Автоматизовані системи управління та прилади автоматики. 2026. № 2 (189) *ISSN 3083-7715 (online)*
Management Information System and Devices. 2026. No. 2 (189) *ISSN 3083-7650 (0135-1710) (print)*

УДК 004.4:004.65:004.75 DOI: https://doi.org/10.30837/0135-1710.2026.189.095

Бобровнікова К. Ю., Гурман І. В., Олексюк Д. А.

ПОРІВНЯЛЬНИЙ АНАЛІЗ МІЖСЕРВІСНОЇ ВЗАЄМОДІЇ З ВИКОРИСТАННЯМ REST І GRPC У СЕРЕДОВИЩІ .NET 8

Предметом дослідження є підходи до реалізації синхронної міжсервісної взаємодії в мікросервісній архітектурі із застосуванням REST і gRPC у середовищі .NET 8. **Мета роботи** – провести порівняльний аналіз REST API на базі HTTP/1.1 і JSON і gRPC-сервісу на основі HTTP/2 і Protocol Buffers, а також оцінити їх вплив на продуктивність, обсяг мережевого обміну й використання серверних ресурсів. З огляду на окреслену мету необхідно виконати такі **завдання:** проаналізувати архітектурні розбіжності REST і gRPC; реалізувати два сервіси з однаковою прикладною логікою в середовищі ASP.NET Core 8; забезпечити однакові умови виконання експерименту; виміряти середню затримку відповіді, P95, кількість успішних операцій за секунду, обсяг переданих даних, а також використання CPU і RAM; інтерпретувати досягнуті результати, зважаючи на межі застосування кожного підходу. **Методи дослідження.** Упроваджено порівняльний аналіз архітектурних підходів, експериментальне навантажувальне тестування й кількісне оцінювання продуктивності. Реалізовано два сервіси з ідентичною прикладною логікою та спільним in-memory набором даних без звернення до зовнішньої бази даних. Навантаження генерувалося за допомогою Grafana k6 у режимі constant-arrival-rate. **Результати дослідження.** У межах експерименту gRPC на HTTP/2 з Protocol Buffers продемонстрував нижчі значення середньої затримки та P95, менший обсяг мережевого обміну й вищу пропускну здатність, якщо порівнювати з REST API на HTTP/1.1 з JSON. Загальний обсяг обміну для gRPC виявився приблизно у 2,5 раза меншим, а на верхніх рівнях навантаження середня затримка була нижчою орієнтовно в 1,7–1,8 раза. Крім того, для gRPC зафіксовано більш помірне використання процесорних і пам'яттєвих ресурсів серверного вузла. **Висновки.** Досягнуті результати підтверджують доцільність застосування gRPC насамперед для внутрішньої міжсервісної взаємодії, де критичними є низькі затримки, висока пропускна здатність і ефективне використання мережевих ресурсів. REST доцільно застосовувати в сценаріях, де пріоритетними є сумісність, простота інтеграції, прозорість HTTP-інтерфейсу й зручність використання в публічних API.

Ключові слова: мікросервісна архітектура; міжсервісна взаємодія; розподілені системи; продуктивність; затримка відповіді; пропускна здатність.

Вступ

Сучасний етап розвитку програмної інженерії визначається переходом від монолітних архітектур до розподілених систем, зокрема мікросервісних підходів, які забезпечують гнучкість, масштабованість і незалежність розгортання окремих компонентів. Така архітектурна парадигма дає змогу підвищити адаптивність програмних систем до змін вимог і навантаження, однак водночас суттєво ускладнює організацію взаємодії між окремими сервісами. У мікросервісній архітектурі міжсервісна взаємодія є одним із ключових факторів, що визначає ефективність функціювання системи загалом. Вона безпосередньо впливає на продуктивність, затримку оброблення запитів, надійність передачі даних, а також складність розроблення й супроводу програмного забезпечення. Неефективно спроєктовані механізми взаємодії можуть призводити до виникнення вузьких місць, надмірного мережевого навантаження та зниження масштабованості системи. З огляду на це вибір механізму міжсервісної комунікації доцільно розглядати не лише як питання реалізації API, а як архітектурне рішення, що залежить від зрозумілості інтерфейсу, семантики HTTP, можливостей транспортного рівня та витрат на подання повідомлень.

На практиці для реалізації міжсервісної взаємодії найчастіше застосовуються REST-орієнтовані підходи й протокол gRPC. REST, який у прикладних системах зазвичай реалізується поверх HTTP із використанням ресурсно-орієнтованого підходу й форматів на кшталт JSON, набув значного поширення завдяки простоті реалізації, універсальності та сумісності з різними клієнтськими платформами. Водночас gRPC, що базується на HTTP/2 і за замовчуванням використовує Protocol Buffers як формат серіалізації повідомлень, забезпечує вищу продуктивність, підтримку стримінгових сценаріїв і зменшення накладних витрат на передачу інформації.

Платформа .NET 8 і ASP.NET Core 8 надають сучасні засоби для побудови високопродуктивних

ISSN 3083-7715 (online) Автоматизовані системи управління та прилади автоматики. 2026. № 2 (189)
ISSN 3083-7650 (0135-1710) (print) Management Information System and Devices. 2026. No. 2 (189)

розподілених систем, зокрема підтримку HTTP/2, gRPC і вдосконалення продуктивності вебзастосунків. Подібні можливості для реалізації міжсервісної взаємодії на основі REST, HTTP/2 та gRPC надають також інші сучасні технологічні платформи, зокрема Java/Spring Boot, Quarkus, Go й Node.js. Водночас у межах цієї роботи обрано .NET 8 і ASP.NET Core 8, оскільки вони дають змогу реалізувати REST API і gRPC-сервіси в єдиному технологічному середовищі та зменшити вплив розбіжностей між runtime-платформами, бібліотеками й фреймворками на результати порівняння.

Незважаючи на наявність численних досліджень у сфері мікросервісних архітектур, питання обґрунтованого вибору підходу до міжсервісної взаємодії з огляду на сучасні інструменти розроблення залишається відкритим. Зокрема недостатньо досліджено співвідношення між продуктивністю, складністю реалізації та експлуатаційними характеристиками REST і gRPC у середовищі .NET 8. Отже, науково-практична проблема полягає у визначенні доцільності застосування REST і gRPC для реалізації міжсервісної взаємодії в мікросервісній архітектурі, зважаючи на їх вплив на продуктивність, ефективність передачі інформації та складність розроблення в середовищі .NET 8.

Аналіз сучасних наукових публікацій і визначення проблеми дослідження

Проблематика мікросервісної архітектури в сучасних дослідженнях розглядається передусім крізь призму складності розподіленого проєктування. У працях [1, 2] наголошено, що мікросервіси дають змогу підвищити незалежність розроблення, розгортання й масштабування окремих компонентів, однак водночас ускладнюють визначення меж сервісів, керування контрактами, моніторинг і забезпечення стійкості системи. Отже, ефективність мікросервісної архітектури залежить не лише від декомпозиції, а й від якості взаємодії між її компонентами.

Окремий напрям досліджень пов’язаний із міжсервісною комунікацією як чинника продуктивності та експлуатаційної складності. У роботі [3] доведено, що спосіб взаємодії між сервісами впливає на затримки, пропускну здатність і використання ресурсів. Це дає підстави розглядати комунікаційний механізм не як другорядну деталь реалізації, а як архітектурне рішення, що потребує кількісного оцінювання.

Значну частину публікацій присвячено якості та зрозумілості API. Зокрема в дослідженні [4] доведено, що дотримання правил RESTful-проєктування впливає на сприйняття й коректне використання Web API-розробниками. У публікаціях [5–7] сформовано технічну основу для аналізу HTTP-взаємодії, а в роботі [8] зосереджено увагу на тому, що формат серіалізації даних може суттєво впливати на обсяг повідомлень і ефективність комунікації. Отже, у науковій літературі формується підхід, за якого API розглядається не лише з погляду зручності, а й з позиції витрат взаємодії.

Порівнянню комунікаційних технологій у мікросервісних системах присвячені праці [9–12]. У них проаналізовано REST, gRPC та інші підходи з огляду на продуктивність, масштабованість, ресурсні характеристики й практичні межі застосування HTTP API та gRPC. Водночас такі результати істотно залежать від середовища виконання, структури повідомлень, сценарію навантаження й обраних метрик, тому не можуть автоматично переноситися на інші технологічні платформи.

Окремі дослідження, дотичні до окресленої проблеми, розглядають не самі протоколи взаємодії, а умови стабільної роботи сервісів у розподілених середовищах. У статті [13] увагу приділено відновленню критичних сервісів в умовах керованої деградації, що є важливим для ширшого розуміння стійкості сервісно-орієнтованих систем. У праці [14] досліджено прогнозування використання ресурсів у хмарній інфраструктурі, а публікацію [15] присвячено моніторингу й балансуванню навантаження в мікросервісних застосунках. Згадані праці важливі для обґрунтування того, що оцінювання міжсервісної взаємодії має передбачати не лише час відповіді, а й ресурсні й експлуатаційні показники.

Отже, аналіз джерел демонструє, що сучасні дослідження охоплюють загальні принципи мікросервісної архітектури, якість API, транспортні й серіалізаційні чинники, а також експериментальні порівняння REST і gRPC. Водночас залишається необхідність у прикладному порівнянні цих підходів в одному технологічному середовищі, за однакової прикладної логіки й контрольованого навантаження.

Метою роботи є проведення порівняльного аналізу підходів REST і gRPC до реалізації міжсервісної взаємодії в мікросервісній архітектурі, дослідження їх впливу на продуктивність і ефективність передачі даних, а також

Автоматизовані системи управління та прилади автоматики. 2026. № 2 (189) *ISSN 3083-7715 (online)*
Management Information System and Devices. 2026. No. 2 (189) *ISSN 3083-7650 (0135-1710) (print)*

обґрунтування рекомендацій щодо їх застосування в середовищі .NET 8.

Матеріали й методи дослідження

Архітектура міжсервісної взаємодії в мікросервісній системі

Мікросервісна архітектура передбачає побудову програмної системи у вигляді сукупності незалежних сервісів, кожен з яких реалізує окрему бізнес-функцію та взаємодіє з іншими сервісами через мережеві інтерфейси. Такий підхід сприяє масштабованості та гнучкості системи, однак висуває підвищені вимоги до організації міжсервісної взаємодії [1, 2]. Загалом взаємодія між мікросервісами може здійснюватися за двома основними моделями: синхронною та асинхронною. Синхронна взаємодія передбачає безпосередній виклик одного сервісу іншим з очікуванням відповіді, що властиво для REST і gRPC. Асинхронна взаємодія реалізується за допомогою брокерів повідомлень і впроваджується для зменшення зв'язаності сервісів і підвищення відмовостійкості системи [1–3].

У межах синхронної моделі ключовим аспектом є не лише вибір конкретного протоколу, а й визначення способу формалізації контракту між сервісами. REST-орієнтований підхід спирається на ресурсну модель, стандартні HTTP-методи й текстове подання інформації, що робить інтерфейси прозорими для розробників, зручними для тестування й сумісними з широким колом клієнтських застосунків. Водночас така гнучкість часто супроводжується більш слабкою формалізацією контрактів, потребою в окремому описі API та вищими накладними витратами під час передавання структурованих JSON-повідомлень. Підхід gRPC, навпаки, орієнтований на контрактно визначені віддалені виклики процедур, де структура сервісів і повідомлень задається у .proto-файлах, а клієнтський і серверний код можуть генеруватися автоматично. Це підвищує типобезпечність взаємодії, спрощує узгодження інтерфейсів між сервісами та зменшує ризик помилок, пов'язаних із некоректною інтерпретацією повідомлень. Крім того, використання HTTP/2 і двійкового формату Protocol Buffers створює передумови для зменшення обсягу переданих даних, кращого використання з'єднання й більш стабільної поведінки сервісів за інтенсивної внутрішньої комунікації [6, 8, 10, 11].

У сучасних системах, реалізованих на платформі .NET 8, обидва підходи підтримуються на рівні фреймворку ASP.NET Core. Це дає змогу реалізовувати як REST API, так і gRPC-сервіси в межах єдиного технологічного стеку, що спрощує їх порівняння та інтеграцію.

Подана на рис. 1 схема відтворює гібридний підхід до організації синхронної взаємодії в мікросервісній системі, за якого зовнішній і внутрішній контури комунікації розглядаються як різні відповідно до вимог архітектурні зони.

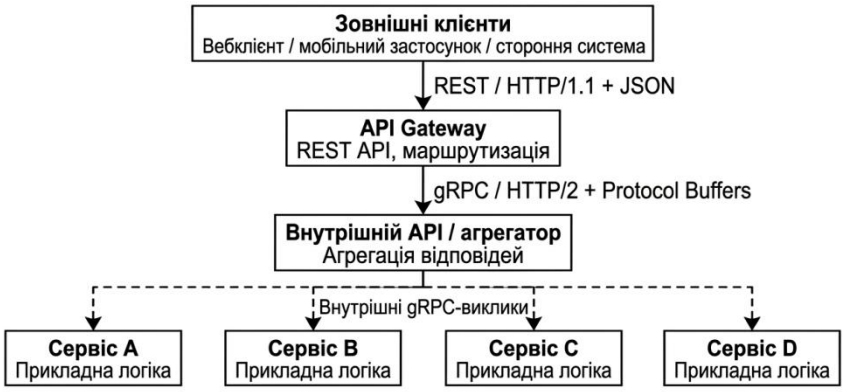

Рис. 1. Архітектура міжсервісної взаємодії в мікросервісній системі з використанням REST і gRPC

Зовнішній контур орієнтований на взаємодію з вебклієнтами, мобільними застосунками та сторонніми системами, тому для нього пріоритетними є сумісність, простота інтеграції, прозорість HTTP-інтерфейсу й зручність тестування. Внутрішній контур, навпаки, працює в більш контрольованому середовищі, де склад клієнтів і серверів відомий наперед, а основними вимогами стають мінімізація затримок, зменшення обсягу службових даних, типобезпечність контрактів і стабільність поведінки під

ISSN 3083-7715 (online) Автоматизовані системи управління та прилади автоматики. 2026. № 2 (189)
ISSN 3083-7650 (0135-1710) (print) Management Information System and Devices. 2026. No. 2 (189)

навантаженням. У такій архітектурі API Gateway виконує роль межового компонента, який ізолює зовнішніх клієнтів від внутрішньої структури сервісів, спрощує маршрутизацію запитів і дає змогу незалежно еволюціонувати зовнішні та внутрішні інтерфейси. Завдяки цьому REST може залишатися зручним механізмом публічної взаємодії, тоді як gRPC доцільно застосовувати для внутрішніх викликів між сервісами, де переваги контрактної моделі, HTTP/2 та Protocol Buffers мають найбільше практичне значення.

Реалізація REST-взаємодії в середовищі .NET 8

REST-підхід залишається одним із найпоширеніших способів організації синхронної взаємодії в розподілених і мікросервісних системах завдяки застосуванню стандартних можливостей HTTP, ресурсно-орієнтованої моделі та високої сумісності з різними клієнтськими платформами [4, 5, 10]. У середовищі .NET 8 реалізація REST-сервісів здійснюється засобами ASP.NET Core, що підтримує побудову HTTP API на основі контролерів або Minimal API, маршрутизацію запитів, серіалізацію JSON, ін'єкцію залежностей, middleware-конвеєр, а також механізми автентифікації, авторизації та оброблення помилок. Це сприяє створенню міжсервісних API в межах єдиного технологічного стеку без залучення додаткових фреймворків для базової HTTP-взаємодії.

У мікросервісній архітектурі REST-сервіс зазвичай надає набір кінцевих точок, які відповідають окремим ресурсам або прикладним операціям. Запити між сервісами передаються за допомогою HTTP-методів, а дані найчастіше подаються у форматі JSON. Така модель є зручною для зовнішніх інтеграцій, публічних API та систем, у яких важливими є прозорість інтерфейсу, простота налагодження, можливість тестування запитів стандартними інструментами й підтримка широкого кола клієнтів. У практичній реалізації ASP.NET Core це може бути реалізовано як через класичні контролери, так і Minimal API, що дає змогу скоротити обсяг службового коду для невеликих сервісів або експериментальних реалізацій.

Водночас використання REST у внутрішній міжсервісній взаємодії має низку особливостей, які необхідно брати до уваги під час проєктування. По-перше, контракт REST API часто є менш формалізованим, ніж контракт gRPC, оскільки структура запитів і відповідей може описуватися окремо, наприклад за допомогою OpenAPI/Swagger, але не завжди безпосередньо застосовується для генерації типобезпечного клієнтського коду. По-друге, текстова природа JSON підвищує зручність читання й діагностики повідомлень, але водночас збільшує обсяг переданих даних і витрати на серіалізацію та десеріалізацію [8]. По-третє, у разі значної кількості внутрішніх синхронних викликів важливими стають повторне використання HTTP-з'єднань, коректне налаштування клієнтських викликів, тайм-аутів, оброблення помилок і контролю каскадних відмов.

Окремим аспектом є еволюція REST API. У мікросервісних системах інтерфейси сервісів змінюються незалежно, тому важливими стають версіювання API, зворотна сумісність контрактів, уніфікована структура помилок, узгоджені коди HTTP-відповідей і спостережуваність запитів. REST добре підтримує інтеграційні сценарії, у яких клієнти можуть бути різнорідними й незалежними від внутрішньої реалізації сервісу. Проте саме ця гнучкість у внутрішніх високонавантажених взаємодіях може перетворюватися на додатковий інженерний тягар, оскільки вимагає окремого контролю контрактів, схем повідомлень, політик повторення запитів і механізмів трасування.

Отже, REST у середовищі .NET 8 є зручним і технологічно зрілим механізмом побудови HTTP API, особливо для публічних інтерфейсів, зовнішніх інтеграцій і сценаріїв, де пріоритетними є простота доступу, сумісність і прозорість обміну. Водночас для інтенсивної внутрішньої міжсервісної взаємодії його ефективність значною мірою залежить від формату даних, версії HTTP, налаштування клієнтських під'єднань і характеру навантаження.

Реалізація gRPC-взаємодії в середовищі .NET 8

Сервіс gRPC є сучасним підходом до реалізації синхронної міжсервісної взаємодії, який базується на моделі віддаленого виклику процедур і орієнтований на побудову продуктивних, контрактно визначених інтерфейсів у розподілених системах [6, 10, 11]. На відміну від REST, де взаємодія зазвичай організовується навколо ресурсів і HTTP-методів, у gRPC основою є сервісний контракт, описаний у .proto-файлі. У цьому контракті визначаються методи сервісу, типи запитів і відповідей, а також структура повідомлень. Такий підхід забезпечує формалізовану модель взаємодії між сервісами

Автоматизовані системи управління та прилади автоматики. 2026. № 2 (189) *ISSN 3083-7715 (online)*
Management Information System and Devices. 2026. No. 2 (189) *ISSN 3083-7650 (0135-1710) (print)*

та зменшує неоднозначність під час інтерпретації переданих даних.

У середовищі .NET 8 реалізація gRPC-сервісів здійснюється засобами ASP.NET Core gRPC. На основі .proto-контракту автоматично генеруються серверні базові класи й клієнтські проксі, що дає змогу працювати з віддаленими викликами як із типізованими методами [10]. Це спрощує супровід міжсервісних інтерфейсів, оскільки помилки невідповідності типів або структури повідомлень можуть бути виявлені ще на етапі компіляції. Для мікросервісної архітектури така властивість є важливою, оскільки зміни контрактів між сервісами мають контролюватися особливо ретельно.

Ключовою технічною особливістю gRPC є використання Protocol Buffers як основного формату серіалізації. На відміну від JSON, який є текстовим і зручним для читання людиною, Protocol Buffers застосовує компактне двійкове подання повідомлень. Це сприяє зменшенню обсягів переданої інформації та витрат на оброблення повідомлень у сценаріях інтенсивної міжсервісної взаємодії. Крім того, gRPC працює поверх HTTP/2, що забезпечує мультиплексування кількох запитів у межах одного з'єднання, більш ефективне використання мережевого каналу й підтримку потокових моделей взаємодії [6, 10].

Практична реалізація gRPC у .NET 8 передбачає не лише опис сервісу в .proto-файлі, а й правильну організацію життєвого циклу викликів. Для внутрішніх сервісних комунікацій важливими є повторне використання gRPC-каналів, контроль часу очікування відповіді, підтримка скасування запитів, передавання службових метаданих і коректне оброблення статусів помилок. На відміну від REST, де помилки зазвичай інтерпретуються через HTTP-коди відповіді, gRPC застосовує власну модель статусів, що дає змогу уніфіковано описувати результати віддалених викликів у різних мовних реалізаціях.

Важливою перевагою gRPC є підтримка кількох моделей взаємодії: унарних викликів, серверного стримінгу, клієнтського стримінгу та двоспрямованого стримінгу. Унарні виклики найближчі до традиційної моделі "запит – відповідь" і ефективні для порівняння з REST API. Стримінгові режими є доцільними для сценаріїв, де потрібно передавати послідовності повідомлень без постійного створення нових запитів, наприклад, для телеметрії, оброблення подій або потокового передавання даних. У межах цієї роботи основну увагу приділено саме унарним викликам, оскільки вони дають змогу коректніше зіставити gRPC із REST API в однаковій прикладній операції.

Крім переваг, gRPC має й певні обмеження. Його використання є найбільш природним у внутрішньому контурі мікросервісної системи, де можна контролювати версії сервісів, клієнтів, контрактів і транспортного середовища. Для безпосередньої взаємодії з браузерними клієнтами gRPC менш зручний, оскільки стандартні браузерні API не забезпечують повної підтримки звичайного gRPC без додаткових механізмів, як-от gRPC-Web або JSON transcoding [10]. Крім того, контрактна модель .proto вимагає узгодженого підходу до еволюції API: зміни повідомлень мають бути зворотно сумісними, а поля – додаватися й вилучатися з огляду на правила версіювання Protocol Buffers.

Отже, gRPC у середовищі .NET 8 є ефективним механізмом реалізації внутрішньої міжсервісної взаємодії, коли важливими є продуктивність, типобезпечність, компактність повідомлень і стабільність контрактів. Його переваги найбільш помітні в сценаріях інтенсивного обміну інформацією між сервісами, тоді як для публічних API та інтеграції з невідомими клієнтами REST часто залишається більш простим і універсальним рішенням. Саме тому в мікросервісній архітектурі доцільним є не протиставлення REST і gRPC як несумісних технологій, а їх комбіноване впровадження відповідно до межі системи, типу клієнтів і вимог до продуктивності.

Порівняльний аналіз REST і gRPC у мікросервісній архітектурі

Порівняльний аналіз REST і gRPC доцільно проводити не лише з позиції швидкодії, а й з огляду на архітектурну роль кожного підходу в мікросервісній системі. У практичних проєктах ці технології часто розв'язують різні класи завдань: REST переважно використовується як універсальний механізм публічної або міжсистемної інтеграції, тоді як gRPC частіше застосовується у внутрішньому контурі системи, де склад сервісів, клієнтів і контрактів контролюється розробниками. Тому коректне порівняння має зважати не лише на середню затримку чи пропускну здатність, а й на спосіб опису контрактів, модель еволюції API, накладні витрати серіалізації, зручність супроводу й вимоги до клієнтського середовища [9–12].

ISSN 3083-7715 (online) Автоматизовані системи управління та прилади автоматики. 2026. № 2 (189)
ISSN 3083-7650 (0135-1710) (print) Management Information System and Devices. 2026. No. 2 (189)

З погляду архітектурної моделі REST і gRPC мають різну логіку організації взаємодії. REST спирається на ресурсно-орієнтований підхід, у межах якого операції над даними реалізуються за допомогою стандартних HTTP-методів і кінцевих точок API. Така модель добре узгоджується з публічними інтерфейсами, інтеграцією з браузерними клієнтами, сторонніми системами та інструментами тестування. Натомість gRPC орієнтований на віддалений виклик процедур, де основною одиницею взаємодії є не ресурс, а метод сервісу, явно описаний у .proto-контракті. Це робить gRPC ближчим до внутрішньої сервісної взаємодії, у якій важливими є типобезпечність, передбачуваність структури повідомлень і можливість автоматичної генерації клієнтського коду.

Важливою особливістю є спосіб формалізації контрактів. REST API може описуватися за допомогою документації або специфікацій OpenAPI, однак у багатьох реалізаціях контракт існує окремо від клієнтського коду й потребує додаткового контролю узгодженості. Це забезпечує гнучкість, але водночас створює ризик розбіжностей між фактичною поведінкою сервісу й очікуваннями клієнтів. У gRPC контракт є центральним елементом розроблення: структура сервісів і повідомлень визначається у .proto-файлі, на основі якого генеруються серверні та клієнтські компоненти. Завдяки цьому частина помилок інтеграції може виявлятися ще на етапі компіляції, що є суттєвою перевагою для систем із значною кількістю внутрішніх сервісних викликів.

Щодо продуктивності, то gRPC має технічні передумови для кращих результатів у високонавантажених внутрішніх взаємодіях. До таких передумов належать компактне двійкове подання інформації в Protocol Buffers, використання HTTP/2, мультиплексування запитів у межах одного з'єднання й більш ефективна робота з повторюваними службовими даними. REST API, особливо у варіанті HTTP/1.1 + JSON, зазвичай формує більший обсяг повідомлень і потребує додаткових витрат на оброблення текстового формату. Водночас це не означає, що REST є "повільним" у загальному сенсі: за невисокого навантаження, невеликих повідомлень або використання HTTP/2 різниця між підходами може бути менш помітною. Тому твердження про перевагу gRPC варто пов'язувати з конкретною постановкою експерименту, типом повідомлень і профілем навантаження.

З позиції експлуатації REST має перевагу в прозорості та простоті діагностики. HTTP-запити легко перевіряти стандартними інструментами, JSON-повідомлення є читабельними для людини, а коди HTTP-відповідей добре відомі розробникам і адміністраторам. Це спрощує налагодження, інтеграційне тестування й підтримку публічних API. Сервіс gRPC, навпаки, використовує двійковий формат повідомлень і власну модель статусів, тому потребує спеціалізованих інструментів для перегляду запитів, тестування сервісів і трасування взаємодії. Водночас у контрольованому внутрішньому середовищі ця складність компенсується чіткістю контрактів, нижчими мережевими накладними витратами та кращою придатністю до інтенсивних сервісних викликів.

Окремо необхідно брати до уваги питання еволюції API. REST забезпечує високу гнучкість зміни інтерфейсів, але вимагає ретельного керування версіями, сумісністю форматів відповідей і поведінкою клієнтів. У gRPC еволюція API тісно пов'язана з правилами зміни .proto-контрактів: нові поля мають додаватися без порушення сумісності, а вилучення або зміна наявних полів потребує особливої обережності. Отже, REST є більш зручним там, де клієнти різнорідні й не завжди контрольовані розробником сервісу, тоді як gRPC краще підходить для середовищ, де команди можуть централізовано керувати контрактами й оновленням клієнтських бібліотек.

З огляду на наведені розбіжності REST і gRPC доцільно розглядати не як несумісні технології, а як інструменти для різних архітектурних зон мікросервісної системи. REST є природним вибором для зовнішнього контуру, публічних API, інтеграцій із вебклієнтами та сторонніми системами. Підхід gRPC зі свого боку є доцільним для внутрішньої міжсервісної взаємодії, де важливими є низькі затримки, компактність повідомлень, типобезпечність контрактів і стабільна поведінка під навантаженням. Узагальнення ключових характеристик цих підходів подано в табл. 1.

Доцільність такого розмежування підтверджується також сучасними дослідженнями продуктивності комунікаційних технологій у мікросервісних системах. Зокрема в роботах [9, 11, 12] продемонстровано, що вибір комунікаційної технології може істотно впливати на затримки, пропускну здатність, масштабованість і використання ресурсів. У цьому разі gRPC зазвичай демонструє

Автоматизовані системи управління та прилади автоматики. 2026. № 2 (189) *ISSN 3083-7715 (online)*
Management Information System and Devices. 2026. No. 2 (189) *ISSN 3083-7650 (0135-1710) (print)*

кращі результати у внутрішніх сервісних викликах і сценаріях інтенсивного обміну даними, тоді як REST зберігає переваги в простоті інтеграції та сумісності з різнорідними клієнтами.

Таблиця 1. *Загальна порівняльна характеристика REST і gRPC*

Критерій	REST	gRPC
Протокол	HTTP/1.1 або HTTP/2	HTTP/2
Формат даних	JSON, XML та інші текстові формати	Protocol Buffers за замовчуванням
Продуктивність	Помірна, залежить від реалізації	Вища у внутрішніх RPC-сценаріях
Затримки	Зазвичай вищі в сценаріях інтенсивного JSON-обміну	Зазвичай нижчі у внутрішніх RPC-викликах
Контракти	Необов'язкові	Обов'язкові (.proto)
Складність реалізації	Нижча	Вища на початковому етапі
Стримінг	Можливий через SSE, WebSockets або chunked responses, але не є нативною моделлю класичного REST API	Нативна підтримка client, server та bi-directional streaming
Сумісність	Висока	Обмежена в браузерах без додаткових засобів
Основне застосування	Публічні API, зовнішня інтеграція	Внутрішня взаємодія між сервісами

Отже, результати порівняльного аналізу свідчать, що REST і gRPC не є несумісними підходами, а доповнюють один одного. REST доцільно застосовувати там, де важливими є універсальність і сумісність, тоді як gRPC є більш ефективним для внутрішньої міжсервісної взаємодії у високонавантажених системах.

Експериментальне дослідження ефективності REST і gRPC

Випробування проводилося в локальній мережі 1 Гбіт/с на обчислювальному вузлі з процесором Intel Core i7-12700, 32 GB оперативної пам'яті та SSD-накопичувачем. Як серверне середовище використовувалася Linux-платформа, на якій REST- і gRPC-сервіси були розгорнуті в однакових контейнерах з обмеженням 2 vCPU та 2 GB RAM для кожної реалізації. Навантажувальний клієнт Grafana k6 запускався в тому самому мережевому сегменті, що дало змогу мінімізувати вплив зовнішніх мережевих чинників і зосередити вимірювання на розбіжностях транспортного рівня, серіалізації та оброблення запитів.

Для порівняння REST API на HTTP/1.1 з JSON і gRPC на HTTP/2 з Protocol Buffers було реалізовано два окремі серверні застосунки з однаковою прикладною логікою. Така постановка відповідає типовій схемі зіставлення HTTP API з JSON і gRPC, де gRPC спирається на .proto-контракт, двійкове кодування повідомлень і HTTP/2 [6, 8, 10, 11]. Водночас необхідно зважати, що HTTP API з JSON також можуть працювати поверх HTTP/2, тому в цьому дослідженні використання HTTP/1.1 для REST було свідомим елементом постановки експерименту, а не універсальною властивістю REST як архітектурного стилю [5–7, 10].

У межах експерименту обидва сервіси реалізовували одну й ту саму операцію GetTelemetryProfile. Запит містив ідентифікатор об'єкта, часовий інтервал, прапорець увімкнення агрегованої статистики й параметр локалізації. Відповідь повертала структурований об'єкт середнього розміру: 14 скалярних полів, вкладений блок стану, масив із 5–7 коротких рядкових міток і блок агрегованих показників. Такий формат обрано свідомо: він достатньо простий, щоб не впроваджувати складну бізнес-логіку, але водночас достатньо репрезентативний за структурою, щоб різниця між JSON і Protocol Buffers була не лише теоретичною, а й у мережевому навантаженні та затримках. Щоб уникнути впливу сторонніх чинників, сервери не зверталися до зовнішньої бази даних. Дані поверталися з попередньо підготовленого in-memory набору, ідентичного для обох реалізацій. Це дало змогу зосередити вимірювання на накладних витратах транспорту, серіалізації, десеріалізації та оброблення HTTP-з'єднань. Для обох сервісів було вимкнено стиснення тіла відповіді, щоб не змішувати ефект протоколу з ефектом компресії. REST-сервіс працював через HTTP/1.1 з keep-alive, а gRPC-сервіс – через HTTP/2.

Обидві реалізації створювалися на ASP.NET Core 8: для REST – Minimal API з System.Text.Json, для gRPC – стандартний стек ASP.NET Core gRPC з кодом, згенерованим із .proto-контракту. Тестування виконувалося в режимі Release; для обох сервісів використовувався один і той самий runtime .NET 8. Така технологічна база є зручною для відтворюваного

ISSN 3083-7715 (online) Автоматизовані системи управління та прилади автоматики. 2026. № 2 (189)
ISSN 3083-7650 (0135-1710) (print) Management Information System and Devices. 2026. No. 2 (189)

порівняння, оскільки одна платформа сприяє зміні лише транспортного й серіалізаційного рівня, не змішуючи ефекти різних runtime-середовищ. У навантажувальному клієнті для REST-сценарію застосовувався стандартний HTTP-клієнт Grafana k6, а для gRPC – модуль k6/net/grpc, який підтримує унарні gRPC-виклики. Режим навантаження задавався за допомогою constant-arrival-rate, тобто як відкрита модель із фіксованою інтенсивністю запуску ітерацій, незалежною від часу відповіді системи. Успішною операцією вважався запит, що завершувався без помилки транспортного рівня й повертав коректну відповідь сервісу в межах вимірювального інтервалу. Для сценаріїв constant-arrival-rate у Grafana k6 цільова інтенсивність задавалася відповідно до кожної точки навантаження, а значення preAllocatedVUs і maxVUs підбиралися так, щоб генератор навантаження не ставав обмежувальним чинником експерименту: для навантаження 100–500 RPS використовувалося preAllocatedVUs = 50, maxVUs = 200, а для 700–1100 RPS – preAllocatedVUs = 150, maxVUs = 400. TLS-шифрування не застосовувалося, оскільки метою експерименту було порівняння накладних витрат транспорту й серіалізації без додаткового впливу криптографічного оброблення. Під час тестування було вимкнено надлишкове прикладне логування; обидві реалізації запускалися в однакових умовах контейнерного середовища.

Усі наведені нижче числові результати отримано в межах описаного експериментального стенда. Дослідження було виконано серіями за умови навантаження 100, 300, 500, 700, 900 і 1100 запитів за секунду. Для кожної точки проводився 30-секундний прогрів, після чого фіксувався 60-секундний інтервал вимірювання. Кожний сценарій повторювався п’ять разів, а до таблиць заносилися усереднені значення. Основними метриками були: середня затримка відповіді, P95, кількість успішних операцій за секунду, середній ефективний обсяг переданих даних на один виклик, а також середнє використання CPU і RAM на серверному боці. CPU і RAM фіксувалися раз за секунду засобами моніторингу контейнерів із подальшим усередненням за інтервал вимірювання.

Затримка й пропускна здатність

У табл. 2 наведено результати для середньої затримки, P95 і кількості успішних операцій за секунду.

Таблиця 2. *Результати вимірювання затримки й пропускної здатності*

Навантаження, RPS	REST			gRPC		
	середня затримка відповіді, мс	P95, мс	успішних оп./с	середня затримка відповіді, мс	P95, мс	успішних оп./с
100	13	22	100	9	14	100
300	19	33	298	12	19	300
500	29	48	492	18	27	498
700	42	71	669	25	39	692
900	68	118	824	38	59	887
1100	111	192	861	61	96	1058

Для наочного подання результатів, що містяться в табл. 2, побудовано діаграми зміни середньої затримки, P95 і кількості успішних операцій за секунду залежно від рівня навантаження. Динаміку середньої затримки для REST і gRPC наведено на рис. 2, значення P95 – на рис. 3, а зміну пропускної здатності – на рис. 4.

Як видно з табл. 2 та рис. 2–4, до рівня приблизно 500 RPS обидва підходи працюють стабільно, однак реалізація gRPC на HTTP/2 з Protocol Buffers уже на цьому етапі демонструє нижчі значення середньої затримки й вужчий "хвіст" розподілу, відображений через P95. Після 700 RPS різниця між підходами стає більш помітною: REST швидше входить у зону деградації, де приріст цільового навантаження вже не перетворюється на пропорційний приріст кількості успішно оброблених запитів. Для gRPC ця межа настає за вищого рівня навантаження, що є очікуваним для стеку, який використовує більш компактні двійкові повідомлення та HTTP/2-механізми мультиплексування.

Автоматизовані системи управління та прилади автоматики. 2026. № 2 (189) *ISSN 3083-7715 (online)*
Management Information System and Devices. 2026. No. 2 (189) *ISSN 3083-7650 (0135-1710) (print)*

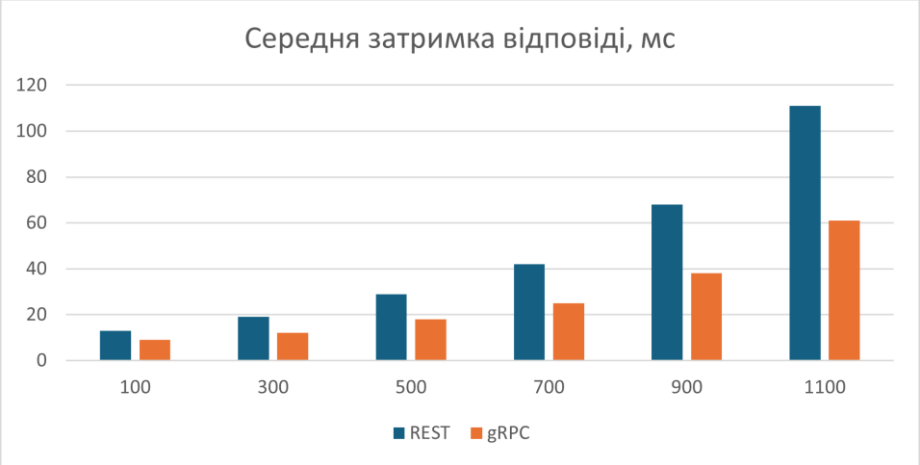

Рис. 2. Динаміка середньої затримки відповіді для REST і gRPC за різних рівнів навантаження

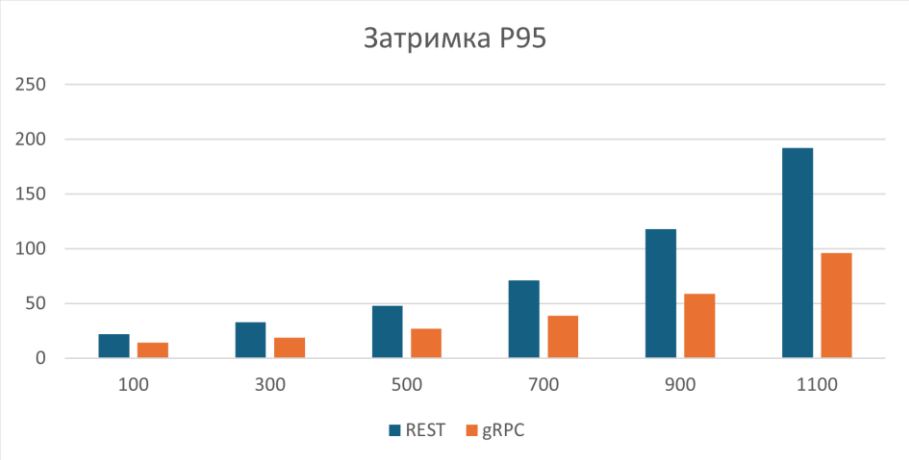

Рис. 3. Динаміка P95 затримки відповіді для REST і gRPC за різних рівнів навантаження

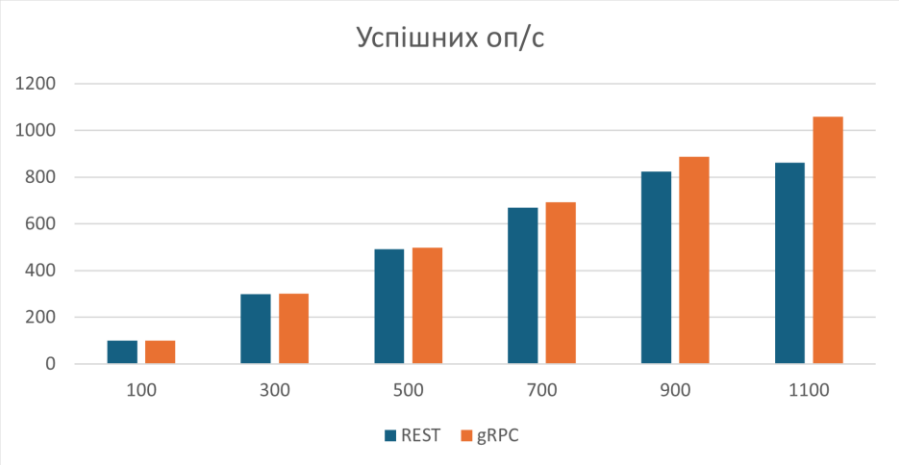

Рис. 4. Кількість успішних операцій за секунду для REST і gRPC за різних рівнів навантаження

Порівняння обсягу даних

Окремо оцінено середній ефективний обсяг даних на один виклик. Для такого порівняння важливо зважати, що службова частина мережевого обміну в REST/HTTP/1.1 і gRPC/HTTP/2 формується по-різному. У REST-запитах заголовки здебільшого передаються як текстові поля, тоді як у HTTP/2 застосовуються механізми кодування та стискання

ISSN 3083-7715 (online) *Автоматизовані системи управління та прилади автоматики. 2026. № 2 (189)*
ISSN 3083-7650 (0135-1710) (print) *Management Information System and Devices. 2026. No. 2 (189)*

заголовків, а їх фактичний внесок залежить від стану з'єднання й повторного використання попередньо переданих значень. Тому в табл. 3 подано не буквальний розмір кожного окремого HTTP-заголовка, а усереднений на один успішний виклик обсяг службової та корисної частини мережевого обміну. Під ефективним обсягом заголовків у цьому дослідженні розуміється середній обсяг службових даних протоколу, що припадає на один виклик у межах вимірювального сценарію. Такий підхід дає змогу коректніше зіставити REST/JSON і gRPC/Protobuf, оскільки бере до уваги не лише розмір тіла повідомлення, а й практичний внесок протокольних накладних витрат. Очікувано, що Protocol Buffers забезпечує більш компактне двійкове подання структурованих даних порівняно з текстовим JSON [8, 10].

Таблиця 3. *Значення середнього обсягу даних на один виклик*

Підхід	Ефективний обсяг заголовків запиту, В	Тіло запиту, В	Ефективний обсяг заголовків відповіді, В	Тіло відповіді, В	Загальний обсяг обміну, В
REST (HTTP/1.1 + JSON)	410	278	430	932	2050
gRPC (HTTP/2 + Protobuf)	92	146	108	472	818

Наведені дані відтворюють типову для такого сценарію тенденцію: основний виграш gRPC формується не лише завдяки меншим накладним витратам на заголовки, а й більш компактному поданню корисного навантаження.

Застосування ресурсів CPU і RAM

Оскільки нижча затримка ще не гарантує кращої експлуатаційної ефективності, додатково оцінено витрати CPU і RAM на серверному боці. У табл. 4 подано усереднені значення для характерних точок навантаження.

Таблиця 4. *Значення використання серверних ресурсів*

Навантаження, RPS	REST		gRPC	
	CPU, %	RAM, MB	CPU, %	RAM, MB
300	21	286	16	268
700	49	431	37	395
900	66	522	49	451
1100	78	603	61	512

Для візуального порівняння застосування серверних ресурсів за різних рівнів навантаження результати табл. 4 продемонстровано у вигляді діаграми (рис. 5).

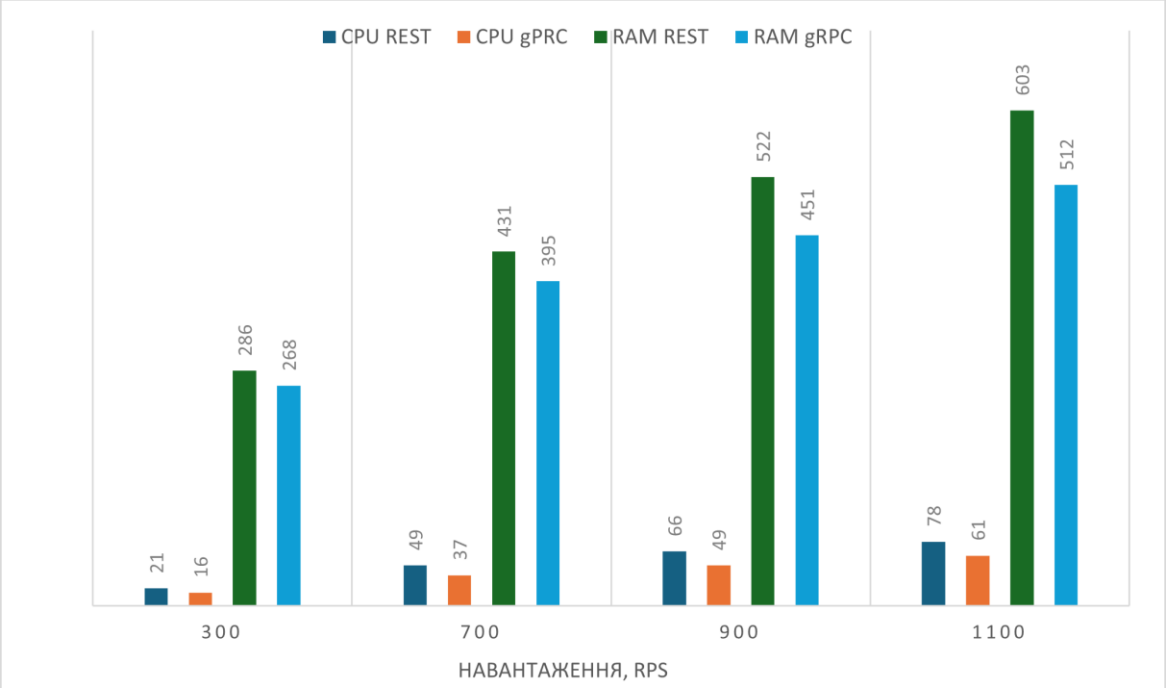

Рис. 5. Використання CPU і RAM серверного контейнера для REST і gRPC за різних рівнів навантаження

Автоматизовані системи управління та прилади автоматики. 2026. № 2 (189) *ISSN 3083-7715 (online)*
Management Information System and Devices. 2026. No. 2 (189) *ISSN 3083-7650 (0135-1710) (print)*

Відповідно до отриманих показників, зі зростанням навантаження REST швидше підходить до межі процесорної насиченості. Це узгоджується з очікуванням, що текстова серіалізація JSON і більший обсяг мережевого обміну створюють вищі накладні витрати на оброблення запитів. Для gRPC витрати CPU і RAM також зростають, але повільніше, що добре узгоджується з використанням більш компактного двійкового формату й ефективнішої транспортної взаємодії поверх HTTP/2. Значення RAM варто інтерпретувати як середнє використання пам'яті контейнером у межах вимірювального інтервалу, що містить не лише об'єкти прикладної логіки, а й службові структури runtime, мережеві буфери та механізми оброблення запитів.

Інтерпретація результатів

Проведений експеримент дає змогу дійти певних висновків. По-перше, зіставлення середньої затримки та P95 дає змогу оцінити не лише середній час відповіді, а й стабільність сервісу під навантаженням. По-друге, таблиця обсягу даних демонструє, чи справді виграш пов'язаний із компактністю бінарної серіалізації, а не лише з особливостями реалізації серверного коду. По-третє, CPU і RAM дають змогу відокремити мережеву ефективність від суто обчислювальної вартості оброблення запиту.

У межах цієї постановки експерименту реалізація gRPC на HTTP/2 з Protocol Buffers продемонструвала перевагу над REST API на HTTP/1.1 з JSON у сценарії міжсервісної взаємодії, де важливими є низькі затримки, висока інтенсивність запитів і стабільна поведінка в умовах зростання навантаження. Водночас цей висновок необхідно інтерпретувати в межах заданих умов експерименту. Його не можна автоматично поширювати на всі ймовірні реалізації REST і gRPC, оскільки HTTP API з JSON також здатні працювати поверх HTTP/2, а для браузерних клієнтів стандартні HTTP API часто залишаються більш природним варіантом. Звичайний gRPC у таких сценаріях потребує додаткових механізмів, зокрема gRPC-Web або JSON transcoding [10].

Висновки

У роботі досліджено підходи до реалізації міжсервісної взаємодії в мікросервісній архітектурі з використанням REST і gRPC у середовищі .NET 8. Розглянуто архітектурні особливості синхронного обміну даними між сервісами, а також проаналізовано практичні аспекти реалізації HTTP API з JSON і gRPC-сервісів на платформі ASP.NET Core 8. Основну увагу зосереджено на впливі транспортного рівня, форматі серіалізації та способі організації викликів на ефективність міжсервісної взаємодії.

У межах роботи проведено експериментальне дослідження, в якому реалізовано два сервіси з однаковою прикладною логікою: REST-сервіс на основі HTTP/1.1 та JSON і gRPC-сервіс на основі HTTP/2 й Protocol Buffers. Для забезпечення коректності порівняння обидва варіанти працювали з однаковим in-memory набором даних, без звернення до зовнішньої бази даних, а навантаження генерувалося засобами Grafana k6 в однакових режимах інтенсивності запитів. Така постановка дала змогу зосередити аналіз саме на відмінностях транспорту, серіалізації та накладних витрат оброблення запитів. Порівняльний аналіз продемонстрував, що REST і gRPC мають різні сфери ефективного застосування. REST властива простота реалізації, висока сумісність, прозорість інтерфейсу та зручність інтеграції, що робить його доцільним для побудови публічних API, зовнішніх інтеграцій і взаємодії з гетерогенними системами. Натомість gRPC є більш придатним для внутрішньої міжсервісної взаємодії, де критичними є низькі затримки, стабільність часу відповіді, висока пропускна здатність і ефективне використання мережевих ресурсів.

За результатами проведеного експерименту встановлено, що в межах запропонованого сценарію реалізація gRPC на HTTP/2 з Protocol Buffers продемонструвала нижчі значення середньої затримки та P95, менший обсяг мережевого обміну й вищу пропускну здатність порівняно з REST API на HTTP/1.1 з JSON. Результати показали, що на верхніх рівнях навантаження середній час відповіді для gRPC був нижчим орієнтовно в 1,7–1,8 раза, тоді як загальний обсяг переданих даних на один виклик виявився меншим приблизно у 2,5 раза, якщо порівнювати з REST. Крім того, для gRPC зафіксовано більш помірне використання процесорних і пам'яттєвих ресурсів серверного вузла. Це дає змогу зробити висновок, що поєднання HTTP/2 і Protocol Buffers дійсно створює передумови для підвищення ефективності внутрішньої сервісної взаємодії.

Водночас досягнуті результати не дають підстав розглядати gRPC як універсальну заміну REST у всіх

ISSN 3083-7715 (online) *Автоматизовані системи управління та прилади автоматики. 2026. № 2 (189)*
ISSN 3083-7650 (0135-1710) (print) *Management Information System and Devices. 2026. No. 2 (189)*

сценаріях. У разі невисокого навантаження, потреби в максимально простій інтеграції, широкій клієнтській сумісності або орієнтації на публічний вебінтерфейс REST зберігає свою практичну перевагу. Отже, REST і gRPC доцільно розглядати не як несумісні, а як взаємодоповнювальні підходи, що можуть ефективно комбінуватися в межах однієї мікросервісної архітектури залежно від призначення окремих сервісів і вимог до їх взаємодії. Отже, вибір підходу до реалізації міжсервісної взаємодії має здійснюватися з огляду на вимоги до затримок, пропускної здатності, масштабованості, сумісності та експлуатаційної складності. Практична цінність роботи полягає в тому, що запропонована методика експерименту й досягнуті результати можуть бути використані для подальшого розширення сценаріїв тестування й обґрунтованого вибору механізму міжсервісної взаємодії.

Перспективи подальших досліджень полягають у проведенні повномасштабних експериментів із реальними навантаженнями, у розширенні набору сценаріїв унаслідок різних типів повідомлень і конфігурацій мережевого середовища, а також у дослідженні поєднання синхронних і асинхронних механізмів взаємодії в мікросервісних системах.

Конфлікт інтересів

Автори декларують, що не мають конфлікту інтересів, зокрема фінансового, особистого, авторського чи будь-якого іншого характеру, який міг би вплинути на дослідження, а також на результати, опубліковані в цій статті.

Фінансування

Дослідження проводилося без фінансової підтримки.

Доступність даних

Рукопис не має пов’язаних даних.

Використання засобів штучного інтелекту

Автори підтверджують, що не застосовували технології ШІ для написання роботи.

References

1. Dragoni, N., Giallorenzo, S., Lluch Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., Safina, L. (2017), "Microservices: Yesterday, Today, and Tomorrow", In: *Present and Ulterior Software Engineering*. Cham: Springer, pp. 195–216. DOI: https://doi.org/10.1007/978-3-319-67425-4_12
2. Velepucha, V., Flores, P. (2023), "A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges", *IEEE Access*, Vol. 11, pp. 88339–88358. DOI: https://doi.org/10.1109/ACCESS.2023.3305687
3. Weerasinghe, L. D. S. B., Perera, I. (2022), "Evaluating the Inter-Service Communication on Microservice Architecture»", *7th International Conference on Information Technology Research (ICITR)*, pp. 1–6. DOI: https://doi.org/10.1109/ICITR57877.2022.9992918
4. Bogner, J., Kotstein, S., Pfaff, T. (2023), "Do RESTful API design rules have an impact on the understandability of Web APIs?", *Empirical Software Engineering*, Vol. 28, Article 132. DOI: https://doi.org/10.1007/s10664-023-10367-y
5. Fielding, R. T., Nottingham, M., Reschke, J. (2022), *HTTP Semantics*. RFC 9110. RFC Editor. DOI: https://doi.org/10.17487/RFC9110
6. Thomson, M., Benfield, C. (2022), *HTTP/2*. RFC 9113. RFC Editor. DOI: https://doi.org/10.17487/RFC9113
7. Fielding, R. T., Nottingham, M., Reschke, J. (2022), *HTTP/1.1*. RFC 9112. RFC Editor. DOI: https://doi.org/10.17487/RFC9112
8. Maltsev, E. Y., Muliarevych, O. (2024), "Beyond JSON: Evaluating Serialization Formats for Space-Efficient Communication", *Advances in Cyber-Physical Systems*, Vol. 9, No. 1, pp. 9–15. DOI: https://doi.org/10.23939/acps2024.01.009
9. Niswar, M., Arisandy Safruddin, R., Bustamin, A., Aswad, I. (2024), "Performance evaluation of microservices communication with REST, GraphQL, and gRPC", *International Journal of Electronics and Telecommunications*, Vol. 70, No. 2, pp. 429–436. DOI: https://doi.org/10.24425/ijet.2024.149562
10. Microsoft. Compare gRPC services with HTTP APIs. URL: https://learn.microsoft.com/en-us/aspnet/core/grpc/comparison
11. Bolanowski, M., Żak, K., Paszkiewicz, A., Ganzha, M., Paprzycki, M., Sowiński, P., Lacalle, I., Palau, C. E. (2022), "Efficiency of REST and gRPC Realizing Communication Tasks in Microservice-Based Ecosystems", In: *New Trends in Intelligent Software Methodologies, Tools and Techniques*. IOS Press, pp. 97–108. DOI: 1 https://doi.org/0.3233/FAIA220242

12. Jarmoszewicz, J., Iwanowski, P., Plechawska-Wójcik, M. (2024), "Analysis of the performance and scalability of microservices depending on the communication technology", *Journal of Computer Sciences Institute*, Vol. 33, pp. 323–330. DOI: https://doi.org/10.35784/jcsi.6499
13. Tkachov, V. (2026), "Method of restoring critical information system services on a mobile platform under conditions of controlled degradation", *Automated Control Systems and Automation Devices*, No. 188, pp. 78–97. DOI: https://doi.org/10.30837/0135-1710.2026.188.078
14. Mykhailichenko, I. V., Liashenko, O. S. (2025), "Resource usage prediction model in cloud computing using Informer architecture", *Automated Control Systems and Automation Devices*, No. 186, pp. 17–28. DOI: https://doi.org/10.30837/0135-1710.2025.186.017
15. Glavchev, M. I., Glavchev, D. M., Panchenko, V. I. (2025), "Evaluating the effectiveness of open-source solutions for monitoring and load balancing in microservice applications ", *Automated Control Systems and Automation Devices*, No. 187, pp. 182–199. DOI: https://doi.org/10.30837/0135-1710.2025.187.182

Received (Надійшла) 19.03.2026
Accepted for publication (Прийнята до друку) 20.04.2026
Publication date (Дата публікації) 29.05.2026

Відомості про авторів / About the Authors

Бобровнікова Кіра Юліївна – кандидат технічних наук, доцент, Університет економіки і підприємництва, завідувач кафедри математики та інформаційних технологій, Хмельницький, Україна;
Kira Bobrovnikova – Candidate of Technical Sciences, Associate Professor, University of Economics and Entrepreneurship, Head of the Department of Mathematics and Information Technologies, Khmelnytskyi, Ukraine;
e-mail: bobrovnikova.kira@gmail.com
ORCID ID: https://orcid.org/0000-0002-1046-893X

Гурман Іван Васильович – кандидат технічних наук, доцент, Університет економіки і підприємництва, доцент кафедри математики та інформаційних технологій, Хмельницький, Україна;
Ivan Hurman – Candidate of Technical Sciences, Associate Professor, University of Economics and Entrepreneurship, Associate Professor of the Department of Mathematics and Information Technologies, Khmelnytskyi, Ukraine;
e-mail: devastator167384@gmail.com
ORCID ID: https://orcid.org/0000-0002-2282-3484

Олексюк Дмитро Андрійович – Університет економіки і підприємництва, старший викладач кафедри математики та інформаційних технологій, Хмельницький, Україна;
Dmytro Oleksiuk – University of Economics and Entrepreneurship, Senior Lecturer of the Department of Mathematics and Information Technologies, Khmelnytskyi, Ukraine;
e-mail: oleksuk.dima@gmail.com
ORCID ID: https://orcid.org/0009-0006-3735-1930

COMPARATIVE ANALYSIS OF INTER-SERVICE COMMUNICATION USING REST AND GRPC IN .NET 8 ENVIRONMENT

The subject of the study is approaches to implementing synchronous inter-service communication in a microservice architecture using REST and gRPC in the .NET 8 environment. The aim of the work is to conduct a comparative analysis of a REST API based on HTTP/1.1 and JSON and a gRPC service based on HTTP/2 and Protocol Buffers, as well as to evaluate their impact on performance, network exchange volume, and server resource utilization. To achieve this aim, the following tasks are defined: to analyze the architectural differences between REST and gRPC; to implement two services with identical application logic in the ASP.NET Core 8 environment; to ensure identical experimental conditions; to measure average response latency, P95 latency, the number of successful operations per second, the volume of transmitted data, and CPU and RAM utilization;

ISSN 3083-7715 (online) *Автоматизовані системи управління та прилади автоматики. 2026. № 2 (189)*
ISSN 3083-7650 (0135-1710) (print) *Management Information System and Devices. 2026. No. 2 (189)*

and to interpret the obtained results with regard to the application limits of each approach. **Methods.** The study uses comparative analysis of architectural approaches, experimental load testing, and quantitative performance evaluation. Two services with identical application logic and a shared in-memory dataset were implemented without accessing an external database. The workload was generated using Grafana k6 in constant-arrival-rate mode. **Research results.** Within the proposed experimental setup, gRPC over HTTP/2 with Protocol Buffers demonstrated lower average latency and P95 values, a smaller network exchange volume, and higher throughput compared with the REST API over HTTP/1.1 with JSON. The total exchange volume for gRPC was approximately 2.5 times smaller, while at the upper load levels the average latency was about 1.7-1.8 times lower. More moderate CPU and memory resource utilization of the server node was also recorded for gRPC. **Conclusions.** The obtained results confirm the feasibility of using gRPC primarily for internal inter-service communication, where low latency, high throughput, and efficient use of network resources are critical. REST is appropriate in scenarios where compatibility, ease of integration, transparency of the HTTP interface, and usability in public APIs are priorities.

Keywords: microservice architecture; inter-service communication; distributed systems; performance; response latency; throughput.

Бібліографічні описи / Bibliographic descriptions

Бобровнікова К. Ю., Гурман І. В., Олексюк Д. А. Порівняльний аналіз міжсервісної взаємодії з використанням REST і gRPC у середовищі .NET 8. *Автоматизовані системи управління та прилади автоматики*. 2026. № 2 (189). С. 95–108. DOI: https://doi.org/10.30837/0135-1710.2026.189.095

Bobrovnikova, K., Hurman, I., Oleksiuk, D. (2026), "Comparative analysis of inter-service communication using REST and gRPC in .NET 8 environment", *Management Information System and Devices*, No. 2 (189), P. 95–108. DOI: https://doi.org/10.30837/0135-1710.2026.189.095