DOI: https://doi.org/10.36910/6775-2524-0560-2025-59-21
УДК 004.42
Круглик Андрій Сергійович, аспірант
Мороз Дмитро Максимович, доктор філософії
https://orcid.org/0000-0003-2577-3352
Національний технічний університет «Дніпровська політехніка», м. Дніпро, Україна.

АНАЛІЗ ІСНУЮЧИХ ПРОГРАМНИХ РІШЕНЬ В ОБЛАСТІ ОБРОБКИ ПОВІДОМЛЕНЬ. БРОКЕРИ ПОВІДОМЛЕНЬ APACHE KAFKA ТА RABBIT MQ.

Круглик А. С., Мороз Д. М. Аналіз існуючих програмних рішень в області обробки повідомлень. Брокери повідомлень Apache Kafka та Rabbit MQ. У межах статті розглянуто можливості брокерів повідомлень Apache Kafka та Rabbit MQ до обробки повідомлень. Розглянуто можливості кожного з цих брокерів не лише з точки зору безпосередньої обробки повідомлень, але також інші можливості даного виду програмного забезпечення. Було проаналізовано принципові відмінності в архітектурному підході до побудови вказаних брокерів. Такий аналіз дозволяє визначити переваги та недоліки кожного з архітектурних підходів. Серед іншого, було також розглянуто можливості брокерів до масштабування, інтеграції, способів шифрування трафіку. Розглянуто можливості кожного з брокерів щодо обробки виключних ситуацій. Адже в умовах сьогодення ці та інші фактори вважаються де-факто обов'язковими вимогами що пред'являються до побудови сучасного програмного забезпечення. Проведений аналіз допоможе архітекторам та розробникам програмного забезпечення виокремити вдалі підходи при розробці нових рішень, а також при вдосконаленні існуючих рішень. Окрім того, в роботі показано практичне використання найбільш розповсюджень можливостей обох брокерів. Це повинно допомогти розробникам без досвіду роботи з даними брокерами швидко адаптуватись та приступити до використання даного ПЗ.
Ключові слова: Rabbit MQ, Apache Kafka, брокер повідомлень, черга повідомлень, producer, consumer, binding key, гарантії доставки.

Kruhlyk A., Moroz D. An analysis of existing software solutions in the field of message processing. Apache Kafka and RabbitMQ message brokers. This article examines the capabilities of the Apache Kafka and RabbitMQ message brokers in the context of message processing. The analysis considers not only the basic message handling functionalities of each broker but also explores the broader range of features provided by this category of software. A detailed comparison of the fundamental architectural approaches underlying each broker is presented, highlighting their respective advantages and limitations. Furthermore, the study addresses the brokers' abilities in terms of scalability, integration with external systems, and traffic encryption mechanisms. The brokers' approaches to handling exceptional situations are also evaluated, recognizing that in the contemporary landscape, such capabilities are increasingly regarded as de facto requirements for the development of modern software systems. The conducted analysis offers valuable insights for software architects and developers, aiding them in selecting effective design strategies for both the creation of new solutions and the enhancement of existing systems. In addition, the paper demonstrates the practical application of the most commonly used features of both brokers. This is intended to assist developers unfamiliar with these technologies in quickly adapting and beginning to work with the software efficiently.
Keywords: Rabbit MQ, Apache Kafka, message broker, message queue, producer, consumer, binding key, delivery guarantees.

Постановка проблеми. У роботах [1, 2] автори описали загальний алгоритм обробки інформаційних потоків повідомлень у системах літальних апаратів. Даний алгоритм базується на математичній моделі обробки інформаційних потоків з урахуванням цінності повідомлення і часу його старіння. Описаний алгоритм дозволяє в найбільш загальному випадку вирішити питання обробки інформаційних потоків. Але в сьогоднішніх реаліях така система не може вважатися повноцінною, тобто здатною до повного вирішення питання обробки повідомлень. Адже у сучасних умовах до систем обробки повідомлень пред'являються певні вимоги. Це і відмовостійкість, і здатність до відновлення у випадках збоїв, і можливості до масштабування, можливості шифрування трафіку та багато інших. Усі ці вимоги до розробки програмного забезпечення (ПЗ) диктуються умовами сьогодення, тож їх неможливо ігнорувати під час розробки сучасного ПЗ.

В даній роботі автори пропонують проаналізувати такий вид програмних засобів як брокери повідомлень (message brokers). В межах роботи пропонується проаналізувати можливості даного виду програмних засобів. Попередньо, було сформовано список запитань, відповіді на які потрібно отримати при розгляді брокерів повідомлень. Список запитань для аналізу наведено нижче.

- Архітектура брокерів повідомлень. Як відрізняються архітектури найбільш популярних брокерів повідомлень. Які способи обробки черг існують.
- Продуктивність та оптимізація. Які існують можливості для масштабування брокерів повідомлень. Які моделі доставки повідомлень існують.

- Які гарантії доставки є у різних брокерів. Як працюють механізми підтвердження повідомлень. Як обробляються ситуації при відмовах. Які стратегії відновлення черг існують при «падінні» брокера.
- Безпека. Які існують механізми авторизації чи автентифікації у брокерах повідомлень. Чи використовуються відомі протоколи шифрування, такі як SSL/TSL у сучасних брокерах повідомлень. Які механізми використовуються для логування подій у брокерах.
- Чи можна інтегрувати брокери повідомлень із зовнішніми технологіями. В першу чергу цікавить питання інтеграції із системами управління базами даних (СУБД), так як повідомлення повинні мати можливість зберігання і накопичення.
- Практичні приклади побудови систем з використанням брокерів повідомлень. Приклади коду на різних мовах програмування, що спростить базові налаштування та використання того чи іншого брокеру в конкретних сценаріях.

У підсумку автори розраховують, що такий аналіз дозволить сформувати коректні вимоги до розробки повноцінної системи обробки інформаційних потоків в системах доставки літальними апаратами. Уникнути типових помилок при розробці рішень стосовно того чи іншого аспекту даного виду програмних засобів, та, за можливості, використати найкращі практики до побудови повноцінного ПЗ.

Аналіз останніх досліджень і публікацій. Брокери повідомлень є досить відомим ПЗ, що охоплює достатньо широке коло застосування. З цього випливає, що таке ПЗ є достатньо дослідженим у багатьох сценаріях застосування. Достатньо широка дослідженість даного виду ПЗ дозволяє зробити попередній висновок, що брокери повідомлень широко застосовуються при розробці повноцінних систем як допоміжне ПЗ, і відповідно, існують достатньо документовані приклади використання та способи вирішення тих чи інших проблем.

Серед подібних робіт хотілось би виділити роботу [3], де автор проводить базовий аналіз декількох сучасних фреймворків обробки повідомлень, таких як Apache Kafka, Active MQ Artemis, RabbitMQ, Redis. У роботі наведено порівняльний аналіз цих фреймворків, їхні базові можливості. Розглянуто можливості маршрутизації повідомлень та способи взаємодії між брокером та споживачем. Під способом взаємодії хотілось би акцентувати увагу на підходи, за допомогою яких взаємодіють брокер та споживач, вказана принципова відмінність в цій взаємодії (Pull/Push – based approach). Наведено практичний приклад використання фреймворків у розробці ПЗ для роботи з повідомленнями. Іншим цікавим матеріалом є робота [13], де автор проводить порівняльний аналіз таких брокерів повідомлень як RabbitMQ та Apache Kafka. Розповідає про їх архітектурні відмінності а також сфери можливого застосування. Описує протоколи роботи брокерів та можливості для розширення в загальному виді. Також було проаналізовано роботу [14], де автори приводять практичний кейс до побудови ПЗ з використанням RabbitMQ брокера повідомлень. Приклади коду на мові програмування C# дають можливість інженерам ПЗ оцінити загальний принцип побудови мікросервісної архітектури, де у якості зв'язку буде використовуватись допоміжний програмний засіб – брокер повідомлень.

Постановка завдання. Аналіз існуючих найбільш популярних брокерів повідомлень для оцінки сучасного стану програмних засобів обробки черг повідомлень, їх можливостей, обмежень, розповсюджених сфер та практик використання.

Викладення основного матеріалу дослідження. В загальному вигляді брокер повідомлень, це програма-посередник, що забезпечує комунікацію між окремими сервісами, модулями, системами. Брокер повідомлень являє собою тип побудови архітектури ПЗ, при якому елементи системи «спілкуються» одна з одною за допомогою посередника – брокера повідомлень. Брокер повідомлень перекладає повідомлення з формального протоколу обміну повідомленнями відправника на формальний протокол обміну повідомленнями одержувача. Таким чином, окремі сервіси, служби, модулі чи системи можуть комунікувати між собою навіть якщо вони реалізовані на різних платформах чи написані на різних мовах[3, 4]. Брокери повідомлень використовують для передачі, зберігання, перевірки та доставки повідомлень. Основна задача брокера повідомлень бути посередником між взаємозалежними сервісами, завдяки чому досягається відокремленість процесів всередині системи [3]. В роботі будь-якого брокера повідомлень використовуються як мінімум дві обов'язкові сутності: producer – сутність, що надсилає повідомлення, або виробник повідомлень, та consumer – сутність що споживає, або обробляє повідомлення, іншими словами споживач повідомлення.

Як правило, для комунікації між брокером та виробником/споживачем використовується одна з двох наступних моделей.

Point-to-point - при такій моделі комунікації один або декілька відправників комунікують через брокер з одним або декількома споживачами. При цьому така модель взаємодії гарантує, що кожне повідомлення буде надіслано та оброблено лише один раз [5, 6]. Також ця модель гарантує, що повідомлення буде обов'язково оброблене. Навіть якщо споживач не доступний у даний час, повідомлення буде надіслано йому як тільки відновиться зв'язок. Як правило, для обробки повідомлень у цьому випадку використовується черга типу FIFO (first in first out). При такому способі організації черги повідомлення будуть оброблені у тій же послідовності, що були відправлені виробником повідомлень.

Publish-Subscribe – модель комунікації при якій publisher (виробник) публікує повідомлення в тему, а subscriber (підписник, або споживач повідомлення), отримує повідомлення із тих тем, на які він підписаний. У даній моделі декілька виробників можуть відправляти повідомлення в одну й ту ж тему. Те ж саме стосується і підписників: кожне повідомлення, що надійшло в певну тему, буде відправлено усім споживачам, підписаним на дану тему. Дана модель не гарантує, що повідомлення буде доставлено усім підписникам.

Існує дві моделі доставки повідомлень: push based та pull based. Pull based модель доставки повідомлень працює наступним чином: брокер повідомлень очікує на запит від споживача. Тобто, усі повідомлення накопичуються в черзі самого брокера. Push based модель дозволяє брокеру відправити усі повідомлення відразу споживачеві. Ці відмінності потрібно обов'язково враховувати при розробці системи, оскільки в останньому випадку потрібно архітектурно передбачити можливість накопичення повідомлень в ПЗ споживача.

Існує декілька типів гарантій доставки повідомлень. At most once – повідомлення може не дійти, але ніколи не буде продубльовано. At least once – повідомлення обов'язково буде доставлено, але може бути продубльовано. Exactly once – буде доставлено раз і тільки раз.

Одним з найрозповсюдженіших на сьогоднішній день брокерів повідомлень є Apache Kafka [7]. Сам брокер є досить відомим, добре документованим та має широке ком'юніті. Тож встановлення, базові налаштування даного брокера виходять за межі даної роботи. Пропонується розглянути загальний принцип роботи та проаналізувати можливості обробки повідомлень і черг повідомлень за допомогою Apache Kafka.

Apache Kafka є лог-орієнтованим брокером подій (log-based event broker). Це означає що даний брокер зберігає повідомлення на диск комп'ютера. Окрім того, після прочитання повідомлення споживачем, воно відразу не видаляється, а зберігається певний час залежно від налаштувань, за замовчуванням 1 тиждень. Лог-орієнтована архітектура означає, що повідомлення від виробника будуть потрапляти не в чергу, а в певний лог файл, звідки їх можна буде прочитати одним або декількома споживачами в залежності від конфігурації системи. Схема роботи лог-орієнтованої архітектури з декількома виробниками подій та одним споживачем зображена на рис. 1.

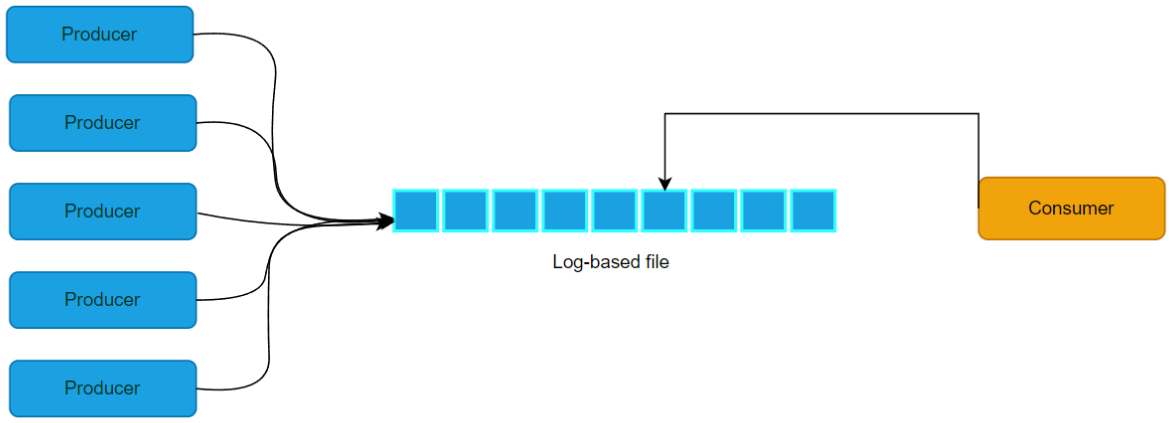

Рис. 1. Лог-орієнтована модель обробки повідомлень.

На рис. 1 видно, як декілька виробників додають повідомлення у лог-файл, а споживач по черзі прочитує повідомлення з лог-файлу і обробляє його. Після зчитування повідомлення споживач зміщується у лог-файлі на одну позицію, що дорівнює величині попереднього повідомлення, або оффсет. Конструкція здається простою, доки існує лише один споживач. Але даний брокер підтримує дуже велику кількість споживачів і виробників одночасно. Звісно кожне повідомлення може оброблятися різний час. Відповідно, потрібно мати механізм, який забезпечить синхронізацію між різними споживачами, що дозволить уникати повторної обробки раніше обробленого повідомлення. Згідно із роботою [11], для вирішення цього завдання Apache Kafka використовує механізм так званого курсору, який знаходиться безпосередньо всередині брокера повідомлень і дозволяє безпечно зміщуватись в лог-файлі незалежно від швидкості обробки кожного окремого повідомлення.

Як було сказано раніше, логи після їх обробки споживачами не видаляються відразу а певний час зберігаються. Така особливість дозволяє проводити аналіз роботи брокеру, аналіз повідомлень, середній час обробки повідомлення і т. ін. Це у свою чергу дозволить приймати певні рішення стосовно масштабування всієї системи у випадках, коли навантаження є піковим. Серед мінусів такого аналізу виявилось те, що аналіз логів є відносно повільною операцією і більш складною, якщо порівнювати наявність в іншому брокері графічного інтерфейсу для перегляду його роботи в режимі реального часу.

Даний брокер працює за принципом publish-subscribe. Усі споживачі, що будуть обробляти повідомлення, повинні бути підписаними на одну тему (topic). Такий архітектурний підхід дозволяє легко масштабуватись горизонтально через шардінг або партиції. Брокер використовує pull based модель доставки повідомлень.

Згідно з офіційною документацією Kafka підтримує усі три види доставки повідомлень: at most once, at least once, exactly once. Також, Apache Kafka підтримує механізм підтвердження доставки повідомлень. Підтримується як механізм підтверджень доставки повідомлень від виробника брокеру, так і механізм підтверджень доставки від брокера споживачеві. В з'єднанні виробник – брокер, виробник надсилає повідомлення з параметром acks. Даний параметр може мати одне з трьох значень: acks = 0 – виробник не очікує підтвердження доставки, acks = 1 – виробник чекає підтвердження від лідера партиції, acks = all – очікує доки всі репліки підтвердять отримання повідомлення. Стосовно механізму підтвердження отримання та обробки повідомлення споживачем від брокера, Kafka не потребує такого підтвердження, тому що повідомлення не видаляються з лог-файлу після обробки і є можливість обробити його знову. Натомість, Kafka використовує механізм комітів, тобто підтвердження обробки повідомлення і зміни офсету в лог-файлі. Такий механізм працює у двох режимах: автоматичний коміт (auto.commit = true) – брокер автоматично комітить через деякі проміжки часу. Такий механізм простіший у використанні, але може призвести до ситуації, коли повідомлення було отримано споживачем, але не оброблено через виникнення непередбачуваної помилки і «падіння» споживача. В такій ситуації повідомлення фактично не було оброблено. Тож в Kafka існує також механізм ручного коміту (auto.commit = false) – механізм, при якому споживач повинен в ручному режимі підтверджувати отримання повідомлення. Такий механізм може гарантувати, що повідомлення було отримано та оброблено, до виклику коміту і відповідно офсету в лог-файлі.

Брокер має добре продуману стратегію відновлення у випадку «падіння» брокера. Так як в якості черг використовується лог-орієнтований розподілений журнал, черги (топіки) можуть бути легко відновлені завдяки реплікації. Кожна черга має n реплік на інших брокерах. Брокери запускаються в наступному порядку – один лідер або ведучий, інші – ведені. Kafka Controller – програмний модуль брокеру, що відповідає за стан брокерів. Тобто саме цей модуль має можливість визначати коли лідер активний і працює. Якщо лідер «падає», Kafka автоматично обирає нового. Окрім того, Kafka використовує не тільки брокер безпосередньо, але також Zookeeper – допоміжне ПЗ яке використовується для координації між брокерами а також відновлення ролей брокерів у випадку «падіння». Тобто, стратегія відновлення наступна: «падає» брокер, Kafka Controller бачить зміну і переобирає лідера. Після цього робота відновлюється. Коли попередній брокер підніметься, відбудеться синхронізація лог-файлу і робота продовжиться.

Відповідно до документації у вказаного брокера є підтримка автентифікації та авторизації. Для автентифікації використовуються наступні механізми:

- SASL (Simple Authentication and Security Layer).

- SSL (Secure Sockets Layer).
- Kerberos.

Після автентифікації, для визначення дій, які дозволено робити користувачеві, брокер підтримує авторизацію. Для забезпечення цієї функціональності брокер підтримує ACL (Access Control Lists) для обмеження доступу до топіків, груп споживачів та адміністративних команд.

Окрім сказаного вище, Apache Kafka може бути інтегрованим із зовнішніми системами такими як СУБД. Для цього брокер використовує Kafka Connect – вбудований механізм для інтеграції із зовнішніми системами, включаючи SQL та NoSQL бази даних. Найбільш популярними коннекторами для баз даних є Debezium [9] – який відстежує зміни в реляційних базах даних (БД), та JDBC Connector – що забезпечує роботу читання/запису в SQL бази даних. Це не єдині коннектори, що можуть використовуватися в даному брокері, є також спеціалізовані, що працюють лише з певною БД, такі як MongoDB Connector та інші.

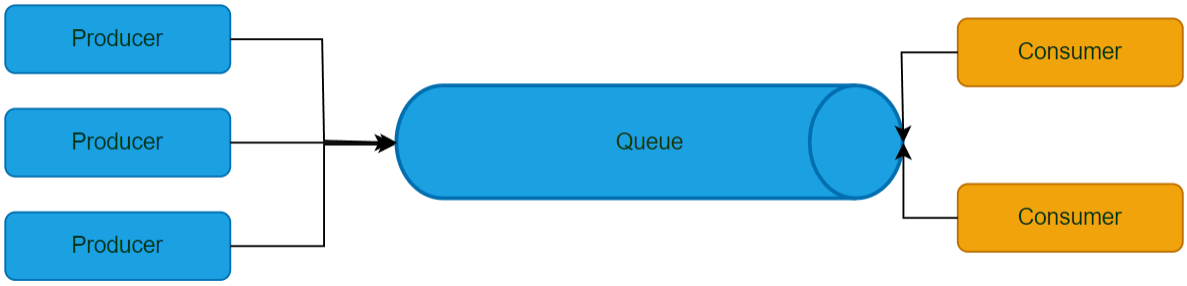

Рис. 2. Архітектура брокера повідомлень із класичною чергою

Іншим досить відомим брокером повідомлень є RabbitMQ. Він має принципові відмінності від попереднього як за архітектурними характеристиками, так і за більшістю інших показників. Даний брокер використовує класичну чергу повідомлень, де черга зберігається або в оперативній пам'яті комп'ютера, або на диску (залежить від налаштувань). Така архітектура забезпечує зберігання повідомлення лише до того часу, доки повідомлення не було оброблене і, при необхідних налаштуваннях, не було підтвердження оброблення повідомлення. Після цього спожиті повідомлення видаляються з черги.

Така конструкція працює за принципом FIFO (first in first out). Тобто повідомлення по черзі забираються з черги, обробляються і видаляються.

На відміну від лог-орієнтованої моделі обробки повідомлень, RabbitMQ не зберігає усі повідомлення. Тож детально проаналізувати інформаційні потоки що надходять, або нещодавно надходили буде складніше. Але, в той же час хотілось би акцентувати увагу на тому, що даний брокер повідомлень має графічний інтерфейс користувача, який, на відміну від логів дозволить у зручному форматі переглядати роботу брокера у режимі реального часу (рис. 3)

Даний брокер також підтримує масштабування. Згідно із матеріалами [12], RabbitMQ можна масштабувати за допомогою маршрутизації. Для цього можна використовувати ключі маршрутизації (routing key). Є декілька видів маршрутизації. Fanout Exchange – повідомлення буде скопійоване до всіх черг, що пов'язані з точкою обміну. Direct Exchange - повідомлення відправляється лише у відповідну чергу за умови, якщо binding-ключ такої черги буде рівний ключу маршрутизації повідомлення (routing key). Topic Exchange - При такому типу маршрутизації, binding-ключ черги може задаватись як регулярний вираз. Отже, якщо routing ключ повідомлення покривається регулярним виразом binding-ключа - повідомлення потрапляє у відповідну чергу. Див. приклад рис. 4

Брокер використовує push based модель доставки повідомлень. Коли повідомлення надходить в чергу, брокер відразу шле його споживачеві, якщо останній підключений. Згідно із документацією обсяг повідомлень можна регулювати за допомогою параметру prefetch, щоб вказати скільки повідомлень може бути доставлено одночасно споживачеві. Це необхідно у випадках пікових навантажень щоб не перевантажити споживача.

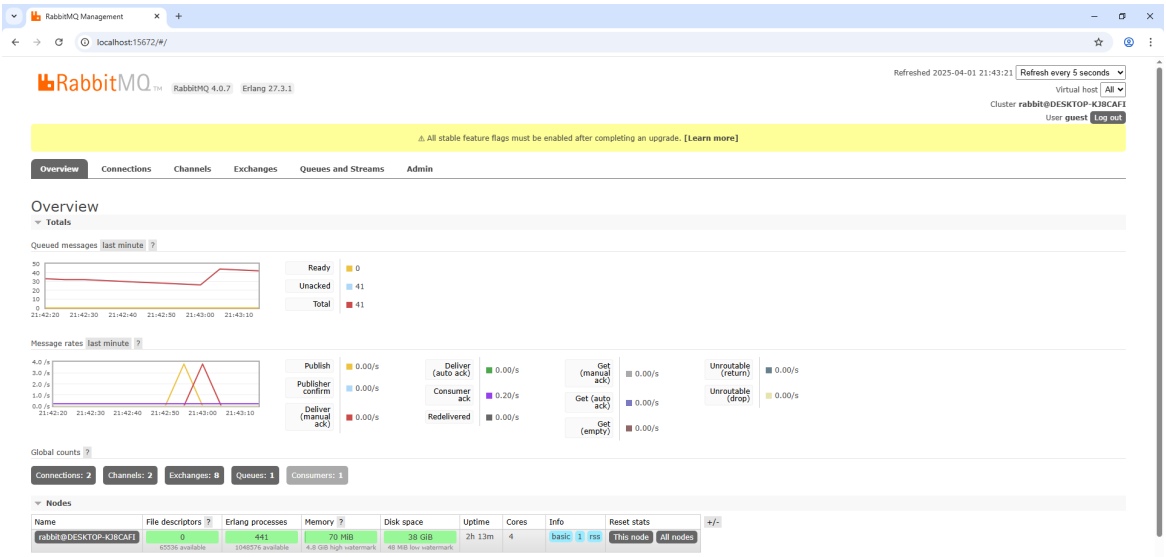

Рис. 3. Графічний інтерфейс брокеру повідомлень RabbitMQ

Згідно з офіційними джерелами RabbitMQ із коробки підтримує два види доставки повідомлень: at most once, at least once. Іншим важливим моментом, який цікавить авторів, є можливість підтвердження доставки повідомлень брокером. RabbitMQ підтримує доставку повідомлень від продюсера брокеру. У цьому випадку виробник буде очікувати підтвердження доставки повідомлення від брокера у вигляді одного з двох типів повідомлень: ack – тобто повідомлення успішно доставлено, або nack – сталася помилка при обробці повідомлення можливостями брокера. Також, RabbitMQ підтримує підтвердження доставки повідомлення від брокера до споживача. У такому випадку, коли споживач обробив повідомлення, він має надіслати ack. Якщо ack не отримано брокером – повідомлення повертається до черги або знищується, залежно від налаштувань [10]. Окрім того, для підтвердження доставки і обробки повідомлень споживачем існує два способи – перший, автоматичне підтвердження (autoAck = true), повідомлення вважається доставленим як тільки воно було отримано споживачем. Автоматичне підтвердження частково спрощує налаштування, але може мати неочікувані наслідки, такі як втрата повідомлення у випадку збою роботи споживача. Другий спосіб підтвердження доставки повідомлення, це ручне підтвердження (autoAck = false). В цьому випадку споживач повинен обов'язково надіслати підтвердження обробки повідомлення.

Відповідно до документації RabbitMQ також має стратегію відновлення при «падінні». По-перше, є можливість зберігати черги на диск. Для цього використовується опція durable = true. Брокер підтримує кластеризацію, іншими словами розподілення на декілька вузлів. Якщо один вузол «падає» - інші вузли мають копії. Відповідно використовується наступна стратегія відновлення: якщо брокер «падає», решта кластеру продовжує працювати. Коли брокер перезапустився, він синхронізується з іншими вузлами, підключає споживачів повідомлень і продовжує роботу.

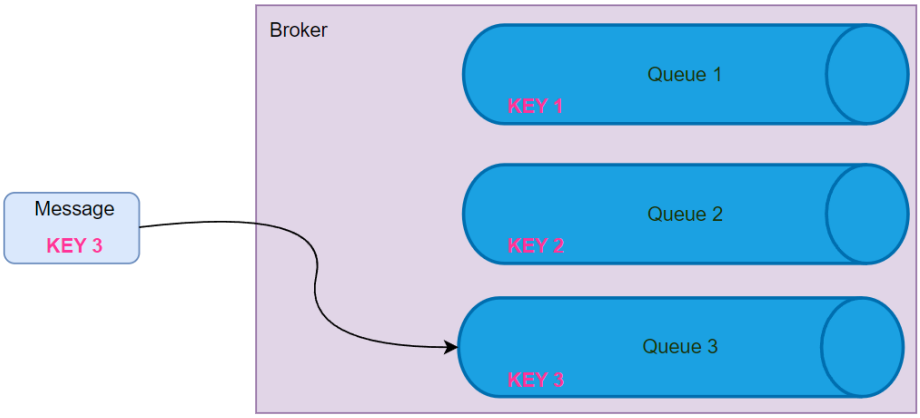

Рис. 4. Приклад маршрутизації черги за допомогою routing key

RabbitMQ має підтримку декількох способів автентифікації. Першим способом є вбудована автентифікація (Built-in), яка працює «з коробки» та може налаштовуватись через веб інтерфейс. Другим способом налаштування автентифікації є підключення зовнішнім плагінів, таких як LDAP або OAuth. Після автентифікації брокер контролює доступ до ресурсів за допомогою політики доступу.

Проаналізувавши відповідні джерела стає зрозуміло, що RabbitMQ напряму не може бути інтегрованим із СУБД безпосередньо, але може використовуватись як допоміжне ПЗ для організації обробки черг повідомлень у системах, де споживач та/або виробник можуть мати безпосереднє підключення до баз даних.

Окрім наведеного вище аналізу, було також протестовано декілька практичних кейсів, для апробацій певних тверджень технічної документації. Звісно в межах роботи неможливо покрити усі можливі варіанти подій, тож автори сфокусувалися на декількох найбільш загальних тестових сценаріях для розуміння принципів роботи вказаних брокерів повідомлень, можливостей для написання вихідного коду для взаємодії із цими брокерами.

По аналогії з попередніми викладками спершу протестуємо брокер повідомлень Apache Kafka. Програмні інструменти що використовуються в тестуванні, окрім безпосередньо брокера: Producer – програмне забезпечення, написане на мові програмування Java з використанням фреймворку Spring. Producer має можливість комунікувати з користувачем через http протокол. Це надає можливість керувати генерацією повідомлень з кожного брокера в порядку та з параметрами, що планується протестувати. Consumer - програмне забезпечення, написане на мові програмування Java з використанням фреймворку Spring. Consumer отримує повідомлення від брокера, затримує потік виконання на середній час і виводить повідомлення про успішну обробку. Curl – утиліта командного рядка для виконання HTTP-запитів. Вона дозволяє взаємодіяти з веб-сервісами, API, завантажувати або відправляти дані через різні протоколи (HTTP, HTTPS, FTP тощо). Цією утилітою буде відбуватись керування виробниками.

Вихідний код виробника повідомлень:

```
//файл конфігурації:
@Configuration
public class ProgramConfig {

   @Value("${spring.kafka.bootstrap-servers}")
   private String bootstrapServers;

   @Bean
   public ProducerFactory<String, String> producerFactory() {
      Map<String, Object> configProps = new HashMap<>();
      configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
      configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
      configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
      return new DefaultKafkaProducerFactory<>(configProps);
   }

   @Bean
   public KafkaTemplate<String, String> kafkaTemplate() {
      return new KafkaTemplate<>(producerFactory());

   }

}
```

Контролер виробника повідомлень. Саме він буде отримувати повідомлення від користувача (тобто від нас) та відправляти ту кількість повідомлень до брокера, що задана параметром. Обидва параметри, повідомлення та їхня кількість задаються через http запити.

```
@RestController
@RequestMapping("/api/kafka")
public class KafkaController {
```

```
    private final KafkaProducer kafkaProducer;

    public KafkaController(KafkaProducer producer)
    {
        this.kafkaProducer = producer;
    }

    @PostMapping("/send")
    public String sendMessage(@RequestParam(required = true) String message,
                              @RequestParam(required = false, defaultValue = "1") String count)
    {
        int messagesToSend = Integer.parseInt(count);
        for (int i = 0; i < messagesToSend; ++i)
        {
            kafkaProducer.sendMessage(message);
        }

        return ("Message: " + message + " sent from Producer " + messagesToSend + " times");
    }
}
```

Сервіс виробника для взаємодії з брокером:

```
@Service
public class KafkaProducer {

    private static int messageId = 0;
    private final KafkaTemplate<String, String> kafkaTemplate;

    public KafkaProducer(KafkaTemplate<String, String> kafkaTemplate)
    {
        this.kafkaTemplate = kafkaTemplate;
    }

    public void sendMessage(String message)
    {
        String fullMessage = "Message ID: " + String.valueOf(messageId) + "message: " + message;
        kafkaTemplate.send("test-topic", fullMessage);
        System.out.println("Message sent: " + fullMessage);
        ++messageId;
    }
}
```

Вихідний код отримувача повідомлень. В цьому сервісі ми імітуємо ресурсозатратний процес, що триває 5 секунд:

```
@Service
public class KafkaConsumer {

    @KafkaListener(topics = "test-topic", groupId = "test-consumer-group")
    public void listen(String message) throws InterruptedException {
        System.out.println("Get message " + message);
        Thread.sleep((long) 5000);
    }
}
```

Тестовий сценарій (ТС1): нехай декілька виробників повідомлень почергово генерують певну кількість повідомлень, кількість залежить від вхідних параметрів. В даному сценарії відправимо 5 повідомлень з кожного виробника. Кожне повідомлення є унікальне, тобто має свій порядковий номер, що не повинен повторюватись в межах одного виробника. Це дозволить

відслідковувати порядок надходження повідомлень в чергу брокера. Що у свою чергу дозволить зробити висновки про можливість чи неможливість обробляти повідомлення згідно з пріоритетом чи іншими характеристиками, в залежності від можливостей налаштувань брокера. Брокер отримує повідомлення та відправляє його споживачу. Споживач імітує ресурсоємну роботу, тобто кожне повідомлення обробляється певний час. В цьому конкретному сценарії обумовимось, що середній час обробки повідомлення споживачем складає 5 секунд. Час обробки повідомлення будемо вважати середнім для всіх повідомлень.

Запустимо брокера:

```
bin\windows\zookeeper-server-start.bat config\zookeeper.properties
bin\windows\kafka-server-start.bat config\server.properties
```

Створимо тему. Тут треба врахувати що назва теми (topic), повинна співпадати в брокері-виробникові-споживачеві. Це дозволить правильно організувати процес обробки повідомлень.

```
bin\windows\kafka-topics.bat --create --topic test-topic --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
```

Перевіримо список тем:

```
bin\windows\kafka-topics.bat --list --bootstrap-server localhost:9092
```

Тепер запустимо 3 виробників на різних портах і споживача.

```
java -jar target/Producer.jar --server.port=8081
java -jar target/Producer.jar --server.port=8082
java -jar target/Producer.jar --server.port=8083
java -jar target/Consumer.jar
```

Відправимо 5 повідомлень з кожного виробника почергово, не очікуючи завершення обробки повідомлень споживачем:

```
curl -X POST "http://localhost:8081/api/kafka/send?message=HelloFromFirstBroker&count=5"
curl -X POST "http://localhost:8082/api/kafka/send?message=HelloFromSecondBroker&count=5"
curl -X POST "http://localhost:8083/api/kafka/send?message=HelloFromThirdBroker&count=5"
```

У виводі споживача отримаємо наступні повідомлення:

```
Get message Message ID: 0 message: HelloFromFirstBroker
Get message Message ID: 1 message: HelloFromFirstBroker
Get message Message ID: 2 message: HelloFromFirstBroker
…
Get message Message ID: 0 message: HelloFromSecondBroker
Get message Message ID: 1 message: HelloFromSecondBroker
Get message Message ID: 2 message: HelloFromSecondBroker
…
Get message Message ID: 0 message: HelloFromThirdBroker
Get message Message ID: 1 message: HelloFromThirdBroker
Get message Message ID: 2 message: HelloFromThirdBroker
```

Запустивши на виконання систему 5 разів, результат був таким самим. Звідси можна зробити попередній висновок, що налаштування брокера «із коробки» не розподіляють повідомлення ніяким чином, тобто повідомлення потрапляють у лог-файл згідно із надходженням у брокер і будуть оброблені за принципом FIFO.

ТС2. Продовжимо попередній сценарій. Вхідні умови та змінні залишимо без змін. Змінимо лише взаємодію брокера зі споживачем. Припустимо, споживач періодично виходить з ладу в процесі обробки повідомлення, а повідомлення продовжують надходити. Через деякий час запустимо споживача. Запустивши програму на виконання, та перезапустивши споживача 10 разів було проаналізовано його вихідну інформацію. Згідно з цією інформацією, consumer продовжує працювати з наступним повідомленням, що надійшло в чергу брокера. Тобто у випадках, коли виробник повідомлень продовжує надсилати повідомлення брокеру, а споживачі відсутні – брокер накопичує їх у лог-файлах і почне відправляти споживачам, щойно останні під'єднаються. Цей сценарій дає певне розуміння того, яким чином можна обробляти помилки при роботі споживача, а також яким способом можливо перерозподілити повідомлення між споживачами у моменти пікових навантажень.

Окрім того, було проведено аналіз можливостей логування роботи брокера. Згідно з документацією Apache Kafka має 3 рівні логування: INFO, ERROR, DEBUG. За замовчуванням

використовується рівень логів INFO. Це дає змогу проглядати роботу брокера в режимі реального часу і аналізувати нетипові події, що виникають. При цьому рівні логування не видно інформації, що consumer відключився. Але в той же час видно в файлах логів, коли споживач підключається. З чого можна зробити опосередковані висновки, щодо відключень споживача, тобто фактичного кінцевого обробника повідомлень і продовжити досліджувати причини таких відключень безпосередньо в логах самого споживача.

ТС3. Розглянемо сценарій, коли в процесі роботи виникають непередбачувані ситуації, і споживач через збій не може обробити повідомлення. Пропонується розглянути такий вид збою, де повідомлення саме по собі є правильним, тобто може бути оброблено повторно. Згідно з документацією до фреймворка, Apache Kafka має налаштування щоб повторно надсилати повідомлення, якщо ті не були оброблені. Зімітуємо ситуацію, коли кожне третє повідомлення призводить до збоїв. Для імітації збоїв, а також підтвердження обробки повідомлень, змінимо вихідний код Consumer'a.

```
@Service
public class KafkaConsumer {

   static int num = 0;

   @KafkaListener(topics = "test-topic", groupId = "my-consumer-group")
   public void listen(String message, Acknowledgment acknowledgment) throws InterruptedException {
      System.out.println("Get message " + message);
      Thread.sleep((long) 5000);
      if(++num % 3 == 0)
      {
         System.out.println("Unexpected exception");
         throw new InterruptedException("Unexpected exception");
      }
      System.out.println("Processed message " + message);
      acknowledgment.acknowledge();
   }
}
```

Окрім того, змінимо файл конфігурації споживача. Це потрібно для того, щоб правильно обробляти повідомлення і не зациклюватись на одному, навіть якщо збою не виникло:

```
spring.kafka.consumer.enable-auto-commit=false
spring.kafka.consumer.auto-offset-reset=latest
spring.kafka.listener.ack-mode=manual_immediate
```

За допомогою утиліти curl відправлено 30 повідомлень для брокера. У виводі споживача отримаємо наступні логи:

```
…
Get message Message ID: 23 message: HelloFromSecondBroker
Processed message Message ID: 23 message: HelloFromSecondBroker
Get message Message ID: 24 message: HelloFromSecondBroker
Unexpected exception
…
Get message Message ID: 24 message: HelloFromSecondBroker
Processed message Message ID: 24 message: HelloFromSecondBroker
Get message Message ID: 25 message: HelloFromSecondBroker
Processed message Message ID: 25 message: HelloFromSecondBroker
Get message Message ID: 26 message: HelloFromSecondBroker
Unexpected exception
…
Get message Message ID: 26 message: HelloFromSecondBroker
Processed message Message ID: 26 message: HelloFromSecondBroker
```

Як видно з логів споживача, якщо повідомлення не було оброблено, тобто, якщо не викликано метод acknowledgment.acknowledge(); Apache Kafka вважає що повідомлення не було

доставлено і відповідно не змінює офсет у лог-файлі. Це повинно гарантувати, що жодне повідомлення не буде втрачено.

Наступним протестуємо брокер повідомлень RabbitMQ. Інструменти використовуються ті ж, що й у попередньому тестуванні зі змінами, доданими відповідно можливості використання із цим брокером. Окрім того, додамо незначних змін стосовно самих повідомлень, адже згідно із документацією, RabbitMQ (як і Apache Kafka) має можливість обробляти більш складні дані, аніж прості текстові рядки. Тож для прикладу використаємо JSON формат для передачі повідомлень в системі. Пропонується використати простий JSON об'єкт User, з наступними полями:
{"name": "Andrii", "age": 30}

Налаштування Producer'а в application.properties, для можливості підключення до брокера:

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
rabbitmq.queue=myJsonQueue
rabbitmq.exchange=myJsonExchange
rabbitmq.routingKey=myJsonRoutingKey
```

Контролер для взаємодії з користувачем.

```
@RestController
@RequestMapping("/api/rabbitmq")
public class MessageController {

   private final RabbitMQJsonProducer rabbitMQJsonProducer;

   MessageController(RabbitMQJsonProducer rabbitMQJsonProducer) {
      this.rabbitMQJsonProducer = rabbitMQJsonProducer;
   }

   @PostMapping("/send")
   public String sendJsonMessage(@RequestBody User user) {
      rabbitMQJsonProducer.sendJsonMessage(user);
      return "JSON message sent";
   }
}
```

Сервіс відправника:

```
@Service
public class RabbitMQJsonProducer {

   private final RabbitTemplate rabbitTemplate;
   private final ObjectMapper objectMapper;

   @Value("${rabbitmq.exchange}")
   private String exchange;

   @Value("${rabbitmq.routingKey}")
   private String routingKey;

   public RabbitMQJsonProducer(RabbitTemplate rabbitTemplate, ObjectMapper objectMapper) {
      this.rabbitTemplate = rabbitTemplate;
      this.objectMapper = objectMapper;
   }

   public void sendJsonMessage(User user) {
      try {
```

```
            String jsonMessage = objectMapper.writeValueAsString(user);
            System.out.println("Sent JSON message to broker: " + jsonMessage);
            rabbitTemplate.convertAndSend(exchange, routingKey, jsonMessage);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Споживач повідомлень має такі ж налаштування у application.properties для взаємодії з брокером, що й виробник, тож немає необхідності їх дублювати. Вихідний код сервісу споживача повідомлень:

```
@Service
public class RabbitMQJsonConsumer {

    private final ObjectMapper objectMapper;

    public RabbitMQJsonConsumer(ObjectMapper objectMapper) {
        this.objectMapper = objectMapper;
    }

    @RabbitListener(queues = "${rabbitmq.queue}")
    public void receiveJsonMessage(String message) {
        try {
            User user = objectMapper.readValue(message, User.class);
            System.out.println("Message received" + user);
            Thread.sleep((long)5000);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Повторимо тестові сценарії для даного брокера. TC1 - відправимо декілька повідомлень за допомогою утиліти curl виробнику повідомлень. В якості об'єкту JSON використаємо вміст файлу testentity.txt із вмістом, описаним вище.

```
curl -X POST http://localhost:8081/api/rabbitmq/send -H "Content-Type: application/json" -d @testentity.txt
```

Немає сенсу додавати подібні логи до цієї роботи. Проаналізувавши їх, автори зробили висновок подібний попередньому брокеру – з коробки, без налаштувань, повідомлення обробляються у порядку надходження. Окрім того, хотілося б звернути увагу на графічний інтерфейс даного брокера (див. рис. 3), в якому візуально можна побачити роботу брокера, підключених споживачів і т.д.

Повторимо тестовий сценарій 2, використаний для попереднього брокера. На відміну від Apache Kafka, у даному брокері при налаштуваннях «із коробки» в графічному інтерфейсі користувача є закладка Queues and Streams, в якій можна подивитись у режимі часу, наближеного до реального, скільки на даний момент споживачів підключено до брокера. Див. рис. 5, поле Consumers. При працюючій системі, зображеній на рис. 5 відправимо 10 повідомлень споживачу. Згідно зі статистикою на Overview вкладці ГУІ інтерфейсу видно, що повідомлення надійшли до брокера. Після того, як споживача було підключено до системи, повідомлення почали оброблятися згідно із записами на ГУІ інтерфейсі.

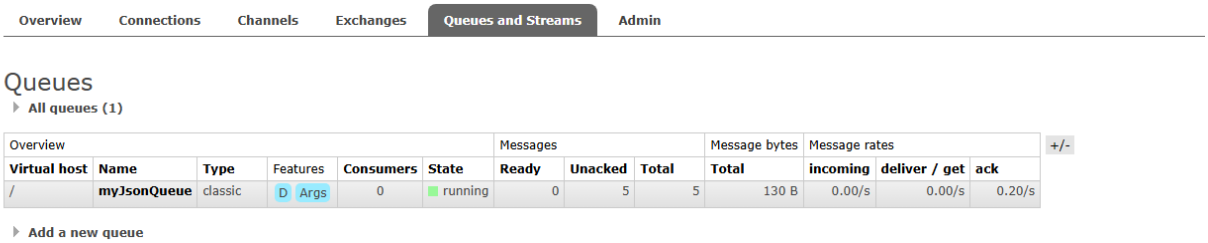

Рис. 5. Перегляд налаштувань брокера RabbitMQ. Кількість підключених споживачів.

Протестувавши також ТС 3 було підтверджено, що RabbitMQ також має можливості для гарантій обробки повідомлень споживачем. Щоб уникнути дублювань коду а також вихідних логів, надамо лише пояснення змін. По-перше, для ручного підтвердження обробки повідомлення споживачем, в його налаштування потрібно додати параметр AcknowledgeMode.MANUAL, що у свою чергу змусить в ручному режимі викликати підтвердження обробки повідомлення в сервісі споживача: channel.basicAck(tag, false), або у випадку помилки обробки, для повернення повідомлення в чергу - channel.basicNack(tag, false, true).

Висновки та перспективи подальших досліджень. У роботі було проаналізовано такий вид програмних засобів як брокери повідомлень. Аналіз проводився на двох принципово різних за архітектурою брокерах повідомлень RabbitMQ та Apache Kafka. Такий аналіз дозволив більш ширше зрозуміти можливості даного виду ПЗ. Зрозуміти, які існують архітектурні підходи до створення брокерів повідомлень. Які можливості доставки повідомлень і способи підтвердження доставки використовуються у сучасних системах. Проаналізовано сучасні підходи до забезпечення захищеності в роботі брокерів за допомогою сучасних методів шифрування.

Дана робота дозволить більш точно сформувати вимоги до розробки власної системи обробки інформаційних потоків в системах доставки літальними апаратами. Після практичного тестування роботи брокерів можна з упевненістю сказати, що для побудови повноцінної системи обробки потоків повідомлень, брокер можна використовувати як допоміжне ПЗ при побудові системи з використанням архітектурної моделі Message-oriented middleware (MOM), коли розподілена система використовує повідомлення для зв'язку між вузлами чи модулями.

Список бібліографічного опису

1. Мороз Б. І., Круглик А. С., Мороз Д. М., Мартиненко А. А. Математична модель раціональної організації обробки інформаційних потоків в системі доставки літальними апаратами. System technologies. 2024. Т. 2, № 151. С. 3–12. DOI: 10.34185/1562-9945-2-151-2024-01
2. Мороз Б.І., Круглик А.С., Мороз Д.М., Мартиненко А.А. Математична модель і загальний алгоритм вирішення задачі обробки повідомлень з урахуванням їх цінності і старіння в системах літальних апаратів. Системні технології. Регіональний міжвузівський збірник наукових праць. 2024. № 5(154). С. 3 – 18. DOI 10.34185/1562-9945-5-154-2024-01
3. Переваги та недоліки сучасних фреймворків черг повідомлень /А. В. Кудякова. Електронний ресурс. URL: https://ekmair.ukma.edu.ua/items/23e5509f-0f5e-4cb0-951c-534bb647ef91. (дата звернення 10.02.2025).
4. What is a message broker? Електронний ресурс. URL: https://www.ibm.com/think/topics/message-brokers. (дата звернення 02.01.2025).
5. Point-to-point messaging. Веб-сайт. URL: https://www.ibm.com/docs/en/wip-mg/2.0.0?topic=concepts-point-point-messaging. (дата звернення 10.02.2025).
6. Point-To-Point Messaging. Веб-сайт. URL: https://www.oreilly.com/library/view/java-message-service/9780596802264/ch04.html. (дата звернення 10.02.2025)
7. Apache Kafka. Електронний ресурс. URL: https://kafka.apache.org/. (дата звернення 15.03.2025).
8. Починаємо роботу з Apache Kafka. Частина І. Веб-сайт. URL: https://dou.ua/forums/topic/39363/. (дата звернення 01.03.2025).
9. Що таке change data capture. Веб-сайт. URL: https://freehost.com.ua/ukr/faq/articles/scho-take-change-data-capture/. (дата звернення 30.03.2025)
10. RabbitMQ. One broker to queue them all. Електронний ресурс. URL: https://www.rabbitmq.com/. (дата звернення 20.03.2025).
11. Message Brokers: Queue-based vs Log-based. Веб-сайт. URL: https://dev.to/oleg_potapov/message-brokers-queue-based-vs-log-based-2f21. (дата звернення 20.03.2025).
12. RabbitMQ Cluster. AMQP protocol. Веб-сайт. URL: https://blog.ipeacocks.info/2016/11/rabbitmq-cluster-amqp-protocol.html. (дата звернення 20.03.2025).
13. When to use RabbitMQ or Apache Kafka. Веб-сайт. URL: https://www.cloudamqp.com/blog/when-to-use-rabbitmq-or-apache-kafka.html. (дата звернення 10.02.2025).
14. A. Ćatović, N. . Buzađija, and S. Lemes, "Microservice development using RabbitMQ message broker ", Sci. Eng. Technol., vol. 2, no. 1, pp. 30–37, Apr. 2022, doi: 10.54327/set2022/v2.i1.19.

References

1. B. Moroz, A. Kruhlyk, D. Moroz, A. Martynenko. Mathematic model of the rational organization of the information flows processing in aircraft delivery system. *System technologies*. 2024. Т. 2, № 151. pp. 3–12. doi: 10.34185/1562-9945-2-151-2024-01.
2. B. Moroz, A. Kruhlyk. Mathematical model and general algorithm for solving the problem of processing messages taking into account their value and aging in aircraft systems. *System technologies*. 2024. № 5(154). P. 3 – 18. DOI 10.34185/1562-9945-5-154-2024-01
3. Advantages and disadvantages of modern message queue frameworks /A. Kudiakova. Website. URL: https://ekmair.ukma.edu.ua/items/23e5509f-0f5e-4cb0-951c-534bb647ef91. (date of app. 10.02.2025).
4. What is a message broker? Website. URL: https://www.ibm.com/think/topics/message-brokers. (date of app. 02.01.2025).
5. Point-to-point messaging. Website. URL: https://www.ibm.com/docs/en/wip-mg/2.0.0?topic=concepts-point-point-messaging. (date of app. 10.02.2025).
6. Point-To-Point Messaging. Website. URL: https://www.oreilly.com/library/view/java-message-service/9780596802264/ch04.html. (date of app. 10.02.2025)
7. Apache Kafka. Website. URL: https://kafka.apache.org/. (date of app. 15.03.2025).
8. Starting to work with Apache Kafka. Part I. Website. URL: https://dou.ua/forums/topic/39363/. (date of app. 01.03.2025).
9. What is change data capture. Website. URL: https://freehost.com.ua/ukr/faq/articles/scho-take-change-data-capture/. (date of app. 30.03.2025).
10. RabbitMQ. One broker to queue them all. Website. URL: https://www.rabbitmq.com/. (date of app. 20.03.2025).
11. Message Brokers: Queue-based vs Log-based. Website. URL: https://dev.to/oleg_potapov/message-brokers-queue-based-vs-log-based-2f21. (date of app. 20.03.2025).
12. RabbitMQ Cluster. AMQP protocol. Website. URL: https://blog.ipeacocks.info/2016/11/rabbitmq-cluster-amqp-protocol.html. (date of app. 20.03.2025).
13. When to use RabbitMQ or Apache Kafka. Website. URL: https://www.cloudamqp.com/blog/when-to-use-rabbitmq-or-apache-kafka.html. (date of app. 10.02.2025).
14. A. Ćatović, N. . Buzađija, and S. Lemes, "Microservice development using RabbitMQ message broker ", Sci. Eng. Technol., vol. 2, no. 1, pp. 30–37, Apr. 2022, doi: 10.54327/set2022/v2.i1.19.