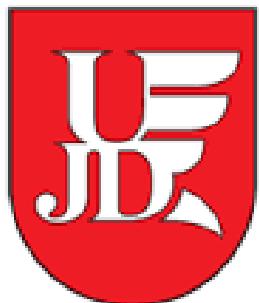


UNIWERSYTET JANA DŁUGOSZA W CZESTOCHOWIE



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

1 Wprowadzenie do problematyki doboru technologii	6
1.1 Znaczenie doboru technologii w aplikacjach webowych	6
1.2 Charakterystyka współczesnych systemów rozproszonych	6
1.3 Protokoły komunikacyjne jako fundament wymiany danych	7
1.4 Uzasadnienie podjęcia badań	7
1.5 Słownik pojęć i skrótów	7
2 Technologie wykorzystane w projekcie	10
2.1 Rust	10
2.2 JavaScript	11
2.3 TypeScript	11
2.4 Framework Actix-web	11
2.5 Framework Tonic	12
2.6 tungstenite-rs	12
2.7 h2load	12
2.8 Python	12
2.9 GraphQL	13
2.10 gRPC	13
2.11 WebAssembly	13
2.12 rumqttc	13
2.13 wasm-pack	14
2.14 Juniper	14
2.15 Apache Avro	14
2.16 BSON	14
2.17 Serde	15
2.18 Prost	15
2.19 quick-xml	15
3 Użyte narzędzia programistyczne	16
3.1 Git	16
3.2 Zed	16

3.3	Visual Studio Code	17
3.4	Bun	17
3.5	Cargo	17
4	Budowa środowiska testowego	19
4.1	Struktura plików aplikacji	19
4.1.1	Katalog communication-mechanisms	20
4.1.2	Katalog data-serialization-formats	21
4.1.3	Katalog graphs	22
4.1.4	Katalog target	22
4.1.5	Katalog wasm	22
4.1.6	Katalog wasm-showcase	23
4.1.7	Plik .gitignore	24
4.2	Testy wydajnościowe	24
4.2.1	Aplikacje zawarte w katalogu communication-mechanisms	24
4.2.2	Aplikacje zawarte w katalogu data-serialization-formats	25
4.2.3	Aplikacje zawarte w katalogu wasm oraz wasm-showcase	26
5	Przeprowadzanie testów	27
5.1	Metodologia testowania protokołów komunikacyjnych	27
5.2	Architektura REST API	27
5.2.1	Wyniki dla protokołu HTTP/1.1	27
5.2.2	Wyniki dla protokołu HTTP/2	30
5.3	Architektura GraphQL	30
5.3.1	Wyniki dla protokołu HTTP/1.1	31
5.3.2	Wyniki dla protokołu HTTP/2	31
5.4	Architektura gRPC	31
5.4.1	Definicja protokołu	31
5.4.2	Wyniki dla protokołu HTTP/2	32
5.5	Architektura WebSocket	33
5.5.1	Wyniki dla protokołu WebSocket	36
5.5.2	Wyniki dla protokołu WebSocket Secure (WSS)	36
5.6	Architektura SOAP	37
5.6.1	Wyniki dla protokołu HTTP/1.1	38
5.6.2	Wyniki dla protokołu HTTP/2	38
5.7	Protokół MQTT	39
5.7.1	Implementacja serwera	39
5.7.2	Implementacja klienta	40
5.7.3	Wyniki pomiarów	43
5.8	Analiza porównawcza protokołów komunikacyjnych	43

5.8.1	Wpływ rozmiaru ładunku na wydajność	43
5.8.2	Wpływ TLS na wydajność	44
5.8.3	Rekomendacje wyboru protokołu	45
5.8.4	Kluczowe wnioski	46
5.9	Analiza porównawcza formatów serializacji danych	46
5.9.1	Apache Avro	47
5.9.2	Binary JSON (BSON)	47
5.9.3	JSON	48
5.9.4	MessagePack	48
5.9.5	Protocol Buffers	49
5.9.6	XML	49
5.9.7	Analiza porównawcza formatów serializacji	49
5.10	Analiza porównawcza WebAssembly i JavaScript	54
5.10.1	Web Workers	54
5.10.2	Operacje na Canvas	57
5.10.3	Wnioski	60
Zakończenie		61

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.

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, WebSocket, GraphQL. 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, definicji oraz mechanizmów umożliwiających komunikację pomiędzy różnymi komponentami oprogramowania [27].

HTTP (*Hypertext Transfer Protocol*) – Protokół komunikacyjny wykorzystywany do przesyłania danych w sieci WWW w modelu klient–serwer [26].

WebSocket – Protokół umożliwiający utrzymywanie stałego, dwukierunkowego połączenia pomiędzy klientem a serwerem, wykorzystywany do komunikacji w czasie rzeczywistym [35].

gRPC – Wysokowydajny framework komunikacyjny typu RPC, oparty na protokole HTTP/2 oraz binarnym formacie *Protocol Buffers* [24].

Protocol Buffers (*Protobuf*) – Binarny format serializacji danych opracowany przez firmę Google, wykorzystywany m.in. w komunikacji gRPC [6].

SOAP (*Simple Object Access Protocol*) – Protokół komunikacyjny oparty na wymianie komunikatów XML, stosowany w architekturach usług sieciowych [32].

MQTT (*Message Queuing Telemetry Transport*) – Lekki protokół komunikacyjny typu publish–subscribe, przeznaczony do systemów o ograniczonych zasobach oraz komunikacji w sieciach o niskiej przepustowości [9].

Serializacja – Proces przekształcania struktury danych do postaci umożliwiającej jej zapis lub transmisję pomiędzy systemami [31].

Deserializacja – Proces odtwarzania pierwotnej struktury danych na podstawie jej zserializowanej reprezentacji [22].

XML (*eXtensible Markup Language*) – Rozszerzalny język znaczników wykorzystywany do opisu i strukturyzacji danych tekstowych [23].

Strumieniowanie danych – Technika przetwarzania i przesyłania danych w sposób ciągły, bez konieczności oczekiwania na kompletny zbiór danych [33].

Head-of-Line Blocking (*HOL*) – Zjawisko polegające na blokowaniu przetwarzania kolejnych żądań przez opóźnienie jednego z nich w ramach tego samego połączenia [25].

WebAssembly (*Wasm*) – Binarny format wykonywalnego kodu umożliwiający uruchamianie aplikacji o wysokiej wydajności w środowisku przeglądarek internetowych [21].

TLS (*Transport Layer Security*) – Protokół kryptograficzny zapewniający poufność, integralność oraz uwierzytelnianie danych przesyłanych w sieci komputerowej [34].

Architektura klient–serwer – Model architektoniczny, w którym klient inicjuje żądania, a serwer przetwarza je i zwraca odpowiedzi [29].

Silnik JavaScript V8 – Wysokowydajny silnik JavaScript opracowany przez firmę Google, wykorzystywany m.in. w przeglądarce Google Chrome oraz środowisku Node.js [5].

DOM (*Document Object Model*) – Model obiektowy dokumentu HTML lub XML, który reprezentuje jego strukturę w postaci drzewa obiektów, umożliwiając programom (np. skryptom JavaScript) dynamiczny dostęp do zawartości, struktury oraz stylów strony internetowej [?].

Rozdział 2

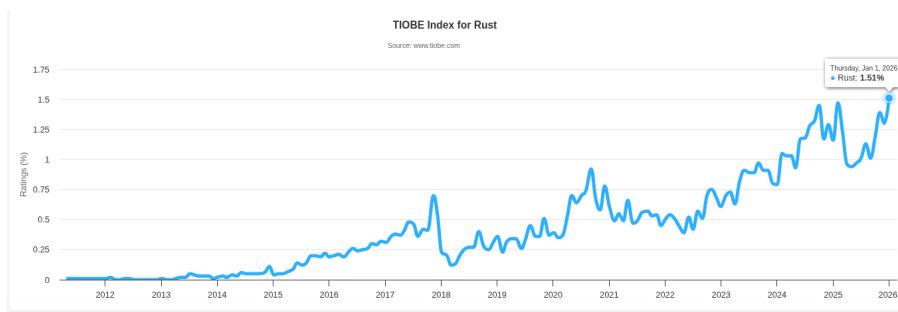
Technologie wykorzystane w projekcie

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 komplikacji, Rust minimalizuje ryzyko występowania błędów takich jak wycieki pamięci czy dereferencja pustych wskaźników. Rust zdobył popularność również dzięki nowoczesnemu systemowemu typów i możliwości pisania kodu niskopoziomowego bez rezygnacji z bezpieczeństwa [15].

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 wykorzystywany do tworzenia aplikacji webowych [28]. 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 TypeScript

TypeScript jest statycznie typowanym nadzbiorem języka JavaScript, rozwijanym przez firmę Microsoft. Rozszerza on JavaScript o system typów oraz mechanizmy weryfikacji poprawności kodu na etapie komplikacji, co znacząco ułatwia tworzenie, utrzymanie i rozwój większych aplikacji [7].

2.4 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 [18].

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.

2.5 Framework Tonic

Tonic jest frameworkiem do implementacji gRPC w języku Rust, opartym na bibliotece **tokio** oraz protokole HTTP/2 [4]. 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.

2.6 tungstenite-rs

tungstenite-rs jest biblioteką w Rust umożliwiającą obsługę protokołu WebSocket. Została wykorzystana do [16]:

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

2.7 h2load

h2load jest narzędziem służącym do testowania wydajności aplikacji korzystających z protokołu HTTP/2 [17]. W projekcie zostało wykorzystane do:

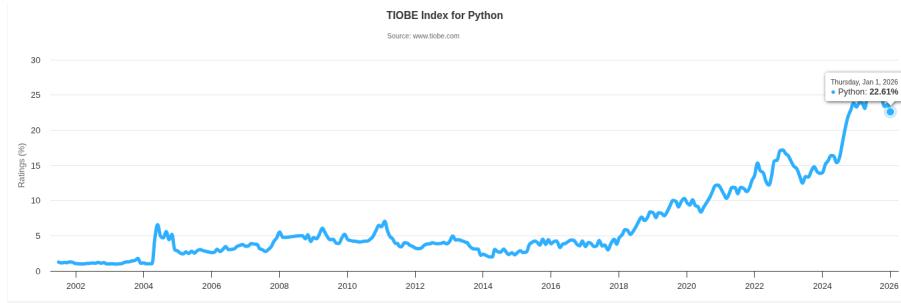
- generowania dużej liczby równoległych zapytań,
- pomiaru czasów odpowiedzi serwera,
- analizy zachowania systemu pod obciążeniem.

2.8 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 [30], co obrazuje Rysunek 2.3.



Rysunek 2.3: Pozycja języka Python w rankingu Tiobe Index.

2.9 GraphQL

GraphQL jest językiem zapytań do API, pozwalającym klientowi precyzyjnie określić, jakie dane są potrzebne [20]. W projekcie został wykorzystany do:

- realizacji zapytań o złożone struktury danych,
- testów wydajnościowych i porównawczych.

2.10 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,

2.11 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 wydajności aplikacji webowej.

2.12 rumqttc

rumqttc jest asynchroniczną biblioteką w języku Rust służącą do obsługi protokołu MQTT (Message Queuing Telemetry Transport). Umożliwia ona implementację klientów MQTT, oferując niskie opóźnienia oraz wysoką wydajność komunikacji [13].

W projekcie biblioteka **rumqttc** została wykorzystana do:

- testów komunikacji typu publish–subscribe,

- analizy wydajności lekkich protokołów komunikacyjnych,
- porównania MQTT z innymi mechanizmami wymiany danych.

2.13 wasm-pack

wasm-pack jest narzędziem wspierającym proces komplikacji projektów napisanych w języku Rust do formatu WebAssembly. Automatyzuje on generowanie pakietów WASM, integrację z ekosystemem JavaScript oraz publikację modułów.

W projekcie **wasm-pack** został wykorzystany do:

- komplikacji modułów Rust do WebAssembly,
- przygotowania interfejsów JavaScript do komunikacji z WASM,
- budowy aplikacji demonstracyjnych uruchamianych w przeglądarce.

2.14 Juniper

Juniper jest biblioteką w języku Rust służącą do implementacji serwerów GraphQL. Umożliwia definiowanie schematów, zapytań oraz mutacji w sposób silnie typowany, zgodny z paradygmatami języka Rust [3].

W projekcie biblioteka **Juniper** została użyta do:

- implementacji API GraphQL,
- realizacji zapytań o złożone struktury danych,
- testów wydajnościowych komunikacji GraphQL.

2.15 Apache Avro

Apache Avro jest binarnym formatem serializacji danych opartym na schematach, wykorzystywanym głównie w systemach przetwarzania danych i architekturach rozproszonych. Zapewnia kompaktową reprezentację danych oraz możliwość ewolucji schematów [19].

W projekcie Apache Avro został wykorzystany do:

- testów wydajności serializacji i deserializacji danych,
- porównania z innymi formatami binarnymi i tekstowymi,
- analizy rozmiaru wynikowych struktur danych.

2.16 BSON

BSON (Binary JSON) jest binarną reprezentacją formatu JSON, zaprojektowaną z myślą o efektywnej serializacji danych. Jest powszechnie stosowany m.in. w bazach danych MongoDB

[?].

W projekcie format BSON został wykorzystany do:

- testów wydajności serializacji danych,
- porównania binarnych i tekstowych formatów wymiany danych,
- analizy narzutu pamięciowego.

2.17 Serde

Serde jest biblioteką w języku Rust służącą do serializacji i deserializacji struktur danych [12]. Zapewnia zunifikowany interfejs obsługujący wiele formatów, takich jak JSON, BSON, XML czy MessagePack.

W projekcie **Serde** została wykorzystana jako:

- warstwa abstrakcji dla procesów serializacji danych,
- wspólna baza dla testów różnych formatów danych,
- narzędzie upraszczające implementację testów porównawczych.

2.18 Prost

Prost jest biblioteką w języku Rust służącą do obsługi formatu Protocol Buffers. Umożliwia generowanie kodu Rust na podstawie plików `.proto` oraz efektywną serializację danych binarnych [10].

W projekcie **Prost** została wykorzystana do:

- obsługi komunikacji gRPC,
- serializacji danych w formacie Protocol Buffers,
- testów wydajności komunikacji RPC.

2.19 quick-xml

quick-xml jest szybką biblioteką w języku Rust przeznaczoną do parsowania i generowania dokumentów XML. Charakteryzuje się niskim narzutem pamięciowym oraz wysoką wydajnością [11].

W projekcie biblioteka **quick-xml** została użyta do:

- testów serializacji i deserializacji danych w formacie XML,
- porównania XML z innymi formatami wymiany danych,
- analizy wydajności przetwarzania danych tekstowych.

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 [2]. 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 [36]. 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 [8]. 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 frameworkm.

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 Bun

Bun jest nowoczesnym środowiskiem uruchomieniowym dla języka JavaScript, TypeScript oraz WebAssembly, pełniącym jednocześnie rolę menedżera pakietów, narzędzia do budowania aplikacji oraz serwera uruchomieniowego [1]. Jego głównym celem jest zwiększenie wydajności i uproszczenie procesu tworzenia aplikacji webowych poprzez integrację funkcjonalności, które w tradycyjnym ekosystemie Node.js realizowane są przez wiele odrębnych narzędzi.

Jednym z kluczowych elementów środowiska Bun jest wbudowany menedżer pakietów, stanowiący alternatywę dla NPM. Umożliwia on instalowanie, aktualizowanie oraz usuwanie zależności projektowych, a także korzystanie z pakietów dostępnych w rejestrze NPM, przy jednoczesnym zachowaniu znacznie krótszych czasów instalacji. Konfiguracja projektu opiera się na pliku **package.json**, analogicznie do standardowego ekosystemu JavaScript, natomiast dodatkowym elementem jest plik **bun.lock**, zapewniający deterministyczność i powtarzalność procesu budowania aplikacji.

Bun obsługuje również natywne uruchamianie kodu TypeScript bez konieczności wcześniejszej transpilacji oraz oferuje wbudowany system budowania i bundlowania zasobów. Dzięki temu środowisku to może pełnić rolę kompleksowej platformy deweloperskiej, szczególnie przydatnej w aplikacjach frontendowych, backendowych oraz projektach wykorzystujących technologię WebAssembly.

3.5 Cargo

Cargo jest oficjalnym narzędziem do zarządzania projektami w języku Rust [14]. Odpowiada ono za proces komplikacji 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.

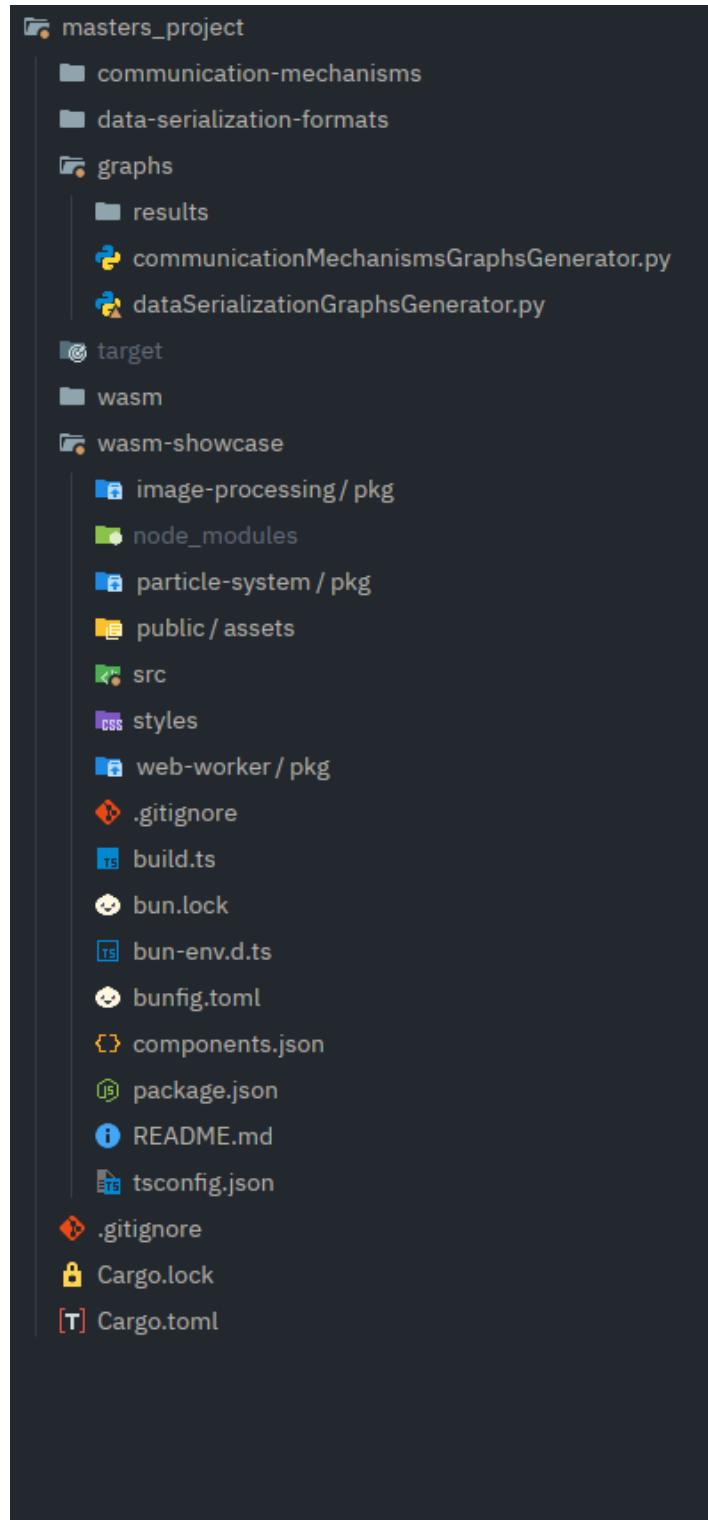
Rozdział 4

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 komplikacji 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 komplikowane 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 komplikacji 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 komplikacji 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 komplikacji, 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. Testom został pođdany tylko protokół HTTP1.1 oraz HTTP2.
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. Testom został pođdany tylko protokół HTTP1.1 oraz HTTP2.

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:

```
1  let count = 1000;
2  let users: Vec<User> = (0..count)
3  .map(|i| User {
4      id: i as i64,
5      name: format!("User {}", i),
6      email: format!("user{}@example.com", i),
7      age: 20 + (i % 50) as i32,
8      is_active: i % 2 == 0,
9      tags: vec![
10         "tag1".to_string(),
11         "tag2".to_string(),
12         "tag3".to_string(),
13     ],
14 })
15 .collect();
16
17 UserCollection {
18     users,
19     metadata: Metadata {
20         version: "1.0.0".to_string(),
21         created_at: "2025-01-23T00:00:00Z".to_string(),
22         total_count: count,
23     },
24 }
```

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.

4.2.3 Aplikacje zawarte w katalogu wasm oraz wasm-showcase

Katalogi *wasm* oraz *wasm-showcase* obejmują zbiór aplikacji eksperymentalnych napisanych w języku Rust, które zostały skompilowane do formatu WebAssembly (WASM) z wykorzystaniem narzędzia *wasm-pack*. Celem tych aplikacji jest analiza wpływu wykorzystania technologii WebAssembly na wydajność wykonywania obliczeń po stronie klienta w środowisku przeglądarek internetowej.

W ramach przeprowadzonych eksperymentów porównywany jest czas wykonania wybranych operacji matematycznych realizowanych bezpośrednio w języku JavaScript oraz analogicznych operacji wykonywanych przez skompilowane moduły WebAssembly. Takie zestawienie umożliwia ocenę potencjalnych korzyści wynikających z wykorzystania kodu natywnego kompilowanego do WASM w kontekście obliczeń o zwiększonej złożoności.

Testy zostały przeprowadzone z wykorzystaniem dwóch interfejsów programistycznych dostępnych w środowisku webowym:

- **Canvas API** – wykorzystywanego do wykonywania obliczeń związanych z przetwarzaniem grafiki oraz renderowaniem dynamicznych scen, co pozwala ocenić wydajność obliczeń realizowanych w głównym wątku aplikacji,
- **Web Workers API** – umożliwiającego uruchamianie modułów WebAssembly w osobnych wątkach roboczych, co pozwala na analizę wpływu przetwarzania równoległego na czas wykonania obliczeń oraz responsywność interfejsu użytkownika.

Zastosowanie zarówno Canvas API, jak i Web Workers pozwala na kompleksową ocenę wydajności modułów WebAssembly w różnych scenariuszach użycia, obejmujących zarówno obciążenie głównego wątku aplikacji, jak i przetwarzanie asynchroniczne. Wyniki uzyskane w ramach tych testów stanowią podstawę do dalszej analizy porównawczej przedstawionej w kolejnym rozdziale pracy.

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 ujednolicionych 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: 1 048 576 bajtów (1 MB)

5.2 Architektura REST API

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

5.2.1 Wyniki dla protokołu HTTP/1.1

Listing 5.1 przedstawia implementację punktów końcowych (ang. *endpoints*) obsługujących zapytania ze standardowym oraz rozszerzonym ładunkiem danych.

```

1  use actix_web::{get, HttpResponse};
2
3  #[get("/test")]
4  async fn test() -> HttpResponse {
5      HttpResponse::Ok().json(serde_json::json!({
6          "message": "x".repeat(1024)
7      }))
8  }
9
10 #[get("/test_high_payload")]
11 async fn test_high_payload() -> HttpResponse {
12     HttpResponse::Ok().json(serde_json::json!({
13         "message": "x".repeat(1024 * 1024)
14     }))
15 }
```

Listing 5.1: Implementacja zapytań dla ładunków 1KB oraz 1MB dla protokołu HTTP/1.1 oraz HTTP/2 dla architektury REST API

Wyniki pomiarów dla standardowego ładunku (1 KB): Przepustowość wyniosła 764 076 żądań na sekundę, co stanowi najwyższą wartość spośród wszystkich testowanych konfiguracji dla małych pakietów danych.

Wyniki pomiarów dla rozszerzonego ładunku (1 MB): Odnotowano przepustowość na poziomie 2 759 żądań na sekundę, co stanowi spadek o 99,6% w porównaniu ze standardowym ładunkiem. Ta znacząca degradacja wynika z narzutu związanego z nawiązywaniem nowego połączenia TCP dla każdego żądania.

Transmisja strumieniowa

Listing 5.2 pokazuje implementację strumieniowania danych, gdzie odpowiedź jest wysyłana w postaci sekwencji fragmentów (ang. *chunks*).

```

1  use actix_web::{get, HttpResponse};
2  use futures_util::stream::{self};
3  use bytes::Bytes;
4
5  #[get("/stream")]
6  async fn stream_handler() -> HttpResponse {
7      let response_stream = stream::unfold(0, |count| async move {
8          if count >= 100 {
9              None
10         } else {
```

```

11         let chunk = format!("chunk {}\n", count);
12         Some((Ok::<Bytes, actix_web::Error>(Bytes::from(chunk)), 
13             count + 1))
14     );
15
16     HttpResponse::Ok()
17     .content_type("text/plain")
18     .streaming(response_stream)
19 }
```

Listing 5.2: Implementacja strumienowania dla protokołu HTTP/1.1 oraz HTTP/2 dla architektury REST API

Transmisja strumieniowa składająca się ze 100 fragmentów osiągnęła przepustowość 239 056 żądań na sekundę, co stanowi wartość pośrednią między scenariuszami małych i dużych pakietów.

Problem Head-of-Line Blocking

Listing 5.3 przedstawia implementację testową symulującą problem HOL, w którym opóźnienie jednego fragmentu danych blokuje całą kolejkę oczekujących pakietów.

```

1 use actix_web::{get, HttpResponse};
2 use futures_util::stream::{self};
3 use bytes::Bytes;
4
5 #[get("/stream")]
6 async fn stream_handler() -> HttpResponse {
7     let response_stream = stream::unfold(0, |count| async move {
8         if count >= 100 {
9             None
10        } else {
11            let chunk = format!("chunk {}\n", count);
12            Some((Ok::<Bytes, actix_web::Error>(Bytes::from(chunk)), 
13                count + 1))
14        }
15    });
16
17     HttpResponse::Ok()
18     .content_type("text/plain")
19     .streaming(response_stream)
}
```

Listing 5.3: Implementacja symulacji problemu HOL dla protokołu HTTP/1.1 oraz HTTP/2 dla architektury REST API

W warunkach występowania problemu Head-of-Line blocking zmierzono przepustowość wynoszącą 1 954 żądań na sekundę, demonstrując znaczącą degradację wydajności względem normalnego strumieniowania.

5.2.2 Wyniki dla protokołu HTTP/2

Protokół HTTP/2 wykorzystuje ten sam kod implementacyjny co HTTP/1.1. Jedyną różnicą jest dodanie warstwy TLS, która zapewnia bezpieczną komunikację poprzez szyfrowanie danych, integralność oraz uwierzytelnianie serwerów [34].

Wyniki pomiarów – standardowy ładunek: Implementacja z wykorzystaniem HTTP/2 osiągnęła przepustowość 198 272 żądań na sekundę dla standardowego ładunku – wartość o 74% niższą niż w przypadku HTTP/1.1. Degradacja ta wynika z narzutu związanego z negocjacją TLS oraz multipleksowaniem strumieni.

Wyniki pomiarów – rozszerzony ładunek: Dla dużych pakietów danych zaobserwowano przepustowość 1 690 żądań na sekundę, co stanowi spadek o 39% względem HTTP/1.1. Mimo niższej wartości bezwzględnej, proporcjonalnie mniejsza degradacja wskazuje na lepszą optymalizację HTTP/2 dla transferu dużych wolumenów danych.

Wyniki pomiarów – transmisja strumieniowa: Transmisja strumieniowa w protokole HTTP/2 uzyskała przepustowość 91 687 żądań na sekundę, wykazując degradację o 62% w stosunku do HTTP/1.1.

Wyniki pomiarów – HOL blocking: W scenariuszu testowym HOL blocking protokół HTTP/2 osiągnął przepustowość 135 372 żądań na sekundę, demonstrując 69-krotną poprawę względem HTTP/1.1. Ten wynik potwierdza efektywność mechanizmu multipleksowania strumieni w mitigacji problemu blokowania kolejki.

5.3 Architektura GraphQL

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

Listing 5.4 przedstawia implementację schematu GraphQL wraz z funkcjami resolver, które są wywoływanie w celu uzyskania żądanego danych.

```
1 #[juniper::graphql_object]
2 impl Query {
3     fn test_payload() -> Payload {
4         Payload {
5             message: "x".repeat(1024),
```

```

6      }
7
8
9     fn large_payload() -> Payload {
10        const PAYLOAD_SIZE: usize = 1024 * 1024;
11        Payload {
12            message: "x".repeat(PAYLOAD_SIZE),
13        }
14    }
15

```

Listing 5.4: Implementacja zapytania GraphQL

5.3.1 Wyniki dla protokołu HTTP/1.1

GraphQL w połączeniu z HTTP/1.1 osiągnął przepustowość 179 760 żądań na sekundę dla standardowego ładunku, co stanowi 76% wydajności REST API w analogicznej konfiguracji.

Dla rozszerzonego ładunku zmierzono przepustowość 2 418 żądań na sekundę, wykazując nieznaczną przewagę (12%) nad REST API. Lepsza wydajność dla dużych pakietów może wynikać z bardziej efektywnej serializacji JSON w bibliotece Juniper.

5.3.2 Wyniki dla protokołu HTTP/2

Konfiguracja z HTTP/2 osiągnęła przepustowość 195 877 żądań na sekundę, demonstrując 9% poprawę względem HTTP/1.1 oraz porównywalne wyniki z REST HTTP/2.

Dla dużych pakietów danych zaobserwowano przepustowość 2 097 żądań na sekundę, co stanowi 24% wzrost względem REST HTTP/2, potwierdzając optymalizacje GraphQL dla złożonych zapytań z dużymi zbiorami danych.

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. Protokół gRPC automatycznie wykorzystuje protokół HTTP/2.

5.4.1 Definicja protokołu

Listing 5.5 przedstawia zawartość pliku definicji protokołu (ang. *Protocol Buffers*), na podstawie którego generowany jest automatycznie kod oraz typy danych dla języka Rust.

```

1 syntax = "proto3";
2
3 package benchmark;
4
5 service BenchmarkService {
6     rpc TestPayload (TestPayloadRequest) returns (TestPayloadResponse)
7         ;
8     rpc TestHighPayload (TestHighPayloadRequest) returns (
9         TestHighPayloadResponse);
10    rpc TestStreamPayload (TestStreamPayloadRequest) returns (stream
11        TestStreamPayloadResponse);
12 }
13
14 message TestPayloadRequest {}
15
16 message TestPayloadResponse {
17     string message = 1;
18 }
19
20 message TestHighPayloadRequest {}
21
22 message TestHighPayloadResponse {
23     string message = 1;
24 }
25
26 message TestStreamPayloadRequest {}
27
28 message TestStreamPayloadResponse {
29     bytes chunk = 1;
30 }
```

Listing 5.5: Zawartość pliku .proto, do którego tworzony jest automatycznie kod oraz typy

5.4.2 Wyniki dla protokołu HTTP/2

Listing 5.6 pokazuje implementację funkcji obsługujących zapytania gRPC.

```

1 async fn test_payload(
2     &self,
3     _request: Request<TestPayloadRequest>,
4     ) -> Result<Response<TestPayloadResponse>, Status> {
5     let payload = "x".repeat(1024);
```

```

6     Ok(Response::new(TestPayloadResponse {
7         message: payload,
8     )))
9 }
10
11    async fn test_high_payload(
12        &self,
13        _request: Request<TestHighPayloadRequest>,
14    ) -> Result<Response<TestHighPayloadResponse>, Status> {
15        let payload = "x".repeat(1024 * 1024);
16        Ok(Response::new(TestHighPayloadResponse {
17            message: payload,
18        )))
19    }

```

Listing 5.6: Funkcje wykorzystywane w testach

Wyniki pomiarów – standardowy ładunek: gRPC osiągnął przepustowość 168 060 żądań na sekundę dla standardowego ładunku, co stanowi 15% spadek względem GraphQL HTTP/2. Niższa wartość może wynikać z narzutu związanego z serializacją Protocol Buffers oraz dodatkowych nagłówków gRPC.

Wyniki pomiarów – rozszerzony ładunek: Dla rozszerzonego ładunku zmierzono przepustowość 2 243 żądań na sekundę, wykazując 7% przewagę nad GraphQL oraz 33% nad REST w konfiguracji HTTP/2. Wynik ten potwierdza optymalizację gRPC dla transferu dużych wolumenów danych binarnych.

5.5 Architektura WebSocket

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

Listing 5.7 przedstawia kompletną implementację serwera WebSocket obsługującego trzy typy zapytań: standardowy ładunek, rozszerzony ładunek oraz transmisję strumieniową.

```

1 use tokio::net::TcpListener;
2 use tungstenite::accept_async;
3 use futures_util::{StreamExt, SinkExt};
4 use tungstenite::Message;
5 use std::sync::atomic::{AtomicUsize, Ordering};
6 use std::sync::Arc;
7
8 const PAYLOAD_SIZE: usize = 1024;

```

```

9   const HIGH_PAYLOAD_SIZE: usize = 1024 * 1024;
10
11 #[tokio::main]
12 async fn main() -> Result<(), Box<dyn std::error::Error>> {
13     let server = TcpListener::bind("127.0.0.1:9001").await?;
14     println!("WebSocket server listening on ws://127.0.0.1:9001");
15
16     let connection_count = Arc::new(AtomicUsize::new(0));
17
18     loop {
19         let (stream, _) = server.accept().await?;
20         let conn_id = connection_count.fetch_add(1, Ordering::SeqCst)
21             ;
22
23         tokio::spawn(async move {
24             let mut websocket = match accept_async(stream).await {
25                 Ok(ws) => ws,
26                 Err(e) => {
27                     eprintln!("WebSocket handshake failed: {}", e);
28                     return;
29                 }
30             };
31
32             println!("Connection {} established", conn_id);
33
34             let mut request_count = 0;
35
36             while let Some(msg) = websocket.next().await {
37                 match msg {
38                     Ok(Message::Text(text)) => {
39                         request_count += 1;
40
41                         match text.as_str() {
42                             "test" => {
43                                 let payload = "x".repeat(PAYLOAD_SIZE);
44                                 if let Err(e) = websocket.send(
45                                     Message::Text(payload.into())
46                                 ).await {
47                                     eprintln!("Connection {} write
48                                         error: {}", conn_id, e);
49                                 }
50                             }
51                         }
52                     }
53                 }
54             }
55         });
56     }
57 }

```

```

45                         break;
46                     }
47                 }
48             "test_high_payload" => {
49                 let payload = "x".repeat(
50                     HIGH_PAYLOAD_SIZE);
51                 if let Err(e) = websocket.send(
52                     Message::Text(payload.into()));
53                     await {
54                         eprintln!("Connection {} write
55                             error: {}", conn_id, e);
56                         break;
57                     }
58             }
59             "stream" => {
60                 for count in 0..100 {
61                     let chunk = format!("chunk {}\n",
62                         count);
63                     if let Err(e) = websocket.send(
64                         Message::Text(chunk.into()));
65                         await {
66                             eprintln!("Connection {}
67                                 write error at chunk {}:"
68                                     "{}", conn_id, count, e);
69                             break;
70                         }
71                 }
72             }
73         _ => {
74             eprintln!("Connection {} unknown
75                 command: {}", conn_id, text);
76         }
77     }
78
79     if request_count % 100 == 0 {
80         println!("Connection {} processed {}
81                         requests", conn_id, request_count);
82     }
83 }
84 Ok(Message::Close(_)) => {
85     println!("Connection {} closed gracefully
86

```

```

75                         after {} requests", conn_id, request_count
76                         );
77             let _ = websocket.send(Message::Close(None)).await;
78             break;
79         }
80     Err(e) => {
81         eprintln!("Connection {} error: {}", conn_id, e);
82         break;
83     }
84     _ => {}
85 }
86 println!("Connection {} handler exiting", conn_id);
87 });
88 }
89 }
```

Listing 5.7: Implementacja serwera WebSocket

5.5.1 Wyniki dla protokołu WebSocket

Wyniki pomiarów – standardowy ładunek: Protokół WebSocket osiągnął przepustowość 122 431 żądań na sekundę, wykazując najwyższą wartość spośród testowanych protokołów dla standardowego ładunku. Niższa wydajność wynika z narzutu związanego z utrzymywaniem trwałych połączeń oraz ramkowaniem wiadomości.

Wyniki pomiarów – rozszerzony ładunek: Dla dużych pakietów danych zaobserwowano przepustowość 3 187 żądań na sekundę – najwyższą wartość wśród wszystkich badanych protokołów, przewyższającą o 42% najbliższego konkurenta (gRPC). Wynik ten potwierdza optymalizację WebSocket dla transferu dużych wolumenów danych poprzez eliminację narzutu związanego z nawiązywaniem połączeń.

5.5.2 Wyniki dla protokołu WebSocket Secure (WSS)

Implementacja protokołu WebSocket Secure jest niemal identyczna z wersją niezabezpieczoną. Kluczową różnicą jest dodanie warstwy TLS, która zapewnia szyfrowanie komunikacji oraz weryfikację tożsamości serwera.

Wyniki pomiarów – standardowy ładunek: Wersja zabezpieczona (WSS) osiągnęła przepustowość 111 086 żądań na sekundę, co stanowi 9% degradację względem niezabezpieczo-

nej implementacji. Spadek ten wynika z narzutu związanego z szyfrowaniem oraz weryfikacją certyfikatów.

Wyniki pomiarów – rozszerzony ładunek: Dla rozszerzonego ładunku zmierzono przepustowość 2 932 żądań na sekundę, wykazując 8% spadek względem wersji niezabezpieczonej. Proporcjonalnie mniejsza degradacja dla dużych pakietów wskazuje, że koszt TLS handshake jest amortyzowany w długotrwałych transferach.

5.6 Architektura SOAP

Implementacja protokołu SOAP została przetestowana w obu wersjach HTTP w celu porównania charakterystyk wydajnościowych. Implementacja opiera się na bibliotece Actix-web, identycznej jak w przypadku REST API.

Listing 5.8 przedstawia implementację punktów końcowych SOAP generujących odpowiedzi w formacie XML zgodnym ze specyfikacją SOAP 1.2.

```
1  use actix_web::{HttpResponse, post};

2
3  #[post("/test")]
4  async fn soap_handler() -> HttpResponse {
5      let payload = "x".repeat(1024);

6
7      let response = format!(
8          r#"
9              <?xml version="1.0" encoding="utf-8"?>
10             <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/
11                 envelope/"
12                 xmlns:test="urn:benchmark">
13                 <soap:Body>
14                     <test:TestPayloadResponse>
15                         <test:message>{}</test:message>
16                     </test:TestPayloadResponse>
17                 </soap:Body>
18             </soap:Envelope>
19             "#,
20             payload
21     );
22
23     HttpResponse::Ok()
24         .content_type("text/xml; charset=utf-8")
25         .body(response)
26 }
```

```

26
27 #[post("/test_large_payload")]
28     fn soap_large_payload_handler() -> HttpResponse {
29         let payload = "x".repeat(1024 * 1024);
30         let response = format!(
31             r#"
32                 <?xml version="1.0" encoding="utf-8"?>
33                 <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/
34                     envelope/" xmlns:test="urn:benchmark">
35                     <soap:Body>
36                         <test:TestPayloadResponse>
37                             <test:message>{}</test:message>
38                         </test:TestPayloadResponse>
39                     </soap:Body>
40                 </soap:Envelope>
41             "#,
42             payload
43         );
44
45         HttpResponse::Ok()
46             .content_type("text/xml; charset=utf-8")
47             .body(response)
48     }

```

Listing 5.8: Implementacja endpointów SOAP

5.6.1 Wyniki dla protokołu HTTP/1.1

Wyniki pomiarów – standardowy ładunek: SOAP z HTTP/1.1 osiągnął przepustowość 198 232 żądań na sekundę, demonstrując porównywalne wyniki z REST HTTP/2. Nieoczekiwane wysoka wydajność wynika z efektywnej implementacji generowania XML oraz braku konieczności parsowania złożonych struktur danych.

Wyniki pomiarów – rozszerzony ładunek: Dla rozszerzonego ładunku zmierzono przepustowość 2 892 żądań na sekundę – najwyższą wartość wśród protokołów wykorzystujących HTTP/1.1, przewyższającą REST o 4,8%.

5.6.2 Wyniki dla protokołu HTTP/2

Protokół SOAP z TLS automatycznie wykorzystuje HTTP/2, jeśli klient i serwer wspierają negocjację ALPN (Application-Layer Protocol Negotiation).

Wyniki pomiarów – standardowy ładunek: Implementacja z HTTP/2 osiągnęła przepustowość 213 973 żądań na sekundę, wykazując 8% poprawę względem HTTP/1.1 oraz najwyższy wynik spośród wszystkich protokołów HTTP/2 dla małych pakietów.

Wyniki pomiarów – rozszerzony ładunek: Dla dużych pakietów danych zaobserwowano przepustowość 2 478 żądań na sekundę, co stanowi 14% spadek względem HTTP/1.1. Degrada-cja ta jest nietypowa i może wynikać z narzutu multipleksowania strumieni dla pojedynczych, dużych odpowiedzi XML.

5.7 Protokół MQTT

MQTT (Message Queuing Telemetry Transport) to lekki protokół publikacji/subskrypcji zaprojektowany dla środowisk z ograniczonymi zasobami oraz niestabilnymi połączeniami sieciowymi.

5.7.1 Implementacja serwera

Listing 5.9 przedstawia główną pętlę zdarzeń (ang. *event loop*) odbiorcy MQTT, która przetwarza przychodzące publikacje i wysyła odpowiedzi na dedykowane tematy.

```
1  loop {
2      match eventloop.poll().await {
3          Ok(Event::Incoming(Incoming::Publish(p))) => {
4              request_count += 1;
5
6              let response_topic = match p.topic.as_str() {
7                  "benchmark/request" => "benchmark/response",
8                  "benchmark/high_payload" => "benchmark/
9                      high_payload_response",
10                 _ => continue,
11             };
12
13             if let Err(e) = client
14                 .publish(
15                     response_topic,
16                     QoS::AtLeastOnce,
17                     false,
18                     p.payload,
19                     )
20                     .await
21             {
22                 eprintln!("Failed to publish response: {:?}", e);
23             }
24         }
25     }
26 }
```

```

22     }
23
24     if request_count % 1000 == 0 {
25         println!("Processed {} requests", request_count);
26     }
27 }
28 Ok(_) => {}
29 Err(e) => {
30     eprintln!("Server error: {:?}", e);
31     tokio::time::sleep(Duration::from_secs(1)).await;
32 }
33 }
34 }
```

Listing 5.9: Główna pętla zdarzeń odbiorcy MQTT

5.7.2 Implementacja klienta

Listing 5.10 pokazuje implementację klienta MQTT wykorzystywanego w testach wydajnościowych, obsługującego publikację wiadomości oraz odbieranie odpowiedzi z mechanizmem timeout.

```

1 let mut latencies = Vec::with_capacity(config.requests_per_worker);
2
3 // Create unique client ID for this worker
4 let client_id = format!("benchmark_worker_{}", worker_id);
5 let mut opts = MqttOptions::new(client_id, "127.0.0.1", 1883);
6 opts.set_keep_alive(Duration::from_secs(30));
7
8 // Increase max packet size to handle 1MB+ payloads
9 opts.set_max_packet_size(10 * 1024 * 1024, 10 * 1024 * 1024); // 10
10 MB max
11
12 let (client, mut eventloop) = AsyncClient::new(opts, 1000);
13
14 let (response_tx, mut response_rx) = mpsc::unbounded_channel();
15
16 // Spawn event loop handler
17 let response_topic = if config.high_payload {
18     "benchmark/high_payload_response"
19 } else {
    "benchmark/response"
```

```

20     };
21
22     tokio::spawn(async move {
23         loop {
24             match eventloop.poll().await {
25                 Ok(Event::Incoming(Incoming::Publish(p))) => {
26                     if p.topic == response_topic {
27                         let _ = response_tx.send(());
28                     }
29                 }
30                 Ok(_) => {}
31                 Err(e) => {
32                     eprintln!("Worker {} eventloop error: {:?}", worker_id, e);
33                     break;
34                 }
35             }
36         }
37     });
38
39     // Subscribe to response topic
40     client
41         .subscribe(response_topic, QoS::AtLeastOnce)
42         .await?;
43
44     // Wait a bit for subscription to be established
45     tokio::time::sleep(Duration::from_millis(100)).await;
46
47     let request_topic = if config.high_payload {
48         "benchmark/high_payload"
49     } else {
50         "benchmark/request"
51     };
52
53     let payload = if config.high_payload {
54         serde_json::json!({
55             "id": format!("worker_{}", worker_id),
56             "payload": "x".repeat(1024 * 1024) // 1 MB payload
57         })
58         .to_string()
59     } else {

```

```

60     serde_json::json!({
61         "id": format!("worker_{}", worker_id),
62         "payload": "x".repeat(config.payload_size)
63     })
64     .to_string()
65 };
66
67 for i in 0..config.requests_per_worker {
68     // Acquire semaphore permit to control concurrency
69     let _permit = semaphore.acquire().await?;
70
71     let start = Instant::now();
72
73     // Send request
74     client
75         .publish(request_topic, QoS::AtLeastOnce, false, payload.clone())
76         .await?;
77
78     // Wait for response with timeout
79     match timeout(Duration::from_secs(10), response_rx.recv()).await
80         {
81             Ok(Some(_)) => {
82                 let latency = start.elapsed();
83                 latencies.push(latency);
84             }
85             Ok(None) => {
86                 eprintln!("Worker {} request {}: Channel closed",
87                         worker_id, i);
88                 break;
89             }
90             Err(_) => {
91                 eprintln!("Worker {} request {}: Timeout", worker_id, i);
92             }
93         }
94
95     drop(_permit);
96 }

```

Listing 5.10: Implementacja klienta MQTT dla testów wydajnościowych

5.7.3 Wyniki pomiarów

Wyniki pomiarów – standardowy ładunek: Protokół MQTT osiągnął przepustowość 112 126 żądań na sekundę dla standardowego ładunku, co lokuje go w dolnej części spektrum wydajnościowego dla małych pakietów.

Wyniki pomiarów – rozszerzony ładunek: Dla rozszerzonego ładunku zmierzono przepustowość 2 167 żądań na sekundę, lokując MQTT w środkowym przedziale wydajnościowym. Wynik ten potwierdza, że protokół jest zoptymalizowany pod kątem niezawodności i niskiego zużycia zasobów, nie zaś maksymalnej przepustowości.

5.8 Analiza porównawcza protokołów komunikacyjnych

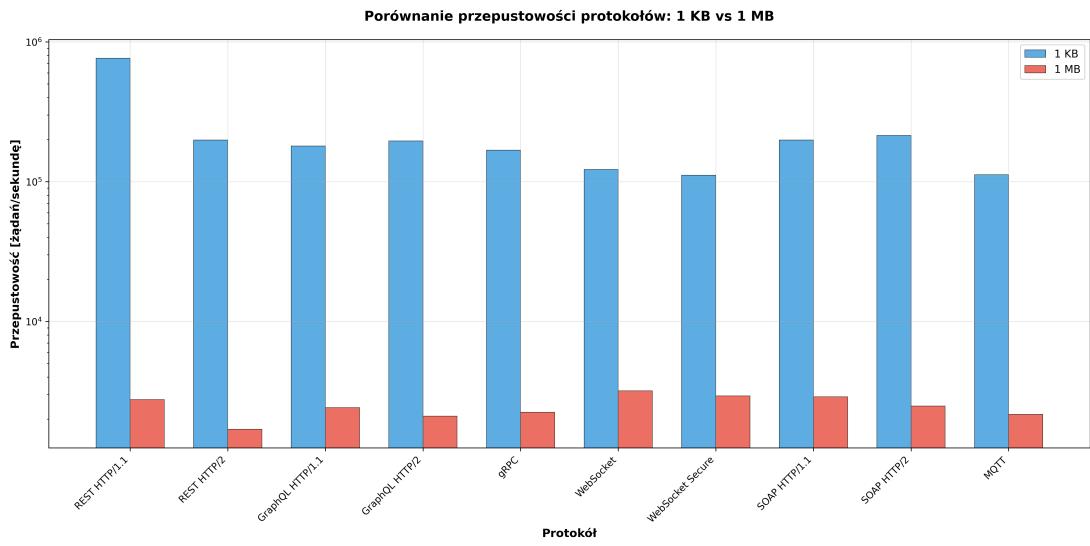
Przeprowadzone badania umożliwiły kompleksową ocenę wydajności protokołów komunikacyjnych w różnych scenariuszach użycia. Poniższa analiza syntetyzuje kluczowe wnioski z testów.

5.8.1 Wpływ rozmiaru ładunku na wydajność

Tabela 5.1: Porównanie przepustowości protokołów komunikacyjnych

Protokół	1 KB [req/s]	1 MB [req/s]	Degradačja
REST HTTP/1.1	764 076	2 759	276,9×
REST HTTP/2	198 272	1 690	117,3×
GraphQL HTTP/1.1	179 760	2 418	74,3×
GraphQL HTTP/2	195 877	2 097	93,4×
gRPC	168 060	2 243	74,9×
WebSocket	122 431	3 187	38,4×
WebSocket Secure	111 086	2 932	37,9×
SOAP HTTP/1.1	198 232	2 892	68,5×
SOAP HTTP/2	213 973	2 478	86,3×
MQTT	112 126	2 167	51,7×

Analiza współczynnika degradacji (tabela 5.1) ujawnia fundamentalne różnice w charakterystykach wydajnościowych protokołów. WebSocket Secure wykazał najniższą degradację (37,9×), co potwierdza jego optymalizację dla transmisji dużych wolumenów danych poprzez utrzymywanie stałego połączenia. W przeciwnieństwie do tego, REST HTTP/1.1 odnotował najwyższą degradację (276,9×), co jest konsekwencją narzutu związanego z nawiązywaniem nowego połączenia TCP dla każdego żądania oraz brakiem multipleksowania.



Rysunek 5.1: Porównanie przepustowości protokołów dla ładunku 1 KB i 1 MB (skala logarytmiczna).

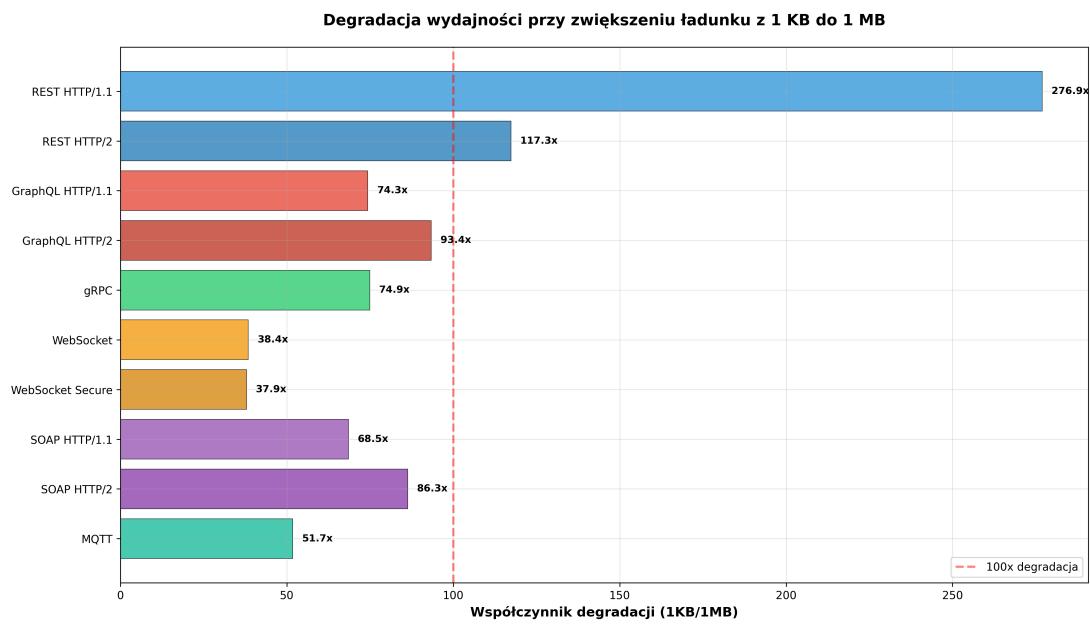
Wykres 5.1 ilustruje wyraźną dychotomię między wydajnością dla małych i dużych ładunków. Protokoły oparte na HTTP/1.1 dominują w scenariuszach małych pakietów, podczas gdy protokoły z trwałymi połączeniami (WebSocket, WSS) wykazują przewagę dla dużych transferów danych.

5.8.2 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 – znaczająco niższą niż REST HTTP/1.1 vs HTTP/2 (74%), co wskazuje, że jednorazowy koszt TLS handshake jest amortyzowany w długotrwałych połączeniach.



Rysunek 5.2: Współczynnik degradacji wydajności przy zwiększeniu ładunku z 1 KB do 1 MB.

5.8.3 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'owność, powszechnie wsparcie
- Wybór HTTP/2: Gdy priorytetem jest mitigacja HOL blocking i bezpieczeństwo

2. GraphQL

- Zastosowanie: Frontend-heavy aplikacje, złożone modele danych
- Zalety: Eliminacja over-fetchingu, redukcja liczby zapytań
- Kompromis: ~1–2% niższa wydajność niż REST przy znaczającej poprawie developer experience

3. gRPC

- Zastosowanie: Komunikacja service-to-service, high-performance backends
- Zalety: Najlepsza wydajność dla dużych ładunków w HTTP/2, silne kontrakty
- Ograniczenie: Słabsze wsparcie w przeglądarkach (wymaga gRPC-web)

4. WebSocket/WSS

- Zastosowanie: Aplikacje real-time (czaty, giełdy, gaming)
- Zalety: Najniższa latencja, najlepsza wydajność dla dużych ładunków
- Kompromis: Zwiększoną złożoność skalowania (sticky sessions)

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, kolejkowanie, QoS levels
- Optymalne: Dla rozproszonych sieci sensorów z intermittent connectivity

5.8.4 Kluczowe wnioski

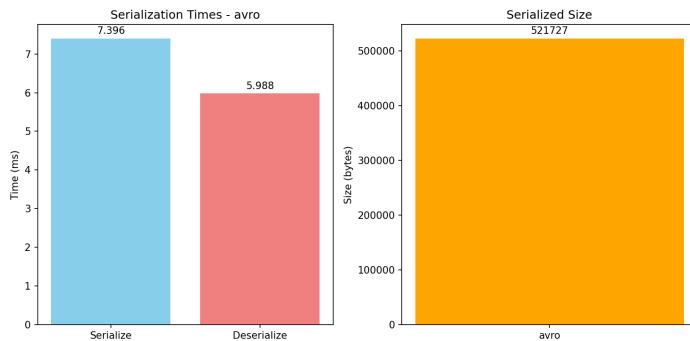
Przeprowadzone badania prowadzą do następujących konkluzji:

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 backend-to-backend.
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 z persistent connections (WebSocket, HTTP/2) wykazują niższą degradację przy wzroście ładunku, co czyni je preferowanymi dla high-throughput scenarios.
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 mechanizmowi QoS i niskiej konsumpcji zasobów.

5.9 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.

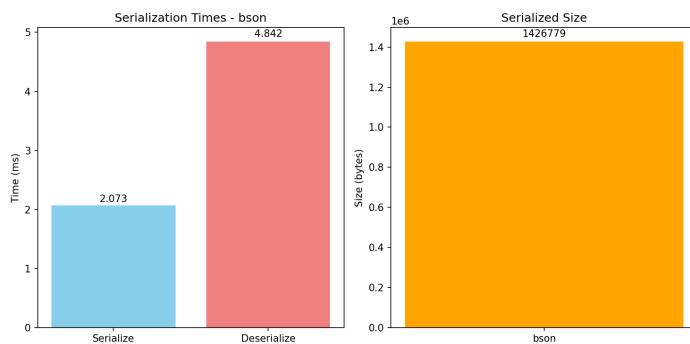
5.9.1 Apache Avro



Rysunek 5.3: Metryki wydajności Apache Avro.

Apache Avro uzyskał czas serializacji 7,396 ms, deserializacji 5,988 ms przy rozmiarze danych wynoszącym 521 727 bajtów (509,5 KB).

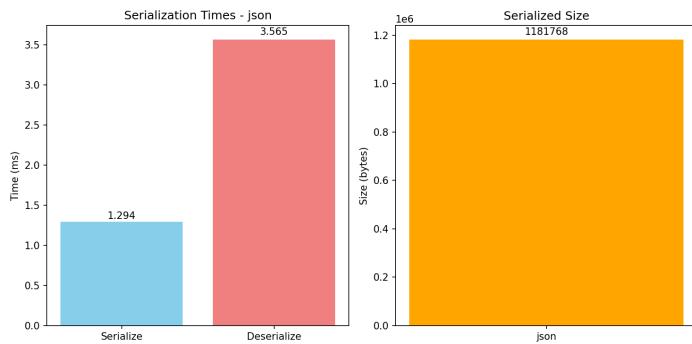
5.9.2 Binary JSON (BSON)



Rysunek 5.4: Metryki wydajności Binary JSON.

Format BSON wykazał czas serializacji 2,073 ms, deserializacji 4,842 ms, generując dane o rozmiarze 1 426 779 bajtów (1 393,3 KB) – największym spośród testowanych formatów.

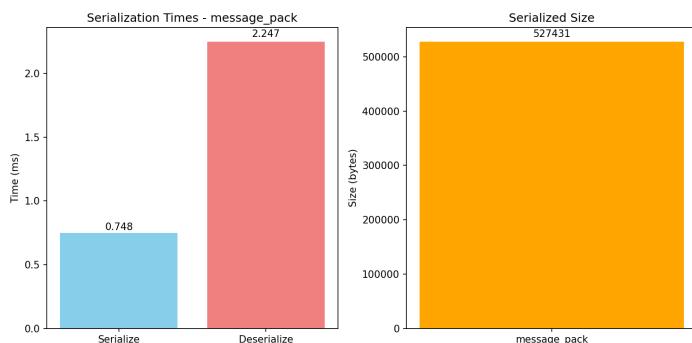
5.9.3 JSON



Rysunek 5.5: Metryki wydajności JSON.

Standardowy JSON osiągnął czas serializacji 1,294 ms oraz deserializacji 3,565 ms przy rozmiarze 1 181 768 bajtów (1 154,1 KB), oferując najszybszą deserializację wśród formatów tekstowych.

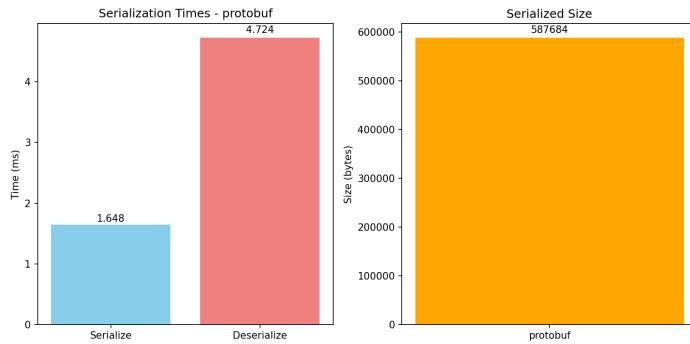
5.9.4 MessagePack



Rysunek 5.6: Metryki wydajności MessagePack.

MessagePack wykazał najlepszą wydajność czasową z serializacją 0,748 ms i deserializacją 2,247 ms, przy kompaktowym rozmiarze 527 431 bajtów (515,1 KB).

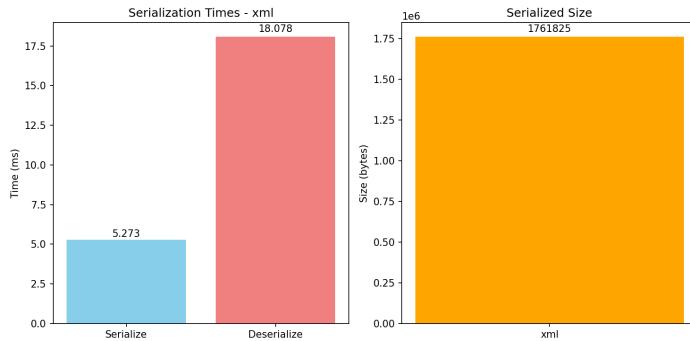
5.9.5 Protocol Buffers



Rysunek 5.7: Metryki wydajności Protocol Buffers.

Protocol Buffers osiągnął czas serializacji 1,648 ms, deserializacji 4,724 ms, generując dane o rozmiarze 587 684 bajtów (573,9 KB).

5.9.6 XML



Rysunek 5.8: Metryki wydajności XML.

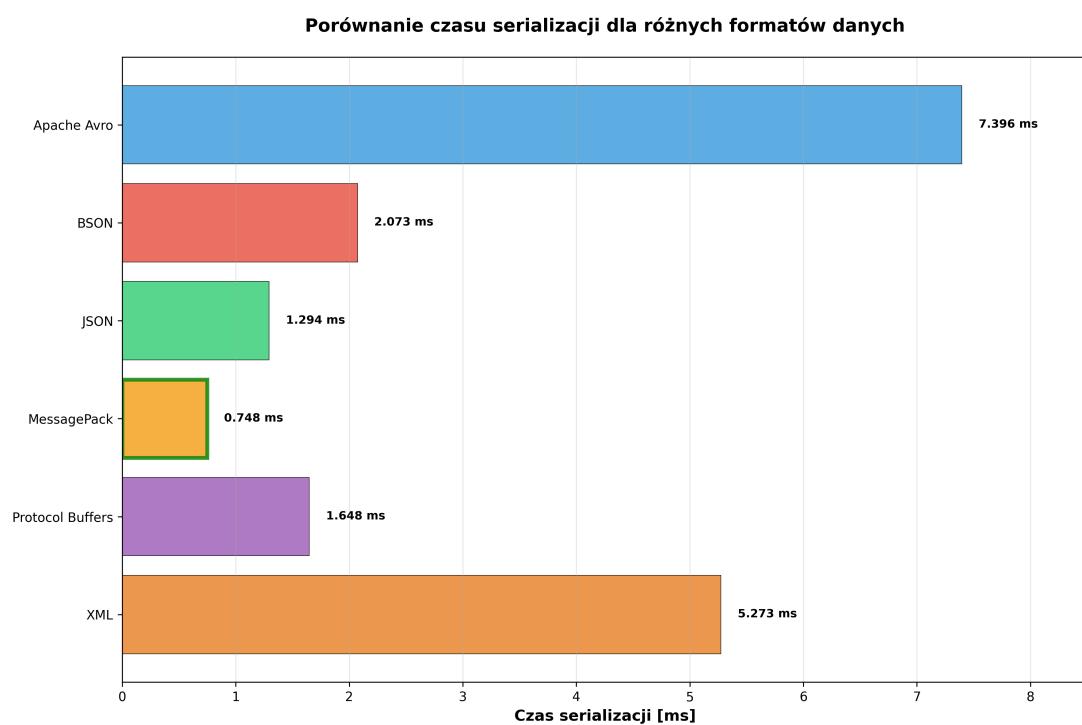
Format XML wykazał najdłuższe czasy przetwarzania – serializację 5,273 ms i deserializację 18,078 ms – przy największym rozmiarze danych 1 761 825 bajtów (1 720,5 KB), co stanowi 3,4-krotność najbardziej efektywnych formatów binarnych.

5.9.7 Analiza porównawcza formatów serializacji

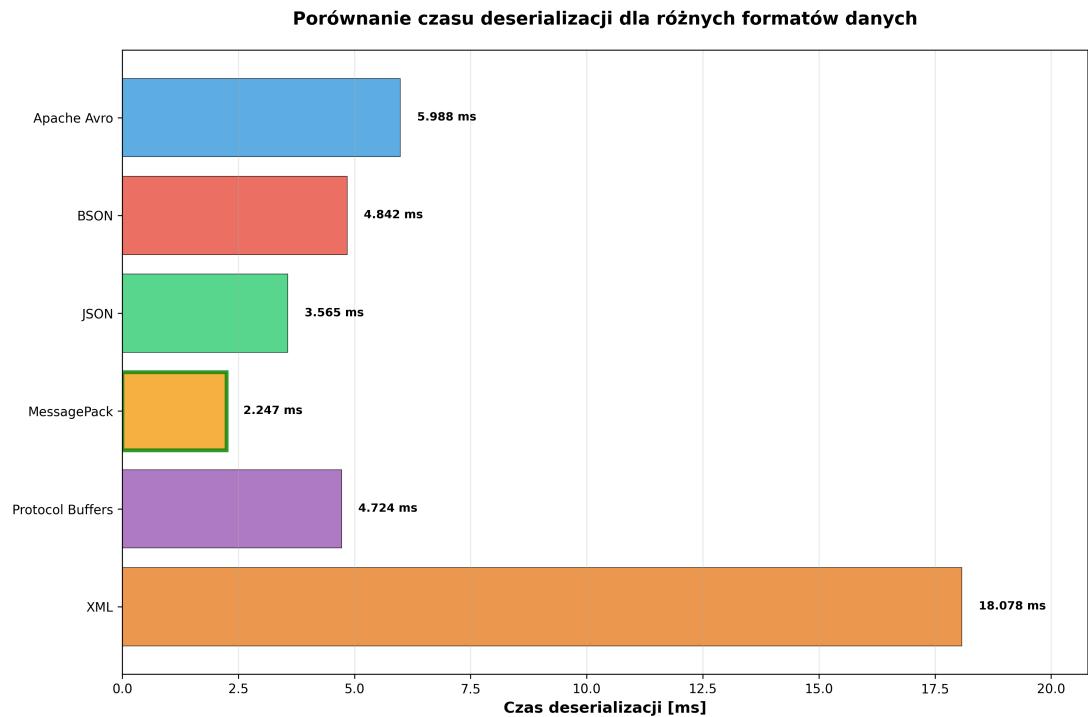
Kompleksowa analiza wydajności formatów serializacji wymaga uwzględnienia trzech klu-czowych metryk: czasu serializacji, czasu deserializacji oraz rozmiaru wynikowych danych.

Tabela 5.2: Porównanie formatów serializacji danych

Format	Serializacja [ms]	Deserializacja [ms]	Rozmiar [KB]
Apache Avro	7,396	5,988	509,5
BSON	2,073	4,842	1393,3
JSON	1,294	3,565	1154,1
MessagePack	0,748	2,247	515,1
Protocol Buffers	1,648	4,724	573,9
XML	5,273	18,078	1720,5



Rysunek 5.9: Porównanie czasu serializacji różnych formatów.



Rysunek 5.10: Porównanie czasu deserializacji formatowania danych.

Wydajność czasowa

MessagePack wykazał najlepszą wydajność czasową we wszystkich kategoriach:

- Najszybsza serializacja: 0,748 ms ($6,0\times$ szybciej niż XML, $9,9\times$ szybciej niż Apache Avro)
- Najszybsza deserializacja: 2,247 ms ($8,0\times$ szybciej niż XML, $2,7\times$ szybciej niż Apache Avro)
- Najniższy całkowity czas przetwarzania: 2,995 ms

JSON i Protocol Buffers osiągnęły porównywalne wyniki czasowe (odpowiednio 4,859 ms i 6,372 ms łącznie), oferując dobry kompromis między wydajnością a czytelnością (JSON) lub efektywnością kompresji (Protobuf).

XML wykazał najgorsze charakterystyki czasowe z deserializacją trwającą 18,078 ms – ponad $8\times$ dłużej niż najbliższy konkurent (Apache Avro). Ta degradacja wynika z parsowania tekstowej struktury hierarchicznej oraz narzutu związanego z walidacją schematu.

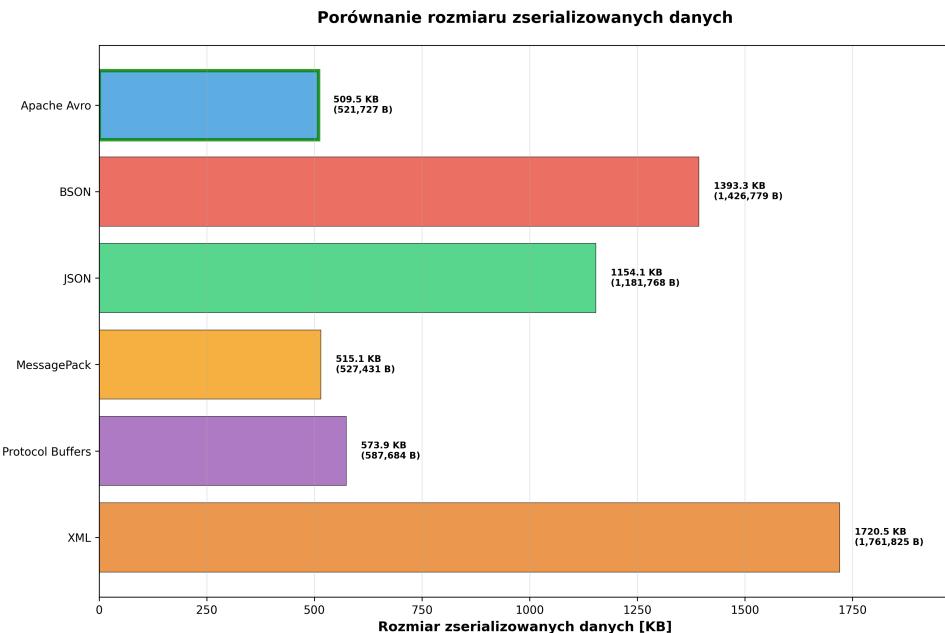
Efektywność kompresji

Apache Avro osiągnął najlepszą kompresję danych z rozmiarem 521 727 bajtów (509,5 KB), co stanowi:

- $2,2\times$ mniejszy rozmiar niż JSON

- $2,7\times$ mniejszy rozmiar niż BSON
- $3,4\times$ mniejszy rozmiar niż XML

MessagePack wykazał rozmiar 527 431 bajtów (515,1 KB) – zaledwie 1,1% większy niż Avro, co w połączeniu z najlepszą wydajnością czasową czyni go najbardziej zrównoważonym formatem.



Rysunek 5.11: Porównanie rozmiaru zserializowanych danych dla różnych formatów.

XML i BSON wykazały najbardziej nieefektywną kompresję, przekraczając 1,4 MB dla tego samego zestawu danych. W przypadku XML nadmiarowy rozmiar wynika z tagów otwierających i zamykających oraz metadanych, podczas gdy BSON przechowuje dodatkowe informacje typów dla każdego pola.

Charakterystyki formatów i rekomendacje

MessagePack – optymalny wybór dla aplikacji wymagających maksymalnej wydajności:

- Najszybszy całkowity czas przetwarzania (2,995 ms)
- Bardzo dobra kompresja (515 KB, +1% względem najlepszego)
- Idealne zastosowanie: real-time gaming, high-frequency trading, cache'owanie
- Ograniczenie: brak natywnej czytelności dla człowieka

Apache Avro – najlepsza kompresja dla big data:

- Najmniejszy rozmiar danych (509,5 KB)

- Schema evolution – wsparcie dla zmian struktury danych w czasie
- Idealne zastosowanie: Apache Kafka, Hadoop, długoterminowe przechowywanie danych
- Ograniczenie: wolniejsza serializacja (7,4 ms) – nie nadaje się dla ultra-low-latency

JSON – balans między wydajnością a developer experience:

- Szybka serializacja (1,294 ms), umiarkowana deserializacja (3,565 ms)
- Natywna czytelność, debugowalność, wsparcie w przeglądarkach
- Idealne zastosowanie: REST APIs, konfiguracje, komunikacja frontend-backend
- Kompromis: 2,2× większy rozmiar niż formaty binarne

Protocol Buffers – wydajność dla kontraktów API:

- Dobra równowaga: średni czas (6,4 ms), średni rozmiar (574 KB)
- Silne typowanie, backward/forward compatibility
- Idealne zastosowanie: gRPC, komunikacja mikroserwisowa, mobile APIs
- Wymaganie: generowanie kodu z plików .proto

BSON – specjalizacja dla baz dokumentowych:

- Szybka serializacja (2,073 ms), dedykowana dla MongoDB
- Wsparcie dla typów binarnych (ObjectId, DateTime, Binary)
- Idealne zastosowanie: MongoDB storage/queries, document databases
- Ograniczenie: 2,7× większy rozmiar niż formaty binarne

XML – legacy i compliance:

- Wsparcie dla złożonych schematów (XSD), przestrzeni nazw, transformacji (XSLT)
- Najgorsze metryki wydajnościowe: 23,4 ms łącznie, 1721 KB
- Idealne zastosowanie: systemy enterprise, SOAP, compliance (HIPAA, financial regulations)
- Uzasadnienie: wymagania regulacyjne często przeważają nad wydajnością

Wnioski

Przeprowadzone badania potwierdzają, że nie istnieje uniwersalnie optymalny format serializacji – wybór powinien być determinowany przez specyficzne wymagania aplikacji:

1. **Dla maksymalnej wydajności:** MessagePack oferuje najlepszy czas przetwarzania przy akceptowalnym rozmiarze
2. **Dla minimalnego rozmiaru:** Apache Avro zapewnia najlepszą kompresję, idealny dla storage i network bandwidth
3. **Dla developer experience:** JSON oferuje czytelność i debugowalność kosztem rozmiaru
4. **Dla typowanych API:** Protocol Buffers zapewnia kontrakt i kompatybilność wsteczną
5. **Dla compliance:** XML pozostaje niezbędny w regulowanych środowiskach enterprise

Kluczowym odkryciem jest, że różnica między najszybszym (MessagePack: 2,995 ms) a najwolniejszym (XML: 23,351 ms) formatem wynosi 7,8 \times , podczas gdy różnica w rozmiarze wynosi 3,4 \times . To wskazuje, że wydajność czasowa wykazuje większą wariancję niż efektywność kompresji, czyniąc ją krytycznym czynnikiem decyzyjnym w aplikacjach real-time.

5.10 Analiza porównawcza WebAssembly i JavaScript

Do stworzenia modułów WebAssembly wykorzystano narzędzie **wasm-pack**. Jest to gotowe rozwiązanie, które kompliluje kod z języka Rust na język JavaScript, gdzie część kodu jest napisana w WebAssembly. Takie podejście umożliwia przeniesienie wybranych fragmentów kodu do WebAssembly, co pozwala na przyśpieszenie działania aplikacji.

5.10.1 Web Workers

Technologia Web Workers umożliwia przerzucenie części kodu, który wykonuje się na wątku głównym, na wątek poboczny. Stworzone zostały dwa przykładowe workery zawierające kod napisany w języku Rust (z równoważnym kodem napisanym w JavaScript) – jeden z małym obciążeniem obliczeniowym i drugi z dużym obciążeniem obliczeniowym.

Implementacja workerów o niskim obciążeniu obliczeniowym

Listing 5.11 przedstawia implementację Web Workera wykonującego podstawowe operacje matematyczne na tablicy liczb zmiennoprzecinkowych. Worker oblicza sumę pierwiastków kwadratowych pomnożonych przez logarytmy naturalne dla każdego elementu tablicy.

```

1  use wasm_bindgen::prelude::*;
2  use js_sys::Float64Array;
3  use web_sys::DedicatedWorkerGlobalScope;
4
5  #[wasm_bindgen(start)]
6  pub fn worker_entry() {
7      let global: DedicatedWorkerGlobalScope =
8          js_sys::global().unchecked_into();
9
10     let global_for_closure = global.clone();
11
12     let onmessage = Closure::<dyn FnMut(_)>::new(move |event: web_sys
13         ::MessageEvent| {
14             let buffer = event.data();
15             let data = Float64Array::new(&buffer);
16
17             let performance = global_for_closure.performance().unwrap();
18             let t0 = performance.now();
19
20             let mut sum = 0.0;
21             for i in 0..data.length() {
22                 let x = data.get_index(i);
23                 sum += x.sqrt() * (x + 1.0).ln();
24             }
25
26             let t1 = performance.now();
27
28             let response = js_sys::Object::new();
29             js_sys::Reflect::set(&response, &"time".into(), &(t1 - t0).
30                 into()).unwrap();
31
32             global_for_closure.post_message(&response).unwrap();
33         });
34
35         global.set_onmessage(Some(onmessage.as_ref().unchecked_ref()));
36         onmessage.forget();
37     }

```

Listing 5.11: Implementacja Web Workera z prostymi operacjami obliczeniowymi

Implementacja workerów o wysokim obciążeniu obliczeniowym

Listing 5.12 pokazuję implementację Web Workera wykonującego złożone operacje obliczeniowe z wykorzystaniem mapy chaotycznej (ang. *chaotic map*). Algorytm ten symuluje zachowanie chaotyczne poprzez iteracyjne obliczenia wykorzystujące mapowanie logistyczne.

```
1  use wasm_bindgen::prelude::*;
2  use js_sys::Float64Array;
3  use web_sys::DedicatedWorkerGlobalScope;
4
5  #[wasm_bindgen(start)]
6  pub fn worker_entry() {
7      let global: DedicatedWorkerGlobalScope =
8          js_sys::global().unchecked_into();
9
10     let global_for_closure = global.clone();
11
12     let onmessage = Closure::<dyn FnMut(_)>::new(move |event: web_sys
13         ::MessageEvent| {
14         let buffer = event.data();
15         let data = Float64Array::new(&buffer);
16
17         let perf = global_for_closure.performance().unwrap();
18         let t0 = perf.now();
19
20         let mut acc = 0.0;
21         let mut seed = 0.123456789_f64;
22
23         for i in 0..data.length() {
24             let v = data.get_index(i);
25
26             // Mapowanie chaotyczne (logistic map)
27             seed = seed * 3.999999 - seed * seed;
28             let x = seed + v;
29
30             seed = seed * 3.999999 - seed * seed;
31             let y = seed + v;
32
33             acc += (x * x + y * y).sqrt().ln();
34         }
35
36         let t1 = perf.now();
```

```

37     let result = js_sys::Object::new();
38     js_sys::Reflect::set(&result, &"value".into(), &acc.into());
39         unwrap();
40     js_sys::Reflect::set(&result, &"time".into(), &(t1 - t0).into()
41         ()).unwrap();
42
43     global_for_closure.post_message(&result).unwrap();
44 }
45
46     global.set_onmessage(Some(onmessage.as_ref().unchecked_ref()));
onmessage.forget();
}

```

Listing 5.12: Implementacja Web Workera z zaawansowanymi operacjami obliczeniowymi

Wyniki pomiarów – Web Workers

Tabela 5.3 przedstawia porównanie czasów wykonania dla obu implementacji (JavaScript i WebAssembly) w zależności od złożoności zadania obliczeniowego.

Tabela 5.3: Porównanie wydajności Web Workers – JavaScript vs. WebAssembly

Typ zadania	JavaScript [ms]	WebAssembly [ms]
Małe zadanie obliczeniowe	9,6	43,1
Duże zadanie obliczeniowe	113,3	91,69

Wyniki wskazują, że dla prostych operacji JavaScript osiąga lepszą wydajność (9,6 ms vs. 43,1 ms), prawdopodobnie ze względu na narzut związany z inicjalizacją modułu WebAssembly oraz przekazywaniem danych między środowiskami. Natomiast w przypadku złożonych obliczeń WebAssembly wykazuje przewagę wydajnościową (91,69 ms vs. 113,3 ms), co potwierdza jego przydatność w scenariuszach wymagających intensywnych operacji numerycznych.

5.10.2 Operacje na Canvas

Drugi zestaw testów dotyczył operacji przetwarzania obrazu na elemencie Canvas. Wszystkie operacje zostały wykonane na obrazie w rozdzielczości 4K (3840×2160 pikseli), co odpowiada przetwarzaniu 8 294 400 pikseli (każdy reprezentowany przez 4 bajty w formacie RGBA).

Implementacja filtrów graficznych

Listing 5.13 zawiera implementację czterech podstawowych operacji przetwarzania obrazu: konwersji do skali szarości, regulacji jasności, kontrastu oraz rozmycia (blur).

```

1  use wasm_bindgen::prelude::*;
2
3  #[wasm_bindgen]
4  pub fn grayscale(data: &mut [u8]) {
5      for i in (0..data.len()).step_by(4) {
6          let r = data[i] as u32;
7          let g = data[i + 1] as u32;
8          let b = data[i + 2] as u32;
9          // Wzor uwzgledniajacy percepcje ludzkiego oka
10         let gray = ((299 * r + 587 * g + 114 * b) / 1000) as u8;
11         data[i] = gray;
12         data[i + 1] = gray;
13         data[i + 2] = gray;
14     }
15 }
16
17 #[wasm_bindgen]
18 pub fn brightness(data: &mut [u8], amount: i32) {
19     for i in (0..data.len()).step_by(4) {
20         data[i] = (data[i] as i32 + amount).clamp(0, 255) as u8;
21         data[i + 1] = (data[i + 1] as i32 + amount).clamp(0, 255) as
22             u8;
23         data[i + 2] = (data[i + 2] as i32 + amount).clamp(0, 255) as
24             u8;
25     }
26 }
27
28 #[wasm_bindgen]
29 pub fn contrast(data: &mut [u8], factor: f32) {
30     let f = (259.0 * (factor + 255.0)) / (255.0 * (259.0 - factor));
31     for i in (0..data.len()).step_by(4) {
32         data[i] = (f * (data[i] as f32 - 128.0) + 128.0).clamp(0.0,
33             255.0) as u8;
34         data[i + 1] = (f * (data[i + 1] as f32 - 128.0) + 128.0).
35             clamp(0.0, 255.0) as u8;
36         data[i + 2] = (f * (data[i + 2] as f32 - 128.0) + 128.0).
37             clamp(0.0, 255.0) as u8;
38     }
39 }
40
41 #[wasm_bindgen]

```

```

37 pub fn blur(data: &mut [u8], width: usize, height: usize, radius:
38     usize) {
39     let temp = data.to_vec();
40     let radius_i = radius as i32;
41
42     for y in 0..height {
43         for x in 0..width {
44             let mut r_sum = 0u32;
45             let mut g_sum = 0u32;
46             let mut b_sum = 0u32;
47             let mut count = 0u32;
48
49             let y_start = (y as i32 - radius_i).max(0) as usize;
50             let y_end = (y + radius + 1).min(height);
51             let x_start = (x as i32 - radius_i).max(0) as usize;
52             let x_end = (x + radius + 1).min(width);
53
54             for ny in y_start..y_end {
55                 for nx in x_start..x_end {
56                     let idx = (ny * width + nx) * 4;
57                     r_sum += temp[idx] as u32;
58                     g_sum += temp[idx + 1] as u32;
59                     b_sum += temp[idx + 2] as u32;
60                     count += 1;
61                 }
62             }
63
64             let idx = (y * width + x) * 4;
65             data[idx] = (r_sum / count) as u8;
66             data[idx + 1] = (g_sum / count) as u8;
67             data[idx + 2] = (b_sum / count) as u8;
68         }
69     }
}

```

Listing 5.13: Implementacja filtrów graficznych dla operacji Canvas

Wyniki pomiarów – operacje Canvas

Tabela 5.4 przedstawia szczegółowe porównanie czasów wykonania dla każdej z czterech operacji graficznych, wraz z wyliczonym współczynnikiem przyspieszenia.

Analiza wyników ujawnia interesujący wzorzec: dla prostych operacji pikselowych (gray-

Tabela 5.4: Porównanie wydajności operacji Canvas – JavaScript vs. WebAssembly

Operacja	JavaScript [ms]	WebAssembly [ms]	Współczynnik
Grayscale	23,6	26,3	0,90×
Brightness	22,1	34,3	0,64×
Contrast	28,4	51,1	0,56×
Blur	1308,5	371,3	3,52×

scale, brightness, contrast) JavaScript przewyższa WebAssembly pod względem wydajności. Natomiast w przypadku operacji rozmycia (blur), która charakteryzuje się znacznie wyższą złożonością obliczeniową ($O(n \cdot r^2)$, gdzie n to liczba pikseli, a r to promień rozmycia), WebAssembly osiąga ponad 3,5-krotne przyspieszenie (371,3 ms kontra 1308,5 ms).

5.10.3 Wnioski

Przeprowadzone testy wydajnościowe pozwalają na sformułowanie następujących wniosków:

1. **Złożoność obliczeniowa jako czynnik decydujący** – WebAssembly wykazuje przewagę wydajnościową głównie w scenariuszach wymagających intensywnych obliczeń o wysokiej złożoności algorytmicznej. Dla operacji prostych narzut związany z inicjalizacją i komunikacją między środowiskami przewyższa potencjalne korzyści.
2. **Optymalizacje specyficzne dla przeglądarki** – nowoczesne silniki JavaScript, takie jak V8, zawierają rozbudowane optymalizacje dla typowych operacji webowych, co może przewajać nad wydajnością WebAssembly w przypadku prostych transformacji danych.
3. **Rekomendacje praktyczne** – WebAssembly stanowi optymalny wybór dla:
 - algorytmów o złożoności $O(n^2)$ lub wyższej,
 - przetwarzania dużych zbiorów danych,
 - operacji wymagających przewidywalnej wydajności,
 - portowania istniejących bibliotek C/C++/Rust.

Z kolei JavaScript pozostaje preferowany dla operacji o niskiej złożoności, szybkiego prototypowania oraz scenariuszy wymagających częstej interakcji z DOM.

Podsumowując, wybór między WebAssembly a JavaScript powinien być uzależniony od specyfiki problemu – optymalne rozwiązanie często stanowi architektura hybrydowa, łącząca elastyczność JavaScript z wydajnością WebAssembly.

Zakończenie

Niniejsza praca magisterska poświęcona była kompleksowej analizie wydajności protokołów komunikacyjnych, formatów serializacji danych oraz technologii wykonawczych stosowanych we współczesnych aplikacjach webowych. Przeprowadzone badania eksperymentalne dostarczyły cennych danych ilościowych oraz pozwoliły na sformułowanie praktycznych rekomendacji dla architektów systemów i programistów.

Podsumowanie wyników badań

Analiza protokołów komunikacyjnych wykazała, że nie istnieje uniwersalne rozwiązanie optymalne dla wszystkich scenariuszy użycia. REST API z protokołem HTTP/1.1 osiągnął najwyższą przepustowość dla małych pakietów danych (764 076 żądań na sekundę), co czyni go doskonałym wyborem dla mikrousług operujących na niewielkich zbiorach danych niewymagających bezpiecznego połączenia. Z kolei WebSocket wykazał najlepszą wydajność dla transferu dużych ładunków (3 187 żądań na sekundę) oraz najniższy współczynnik degradacji ($38,4\times$), potwierdzając przewagę protokołów z trwałymi połączniami w scenariuszach wymagających intensywnej wymiany danych.

Szczególnie istotnym odkryciem była efektywność protokołu HTTP/2 w mitigacji problemu Head-of-Line blocking. Osiągnięta 69-krotna poprawa przepustowości w porównaniu z HTTP/1.1 (135 372 vs. 1 954 żądań na sekundę) potwierdza, że mechanizm multipleksowania strumieni stanowi fundamentalną optymalizację dla nowoczesnych aplikacji webowych. Dodatkowym aspektem wartym podkreślenia jest stosunkowo niewielki narzut wydajnościowy związany z warstwą TLS – degradacja rzędu 5–15% jest w pełni akceptowalna w kontekście krytycznych wymagań bezpieczeństwa współczesnych systemów.

W zakresie formatów serializacji MessagePack osiągnął najkrótszy całkowity czas przetwarzania (2,995 ms), podczas gdy Apache Avro zapewnił najlepszą kompresję danych (509,5 KB). Format JSON, mimo 2,2-krotnie większego rozmiaru względem formatów binarnych, pozostaje wartościowym wyborem ze względu na czytelność, debugowalność oraz natywne wsparcie w przeglądarkach. Z kolei XML, choć wykazał najgorsze metryki wydajnościowe (23,4 ms łącznie, 1721 KB), zachowuje znaczenie w środowiskach enterprise ze względu na wymagania regulacyjne oraz wsparcie dla złożonych schematów validacji.

Analiza technologii WebAssembly dostarczyła niuansowanego obrazu jej zastosowania w

aplikacjach webowych. Kluczowym wnioskiem jest, że złożoność obliczeniowa stanowi czynnik decydujący o opłacalności implementacji w WebAssembly. Podczas gdy proste operacje (grayscale, brightness, contrast) wykazały lepszą wydajność w JavaScript, złożone algorytmy – reprezentowane przez operację blur – osiągnęły 3,52-krotne przyspieszenie w WebAssembly (371,3 ms vs. 1308,5 ms). Ten wzorzec potwierdza, że WebAssembly znajduje optymalne zastosowanie w scenariuszach wymagających intensywnych obliczeń numerycznych, podczas gdy JavaScript pozostaje preferowany dla operacji o niskiej złożoności oraz częstej interakcji z DOM.

Wkład naukowy i praktyczny

Wartość niniejszej pracy przejawia się w kilku kluczowych aspektach. Po pierwsze, przeprowadzone badania opierają się na rzeczywistych implementacjach oraz kontrolowanych eksperymentach, eliminując abstrakcyjne założenia teoretyczne. Ujednolicona metodologia testowania (100 współbieżnych operacji, 1000 zapytań na wątek) zapewnia porównywalność wyników między różnymi protokołami i formatami.

Po drugie, praca dostarcza konkretnych, kwantyfikowalnych metryk wydajnościowych, które mogą stanowić bezpośrednie wsparcie w procesie podejmowania decyzji architektonicznych. Zamiast ogólnych stwierdzeń typu “protokół X jest szybszy od Y”, prezentowane są precyzyjne dane numeryczne wraz z analizą kontekstów, w których dana technologia wykazuje przewagę.

Po trzecie, sformułowane rekomendacje uwzględniają wielowymiarowy charakter wydajności systemu – nie ograniczają się wyłącznie do przepustowości, ale uwzględniają również aspekty takie jak skalowalność, kompatybilność wsteczna, developer experience oraz wymagania bezpieczeństwa i compliance.

Ograniczenia badań

Przy interpretacji wyników należy uwzględnić pewne ograniczenia przeprowadzonych badań. Testy wykonano w kontrolowanym środowisku testowym, które nie w pełni odzwierciedla złożoność środowisk produkcyjnych, charakteryzujących się zmiennym obciążeniem sieciowym, różnorodną infrastrukturą oraz nieprzewidywalnymi wzorcami użytkowania.

Dodatkowo, implementacje testowe zostały zoptymalizowane pod kątem maksymalnej wydajności dla poszczególnych protokołów, co może nie odzwierciedlać typowych wdrożeń produkcyjnych, w których priorytetem jest często równowaga między wydajnością a możliwością utrzymania kodu oraz zgodnością z istniejącą architekturą systemu.

Warto również zaznaczyć, że wyniki dotyczące WebAssembly odnoszą się do konkretnych implementacji algorytmów i mogą różnić się w zależności od charakterystyki przetwarzanych danych oraz specyfiki wykorzystywanych bibliotek.

Kierunki dalszych badań

Przeprowadzone badania otwierają kilka perspektywicznych kierunków dalszych eksploracji. Pierwszym z nich jest analiza wydajności protokołów w środowiskach rozproszonych charakteryzujących się wysokim opóźnieniem sieciowym oraz niestabilnymi połączeniami, co jest szczególnie istotne w kontekście aplikacji mobilnych oraz systemów IoT.

Interesującym kierunkiem jest również badanie wpływu nowych wersji protokołów – w szczególności HTTP/3 z protokołem QUIC – na wydajność komunikacji w aplikacjach webowych. HTTP/3, eliminujący problem Head-of-Line blocking na poziomie warstwy transportowej, może istotnie zmienić krajobraz wydajnościowy prezentowany w niniejszej pracy.

Wreszcie, w kontekście rosnącej popularności architektur serverless oraz edge computing, wartościowe byłoby zbadanie, jak charakterystyki wydajnościowe poszczególnych protokołów i formatów zmieniają się w środowiskach o dynamicznej skalowalności i geograficznie rozproszonych węzłach obliczeniowych.

Wnioski końcowe

Głównym przesłaniem niniejszej pracy jest przekonanie, że świadome i oparte na danych empirycznych decyzje architektoniczne stanowią fundament wydajnych systemów informatycznych. Wybór protokołu komunikacyjnego, formatu serializacji oraz technologii wykonawczej powinien być determinowany przez specyficzne wymagania aplikacji, uwzględniając takie czynniki jak charakterystyka przetwarzanych danych, oczekiwana latencja, wymagania bezpieczeństwa oraz ograniczenia infrastrukturalne.

Przeprowadzone badania potwierdzają, że nowoczesne aplikacje webowe mogą osiągać wysoką wydajność poprzez inteligentne wykorzystanie dostępnych technologii. REST API pozostaje doskonałym wyborem dla większości aplikacji webowych dzięki skalowalności i uniwersalności, WebSocket okazuje się optymalny dla komunikacji w czasie rzeczywistym, gRPC przewyższa konkurencję w komunikacji backend-to-backend, a WebAssembly stanowi wartościowe uzupełnienie JavaScript w scenariuszach wymagających intensywnych obliczeń.

Równocześnie praca podkreśla, że wydajność nie powinna być traktowana jako wyłączny czynnik decyzyjny – aspekty takie jak możliwość utrzymania kodu, ekosystem narzędzi, wsparcie społeczności oraz kompatybilność z istniejącymi systemami często przeważają nad przewagą wydajnościową rzędu kilku czy kilkunastu procent.

Podsumowując, niniejsza praca dostarcza praktycznych wskazówek oraz danych empirycznych, które mogą wspierać proces projektowania i optymalizacji nowoczesnych aplikacji webowych. Mam nadzieję, że przedstawione wyniki i rekomendacje okażą się wartościowe dla architektów systemów, programistów oraz badaczy zajmujących się wydajnością aplikacji internetowych.

Bibliografia

- [1] Anthropic. bun documentation. <https://bun.com//>. [dostęp: 2 luty 2026].
- [2] Git Project. Git. <https://git-scm.com/>. [dostęp: 21 stycznia 2026].
- [3] github. Juniper graphql server library for rust. <https://github.com/graphql-rust/juniper>. [dostęp: 2 luty 2026].
- [4] github. Tonic framework. <https://github.com/hyperium/tonic>. [dostęp: 2 luty 2026].
- [5] Google. Javascript v8 engine. <https://v8.dev/>. [dostęp: 2 luty 2026].
- [6] Google. Protocol buffers. <https://protobuf.dev/>. [dostęp: 2 luty 2026].
- [7] Microsoft. Typescript. <https://www.typescriptlang.org/>. [dostęp: 2 luty 2026].
- [8] Microsoft. Visual studio code. <https://code.visualstudio.com/>. [dostęp: 21 stycznia 2026].
- [9] MQTT.org. Message queuing telemetry transport. <https://mqtt.org/>. [dostęp: 2 luty 2026].
- [10] Open source. Prost. <https://github.com/tokio-rs/prost>. [dostęp: 2 luty 2026].
- [11] Open Source. quick-xml. <https://crates.io/crates/quick-xml/0.24.1>. [dostęp: 2 luty 2026].
- [12] Open source. Serde. <https://serde.rs/>. [dostęp: 2 luty 2026].
- [13] Ravi Teja. rumqttc. <https://crates.io/crates/rumqttc>. [dostęp: 2 luty 2026].
- [14] Rust Project. Cargo – the rust package manager. <https://doc.rust-lang.org/cargo/>. [dostęp: 21 stycznia 2026].
- [15] Rust Team. Rust language. <https://rust-lang.org/>. [dostęp: 2 luty 2026].
- [16] Snapview GmbH. tungstenite-rs. <https://github.com/snapview/tungstenite-rs>. [dostęp: 2 luty 2026].

- [17] Tatsuhiro Tsujikawa. h2load. <https://nghttp2.org/documentation/h2load-howto.html#>. [dostęp: 2 luty 2026].
- [18] @The Actix Team. Actix-web. <https://actix.rs/>. [dostęp: 2 luty 2026].
- [19] The Apache Software Foundation . Apache avro. <https://avro.apache.org/docs/>. [dostęp: 2 luty 2026].
- [20] The GraphQL Foundation. Graphql. <https://graphql.org/>. [dostęp: 2 luty 2026].
- [21] Webassembly.org. Webassembly. <https://webassembly.org/>. [dostęp: 21 stycznia 2026].
- [22] Wikipedia. Deserializacja. [https://en.wikipedia.org/wiki/Marshalling_\(computer_science\)#Unmarshalling](https://en.wikipedia.org/wiki/Marshalling_(computer_science)#Unmarshalling). [dostęp: 21 stycznia 2026].
- [23] Wikipedia. Extensible markup language. <https://pl.wikipedia.org/wiki/XML>. [dostęp: 2 luty 2026].
- [24] Wikipedia. google remote procedure call. <https://en.wikipedia.org/wiki/GRPC>. [dostęp: 21 stycznia 2026].
- [25] Wikipedia. Head-of-line-blocking. https://en.wikipedia.org/wiki/Head-of-line_blocking. [dostęp: 2 luty 2026].
- [26] Wikipedia. Hypertext transfer protocol. <https://pl.wikipedia.org/wiki/HTTP>. [dostęp: 21 stycznia 2026].
- [27] Wikipedia. Interfejs programowania aplikacji. https://pl.wikipedia.org/wiki/Interfejs_programowania_aplikacji. [dostęp: 21 stycznia 2026].
- [28] Wikipedia. Javascript. <https://pl.wikipedia.org/wiki/JavaScript>. [dostęp: 2 luty 2026].
- [29] Wikipedia. Model klient serwer. https://en.wikipedia.org/wiki/Client%E2%80%93server_model. [dostęp: 21 stycznia 2026].
- [30] Wikipedia. Python. [https://en.wikipedia.org/wiki/Python_\(programming_language\)](https://en.wikipedia.org/wiki/Python_(programming_language)). [dostęp: 2 luty 2026].
- [31] Wikipedia. Serializacja. <https://en.wikipedia.org/wiki/Serialization>. [dostęp: 21 stycznia 2026].
- [32] Wikipedia. Simple object access protocol. <https://en.wikipedia.org/wiki/SOAP>. [dostęp: 2 luty 2026].

- [33] Wikipedia. Strumieniowanie. [https://en.wikipedia.org/wiki/Stream_\(computing\)](https://en.wikipedia.org/wiki/Stream_(computing)). [dostęp: 2 luty 2026].
- [34] Wikipedia. Tls - transport layer security. https://pl.wikipedia.org/wiki/Transport_Layer_Security. [dostęp: 2 luty 2026].
- [35] Wikipedia. Websocket. <https://en.wikipedia.org/wiki/WebSocket>. [dostęp: 21 stycznia 2026].
- [36] Zed Industries. Zed editor. <https://zed.dev/>. [dostęp: 21 stycznia 2026].

Spis rysunków

2.1	Pozycja języka Rust w rankingu Tiobe Index.	10
2.2	Framework Actix-web w porównaniu z innymi frameworkami.	11
2.3	Pozycja języka Python w rankingu Tiobe Index.	13
4.1	Struktura plików i katalogów w aplikacji.	20
5.1	Porównanie przepustowości protokołów dla ładunku 1 KB i 1 MB (skala logarytmiczna).	44
5.2	Współczynnik degradacji wydajności przy zwiększeniu ładunku z 1 KB do 1 MB.	45
5.3	Metryki wydajności Apache Avro.	47
5.4	Metryki wydajności Binary JSON.	47
5.5	Metryki wydajności JSON.	48
5.6	Metryki wydajności MessagePack.	48
5.7	Metryki wydajności Protocol Buffers.	49
5.8	Metryki wydajności XML.	49
5.9	Porównanie czasu serializacji różnych formatów.	50
5.10	Porównanie czasu deserializacji formatowania danych.	51
5.11	Porównanie rozmiaru zserializowanych danych dla różnych formatów.	52

Spis listingów

4.1	Struktura danych wykorzystana w testach serializacji.	25
5.1	Implementacja zapytań dla ładunków 1KB oraz 1MB dla protokołu HTTP/1.1 oraz HTTP/2 dla architektury REST API	28
5.2	Implementacja strumienowania dla protokołu HTTP/1.1 oraz HTTP/2 dla ar- chitektury REST API	28
5.3	Implementacja symulacji problemu HOL dla protokołu HTTP/1.1 oraz HTTP/2 dla architektury REST API	29
5.4	Implementacja zapytania GraphQL	30
5.5	Zawartość pliku .proto, do którego tworzony jest automatycznie kod oraz typy .	32
5.6	Funkcje wykorzystywane w testach	32
5.7	Implementacja serwera WebSocket	33
5.8	Implementacja endpointów SOAP	37
5.9	Główna pętla zdarzeń odbiorcy MQTT	39
5.10	Implementacja klienta MQTT dla testów wydajnościowych	40
5.11	Implementacja Web Workera z prostymi operacjami obliczeniowymi	55
5.12	Implementacja Web Workera z zaawansowanymi operacjami obliczeniowymi . . .	56
5.13	Implementacja filtrów graficznych dla operacji Canvas	58