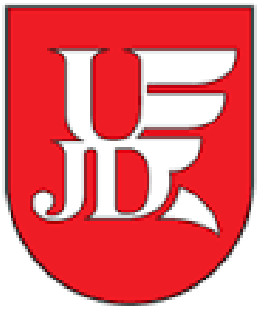
UNIWERSYTET JANA DŁUGOSZA W CZĘSTOCHOWIE



Wydział Nauk Ścisłych, Przyrodniczych i Technicznych

Kierunek: **Informatyka**
Specjalność: **Programowanie gier komputerowych**

Przemysław Kamiński
Nr albumu: **88751**

Sposoby optymalizacji aplikacji internetowych. Ways to optimize web applications

Praca magisterska
przygotowana pod kierunkiem
dr hab. Bożeny Woźnej-Szcześniak, prof. UJD

Częstochowa, 2025

Streszczenie

Celem niniejszej pracy jest porównanie wydajności protokołów komunikacyjnych stosowanych w aplikacjach webowych oraz analiza wpływu formatów wymiany danych na efektywność komunikacji w nowoczesnych systemach informatycznych. Badania obejmują protokoły HTTP/1.1, HTTP/2, GraphQL, gRPC, SOAP, WebSocket Secure oraz WebSocket, a także formaty danych takie jak JSON, XML, Protocol Buffers i MessagePack. Analiza koncentruje się na porównaniu rozmiaru przesyłanych danych, czasu przetwarzania oraz kompatybilności pomiędzy systemami.

Dodatkowo w pracy oceniono wpływ zastosowania technologii WebAssembly na wydajność przetwarzania danych po stronie klienta w porównaniu z tradycyjnym podejściem opartym na języku JavaScript. Eksperymenty zostały przeprowadzone z wykorzystaniem dedykowanej aplikacji testowej, umożliwiającej rzetelny pomiar parametrów wydajnościowych w kontrolowanym środowisku.

Uzyskane wyniki pozwalają określić zależności pomiędzy wyborem protokołów komunikacyjnych, formatów danych oraz technologii wykonawczych a osiąganą wydajnością systemu. Na ich podstawie sformułowano praktyczne wnioski i rekomendacje, które mogą wspierać proces projektowania i optymalizacji nowoczesnych aplikacji webowych oraz architektur rozproszonych.

Abstract

The aim of this thesis is to compare the performance of communication protocols used in web applications and to analyze the impact of data exchange formats on communication efficiency in modern information systems. The study covers protocols such as HTTP/1.1, HTTP/2, GraphQL, gRPC, SOAP, and WebSocket, as well as data formats including JSON, XML, Protocol Buffers, and MessagePack. The analysis focuses on data size, processing time, and cross-system compatibility.

Additionally, the thesis evaluates the impact of using WebAssembly on client-side data processing performance compared to the traditional JavaScript-based approach. The experiments were conducted using a dedicated test application that enabled reliable performance measurements in a controlled environment.

The obtained results allow identifying relationships between the choice of communication protocols, data formats, and execution technologies and the resulting system performance. Based on these findings, practical conclusions and recommendations are formulated to support the design and optimization of modern web applications and distributed architectures.

Słowa kluczowe: Rust, WebAssembly, JavaScript, Sieci komputerowe, Protokoły komunikacyjne, Optymalizacja, Serializacja, Deserializacja, Klient–serwer

Spis treści

Wstęp

Dynamiczny rozwój aplikacji webowych oraz rosnące wymagania dotyczące ich skalowalności, responsywności i niezawodności sprawiają, że wydajność komunikacji sieciowej staje się jednym z kluczowych aspektów projektowania nowoczesnych systemów informatycznych. Współczesne aplikacje coraz częściej opierają się na architekturach rozproszonych, w których wymiana danych pomiędzy klientem a serwerem, a także poszczególnymi usługami, odbywa się z wykorzystaniem różnych protokołów komunikacyjnych oraz formatów danych. Wybór odpowiednich technologii w tym zakresie ma bezpośredni wpływ na czas odpowiedzi systemu, obciążenie sieci oraz koszty przetwarzania po obu stronach komunikacji.

Celem niniejszej pracy magisterskiej jest analiza i porównanie wydajności wybranych protokołów komunikacyjnych stosowanych w aplikacjach webowych, a także ocena wpływu formatów wymiany danych oraz technologii wykonywania kodu po stronie klienta na ogólną efektywność systemu. W pracy szczególną uwagę poświęcono porównaniu protokołów HTTP/1.1, HTTP/2, GraphQL, gRPC, SOAP oraz WebSocket, które reprezentują różne podejścia do komunikacji w środowiskach sieciowych. Analizie poddano również popularne formaty danych, takie jak JSON, XML, Protocol Buffers oraz MessagePack, uwzględniając ich rozmiar, czas serializacji i deserializacji oraz kompatybilność pomiędzy różnymi systemami.

Dodatkowym celem pracy jest zbadanie wpływu zastosowania technologii WebAssembly na wydajność przetwarzania danych po stronie klienta w porównaniu z tradycyjnym podejściem opartym na języku JavaScript. W tym kontekście przeprowadzono eksperymenty mające na celu ocenę różnic w czasie wykonywania operacji, zużyciu zasobów oraz potencjalnych korzyściach wynikających z wykorzystania WebAssembly w aplikacjach webowych.

Praca została podzielona na cztery rozdziały.

W pierwszym rozdziale przedstawiono wprowadzenie do tematyki pracy, obejmujące podstawowe pojęcia związane z komunikacją w aplikacjach webowych oraz uzasadnienie podjęcia badań nad wydajnością protokołów i technologii wykorzystywanych w nowoczesnych systemach informatycznych.

Rozdział drugi poświęcono porównaniu protokołów komunikacyjnych stosowanych w aplikacjach webowych. Opisano w nim charakterystykę wybranych protokołów, takich jak HTTP/1.1, HTTP/2, GraphQL, gRPC, SOAP, WebSocket Secure oraz WebSocket, a także przedstawiono środowisko badawcze, wykorzystane narzędzia, biblioteki oraz sposób przeprowadzania eksperymentów wydajnościowych.

W rozdziale trzecim zaprezentowano różnice pomiędzy formatami danych wykorzystywanymi w komunikacji sieciowej. Omówiono takie aspekty jak rozmiar przesyłanych danych, czas przetwarzania oraz kompatybilność pomiędzy systemami na przykładzie formatów JSON, XML, Protocol Buffers oraz MessagePack.

Czwarty rozdział poświęcono analizie wydajności technologii WebAssembly w porównaniu z JavaScriptem. Przedstawiono w nim wyniki przeprowadzonych eksperymentów oraz ocenę wpływu obu podejść na wydajność aplikacji webowych.

Podsumowując, praca ma na celu dostarczenie praktycznych i eksperymentalnie potwierdzonych wniosków, które mogą stanowić wsparcie przy wyborze protokołów komunikacyjnych, formatów danych oraz technologii wykonawczych w projektowaniu wydajnych i nowoczesnych aplikacji webowych.

Rozdział 1

Wprowadzenie do problematyki doboru technologii

1.1 Znaczenie doboru technologii w aplikacjach webowych

Współczesne aplikacje webowe stanowią podstawę funkcjonowania wielu systemów informatycznych wykorzystywanych w biznesie, administracji publicznej oraz życiu codziennym. Rosnąca liczba użytkowników, potrzeba obsługi dużych wolumenów danych oraz wymagania dotyczące niskich czasów odpowiedzi powodują, że zagadnienia związane z wydajnością i skalowalnością systemów stają się kluczowe już na etapie projektowania architektury. Jednym z najważniejszych czynników wpływających na te cechy jest dobór odpowiednich technologii komunikacyjnych oraz formatów wymiany danych.

W architekturach rozproszonych komunikacja pomiędzy komponentami systemu odbywa się za pośrednictwem sieci komputerowej, która wprowadza dodatkowe opóźnienia oraz ograniczenia przepustowości. Niewłaściwy wybór protokołu komunikacyjnego lub formatu danych może prowadzić do nadmiernego obciążenia sieci, zwiększonego zużycia zasobów obliczeniowych oraz pogorszenia doświadczeń użytkownika końcowego. Z tego względu świadomy dobór technologii powinien być oparty nie tylko na ich popularności, lecz przede wszystkim na analizie wymagań danego problemu.

1.2 Charakterystyka współczesnych systemów rozproszonych

Nowoczesne systemy informatyczne coraz częściej projektowane są w oparciu o architektury rozproszone, takie jak architektura klient–serwer, mikroserwisy czy systemy oparte na komunikacji zdarzeniowej. W tego typu rozwiązaniach poszczególne komponenty systemu działają niezależnie i komunikują się ze sobą za pomocą jasno zdefiniowanych interfejsów.

Taki model projektowy umożliwia łatwiejszą skalowalność oraz rozwój systemu, jednak jed-

nocześnie zwiększa znaczenie wydajnej i niezawodnej komunikacji. Każde wywołanie zdalne wiąże się z kosztami transmisji danych, serializacji i deserializacji komunikatów oraz przetwarzania po obu stronach połączenia. W praktyce oznacza to, że nawet niewielkie różnice w zastosowanych technologiach mogą prowadzić do zauważalnych różnic w wydajności całego systemu.

1.3 Protokoły komunikacyjne jako fundament wymiany danych

Protokoły komunikacyjne definiują sposób, w jaki dane są przesyłane pomiędzy uczestnikami komunikacji. W aplikacjach webowych powszechnie wykorzystywane są protokoły oparte na rodzinie HTTP, jednak wraz z rozwojem technologii pojawiły się również alternatywne rozwiązania, takie jak gRPC czy WebSocket. Każdy z tych protokołów oferuje inne właściwości w zakresie wydajności, sposobu zestawiania połączeń, obsługi strumieniowania danych oraz kompatybilności z istniejącą infrastrukturą.

Dobór protokołu komunikacyjnego powinien uwzględniać charakter wymiany danych, częstotliwość komunikacji, wymagania dotyczące opóźnień oraz środowisko uruchomieniowe aplikacji. Przykładowo, protokoły oparte na długotrwałych połączeniach mogą być bardziej efektywne w aplikacjach czasu rzeczywistego, natomiast klasyczne podejście żądanie–odpowiedź sprawdza się w prostszych scenariuszach komunikacyjnych.

1.4 Uzasadnienie podjęcia badań

Różnorodność dostępnych protokołów komunikacyjnych, formatów danych oraz technologii wykonawczych sprawia, że projektanci systemów informatycznych stają przed trudnym zadaniem wyboru najbardziej odpowiednich rozwiązań. Brak jednoznacznych odpowiedzi oraz silne uzależnienie wyników od kontekstu zastosowania powodują, że decyzje te często podejmowane są intuicyjnie.

Celem niniejszej pracy jest dostarczenie empirycznych danych oraz praktycznych wniosków, które mogą wspierać proces podejmowania decyzji technologicznych. Przeprowadzone analizy i eksperymenty pozwalają lepiej zrozumieć zależności pomiędzy wyborem technologii a wydajnością aplikacji webowych.

1.5 Słownik pojęć i skrótów

API (*Application Programming Interface*) - Zbiór reguł i definicji umożliwiających komunikację pomiędzy różnymi komponentami oprogramowania [9].

HTTP (*Hypertext Transfer Protocol*) - Protokół komunikacyjny wykorzystywany do przesyłania danych w sieci WWW [8].

WebSocket - Protokół umożliwiający dwukierunkową komunikację w czasie rzeczywistym pomiędzy klientem a serwerem [12].

gRPC - Wysokowydajny framework komunikacyjny oparty na protokole HTTP/2 i formacie *Protocol Buffers* [7].

Serializacja - Proces przekształcania struktury danych do postaci umożliwiającej jej zapis lub transmisję [11].

Deserializacja - Proces odtwarzania struktury danych na podstawie jej zserializowanej reprezentacji [6].

WebAssembly (*Wasm*) - Binarny format wykonywalnego kodu umożliwiający uruchamianie aplikacji o wysokiej wydajności w przeglądarkach internetowych [5].

Architektura klient–serwer - Model architektoniczny, w którym klient inicjuje żądania, a serwer je obsługuje i zwraca odpowiedzi [10].

Rozdział 2

Technologie wykorzystane w projekcie

2.1 Rust

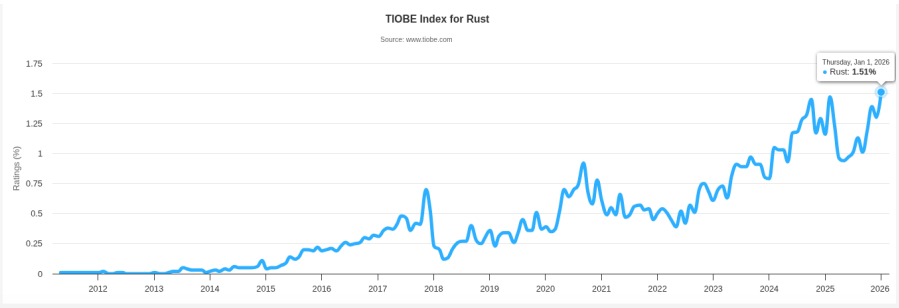
Rust jest nowoczesnym językiem programowania systemowego, który kładzie duży nacisk na bezpieczeństwo pamięci, wydajność oraz wielowątkowość. Dzięki mechanizmowi własności (ownership) oraz statycznej analizie błędów w czasie kompilacji, Rust minimalizuje ryzyko występowania błędów takich jak wycieki pamięci czy dereferencja pustych wskaźników. Rust zdobył popularność również dzięki nowoczesnemu systemowi typów i możliwości pisania kodu niskopoziomowego bez rezygnacji z bezpieczeństwa.

W projekcie Rust został wykorzystany do implementacji backendu aplikacji, w tym:

- obsługi żądań sieciowych,
- komunikacji asynchronicznej,
- implementacji serwisów gRPC oraz GraphQL,
- testów wydajnościowych.

Wybór języka Rust był podyktowany potrzebą uzyskania niskich czasów odpowiedzi oraz stabilności działania aplikacji przy dużej liczbie równoległych zapytań. Dodatkowo paczki wykorzystane do badań nie posiadały zbędnych funkcjonalności, co umożliwiło skupienie się na analizie wyników i minimalnej implementacji rozwiązań.

Według Tiobe Index, Rust zyskał największą dotychczas popularność 1 stycznia 2026 roku.



Rysunek 2.1: Pozycja języka Rust w rankingu Tiobe Index.

Jak widać na Rysunku 2.1, Rust zyskuje coraz większą popularność wśród programistów, co potwierdza rosnące zainteresowanie nim w przemyśle i projektach open source.

2.2 JavaScript

JavaScript jest językiem skryptowym powszechnie wykorzystywanym do tworzenia aplikacji webowych. W projekcie został użyty głównie jako punkt odniesienia dla tradycyjnych interfejsów użytkownika w porównaniu z technologią WebAssembly. Dzięki swojej uniwersalności i dużemu ekosystemowi bibliotek, JavaScript nadal pozostaje podstawowym językiem front-endowym.

2.3 Framework Actix-web

Actix-web jest asynchronicznym frameworkiem webowym dla języka Rust, opartym na modelu aktorów. Charakteryzuje się wysoką wydajnością oraz niskim narzutem czasowym, co sprawia, że jest jedną z najszybszych opcji w ekosystemie Rust.

W projekcie framework Actix-web został wykorzystany do:

- obsługi klasycznego API HTTP,
- implementacji endpointów GraphQL,
- obsługi żądań REST.

Framework Actix-web został wybrany ze względu na:

- wysoką wydajność potwierdzoną testami benchmarkowymi,
- dobrą integrację z ekosystemem Rust,
- wsparcie dla programowania asynchronicznego.

Rysunek 2.2: Framework Actix-web w porównaniu z innymi frameworkami.

Jak widać na Rysunku 2.2, Actix wypada bardzo korzystnie w kontekście wydajności i obsługi dużego ruchu, co jest istotne dla aplikacji wymagających niskich czasów odpowiedzi.

2.4 Framework Tonic

Tonic jest frameworkiem do implementacji gRPC w języku Rust, opartym na bibliotece `tokio` oraz protokole HTTP/2. W projekcie został wykorzystany do:

- implementacji serwera gRPC,
- definiowania kontraktów komunikacyjnych w postaci plików `.proto`,
- realizacji wydajnej komunikacji klient–serwer.

Framework Tonic umożliwił stworzenie stabilnej i szybkiej komunikacji typu RPC, co było kluczowe dla testów wydajnościowych i porównawczych z GraphQL.

2.5 tungstenite-rs

`tungstenite-rs` jest biblioteką w Rust umożliwiającą obsługę protokołu WebSocket. Została wykorzystana do:

- obsługi komunikacji w czasie rzeczywistym,
- przesyłania danych bez konieczności inicjowania nowych połączeń HTTP,
- testów alternatywnych modeli komunikacji.

2.6 h2load

`h2load` jest narzędziem służącym do testowania wydajności aplikacji korzystających z protokołu HTTP/2. W projekcie zostało wykorzystane do:

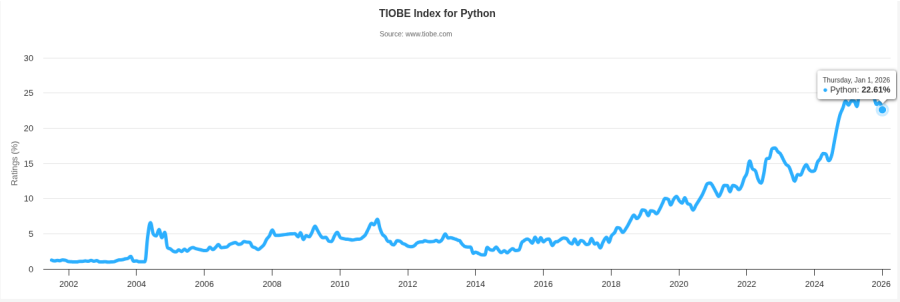
- generowania dużej liczby równoległych zapytań,
- pomiaru czasów odpowiedzi serwera,
- analizy zachowania systemu pod obciążeniem.

2.7 Python

Python jest językiem wysokiego poziomu, który w projekcie został użyty głównie pomocniczo, do:

- analizy wyników testów wydajnościowych,
- generowania wykresów porównawczych,
- przetwarzania danych pomiarowych.

Python pozostaje jednym z najpopularniejszych języków programowania, co obrazuje Rysunek 2.3.



Rysunek 2.3: Pozycja języka Python w rankingu Tiobe Index.

2.8 GraphQL

GraphQL jest językiem zapytań do API, pozwalającym klientowi precyzyjnie określić, jakie dane są potrzebne. W projekcie został wykorzystany do:

- porównania modelu komunikacji GraphQL z gRPC,
- realizacji zapytań o złożone struktury danych,
- testów wydajnościowych i porównawczych.

2.9 gRPC

gRPC jest mechanizmem komunikacji RPC opartym na protokole HTTP/2 oraz formacie binarnym Protocol Buffers. W projekcie zostało wykorzystane do:

- komunikacji między komponentami systemu,
- przeprowadzania testów wydajnościowych,
- porównania z innymi podejściami komunikacyjnymi (np. GraphQL).

2.10 WebAssembly

WebAssembly (Wasm) jest binarnym formatem wykonywalnym, pozwalającym uruchamiać kod napisany w Rust bezpośrednio w przeglądarce. W projekcie zostało wykorzystane do:

- uruchamiania części logiki aplikacji po stronie klienta,
- eksperymentalnego porównania wydajności z klasycznym JavaScriptem,
- zwiększenia bezpieczeństwa i wydajności aplikacji webowej.

Rozdział 3

Użyte narzędzia programistyczne

3.1 Git

Git jest rozproszonym systemem kontroli wersji, którego głównym celem jest zarządzanie historią zmian w kodzie źródłowym projektu programistycznego [1]. Narzędzie to umożliwia rejestrowanie kolejnych wersji plików, analizę wprowadzonych modyfikacji oraz powrót do wcześniejszych stanów projektu. Dzięki rozproszonej architekturze każdy użytkownik posiada pełną kopię repozytorium wraz z całą historią zmian, co zwiększa niezawodność oraz elastyczność pracy zespołowej.

Git oferuje mechanizmy takie jak gałęzie (ang. *branches*) oraz scalanie zmian (ang. *merge*), które pozwalają na równoległą pracę nad różnymi funkcjonalnościami bez ingerencji w główną wersję projektu. System ten jest szeroko stosowany zarówno w małych projektach indywidualnych, jak i w dużych przedsięwzięciach komercyjnych oraz otwartoźródłowych.

3.2 Zed

Zed jest nowoczesnym edytorem kodu źródłowego zaprojektowanym z myślą o wysokiej wydajności, niskich opóźnieniach oraz współczesnych potrzebach programistów [13]. Narzędzie to zostało stworzone przy wykorzystaniu języka Rust, co przekłada się na bezpieczeństwo pamięci oraz wysoką responsywność interfejsu użytkownika.

Edytor oferuje wsparcie dla wielu języków programowania, inteligentne podpowiedzi kodu, integrację z systemami kontroli wersji oraz możliwość pracy zespołowej w czasie rzeczywistym. Zed kładzie duży nacisk na minimalizm interfejsu oraz płynność pracy, co czyni go atrakcyjną alternatywą dla bardziej rozbudowanych środowisk programistycznych.

3.3 Visual Studio Code

Visual Studio Code jest wieloplatformowym edytorem kodu źródłowego rozwijanym przez firmę Microsoft [2]. Narzędzie to łączy w sobie prostotę edytora tekstu z funkcjonalnością zintegrowanego środowiska programistycznego (IDE). Dzięki rozbudowanemu systemowi rozszerzeń możliwe jest dostosowanie edytora do pracy z niemal dowolnym językiem programowania oraz frameworkiem.

Visual Studio Code oferuje funkcje takie jak podświetlanie składni, inteligentne uzupełnianie kodu, debugowanie aplikacji oraz integrację z systemem Git. Jego popularność wynika z dużej elastyczności, aktywnej społeczności oraz regularnych aktualizacji, co czyni go jednym z najczęściej wybieranych narzędzi programistycznych.

3.4 NPM

NPM (Node Package Manager) jest menedżerem pakietów przeznaczonym dla środowiska Node.js [3]. Jego głównym zadaniem jest zarządzanie zależnościami projektu, w tym instalowanie, aktualizowanie oraz usuwanie bibliotek zewnętrznych. NPM umożliwia również publikowanie własnych pakietów w publicznym repozytorium, co wspiera ponowne wykorzystanie kodu.

Centralnym elementem działania NPM jest plik `package.json`, w którym definiowane są informacje o projekcie, jego zależnościach oraz skryptach automatyzujących typowe zadania, takie jak budowanie aplikacji czy uruchamianie testów. Narzędzie to stanowi podstawowy element ekosystemu JavaScript i jest powszechnie wykorzystywane w projektach frontendowych oraz backendowych.

3.5 Cargo

Cargo jest oficjalnym narzędziem do zarządzania projektami w języku Rust [4]. Odpowiada ono za proces kompilacji kodu źródłowego, pobieranie i zarządzanie zależnościami oraz uruchamianie testów jednostkowych. Cargo integruje się bezpośrednio z kompilatorem Rust, zapewniając spójność i automatyzację procesu budowania aplikacji.

Konfiguracja projektu realizowana jest za pomocą pliku `Cargo.toml`, który zawiera informacje o projekcie, wersjach bibliotek oraz ustawieniach kompilacji. Dzięki Cargo proces tworzenia aplikacji w języku Rust jest uproszczony i ustandaryzowany, co sprzyja utrzymaniu wysokiej jakości kodu oraz jego skalowalności.

3.6 Testy wydajnościowe

W ramach przygotowanego środowiska testowego przeprowadzono szereg eksperymentów mających na celu obiektywną ocenę wydajności różnych mechanizmów komunikacji oraz formatów serializacji danych. Poniżej przedstawiono szczegółowy opis metodologii testowania poszczególnych grup aplikacji.

3.6.1 Aplikacje zawarte w katalogu communication-mechanisms

Dla każdego zaimplementowanego protokołu komunikacji przeprowadzono cztery kategorie testów wydajnościowych, pozwalających na kompleksową ocenę charakterystyk przesyłania danych:

1. **Transmisja małych pakietów danych** – test polegający na cyklicznym odpytywaniu serwera o dane o rozmiarze 1 KB. Pomiar ten pozwala ocenić narzut protokołu oraz czas odpowiedzi dla typowych żądań zawierających niewielką ilość informacji.

2. **Transmisja dużych pakietów danych** – test analogiczny do poprzedniego, z tą różnicą, że rozmiar przesyłanych danych wynosi 1 MB. Eksperyment ten umożliwia ocenę wydajności protokołów w scenariuszu transferu większych zasobów.

3. **Head-of-Line Blocking** – test weryfikujący występowanie zjawiska blokowania głowy kolejki, w którym opóźnienie w przetwarzaniu jednego żądania wpływa na obsługę kolejnych żądań w ramach tego samego połączenia. Pomiar ten jest szczególnie istotny dla protokołów wykorzystujących multipleksowanie strumieni.

4. **Strumieniowanie danych** – test przeprowadzany wyłącznie dla protokołów wspierających tryb strumieniowy (streaming). Ocenie podlega efektywność przesyłania danych w sposób ciągły, bez konieczności oczekiwania na kompletną odpowiedź serwera.

3.6.2 Aplikacje zawarte w katalogu data-serialization-formats

Dla każdej z zaimplementowanych bibliotek serializacji danych przeprowadzono pomiary wydajności na podstawie ustandaryzowanej struktury danych. W celu zapewnienia porównywalności wyników, wszystkie testy wykorzystują identyczny zestaw danych testowych składający się z kolekcji 1000 obiektów użytkowników wraz z metadanymi.

Struktura danych testowych jest generowana w następujący sposób:

```
    let count = 1000;
    let users: Vec<User> = (0..count)
    .map(|i| User {
        id: i as i64,
        name: format!("User {}", i),
```

```
        email: format!("user{}@example.com", i),
        age: 20 + (i % 50) as i32,
        is_active: i % 2 == 0,
        tags: vec![
        "tag1".to_string(),
        "tag2".to_string(),
        "tag3".to_string(),
        ],
    })
    .collect();

    UserCollection {
        users,
        metadata: Metadata {
            version: "1.0.0".to_string(),
            created_at: "2025-01-23T00:00:00Z".to_string(),
            total_count: count,
        },
    }
```

Listing 3.1: Struktura danych wykorzystana w testach serializacji.

Każdy obiekt użytkownika zawiera kompletny zestaw atrybutów: identyfikator liczbowy, nazwę, adres email, wiek, status aktywności oraz listę trzech tagów. Dodatkowo, kolekcja jest wzbogacona o metadane zawierające wersję struktury danych, znacznik czasu utworzenia oraz łączną liczbę elementów. Taka konstrukcja zapewnia reprezentatywny zestaw różnorodnych typów danych (liczby całkowite, ciągi znaków, wartości logiczne, kolekcje) występujących w rzeczywistych zastosowaniach.

Dla każdego formatu serializacji mierzono następujące parametry wydajnościowe:

1. **Czas serializacji** – czas potrzebny na przekształcenie struktury danych z reprezentacji obiektowej języka Rust do formatu serializowanego. Pomiar wyrażony w mikrosekundach (μs) i uśredniony na podstawie wielokrotnych iteracji.

2. **Czas deserializacji** – czas wymagany do odtworzenia struktury obiektowej z danych w formacie serializowanym. Pomiar wyrażony w mikrosekundach (μs) i uśredniony na podstawie wielokrotnych iteracji.

3. **Rozmiar zserializowanych danych** – całkowita wielkość danych po procesie serializacji, wyrażona w bajtach (B). Parametr ten pozwala ocenić efektywność kompresji oraz narzut protokołu dla różnych formatów wymiany danych.

Wykorzystanie identycznego zestawu danych testowych dla wszystkich formatów serializacji

umożliwia obiektywne porównanie ich wydajności oraz charakterystyk w zakresie rozmiaru wynikowych struktur danych.

3.6.3 Testy przeglądarkowe

Testy przeglądarkowe służą do porównania wydajności implementacji algorytmów w czystym języku JavaScript oraz w technologii WebAssembly. Eksperyment polega na równoległym uruchomieniu identycznych funkcjonalności w obu środowiskach wykonawczych i pomiarze czasu ich wykonania.

Testowane są następujące scenariusze:

- **Implementacja JavaScript** – natywny kod uruchamiany bezpośrednio przez silnik JavaScript przeglądarki,
- **Implementacja WebAssembly** – kod napisany w języku Rust, skompilowany do formatu WASM za pomocą narzędzia `wasm-pack` i wykonywany w środowisku przeglądarki.

Porównanie obu implementacji pozwala na obiektywną ocenę potencjalnych korzyści wydajnościowych wynikających z zastosowania technologii WebAssembly w aplikacjach webowych wymagających intensywnych obliczeń.

Zakończenie

Bibliografia

[1] Git Project. Git. https://git-scm.com/. [dostęp: 21 stycznia 2026].

[2] Microsoft. Visual studio code. https://code.visualstudio.com/. [dostęp: 21 stycznia 2026].

[3] npm, Inc. npm documentation. https://www.npmjs.com/. [dostęp: 21 stycznia 2026].

[4] Rust Project. Cargo – the rust package manager. https://doc.rust-lang.org/cargo/. [dostęp: 21 stycznia 2026].

[5] Webassembly.org. Webassembly. https://webassembly.org/. [dostęp: 21 stycznia 2026].

[6] Wikipedia. Deserializacja. https://en.wikipedia.org/wiki/Marshalling_(computer_science)#Unmarshalling. [dostęp: 21 stycznia 2026].

[7] Wikipedia. google remote procedure call. https://en.wikipedia.org/wiki/GRPC. [dostęp: 21 stycznia 2026].

[8] Wikipedia. Hypertext transfer protocol. https://pl.wikipedia.org/wiki/HTTP. [dostęp: 21 stycznia 2026].

[9] Wikipedia. Interfejs programowania aplikacji. https://pl.wikipedia.org/wiki/Interfejs_programowania_aplikacji. [dostęp: 21 stycznia 2026].

[10] Wikipedia. Model klient serwer. https://en.wikipedia.org/wiki/Client%E2%80%93server_model. [dostęp: 21 stycznia 2026].

[11] Wikipedia. Serializacja. https://en.wikipedia.org/wiki/Serialization. [dostęp: 21 stycznia 2026].

[12] Wikipedia. Websocket. https://en.wikipedia.org/wiki/WebSocket. [dostęp: 21 stycznia 2026].

[13] Zed Industries. Zed editor. https://zed.dev/. [dostęp: 21 stycznia 2026].

Spis rysunków

Spis listingów