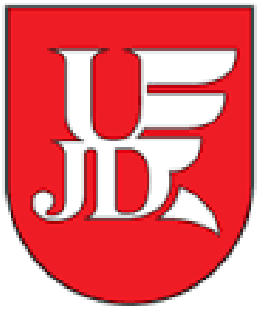
UNIWERSYTET JANA DŁUGOSZA W CZĘSTOCHOWIE



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

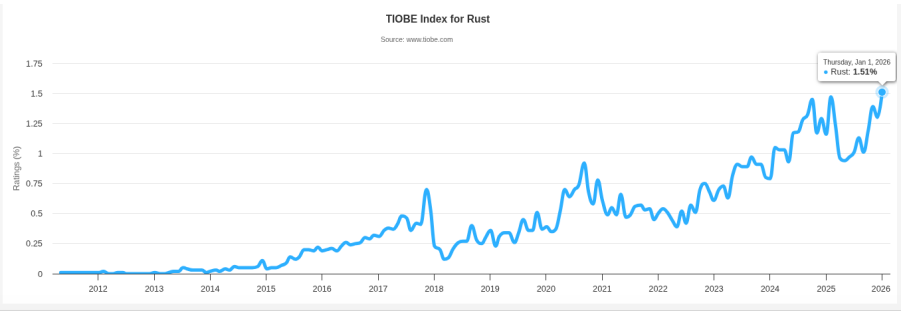
Rust jest nowoczesnym językiem programowania systemowego, który kładzie duży nacisk na bezpieczeństwo pamięci, wydajność oraz wielowątkowość. Dzięki mechanizmowi własności (ownership) oraz statycznej analizie błędów w czasie kompilacji, Rust minimalizuje ryzyko występowania błędów takich jak wycieki 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:

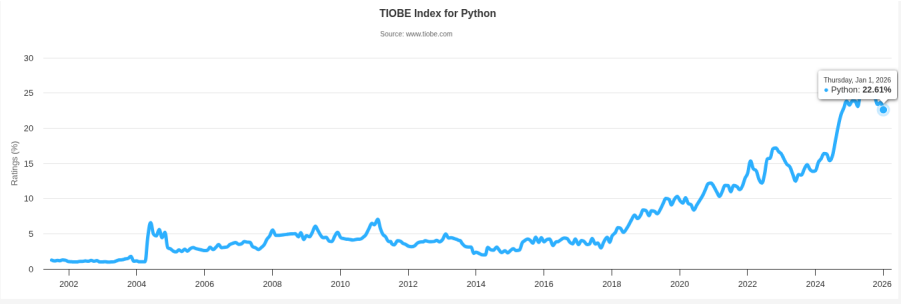
- generowania dużej liczby równoległych zapytań,
- pomiaru czasów odpowiedzi serwera,
- analizy zachowania systemu pod obciążeniem.

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.

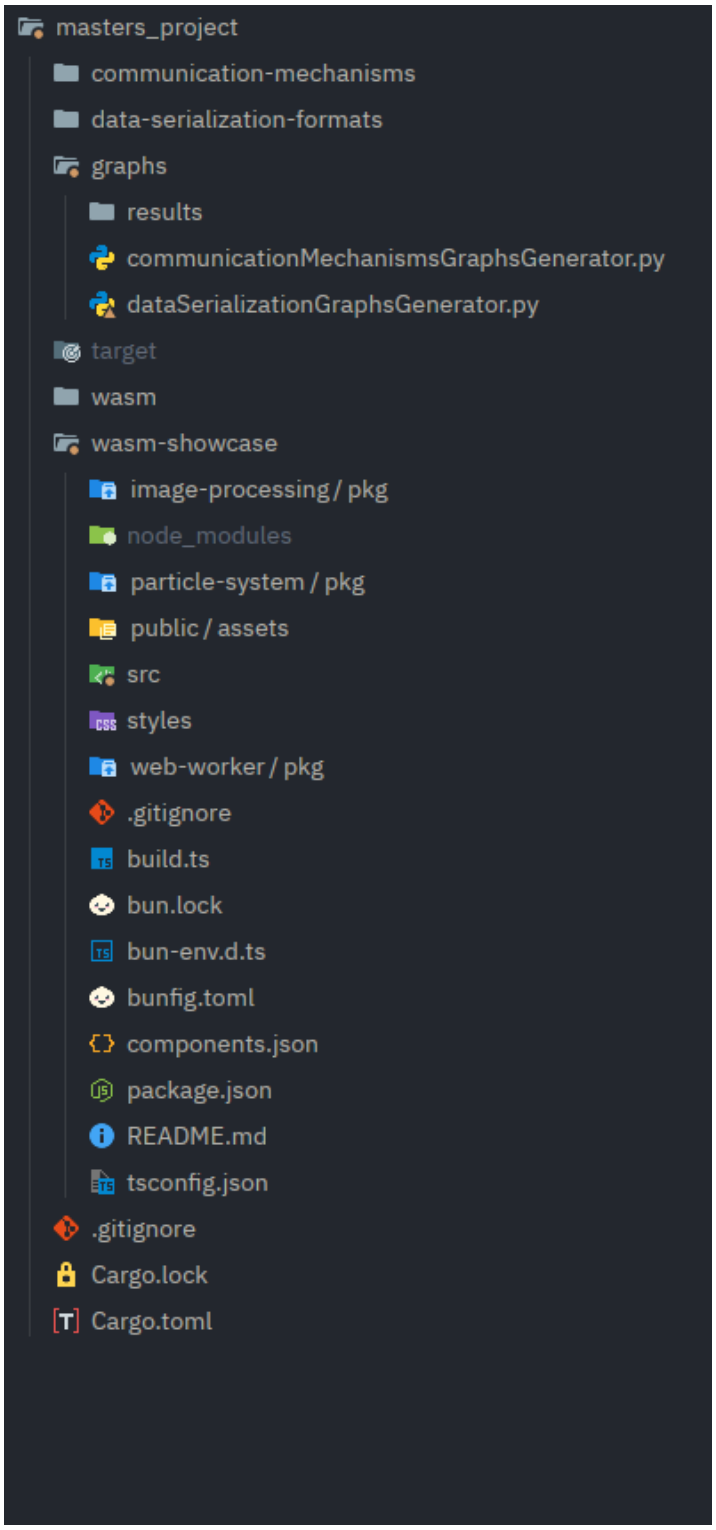
Rozdział 4

Budowa środowiska testowego

W niniejszym rozdziale przedstawiono kluczowe elementy środowiska testowego. W pierwszej kolejności zaprezentowano szczegółową strukturę katalogów projektu wraz z opisem zawartości poszczególnych folderów i plików. Następnie omówiono działanie wybranych aplikacji testowych, których implementację oraz wyniki pomiarów przedstawiono w kolejnym rozdziale.

4.1 Struktura plików aplikacji

Rysunek 4.1 przedstawia architekturę przygotowanego środowiska testowego. Projekt składa się z pięciu głównych katalogów, z których każdy odpowiada za odrębny aspekt przeprowadzanych badań.



Rysunek 4.1: Struktura plików i katalogów w aplikacji.

4.1.1 Katalog communication-mechanisms

Katalog ten zawiera implementacje aplikacji testujących wydajność różnych protokołów komunikacji. Każda aplikacja mierzy czasy odpowiedzi oraz przepustowość dla konkretnego mechanizmu wymiany danych. Wspólna struktura podkatalogów dla wszystkich implementacji

obejmuje:

- **src** – katalog zawierający kod źródłowy aplikacji serwerowej i klienckiej dla danego protokołu,
- **cert.pem** – lokalny certyfikat SSL/TLS niezbędny do nawiązania bezpiecznego połączenia z wykorzystaniem protokołu HTTP/2,
- **key.pem** – prywatny klucz kryptograficzny odpowiadający certyfikatowi, wymagany do obsługi szyfrowanej komunikacji,
- **Cargo.toml** – manifest pakietu w formacie TOML, zawierający metadane projektu, listę zależności oraz parametry kompilacji niezbędne do zbudowania aplikacji w języku Rust.

W katalogu communication-mechanisms znajdują się następujące implementacje protokołów:

- **graphql** – implementacja serwera GraphQL umożliwiającego elastyczne zapytania o dane ze zdefiniowanego schematu,
- **grpc** – implementacja serwera gRPC opartego na Protocol Buffers, wykorzystującego HTTP/2 do efektywnej komunikacji,
- **REST** – standardowa implementacja architektury REST API z wykorzystaniem metod HTTP (GET, POST, PUT, DELETE),
- **soap** – implementacja protokołu SOAP (Simple Object Access Protocol) opartego na wymianie komunikatów XML,
- **websocket** – implementacja serwera WebSocket umożliwiającego dwukierunkową komunikację w czasie rzeczywistym,
- **websocket_secure** – implementacja protokołu WebSocket Secure (WSS) z szyfrowaniem SSL/TLS.

4.1.2 Katalog data-serialization-formats

Katalog zawiera aplikacje służące do porównania wydajności procesów serializacji i deserializacji danych w różnych formatach. Każda implementacja mierzy czas potrzebny na przekształcenie struktur danych oraz rozmiar wynikowych reprezentacji. W katalogu znajdują się następujące moduły testowe:

- **avro** – implementacja testów dla formatu Apache Avro, binarnego systemu serializacji z wbudowanym wsparciem dla schematów danych,

- **bson_benchmark** – testy wydajności formatu BSON (Binary JSON), wykorzystywanego między innymi w bazach danych MongoDB,
- **json** – implementacja testów dla formatu JSON (JavaScript Object Notation), popularnego tekstowego formatu wymiany danych,
- **message-pack** – testy formatu MessagePack, będącego binarną alternatywą dla JSON o mniejszym rozmiarze danych,
- **protobuf** – implementacja testów dla Protocol Buffers (Protobuf), binarnego formatu serializacji opracowanego przez Google,
- **xml** – testy wydajności formatu XML (eXtensible Markup Language), rozszerzalnego języka znaczników.

4.1.3 Katalog graphs

Katalog zawiera narzędzia do wizualizacji wyników przeprowadzonych eksperymentów oraz wygenerowane wykresy. Struktura obejmuje:

- **communicationMechanismsGraphsGenerator.py** – skrypt w języku Python generujący wykresy porównawcze dla testów mechanizmów komunikacji, wizualizujący czasy odpowiedzi oraz przepustowość poszczególnych protokołów,
- **dataSerializationGraphsGenerator.py** – skrypt w języku Python tworzący wykresy przedstawiające wyniki testów serializacji danych, w tym czasy operacji oraz rozmiary zserializowanych struktur,
- **results** – katalog przechowujący wygenerowane wykresy w formatach graficznych (PNG, SVG) gotowe do wykorzystania w dokumentacji.

4.1.4 Katalog target

Katalog target stanowi standardowy folder roboczy kompilatora Rust, zawierający pliki binarne, biblioteki oraz artefakty pośrednie powstałe w procesie budowania projektu. Zawartość tego katalogu jest automatycznie zarządzana przez narzędzie Cargo i nie wymaga ręcznej modyfikacji.

4.1.5 Katalog wasm

Katalog zawiera programy napisane w języku Rust, które są kompilowane do formatu WebAssembly (WASM). Powstałe moduły binarne są następnie wykorzystywane w środowisku przeglądarki internetowej, umożliwiając uruchomienie kodu o wysokiej wydajności po stronie klienta.

4.1.6 Katalog wasm-showcase

Katalog wasm-showcase zawiera kompletną aplikację webową demonstrującą możliwości technologii WebAssembly. Struktura projektu obejmuje następujące elementy:

- **image-processing** – katalog zawierający skompilowane do WebAssembly moduły służące do przetwarzania obrazów. Pliki WASM oraz pomocnicze skrypty JavaScript pozwalają na wykonywanie operacji graficznych bezpośrednio w przeglądarce,
- **particle-system** – katalog z modułami WASM implementującymi system cząsteczek, wykorzystywany do demonstracji wydajności obliczeń graficznych w środowisku webowym,
- **web-worker** – katalog zawierający skompilowane moduły WASM przeznaczone do uruchomienia w Web Workers, umożliwiające wielowątkowe przetwarzanie danych bez blokowania głównego wątku interfejsu użytkownika,
- **styles** – katalog ze stylami CSS definiującymi wygląd aplikacji webowej,
- **build.ts** – natywny skrypt TypeScript odpowiedzialny za proces budowania aplikacji, zawierający logikę kompilacji i optymalizacji zasobów,
- **bun.lock** – plik blokady rejestrujący dokładne wersje wszystkich zależności projektu oraz ich podzależności, zapewniający powtarzalność budowania aplikacji,
- **bunfig.toml** – plik konfiguracyjny środowiska uruchomieniowego Bun, zawierający ustawienia dotyczące kompilacji i wykonania projektu,
- **components.json** – plik konfiguracyjny definiujący strukturę importów komponentów oraz integrację z biblioteką Tailwind CSS,
- **package.json** – manifest projektu Node.js zawierający listę zależności npm wraz z określonymi wersjami, skrypty budowania oraz metadane aplikacji,
- **README.md** – plik dokumentacji zawierający szczegółowy opis procesu instalacji, konfiguracji oraz uruchomienia aplikacji,
- **tsconfig.json** – plik konfiguracyjny kompilatora TypeScript, definiujący opcje transpilacji, ścieżki modułów oraz poziom zgodności ze standardem ECMAScript,
- **bun-env.d.ts** – plik deklaracji typów TypeScript dla środowiska Bun, zapewniający wsparcie IntelliSense i kontrolę typów,
- **src** – katalog zawierający pliki źródłowe aplikacji, w tym komponenty React, logikę biznesową oraz pomocnicze moduły,

- **public** – katalog z zasobami statycznymi (obrazy, ikony, manifesty) udostępnianymi bezpośrednio przez serwer webowy,
- **node_modules** – katalog zawierający zainstalowane biblioteki i moduły npm, automatycznie zarządzany przez menedżer pakietów.

4.1.7 Plik .gitignore

Plik konfiguracyjny systemu kontroli wersji Git, zawierający listę plików i katalogów wykluczonych z repozytorium. Typowo obejmuje katalogi tymczasowe (target, node_modules), pliki konfiguracyjne środowiska oraz artefakty kompilacji, które nie powinny być śledzone w historii zmian projektu.

4.2 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.

4.2.1 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.

4.2.2 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 4.1: 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 milisekundach (ms) i uśredniony na podstawie wielokrotnych iteracji.

2. **Czas deserializacji** – czas wymagany do odtworzenia struktury obiektowej z danych w formacie serializowanym. Pomiar wyrażony w milisekundach (ms) 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.

Rozdział 5

Przeprowadzanie testów

Niniejszy rozdział prezentuje metodologię przeprowadzonych badań eksperymentalnych oraz ich wyniki ilościowe.

5.1 Metodologia testowania protokołów komunikacyjnych

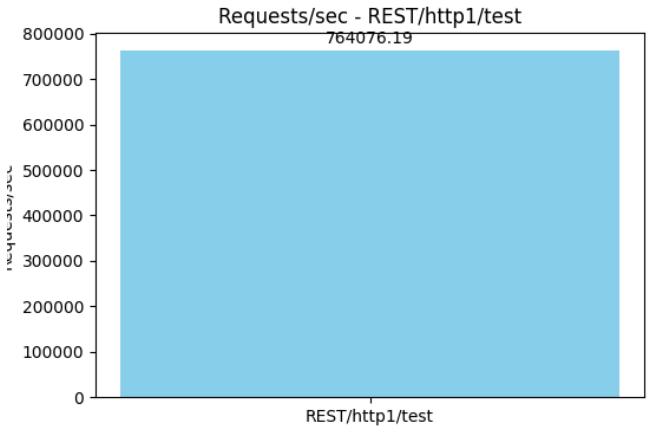
Wszystkie eksperymenty zostały przeprowadzone z wykorzystaniem ujednoliconych parametrów testowych, dostosowanych do specyfiki każdego badanego protokołu:

- Liczba współbieżnych operacji: 100
- Liczba zapytań na wątek: 1000
- Rozmiar standardowego ładunku danych: 1024 bajty (1 KB)
- Rozmiar rozszerzonego ładunku danych: 1048576 bajty (1 MB)

5.2 Architektura REST API

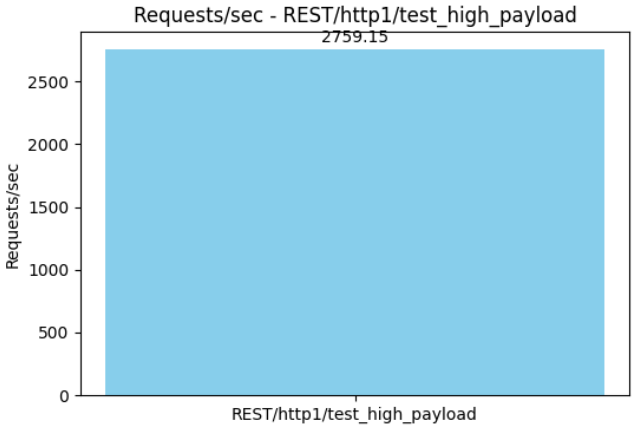
Implementacja architektury REST została zrealizowana przy użyciu biblioteki Actix-web – jednej z najpopularniejszych wysokopoziomowych platform wspierających natywnie strumieniowanie danych oraz protokół HTTP/2 w połączeniu z certyfikatami TLS. W ramach analizy zbadano również problem blokowania na początku kolejki (Head-of-Line blocking, HOL) wraz z proponowanym rozwiązaniem.

5.2.1 Wyniki dla protokołu HTTP/1.1



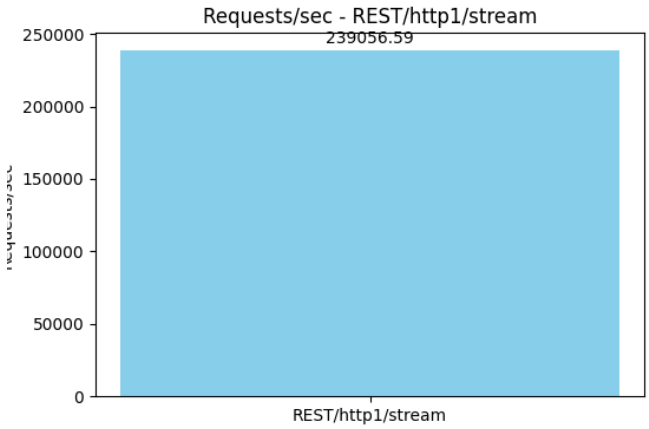
Rysunek 5.1: Przepustowość REST HTTP/1.1 dla ładunku 1 KB.

Przepustowość dla zapytań ze standardowym ładunkiem (1 KB) wyniosła 764076 żądań na sekundę.



Rysunek 5.2: Przepustowość REST HTTP/1.1 dla ładunku 1 MB.

Dla rozszerzonego ładunku (1 MB) odnotowano przepustowość na poziomie 2759 żądań na sekundę, co stanowi spadek o 99,6% w porównaniu ze standardowym ładunkiem.



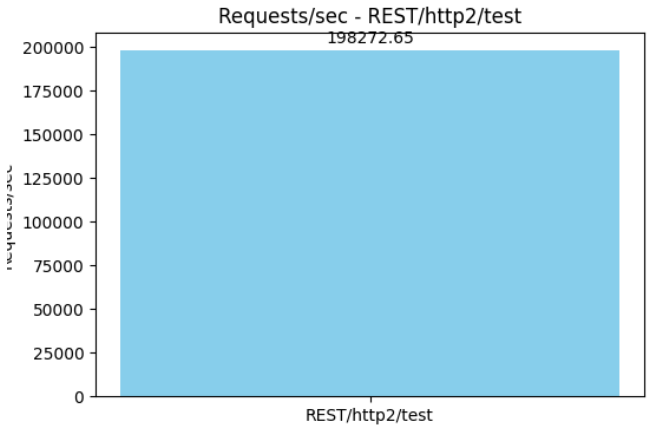
Rysunek 5.3: Przepustowość REST HTTP/1.1 dla transmisji strumieniowej (100 iteracji).

Transmisja strumieniowa składająca się ze 100 fragmentów (chunków) osiągnęła przepustowość 239056 żądań na sekundę.

Rysunek 5.4: Wpływ blokowania HOL na przepustowość REST HTTP/1.1.

W warunkach występowania problemu Head-of-Line blocking zmierzono przepustowość wynoszącą 1954 żądania na sekundę, demonstrując znaczącą degradację wydajności.

5.2.2 Wyniki dla protokołu HTTP/2

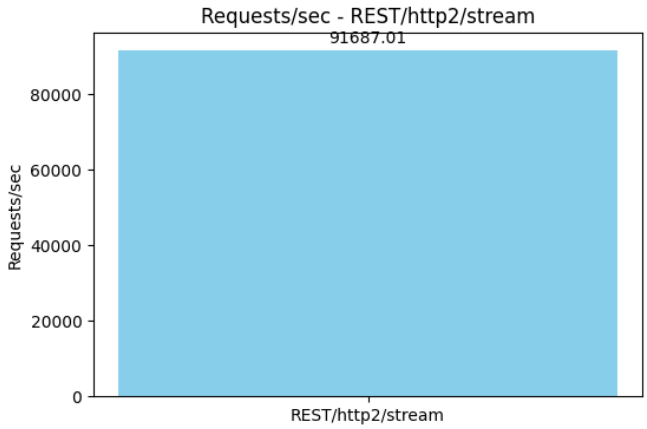


Rysunek 5.5: Przepustowość REST HTTP/2 dla ładunku 1 KB.

Implementacja z wykorzystaniem HTTP/2 osiągnęła przepustowość 198272 żądań na sekundę dla standardowego ładunku – wartość o 74% niższą niż w przypadku HTTP/1.1.

Rysunek 5.6: Przepustowość REST HTTP/2 dla ładunku 1 MB.

Dla dużych pakietów danych zaobserwowano przepustowość 1690 żądań na sekundę, co stanowi spadek o 39% względem HTTP/1.1.



Rysunek 5.7: Przepustowość transmisji strumieniowej REST HTTP/2.

Transmisja strumieniowa w protokole HTTP/2 uzyskała przepustowość 91687 żądań na sekundę, wykazując degradację o 62% w stosunku do HTTP/1.1.

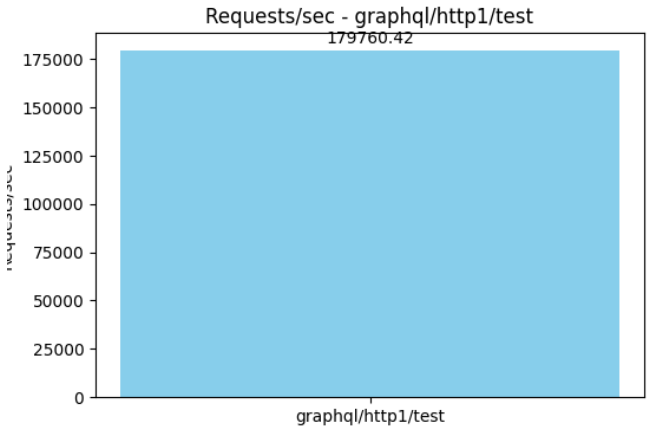
Rysunek 5.8: Mitigacja blokowania HOL w REST HTTP/2.

W scenariuszu testowym HOL blocking protokół HTTP/2 osiągnął przepustowość 135372 żądań na sekundę, demonstrując 69-krotną poprawę względem HTTP/1.1.

5.3 Architektura GraphQL

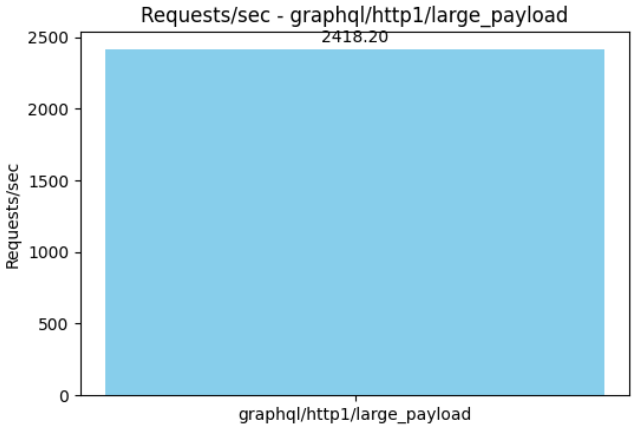
Implementację GraphQL zrealizowano przy użyciu biblioteki Juniper – zaawansowanego frameworka wspierającego natywnie strumieniowanie oraz HTTP/2 z wykorzystaniem TLS.

5.3.1 Wyniki dla protokołu HTTP/1.1



Rysunek 5.9: Przepustowość GraphQL HTTP/1.1 dla ładunku 1 KB.

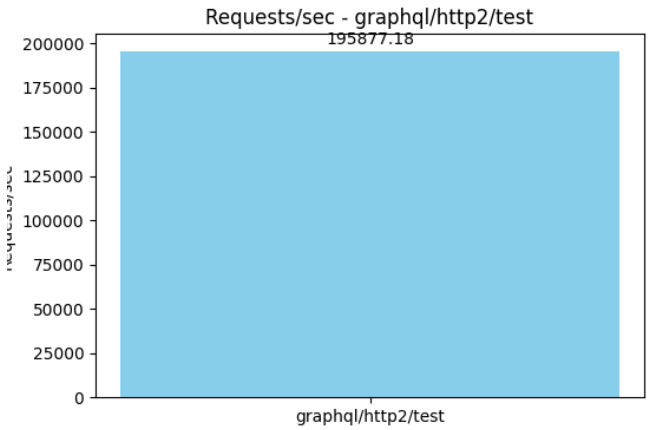
GraphQL w połączeniu z HTTP/1.1 osiągnął przepustowość 179760 żądań na sekundę dla standardowego ładunku, co stanowi 76% wydajności REST API.



Rysunek 5.10: Przepustowość GraphQL HTTP/1.1 dla ładunku 1 MB.

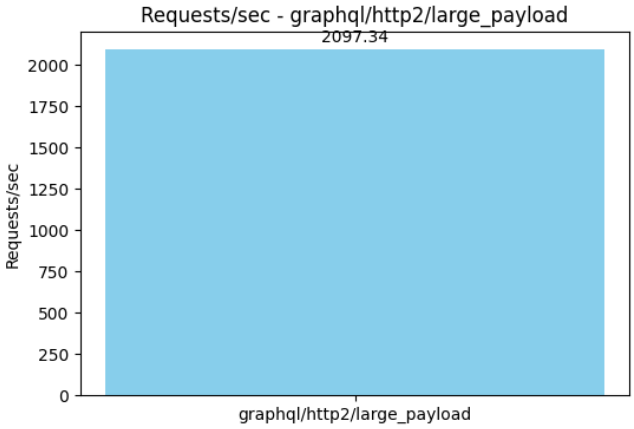
Dla rozszerzonego ładunku zmierzono przepustowość 2418 żądań na sekundę, wykazując nieznaczną przewagę (12%) nad REST API.

5.3.2 Wyniki dla protokołu HTTP/2



Rysunek 5.11: Przepustowość GraphQL HTTP/2 dla ładunku 1 KB.

Konfiguracja z HTTP/2 osiągnęła przepustowość 195877 żądań na sekundę, demonstrując 9% poprawę względem HTTP/1.1 oraz porównywalne wyniki z REST HTTP/2.



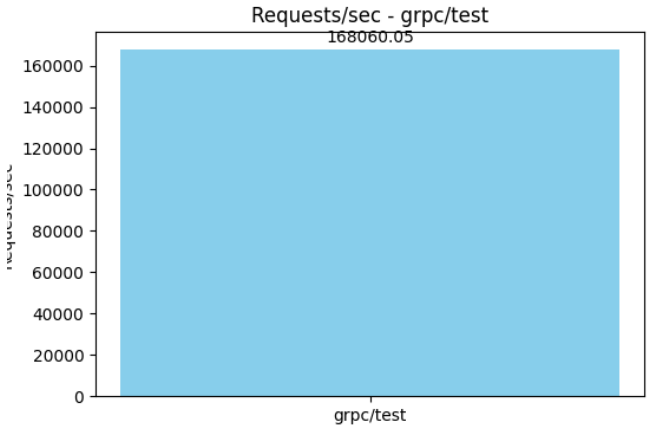
Rysunek 5.12: Przepustowość GraphQL HTTP/2 dla ładunku 1 MB.

Dla dużych pakietów danych zaobserwowano przepustowość 2097 żądań na sekundę, co stanowi 24% wzrost względem REST HTTP/2.

5.4 Architektura gRPC

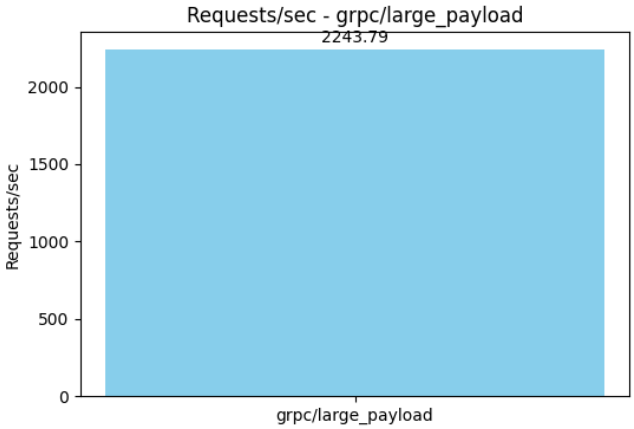
Implementacja gRPC została zrealizowana przy użyciu biblioteki Tonic – wysoko wydajnego frameworka natywnie wspierającego HTTP/2, strumieniowanie danych oraz szyfrowanie TLS. Ze względu na optymalizację wydajności zrezygnowano z implementacji mechanizmu refleksji API.

5.4.1 Wyniki dla protokołu HTTP/2



Rysunek 5.13: Przepustowość gRPC dla ładunku 1 KB.

gRPC osiągnął przepustowość 168060 żądań na sekundę dla standardowego ładunku, co stanowi 15% spadek względem GraphQL HTTP/2.



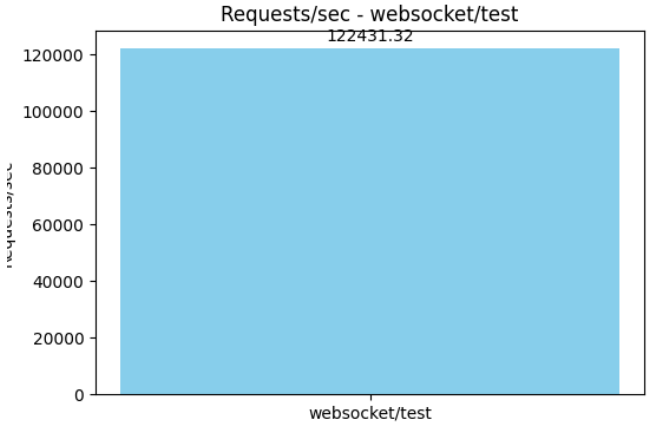
Rysunek 5.14: Przepustowość gRPC dla ładunku 1 MB.

Dla rozszerzonego ładunku zmierzono przepustowość 2243 żądań na sekundę, wykazując 7% przewagę nad GraphQL oraz 33% nad REST w konfiguracji HTTP/2.

5.5 Architektura WebSocket

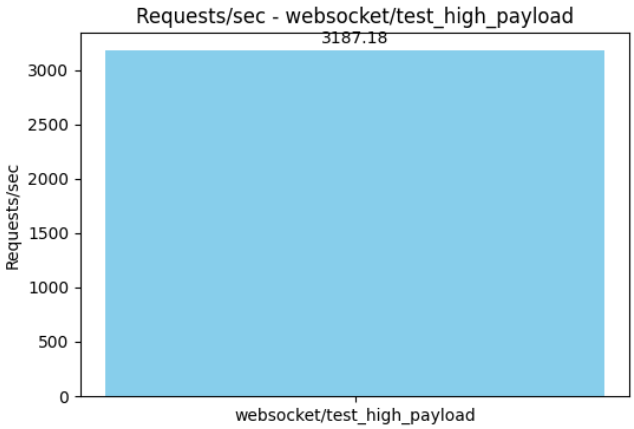
Do implementacji architektury WebSocket wykorzystano bibliotekę tungstenite-rs – popularny framework wysokiego poziomu dedykowany komunikacji dwukierunkowej w czasie rzeczywistym.

5.5.1 Wyniki dla protokołu WebSocket



Rysunek 5.15: Przepustowość WebSocket dla ładunku 1 KB.

Protokół WebSocket osiągnął przepustowość 122431 żądań na sekundę, wykazując najniższą wartość spośród testowanych protokołów dla standardowego ładunku.



Rysunek 5.16: Przepustowość WebSocket dla ładunku 1 MB.

Dla dużych pakietów danych zaobserwowano przepustowość 3187 żądań na sekundę – najwyższą wartość wśród wszystkich badanych protokołów, przewyższającą o 42% najbliższego konkurenta (gRPC).

5.5.2 Wyniki dla protokołu WebSocket Secure

Rysunek 5.17: Przepustowość WSS dla ładunku 1 KB.

Wersja zabezpieczona (WSS) osiągnęła przepustowość 111086 żądań na sekundę, co stanowi 9% degradację względem niezabezpieczonej implementacji.

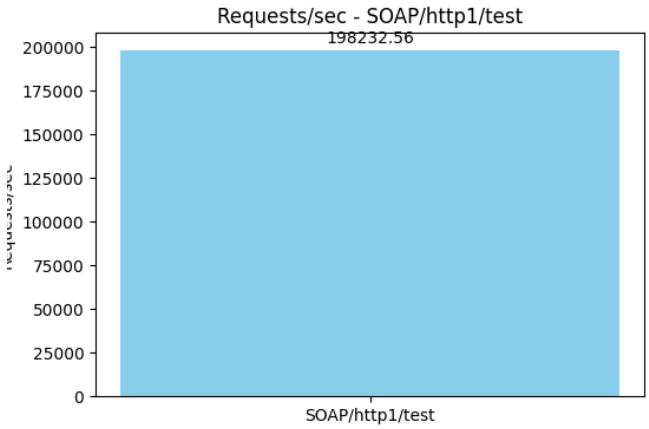
Rysunek 5.18: Przepustowość WSS dla ładunku 1 MB.

Dla rozszerzonego ładunku zmierzono przepustowość 2932 żądań na sekundę, wykazując 8% spadek względem wersji niezabezpieczonej.

5.6 Architektura SOAP

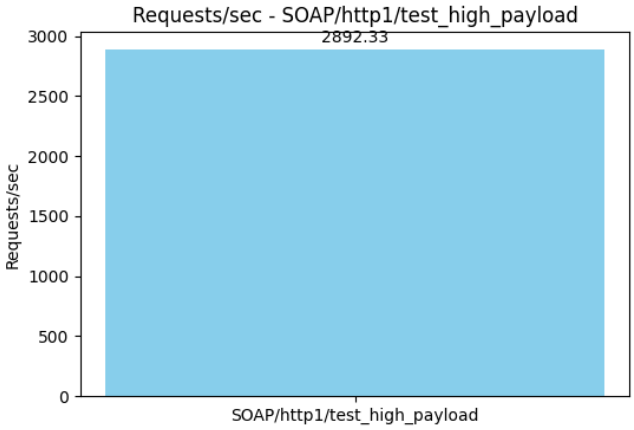
Implementacja protokołu SOAP została przetestowana w obu wersjach HTTP w celu porównania charakterystyk wydajnościowych.

5.6.1 Wyniki dla protokołu HTTP/1.1



Rysunek 5.19: Przepustowość SOAP HTTP/1.1 dla ładunku 1 KB.

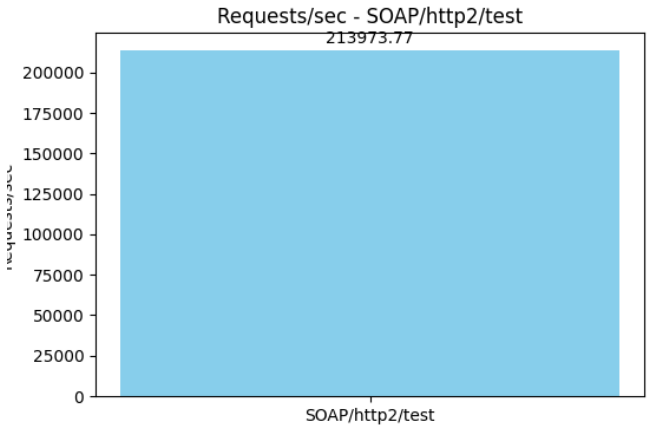
SOAP z HTTP/1.1 osiągnął przepustowość 198232 żądań na sekundę, demonstrując porównywalne wyniki z REST HTTP/2.



Rysunek 5.20: Przepustowość SOAP HTTP/1.1 dla ładunku 1 MB.

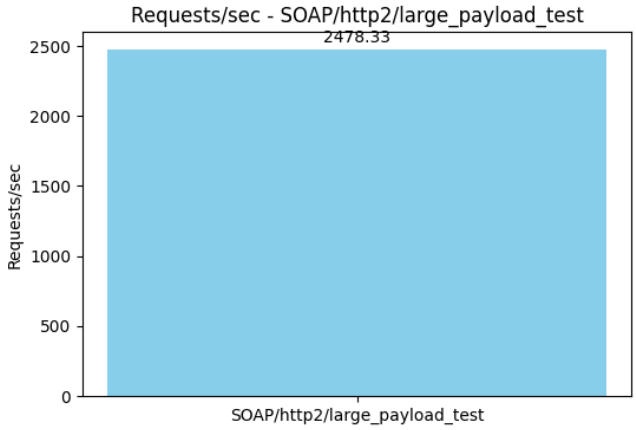
Dla rozszerzonego ładunku zmierzono przepustowość 2892 żądań na sekundę – najwyższą wartość wśród protokołów wykorzystujących HTTP/1.1.

5.6.2 Wyniki dla protokołu HTTP/2



Rysunek 5.21: Przepustowość SOAP HTTP/2 dla ładunku 1 KB.

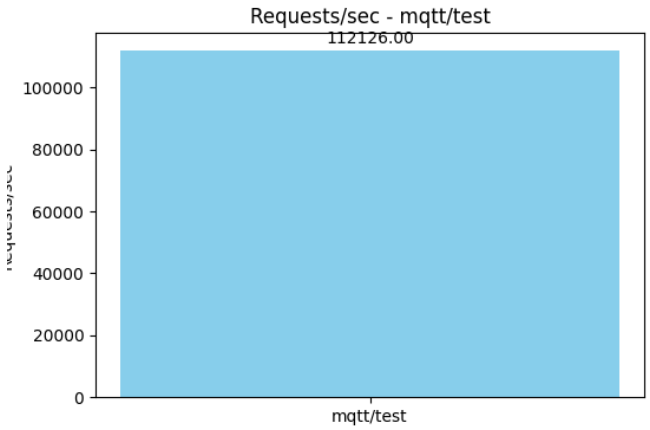
Implementacja z HTTP/2 osiągnęła przepustowość 213973 żądań na sekundę, wykazując 8% poprawę względem HTTP/1.1 oraz najwyższy wynik spośród wszystkich protokołów HTTP/2.



Rysunek 5.22: Przepustowość SOAP HTTP/2 dla ładunku 1 MB.

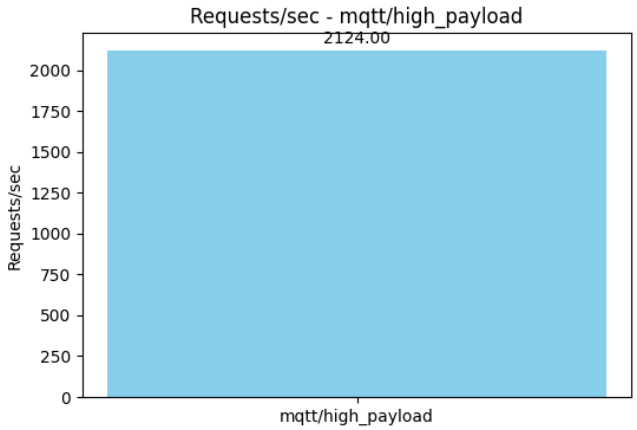
Dla dużych pakietów danych zaobserwowano przepustowość 2478 żądań na sekundę, co stanowi 14% spadek względem HTTP/1.1.

5.7 Protokół MQTT



Rysunek 5.23: Przepustowość MQTT dla ładunku 1 KB.

Protokół MQTT osiągnął przepustowość 112126 żądań na sekundę dla standardowego ładunku.



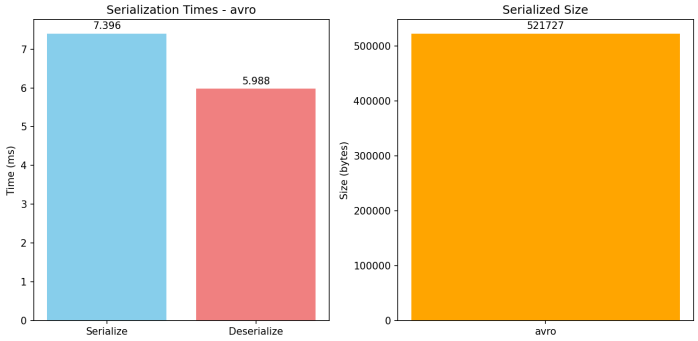
Rysunek 5.24: Przepustowość MQTT dla ładunku 1 MB.

Dla rozszerzonego ładunku zmierzono przepustowość 2167 żądań na sekundę, lokując MQTT w środkowym przedziale wydajnościowym.

5.8 Analiza porównawcza formatów serializacji danych

Przeprowadzono testy wydajnościowe dla sześciu popularnych formatów serializacji, mierząc czas serializacji, deserializacji oraz rozmiar wynikowych danych. Testowy zbiór składał się z 1000 obiektów użytkownika wraz z metadanymi.

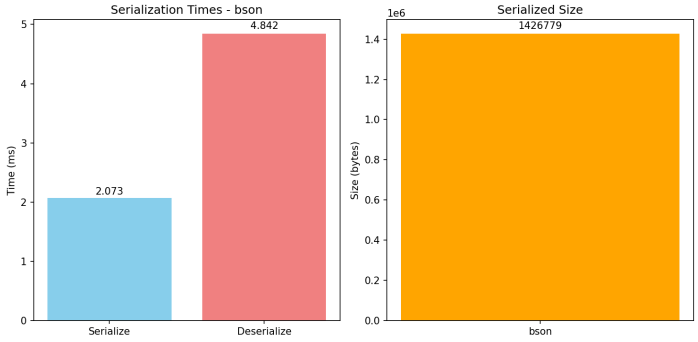
5.8.1 Apache Avro



Rysunek 5.25: Metryki wydajności Apache Avro.

Apache Avro uzyskał czas serializacji 7,396 ms, deserializacji 5,988 ms przy rozmiarze danych wynoszącym 521727 bajtów.

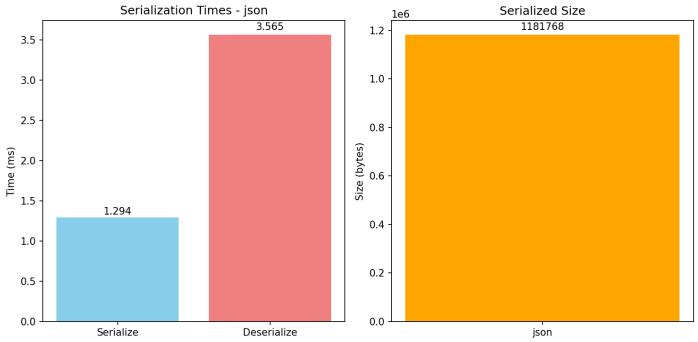
5.8.2 Binary JSON



Rysunek 5.26: Metryki wydajności Binary JSON.

Format BSON wykazał czas serializacji 2,073 ms, deserializacji 4,842 ms, generując dane o rozmiarze 1426779 bajtów – największym spośród testowanych formatów.

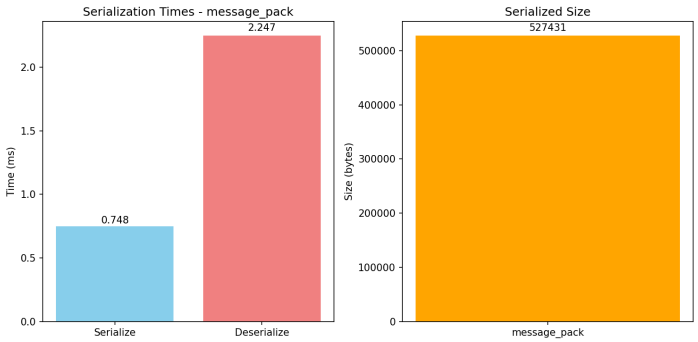
5.8.3 JSON



Rysunek 5.27: Metryki wydajności JSON.

Standardowy JSON osiągnął czas serializacji 1,294 ms oraz deserializacji 3,565 ms przy rozmiarze 1181768 bajtów, oferując najszybszą deserializację wśród formatów tekstowych.

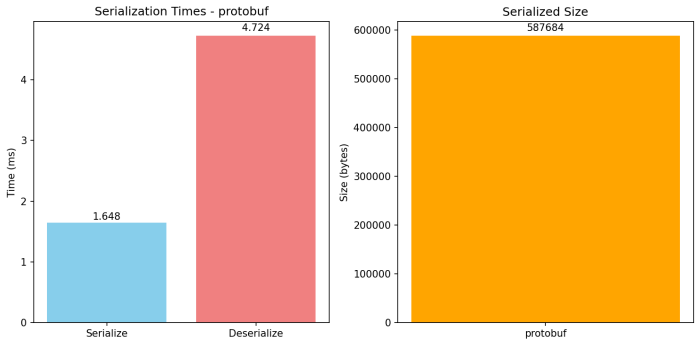
5.8.4 MessagePack



Rysunek 5.28: Metryki wydajności MessagePack.

MessagePack wykazał najlepszą wydajność czasową z serializacją 0,748 ms i deserializacją 2,247 ms, przy kompaktowym rozmiarze 527431 bajtów.

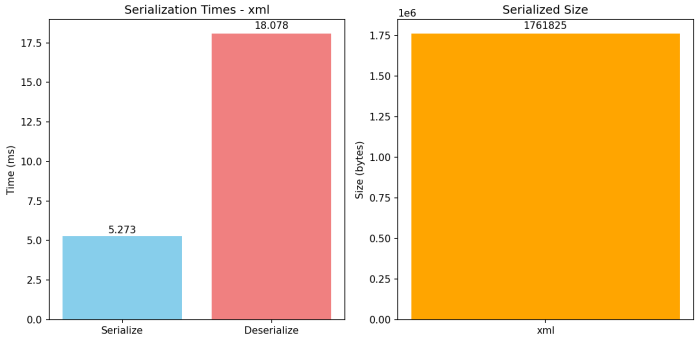
5.8.5 Protocol Buffers



Rysunek 5.29: Metryki wydajności Protocol Buffers.

Protocol Buffers osiągnął czas serializacji 1,648 ms, deserializacji 4,724 ms, generując dane o rozmiarze 587684 bajtów.

5.8.6 XML



Rysunek 5.30: Metryki wydajności XML.

Format XML wykazał najdłuższe czasy przetwarzania – serializację 5,273 ms i deserializację 18,078 ms – przy największym rozmiarze danych 1761825 bajtów, co stanowi 3,4-krotność najbardziej efektywnych formatów binarnych.

5.9 Analiza porównawcza protokołów komunikacyjnych

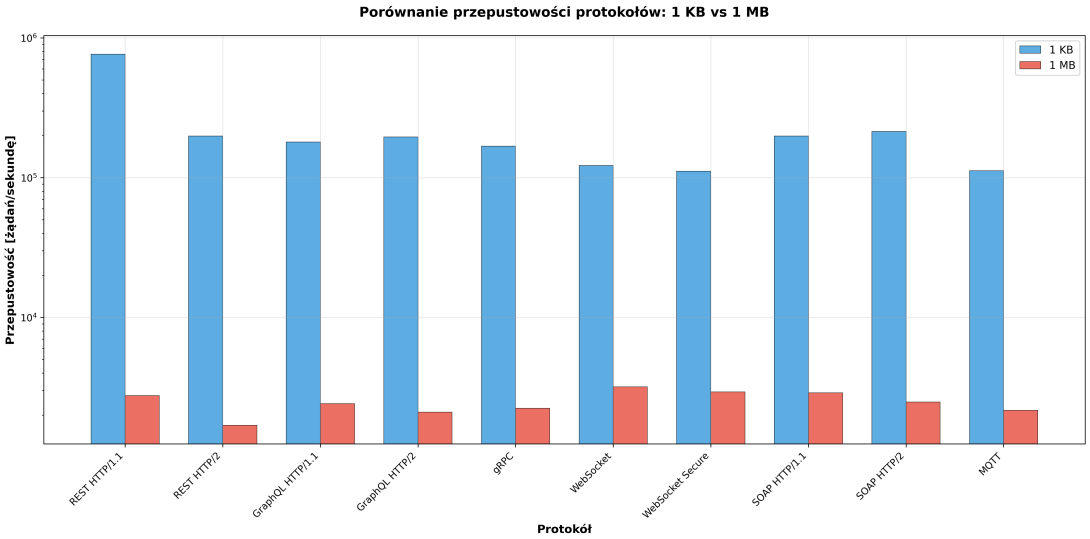
Przeprowadzone badania umożliwiły kompleksową ocenę wydajności protokołów komunikacyjnych w różnych scenariuszach użycia. Poniższa analiza syntetyzuje kluczowe wnioski z testów.

5.9.1 Wpływ rozmiaru ładunku na wydajność

Tabela 5.1: Porównanie przepustowości protokołów komunikacyjnych

Protokół	**1 KB [req/s]**	**1 MB [req/s]**	**Degradacja**
REST HTTP/1.1	764,076	2,759	276.9x
REST HTTP/2	198,272	1,690	117.3x
GraphQL HTTP/1.1	179,760	2,418	74.3x
GraphQL HTTP/2	195,877	2,097	93.4x
gRPC	168,060	2,243	74.9x
WebSocket	122,431	3,187	38.4x
WebSocket Secure	111,086	2,932	37.9x
SOAP HTTP/1.1	198,232	2,892	68.5x
SOAP HTTP/2	213,973	2,478	86.3x
MQTT	112,126	2,167	51.7x

Analiza współczynnika degradacji (tabela 5.1) ujawnia fundamentalne różnice w charakterystykach wydajnościowych protokołów. WebSocket Secure wykazał najniższą degradację (37.9x), co potwierdza jego optymalizację dla transmisji dużych wolumenów danych poprzez utrzymywanie stałego połączenia. W przeciwieństwie do tego, REST HTTP/1.1 odnotował najwyższą degradację (276.9x), co jest konsekwencją narzutu związanego z nawiązywaniem nowego połączenia TCP dla każdego żądania oraz brakiem multipleksowania.



Rysunek 5.31: Porównanie przepustowości protokołów dla ładunku 1 KB i 1 MB (skala logarytmiczna).

Wykres 5.31 ilustruje wyraźną dychotomię między wydajnością dla małych i dużych ładunków. Protokoły oparte na HTTP/1.1 dominują w scenariuszach małych pakietów, podczas gdy

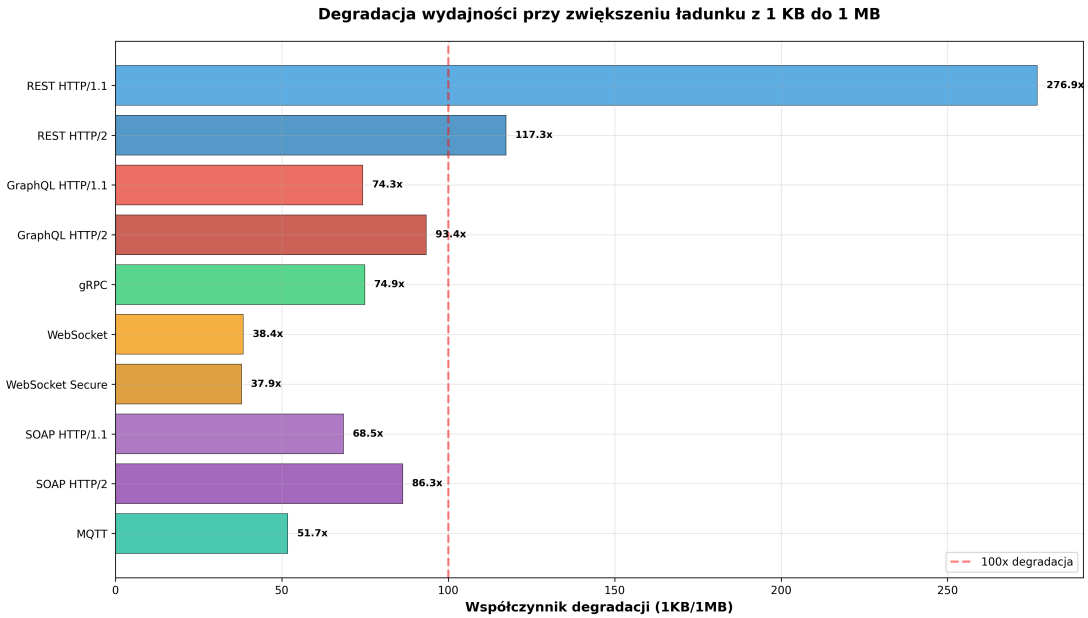
protokoły z trwałymi połączeniami (WebSocket, WSS) wykazują przewagę dla dużych transferów danych.

5.9.2 Wpływ TLS na wydajność

Bezpieczeństwo transmisji za pomocą TLS wprowadza dodatkowy narzut wydajnościowy, jednak jest niezbędne w aplikacjach produkcyjnych. Kluczowe aspekty TLS handshake:

- **Czas inicjalizacji**: TLS handshake typowo dodaje 50-100 ms opóźnienia dla pierwszego połączenia
- **Narzut obliczeniowy**: Szyfrowanie symetryczne (AES) ma minimalny wpływ na przepustowość (<5%)
- **Optymalizacja**: HTTP/2 i WebSocket amortyzują koszt TLS poprzez długotrwałe połączenia
- **Session resumption**: TLS 1.3 redukuje czas ponownego handshake'a do 1 RTT

Porównanie WebSocket (122,431 req/s) vs WebSocket Secure (111,086 req/s) pokazuje 9% degradację – znacząco niższą niż REST HTTP/1.1 vs HTTP/2 (74%), co wskazuje, że jednorazowy koszt TLS handshake jest amortyzowany w długotrwałych połączeniach.



Rysunek 5.32: Współczynnik degradacji wydajności przy zwiększeniu ładunku z 1 KB do 1 MB.

5.9.3 Rekomendacje wyboru protokołu

Na podstawie przeprowadzonych badań, dobór protokołu powinien uwzględniać specyficzne wymagania aplikacji:

1. **REST API (HTTP/1.1 lub HTTP/2)**

   - Zastosowanie: Aplikacje webowe, API publiczne, architektury mikroserwisowe
   - Zalety: Najlepsza skalowalność, cache'owalność, powszechne wsparcie
   - Wybór HTTP/2: Gdy priorytetem jest ograniczenie head of line blocking i bezpieczeństwo

2. **GraphQL**

   - Zastosowanie: aplikacje frontendowe, złożone modele danych
   - Zalety: Eliminacja over-fetchingu, redukcja liczby zapytań
   - Kompromis: 1-2% niższa wydajność niż REST przy znaczącej poprawie pisania oprogramowania

3. **gRPC**

   - Zastosowanie: Komunikacja pomiędzy serwisami, wysoko wydajne współczesne rozwiązania backendowe
   - Zalety: Najlepsza wydajność dla dużych ładunków w HTTP/2, silne kontrakty, eliminacja problemu pomiędzy językami programowania.
   - Ograniczenie: Słabsze wsparcie w przeglądarkach (wymaga gRPC-web)

4. **WebSocket/WSS**

   - Zastosowanie: Aplikacje czasu rzeczywistego (czaty, giełdy, gry)
   - Zalety: Najniższa latencja, najlepsza wydajność dla dużych ładunków
   - Kompromis: Zwiększona złożoność skalowania.

5. **SOAP**

   - Zastosowanie: Systemy enterprise, integracje legacy, transakcje finansowe
   - Zalety: Formalne standardy (WS-Security, WS-Transaction), compliance
   - Ograniczenie: Narzut XML, większa złożoność implementacji

6. **MQTT**

   - Zastosowanie: IoT, urządzenia wbudowane, sieci z ograniczonym pasmem
   - Zalety: Minimalny narzut protokołu, automatyczne kolejkowanie.
   - Optymalne: Dla rozproszonych sieci.

5.9.4 Kluczowe wnioski

Przeprowadzone badania prowadzą do następujących konkluzji:

1. **Brak uniwersalnego lidera**: Każdy protokół wykazuje przewagi w specyficznych scenariuszach użycia. REST HTTP/1.1 dominuje dla małych zapytań, WebSocket dla dużych transferów, gRPC dla komunikacji pomiędzy serwisami lub aplikacjami backendowymi.

2. **TLS jest akceptowalnym kosztem**: Degradacja wydajności 5-15% (w zależności od protokołu) jest w pełni uzasadniona przez krytyczne wymagania bezpieczeństwa. Nowoczesne implementacje TLS 1.3 dodatkowo minimalizują ten narzut.

3. **Długotrwałe połączenia amortyzują koszty**: Protokoły takie jak WebSocket, HTTP/2 wykazują niższą degradację przy wzroście ładunku, co czyni je preferowanymi dla ładunków o wysokiej przepustowości.

4. **Multipleksowanie rozwiązuje HOL blocking**: HTTP/2 osiągnęło 69-krotną poprawę w testach HOL, potwierdzając efektywność multipleksowania strumieni.

5. **Specjalizacja protokołów IoT**: MQTT, mimo umiarkowanej przepustowości, pozostaje optymalnym wyborem dla IoT dzięki mechanizmom QoS i niskiej konsumpcji zasobów.

5.10 Analiza porównawcza WebAssembly i JavaScript

5.10.1 WebWorker

5.10.2 Canvas

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