УДК 004.75
https://doi.org/10.31891/2219-9365-2025-82-38

ОСТАПЕЦЬ Денис
Український державний університет науки і технологій
https://orcid.org/0000-0003-1778-7770
e-mail: odaua@i.ua
НАГОРЯНСЬКИЙ Микола
Український державний університет науки і технологій
https://orcid.org/0009-0000-5396-1305
e-mail: n.nagoryanskiy@gmail.com

ПОРІВНЯЛЬНИЙ АНАЛІЗ СУЧАСНИХ ЗАСОБІВ ВЗАЄМОДІЇ МІЖ КОМПОНЕНТАМИ В РОЗПОДІЛЕНИХ ПРОГРАМНИХ СИСТЕМАХ

В роботі розглядається порівняльний аналіз сучасних засобів взаємодії між компонентами в розподілених програмних системах, що є важливими для побудови ефективних, надійних та масштабованих інформаційних технологій. Дано класифікацію механізмів комунікації за моделлю взаємодії (синхронна і асинхронна). Представлено огляд та аналіз відомих сучасних технологій комунікації, таких як RESTful API, gRPC для синхронної взаємодії і Apache Kafka та RabbitMQ для асинхронної взаємодії. Описано архітектурні особливості кожної з технологій, її сильні та слабкі сторони, технічні обмеження, а також типові сценарії використання. Проведено аналіз характеристик продуктивності, масштабованості, складності інтеграції, гарантій доставки повідомлень і можливостей маршрутизації. Сформовані рекомендацій з вибору засобів комунікації в залежності від потреб архітектури системи.

Ключові слова: інформаційні технології, розподілені програмні системи, моделі взаємодії, RESTful API, gRPC, Apache Kafka, RabbitMQ, архітектура програмного забезпечення.

OSTAPETS Denys, NAHORIANSKYI Mykola
Ukrainian State University of Science and Technologies

COMPARATIVE ANALYSIS OF MODERN INTERACTION MEANS BETWEEN COMPONENTS IN DISTRIBUTED SOFTWARE SYSTEMS

This paper presents an in-depth comparative analysis of contemporary communication mechanisms used in distributed software systems, which play a pivotal role in the development of efficient, reliable, and scalable information technologies. A systematic classification of interaction methods is proposed based on the model of communication—synchronous and asynchronous. This classification serves as a fundamental criterion for selecting the appropriate communication mechanism and is closely tied to the architectural paradigm of the distributed system, significantly influencing its performance, reliability, and adaptability.

The study provides a comprehensive overview and technical evaluation of widely adopted technologies for component interaction. In the realm of synchronous communication, RESTful APIs and gRPC are analyzed for their usability, protocol characteristics, and compatibility. For asynchronous messaging, the paper investigates the features and implementations of Apache Kafka and RabbitMQ, emphasizing their messaging models, persistence capabilities, and event-driven design.

Each technology is assessed in terms of architectural implications, performance metrics, scalability potential, integration complexity, message delivery guarantees, and support for complex routing scenarios. The strengths and limitations of each solution are discussed, supported by real-world application cases and usage patterns.

Based on the comparative insights, the paper provides practical recommendations for selecting communication mechanisms that align with specific architectural and operational requirements. The results aim to support system architects and developers in designing robust and maintainable distributed systems.

Keywords: information technology, distributed software systems, interaction models, RESTful API, gRPC, Apache Kafka, RabbitMQ, software architecture.

Стаття надійшла до редакції / Received 16.04.2025
Прийнята до друку / Accepted 11.05.2025

ПОСТАНОВКА ПРОБЛЕМИ У ЗАГАЛЬНОМУ ВИГЛЯДІ ТА ЇЇ ЗВ'ЯЗОК ІЗ ВАЖЛИВИМИ НАУКОВИМИ ЧИ ПРАКТИЧНИМИ ЗАВДАННЯМИ

Сучасні інформаційні технології все частіше покладаються на розподілені системи для вирішення складних обчислювальних завдань, обробки великих даних та забезпечення високої доступності та надійності сервісів. Ефективна взаємодія між компонентами таких систем є ключовим фактором їх успішного функціонування, тому огляд сучасних засобів забезпечення комунікації між різними програмними компонентами є актуальною задачею [1]. Розподілена система – це комплекс незалежних комп'ютерів, які сприймаються користувачем єдиною об'єднаною системою [2]. Будемо розглядати розподілені програмні системи (РПС) які визначаються як сукупність комп'ютерних програм, що використовують обчислювальні ресурси на кількох окремих обчислювальних вузлах для досягнення спільної, загальної мети. Ці вузли зазвичай представляють окремі фізичні апаратні пристрої, але можуть також представляти окремі програмні процеси. Основна мета РПС полягає в усуненні вузьких місць або центральних точок відмови, властивих

централізованим системам [3]. Такі системи дозволяють ефективно обробляти та зберігати значні обсяги даних, забезпечуючи при цьому високий рівень доступності та надійності сервісів.

Засоби взаємодії в РПС– це сукупність технологій, протоколів, патернів та інструментів, які дозволяють встановлювати з'єднання між віддаленими компонентами (апаратними вузлами, процесами, сервісами), передавати дані та координувати їхні спільні дії. Вони є фундаментальною основою для забезпечення спільного використання ресурсів, таких як дані, обчислювальні потужності, тощо.

У більш широкому сенсі, засоби взаємодії можна розглядати як канали та способи передачі інформаційних повідомлень між учасниками взаємодії. У контексті РПС, взаємодія – це не просто передача байтів даних. Вона охоплює значно ширший спектр завдань, включаючи узгодження форматів даних (серіалізація/десеріалізація) [4], обробку помилок передачі, забезпечення послідовності доставки повідомлень, механізми синхронізації, а в деяких випадках – і підтримку транзакційності на рівні комунікації.

Вибір конкретних засобів взаємодії тісно пов'язаний з архітектурним підходом до побудови РПС і може суттєво впливати на її ключові характеристики, такі як продуктивність, масштабованість, надійність та складність розробки й підтримки [5]. Наприклад, мікросервісна архітектура, що передбачає велику кількість дрібних, незалежно розгорнутих сервісів, висуває особливі вимоги до легкості, швидкості та надійності міжсервісної комунікації [6].

ФОРМУЛЮВАННЯ ЦІЛЕЙ СТАТТІ

Метою роботи є порівняльний аналіз сучасних засобів взаємодії між компонентами в розподілених програмних системах, зокрема технологій RESTful API, gRPC, Apache Kafka та RabbitMQ, для визначення їх переваг, недоліків, специфіки використання та впливу на архітектурні рішення систем. Результати порівняльного аналізу можуть бути використані при реалізації розподілених систем. В роботі, відповідно до мети, поставлені наступні задачі: визначення основних особливостей розподілених програмних систем та моделей взаємодії між їх компонентами; огляд та аналіз відомих сучасних технологій комунікації – RESTful API, gRPC, Apache Kafka, RabbitMQ; формування рекомендацій з вибору засобів комунікації в залежності від потреб архітектури системи.

ВИКЛАД ОСНОВНОГО МАТЕРІАЛУ

Сучасні РПС використовують широкий спектр засобів комунікації, які можна класифікувати за різними критеріями [7]. Розуміння цих класифікацій допомагає систематизувати знання та зробити обґрунтований вибір технологій для конкретних завдань. Їх можна категоризувати на основі декількох ключових аспектів: моделі взаємодії, типу технології або протоколу, а також рівня формалізації та зв'язаності між компонентами.

Існують дві фундаментальні моделі (парадигми) взаємодії: синхронна (запит-відповідь) та асинхронна (подіє орієнтована). Кожна з них має свої переваги, недоліки та оптимальні сценарії застосування, і вибір між ними суттєво впливає на загальну архітектуру, продуктивність, надійність та складність системи [7].

Синхронна взаємодія

При синхронній взаємодії клієнтський компонент надсилає запит до сервісного компонента і блокує свою подальшу роботу, очікуючи на відповідь. Цей процес триває доти, доки відповідь не буде отримана, або доки не мине встановлений тайм-аут. Такий підхід забезпечує негайний зворотний зв'язок, що може спростити логіку клієнта для певних типів операцій. Однак, він створює тимчасову залежність: клієнт не може продовжувати роботу, а сервіс-одержувач повинен бути доступним і швидко відповісти, щоб уникнути тривалого блокування клієнта. Це може призвести до зниження загальної пропускної здатності системи та потенційних каскадних збоїв, якщо сервіс-одержувач стає повільним або недоступним.

Розглянемо найпопулярніші приклади реалізації синхронної взаємодії.

RESTful APIs. REST (Representational State Transfer) є архітектурним стилем, що широко використовується для побудови веб-сервісів, зокрема API (Application Programming Interfaces). RESTful API базуються на стандартних методах протоколу HTTP для взаємодії з ресурсами, які ідентифікуються за допомогою URI (Uniform Resource Identifiers).

Переваги:

- Простота та інтуїтивність. REST використовує загальновідомі стандарти HTTP, що робить його відносно простим для розуміння та використання.
- Масштабованість та надійність. Безстатусність та незалежність запитів сприяють масштабованості як клієнтської, так і серверної частини.
- Мовна та платформна незалежність.
- Використання HTTP-кешування. Можливість кешування відповідей дає можливість покращити продуктивність та зменшити навантаження на сервери.

- Гнучкість формату даних. Підтримуються різноманітні формати даних, такі як JSON, XML, HTML, звичайний текст.

Недоліки:

- Синхронний характер. Типова синхронна модель запит-відповідь може призводити до блокування клієнта та проблем з продуктивністю при високих навантаженнях або повільних сервісах.
- Надлишкове або недостатнє отримання даних. Це призводить до неефективного використання пропускної здатності та збільшення затримок.
- Складність забезпечення обробки стану ресурсу при виконання запиту який запускає асинхронну операцію.
- Тісна зв'язаність через схему даних.

Типові сценарії використання:

- Мікросервісні архітектури: Для комунікації між окремими сервісами де синхронна взаємодія є прийнятною.
- Публічні API: Багато компаній надають публічні REST API для інтеграції своїх сервісів з третіми сторонами , такими як веб-додадки, пристрої інтернету речей, тощо.

gRPC. gRPC (Google Remote Procedure Call) [8] – це сучасний, високопродуктивний, відкритий фреймворк для реалізації віддалених викликів процедур (RPC), розроблений Google. Він використовує HTTP/2 як транспортний протокол, що забезпечує значні переваги над HTTP/1.1, та Protocol Buffers (Protobuf) як мову опису інтерфейсу (IDL) та формат серіалізації даних.

Переваги:

- Висока продуктивність та низька затримка. Завдяки використанню HTTP/2 та ефективній бінарній серіалізації Protocol Buffers, gRPC значно швидший [9] за традиційні REST+JSON API.
- Ефективне використання мережевих ресурсів. Мультиплексування та стиснення заголовків в HTTP/2 зменшують накладні витрати на мережу.
- Потужна підтримка потокової передачі даних:
- Строго типізовані контракти. Використання Protocol Buffers для визначення сервісів та повідомлень забезпечує строгу типізацію та чіткі контракти API. Це допомагає виявляти помилки на етапі компіляції, покращує надійність та полегшує еволюцію API.
- Вбудовані функції. gRPC надає вбудовану підтримку для важливих аспектів РПС, таких як дедлайни/таймаути, скасування RPC, балансування навантаження, обмін метаданими, шифрування (TLS) та аутентифікація.

Недоліки:

- Менша людська читабельність. Оскільки Protocol Buffers використовує бінарний формат серіалізації, повідомлення gRPC не є легко читабельними для людини, на відміну від JSON або XML, що може ускладнити налагодження.
- Крива навчання. Для розробників, не знайомих з RPC, HTTP/2 та Protocol Buffers, початкове вивчення та налаштування gRPC може бути складнішим порівняно з більш звичними REST API.
- Кешування. Традиційні механізми HTTP-кешування, які добре працюють з REST (особливо для GET-запитів), менш застосовні до gRPC, оскільки gRPC-виклики зазвичай використовують HTTP POST, а відповіді не призначені для кешування проміжними вузлами.
- Інструментарій. Хоча екосистема gRPC активно розвивається, інструментарій для тестування, моніторингу та документування може бути менш зрілим або поширеним, ніж для REST API.

Типові сценарії використання:

- Внутрішня комунікація між мікросервісами. Це основний і найпоширеніший випадок використання. Висока продуктивність, низька затримка та строгі контракти роблять gRPC хорошим засобом для швидкої та надійної взаємодії між сервісами всередині однієї системи або дата-центру.

Асинхронна взаємодія

В асинхронній взаємодії клієнтський компонент надсилає запит (повідомленням) і не чекає відповіді від сервісного компонента. Це дозволяє клієнту продовжувати свою роботу, не блокуючись. Повідомлення зазвичай розміщується в проміжному сховищі, такому як черга повідомлень, звідки сервісний компонент може його забрати для обробки у зручний для себе час.

Такий підхід розриває пряму залежність між клієнтом та сервером, що є характерною для синхронної моделі. Це призводить до підвищення стійкості системи, оскільки компоненти можуть функціонувати, навіть якщо інші частини системи тимчасово недоступні або перевантажені. Масштабованість також покращується, оскільки ініціатори та отримувачи повідомлень можуть масштабуватися незалежно. Однак асинхронна комунікація ускладнює отримання негайного зворотного зв'язку та вимагає інших підходів до обробки помилок, забезпечення послідовності операцій та узгодженості даних.

Також слід зазначити, що асинхронна модель є основою для реалізації Подієво-орієнтованої архітектури (EDA) [10].

Черги повідомлень є фундаментальним інструментом для реалізації асинхронної взаємодії в РПС. Вони діють як посередники (брокери), які тимчасово зберігають повідомлення, надіслані продюсерами (сервісами, що генерують повідомлення), доки їх не заберуть та не оброблять консюмери (сервіси, що споживають повідомлення). Це дозволяє роз'єднати продюсерів та консюмерів, підвищуючи гнучкість, стійкість та масштабованість системи. Розглянемо дві найпопулярніші системи черг повідомлень (RabbitMQ та Apache Kafka) які мають відмінні архітектурні особливості та особливості застосування. Інші аналоги розглядати не будемо, оскільки вони так чи інакше дублюють особливості перерахованих вище систем.

Apache Kafka. Kafka [11] – це розподілена платформа потокової передачі подій, розроблена та оптимізована для високої пропускної здатності та обробки даних у реальному часі з низькою затримкою. Повідомлення (записи) організовані в топіки (topics), які, у свою чергу, розділені на партиції (partitions). Партиції є впорядкованими, незмінними послідовностями записів. Kafka використовує модель "pull", де консюмери самі витягують повідомлення з партицій, відстежуючи свою позицію (offset). Повідомлення зберігаються в логах протягом налаштовуваного періоду, що дозволяє повторну обробку

Переваги:

- Висока пропускна здатність та низька затримка. Kafka розроблена для обробки мільйонів повідомлень на секунду [12], що робить можливим її використання для високо-навантажених систем та потокової передачі даних у реальному часі. Це досягається за рахунок оптимізацій, таких як послідовний запис на диск (sequential I/O), пакетна обробка повідомлень та нульове копіювання (zero-copy).
- Масштабованість. Kafka відмінно масштабується горизонтально шляхом додавання нових брокерів до кластера та збільшення кількості партицій для тем.
- Відмовостійкість та надійність. Реплікація партицій на декількох брокерах забезпечує високу доступність та захист від втрати даних у разі відмови окремих вузлів кластера.
- Довготривале зберігання повідомлень (Durability & Retention). Повідомлення можуть зберігатися протягом тривалого часу (дні, тижні, місяці або навіть необмежено), що дозволяє використовувати Kafka не тільки для передачі даних, але і як систему зберігання для потоків подій.
- Екосистема та інтеграції. Kafka має багату екосистему інструментів та бібліотек, включаючи Kafka Connect, Kafka Streams та ksqlDB.
- Гарантії доставки. Kafka підтримує різні семантики доставки повідомлень: "щонайбільше один раз" (at-most-once), "щонайменше один раз" (at-least-once) та, за певних умов та з використанням відповідних механізмів (наприклад, ідемпотентні продюсери, транзакції, Kafka Streams), "точно один раз" (exactly-once).

Недоліки:

- Складність налаштування та управління. Розгортання, конфігурація, моніторинг та оптимізація кластера Kafka, особливо великого, можуть бути складними завданнями, що вимагають глибоких знань та досвіду.
- Високі вимоги до ресурсів.
- Обмежена гнучкість маршрутизації повідомлень. Kafka в першу чергу орієнтована на потокову передачу даних та модель Pub/Sub на основі тем. Вона не надає таких гнучких механізмів маршрутизації окремих повідомлень на основі їх вмісту або атрибутів.
- Управління зміщеннями (Offset Management). Хоча Kafka надає механізми для відстеження зміщень споживачами, відповідальність за правильне управління цими зміщеннями (збереження, відновлення) лежить на клієнтських додатках або бібліотеках, що може додати складності.
- Проблеми зі зберіганням дуже великих обсягів історичних даних. Хоча Kafka може зберігати дані довго, її основне призначення – потокова обробка.

Типові сценарії використання:

- Обробка потоків даних у реальному часі (Real-time Data Processing)
- Агрегація логів та метрик (Log Aggregation & Metrics Collection):
- Відстеження активності користувачів (Website Activity Tracking)
- Побудова подієво-орієнтованих архітектур (Event-Driven Architectures - EDA): Kafka часто виступає як центральна шина подій, що дозволяє різним мікросервісам та компонентам асинхронно реагувати на події, що відбуваються в системі.
- Системи обміну повідомленнями з високим навантаженням: Як заміна традиційним брокерам повідомлень у випадках, коли потрібна екстремальна пропускна здатність та масштабованість.
- Інтеграція даних (Data Integration): Передача даних між різними системами, базами даних, сховищами даних та додатками.
- Event Sourcing: Використання Kafka як надійного та впорядкованого сховища для послідовності подій, що відображають зміни стану сутностей в системі.

RabbitMQ. RabbitMQ [13] є традиційним брокером повідомлень, що реалізує протокол AMQP (Advanced Message Queuing Protocol), а також підтримує інші, як MQTT та STOMP. Він використовує концепцію обмінників (exchanges), які отримують повідомлення від продюсерів і маршрутизують їх до однієї або кількох черг на основі правил маршрутизації та ключів. Консюмери підписуються безпосередньо на черги. RabbitMQ зазвичай використовує модель "push", де брокер активно надсилає повідомлення доступним консюмерам. Універсальність RabbitMQ, зумовлена підтримкою кількох протоколів (AMQP, MQTT, STOMP), робить його зручним інструментом для інтеграції різнорідних систем [14].

Переваги:

- Гнучка маршрутизація повідомлень. Завдяки різноманітним типам обмінників (direct, fanout, topic, headers) та гнучким правилам зв'язування, RabbitMQ дозволяє реалізовувати складні сценарії маршрутизації та доставки повідомлень.
- Підтримка декількох протоколів. Нативно підтримує AMQP 0-9-1, а також через плагіни – MQTT (для IoT), STOMP, що робить його універсальним рішенням для інтеграції різнорідних систем.
- Надійна доставка повідомлень: Підтримує механізми підтвердження отримання повідомлень (acknowledgements) як з боку споживача, так і з боку продюсера (publisher confirms), довговічність (persistence) черг та повідомлень (збереження на диск), що забезпечує високу надійність доставки.
- Зрілість та стабільність. RabbitMQ є добре перевіреною часом технологією з великою та активною спільнотою, розгалуженою документацією та численними клієнтськими бібліотеками для різних мов програмування.
- Підтримка пріоритетних черг.
- Зручний інтерфейс управління. RabbitMQ Management Plugin надає веб-інтерфейс для моніторингу та управління брокером, чергами, обмінниками тощо.

Недоліки:

- Продуктивність при високих навантаженнях. У сценаріях з дуже високою пропускною здатністю (мільйони повідомлень на секунду) RabbitMQ може поступатися спеціалізованим потоковим платформам, таким як Kafka, особливо якщо активно використовується персистентність повідомлень. RabbitMQ оптимізований для обробки кожного повідомлення індивідуально і не підтримує таку ж ефективну пакетну обробку, як Kafka
- Масштабування. Хоча RabbitMQ підтримує кластеризацію, досягнення дуже високої горизонтальної масштабованості може бути складнішим і вимагати ретельнішого планування порівняно з архітектурою.
- Мова розробки (Erlang). Той факт, що RabbitMQ написаний на Erlang, може бути певним бар'єром для команд, які не мають експертизи в цій мові, якщо виникає потреба у глибокому аналізі поведінки брокера, його кастомізації або розробці власних плагінів.
- Складність управління кластером: Налаштування та підтримка відмовостійкого кластера RabbitMQ може бути нетривіальним завданням.

Типові сценарії використання:

- Фонова обробка завдань (Background Jobs/Task Queues). Розвантаження довготривалих операцій (наприклад, генерація звітів, обробка зображень, надсилання електронних листів).
- Інтеграція додатків та мікросервісів: Забезпечення надійної асинхронної комунікації між різними частинами системи або між різними додатками, особливо коли потрібна складна логіка маршрутизації повідомлень.
- Розподіл завдань між багатьма воркерами (Work Queues / Competing Consumers Pattern) для паралельного виконання.
- Системи сповіщень у реальному часі: Доставка сповіщень користувачам або іншим системам.
- Додатки Інтернету речей (IoT): Часто використовується в поєднанні з плагіном MQTT для збору даних з пристроїв та надсилання команд на них.

Вибір між RabbitMQ та Kafka (або іншими брокерами) залежить від специфічних вимог до пропускної здатності, затримок, гнучкості маршрутизації, потреби у відтворенні повідомлень та складності системи, компетенцій команди, наявних ресурсів.

У табл. 1 наведене порівняння основних реалізацій для кожної моделі взаємодії.

ВИСНОВКИ З ДАНОГО ДОСЛІДЖЕННЯ І ПЕРСПЕКТИВИ ПОДАЛЬШИХ РОЗВІДОК У ДАНОМУ НАПРЯМІ

Огляд та аналіз сучасних засобів взаємодії між компонентами в РПС показав, що існує широкий спектр технологій та архітектурних підходів для забезпечення ефективної комунікації. Вибір конкретних засобів залежить від багатьох факторів, таких як вимоги до продуктивності, масштабованості, надійності, складності розробки та підтримки, а також архітектурного стилю системи. Існують дві основних моделі взаємодії: синхронна та асинхронна.

Таблиця 1.

Порівняння основних реалізацій для кожної моделі взаємодії

Назва реалізації	Модель взаємодії	Переваги	Недоліки
RESTful APIs	Синхронна	- Простота та інтуїтивність. - Масштабованість та надійність. - Мовна та платформна незалежність. - Використання HTTP-кешування. - Гнучкість формату даних (JSON, XML, тощо).	- Синхронний характер може призводити до блокування клієнта. - Надлишкове або недостатнє отримання даних. - Складність обробки стану ресурсу при асинхронній операції. - Тісна зв'язаність через схему даних.
gRPC	Синхронна	- Висока продуктивність та низька затримка. - Ефективне використання мережевих ресурсів (HTTP/2). - Потужна підтримка потокової передачі даних. - Строго типізовані контракти (Protocol Buffers). - Вбудовані функції (таймаути, скасування, балансування, тощо).	- Менша людська читабельність (бінарний формат Protocol Buffers). - Крива навчання для новачків. - Обмежене застосування HTTP-кешування. - Менш зрілий інструментарій порівняно з REST API.
Apache Kafka	Асинхронна	- Висока пропускна здатність та низька затримка. - Масштабованість. - Відмовостійкість та надійність (реплікація). - Довготривале зберігання повідомлень. - Багата екосистема та інтеграції. - Гарантії доставки.	- Складність налаштування та управління. - Високі вимоги до ресурсів. - Обмежена гнучкість маршрутизації повідомлень. - Управління зміщеннями (offset management) лежить на клієнтських додатках. - Проблеми зі зберіганням дуже великих обсягів історичних даних.
RabbitMQ	Асинхронна	- Гнучка маршрутизація - Підтримка декількох протоколів (AMQP, MQTT, STOMP). - Надійна доставка повідомлень. - Зрілість та стабільність. - Підтримка пріоритетних черг. - Зручний інтерфейс управління.	- Продуктивність при високих навантаженнях може бути нижчою, ніж у Kafka. - Масштабування може бути складнішим. - Мова розробки Erlang може бути бар'єром для деяких команд. - Складність управління кластером.

Синхронні засоби взаємодії, такі як RESTful API та gRPC, забезпечують прямий зворотний зв'язок і добре підходять для сценаріїв, де важлива швидкість відповіді. При цьому, gRPC має кращі характеристики швидкодії, але вищу складність впровадження. Асинхронні засоби, такі як Apache Kafka, RabbitMQ та інші, дозволяють роз'єднати компоненти, підвищити стійкість системи та ефективно обробляти великі обсяги даних. Kafka виділяється високою пропускною здатністю та можливістю обробки потокових даних у реальному часі, а RabbitMQ є гнучким і надійним засобом інтеграції для систем з ускладненою маршрутизацією повідомлень.

Розуміння переваг і недоліків різних підходів дозволяє розробникам приймати обґрунтовані рішення при проектуванні розподілених систем, забезпечуючи їх ефективну та надійну роботу. Також варто відмітити важливість контексту використання при виборі технології та необхідність врахування як функціональних, так і нефункціональних вимог.

Література

1. Oleh V. Talaver, Tetiana A.Vakaliuk. Reliable distributed systems: review of modern approaches // Journal of Engineering and Cybernetics. – 2022. – Т. 6, № 1. – С. 85. – URL: https://acnsci.org/journal/index.php/jec/article/download/586/657/1120
2. Коваленко А.Є. Розподілені інформаційні системи: навч. посіб. – К. : НТУУ «КПІ», 2011. – 256 с.
3. What is a distributed system // Atlassian. – URL: https://www.atlassian.com/microservices/microservices-architecture/distributed-architecture
4. Maltsev E., Muliarevych O.V. Evaluation of efficiency and performance of serialization formats for distributed systems // Komp'yuterni sistemi ta merezhi. – 2024. – № 2. – С. 155–156. – DOI: 10.23939/csn2024.02.141
5. Ritu, Arora S., Bhardwaj A., Kukkar A., Kaur S. A Comparative Analysis of Communication Efficiency: REST vs. gRPC in Microservice-Based Ecosystems // InnoComp 2024 Conference Proceedings, IEEE. – 2024. – DOI: 10.1109/innocomp63224.2024.00107
6. Таненбаум А.С., ван Стеен М. Розподілені системи: принципи та парадигми. – 2-ге вид. – Прентіс Холл, 2007. – 672 с.
7. Comparative Review of Selected Internet Communication Protocols // Foundations of Computing and Decision Sciences. – 2023. – Vol. 48, No. 1. – DOI: 10.2478/fcds-2023-0003
8. Documentation | gRPC – URL: https://grpc.io/docs/

9. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. – Sebastopol CA: O'Reilly Media, 2021. – 280 p.
10. Kommera A.R. The Power of Event-Driven Architecture: Enabling Real-Time Systems and Scalable Solutions // Turkish Journal of Computer and Mathematics Education. – 2020. – Vol. 11, No. 1. – DOI: 10.61841/turcomat.v11i1.14928
11. Apache Kafka – URL: https://kafka.apache.org/documentation/
12. Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines) // LinkedIn Engineering. – URL: : https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines
13. RabbitMQ Documentation | RabbitMQ – URL: https://www.rabbitmq.com/docs
14. Nguyen Quoc Uy, Vu Hoai Nam. A Comparison of AMQP and MQTT Protocols for Internet of Things // Proceedings of the 6th International Conference on Nature of Computation and Communication (NICS). – 2019. – DOI: 10.1109/NICS48868.2019.9023812

References

o
a 2. K
o 3. What is a distributed system // Atlassian. – URL: https://www.atlassian.com/microservices/microservices-architecture/distributed-architecture
a 4. Maltsev E., Muliarevych O.V. Evaluation of efficiency and performance of serialization formats for distributed systems // Komp'yuterni sistemi ta merezhi. – 2024. – № 2. – C. 155–156. – DOI: 10.23939/csn2024.02.141
e 5. Ritu, Arora S., Bhardwaj A., Kukkar A., Kaur S. A Comparative Analysis of Communication Efficiency: REST vs. gRPC in Microservice-Based Ecosystems // InnoComp 2024 Conference Proceedings, IEEE. – 2024. – DOI: 10.1109/innocomp63224.2024.00107
k 6. Tanenbaum A.S., van Steen M. Distributed Systems: Principles and Paradigms. – 2nd ed. – Prentice Hall, 2007. – 672 p.
o 7. Comparative Review of Selected Internet Communication Protocols // Foundations of Computing and Decision Sciences. – 2023. A Vol. 48, No. 1. – DOI: 10.2478/fcds-2023-0003
E 8. Documentation | gRPC – URL: https://grpc.io/docs/
D 9. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. – Sebastopol CA: O'Reilly Media, 2021. – 280 p.
r 10. Kommera A.R. The Power of Event-Driven Architecture: Enabling Real-Time Systems and Scalable Solutions // Turkish Journal of Computer and Mathematics Education. – 2020. – Vol. 11, No. 1. – DOI: 10.61841/turcomat.v11i1.14928
a 11. Apache Kafka – URL: https://kafka.apache.org/documentation/
t 12. Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines) // LinkedIn Engineering. – URL: : https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines
b 13. RabbitMQ Documentation | RabbitMQ – URL: https://www.rabbitmq.com/docs
fi 14. Nguyen Quoc Uy, Vu Hoai Nam. A Comparison of AMQP and MQTT Protocols for Internet of Things // Proceedings of the 6th International Conference on Nature of Computation and Communication (NICS). – 2019. – DOI: 10.1109/NICS48868.2019.9023812
E
d
g
n
f
o
e
m
a
n
g
o
a
n
s
d
y
s
C
y
m
s
textbook. – K.: NTUU "KPI", 2011. – 256 p.
n
e
t
i
c
s
.
–
2
0
2
2
.

–
T
.
6
,