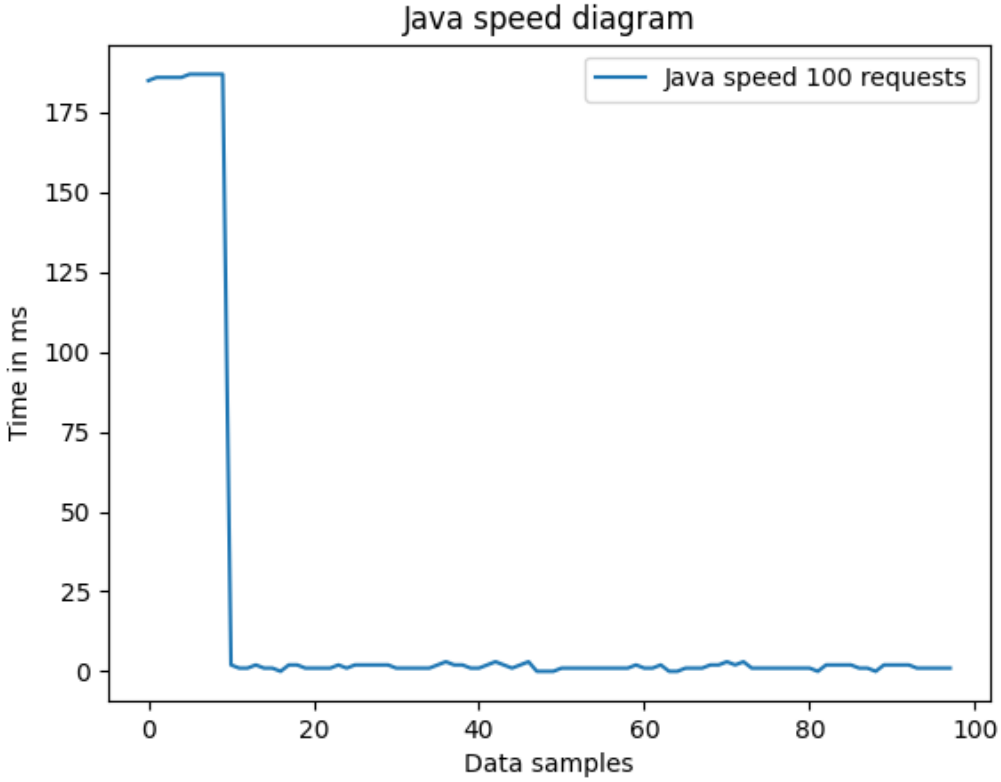
Поређењем 1 и 2 примети се да је први дијаграм изражен у MB (мегабајтима), а други у kB (килобајтима), чиме се примети знатна разлика у заузимању меморије ова два програма. Такође, Rust апликација успева јако лако подржати милион података, док код Java апликаје не постоје услови да се то тестира због великог раста меморије. Rust алоцира меморију доста компактније и мирније, чиме се у меморији од почетка апликације није приметио раст све до 1000 записа.

Додатно, тестирано је са максималном оптимизацијом и без оптимизације и заузимање меморије је исто.



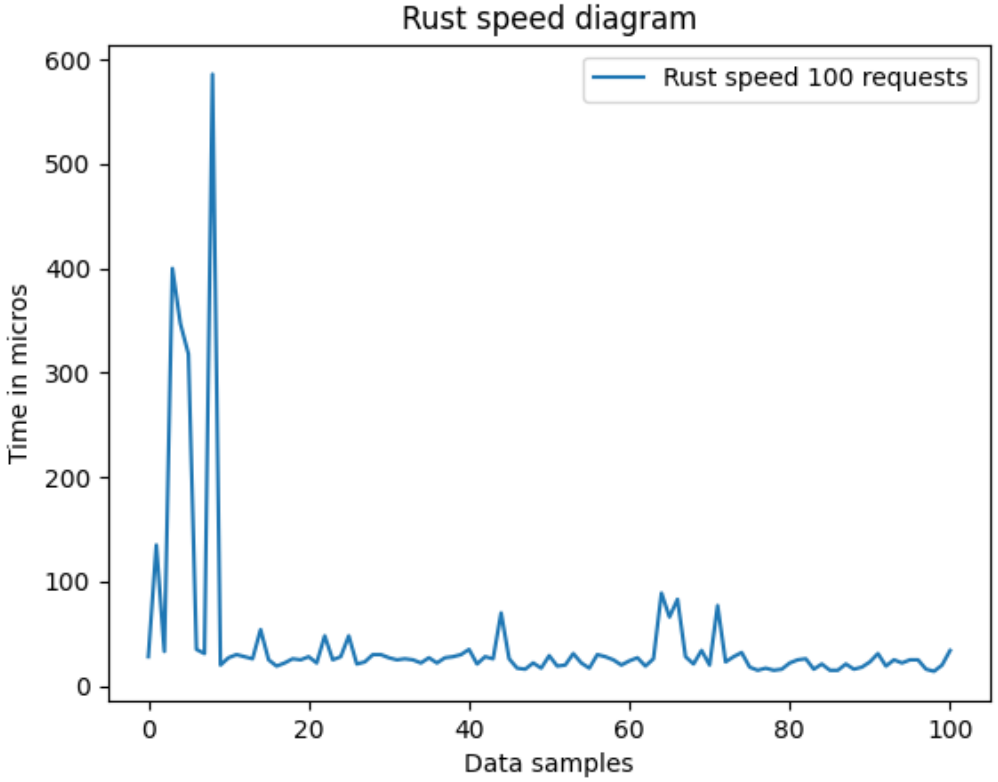

Слика 2: Меморија Rust решења

Приликом мерења брзине извршавања унутар обраде захтева 3 почетне брзине извршавања су знатно спорије од каснијих. Приликом покретања првих нити Java заузима ресурсе чиме ово доводи до обрзања у каснијим итерацијама зато што није потребно да се поново учитају ресурси.




Слика 3: Брзина Java решења

У Rust апликацији, да би се приметила разлика, потребно је презентовати податке у микросекундама. Такође, на почетку програма види се раст за заузимање ресурса, али знатно мање вредности него у Java решењу. Поређењем ова два дијаграма (3 и 4), примети се знатна разлика у брзини обраде захтева код ове две апликације.




Слика 4: Брзина Rust решења

Тестирањем са истог рачунара позивају се подпроцеси који додају или претражују податке. У следећој табели су приказане резултати како се показала која апликација на овај стрес тест.

| **Тестни случај** | **Java апликација (s)** | **Rust апликација (s)** |
|---|---|---|
| 50 подпроцеса, 50.000 података | 7.1 | 6.15 |
| 50 подпроцеса, 200.000 података | 23.5 | 20.1 |
| 10 подпроцеса, 20.000 упита | 2.82 | 2.23 |
| 50 подпроцеса, 200.000 упита | 28.9 | 30.1 |

Табела 1: Поређење извршавања Java і Rust апликација

Тестирањем са другог рачунара позивају се подпроцеси који додају или претражују податке. У следећој табели су приказане резултати како се показала која апликација на овај стрест тест.

| **Тестни случај** | **Java апликација (s)** | **Rust апликација (s)** |
|---|---|---|
| 10 подпроцеса, 20.000 података | 35.3 | 39.5 |
| 10 подпроцеса, 20.000 упита | 35.38 | 42.1 |

Табела 2: Поређење извршавања Java i Rust апликација позивом са другог рачунара

Поређењем табела 1 и 2 закључује се да мрежни приступ са другог рачунара знатно успорава приступ подацима. Такође, у другој табели се перформантније показала Java апликација, чиме се може довести до закључка да Java боље управља мрежним саобраћајем.

Процесор и меморија на рачунару где је покренута апликација 1:

- Ryzen 7 6800H са 16 језгара 3.2GHz
- 16GB DDR5 4800MHz

Процесор и меморија на рачунару где је покренут други стрес тест 2:

- Ryzen 5 2500U са 8 језгара 2GHz
- 8GB DDR4 2400MHz

# 5 Закључак

Да ли постоји боље решење између ова два? Одговор би био зависи. Rust апликација заузима знатно мање меморије и знатно брже врши обраду. Такође, Rust апликација ради без базена нити, који би потенцијално убрзао извршавање апликације. Java прилично добро ради са мрежом, има доста пакета који су намењени за рад са мрежом и претежно је ка томе оријентисана.

У случају потребе за малом и ефикасном апликацијом, Rust језик решава то одлично, ако меморија није ограничење, једноставније је имплементирати Java апликацију и могуће је имати добре перформансе.

# 6 Литература

1. Rust Documentation, https://www.rust-lang.org/learn
2. Java Documentation, https://docs.oracle.com/en/java/