

Міністерство освіти і науки України
Державний заклад
«Луганський національний університет імені Тараса Шевченка»

Факультет технологій та інформаційних систем

Кафедра інформаційних технологій та систем

Тимошенко Володимир Олегович

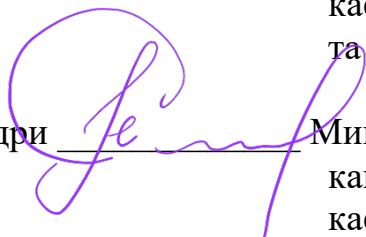
**РЕАЛІЗАЦІЯ КЛІЄНТ-СЕРВЕРНОЇ СИСТЕМИ АСИНХРОННОЇ
ПЕРЕДАЧІ ПОВІДОМЛЕНЬ ІЗ ЗАСТОСУВАННЯМ ЧЕРГ**

кваліфікаційна робота

**здобувача вищої освіти першого (бакалаврського) рівня
освітньої програми «Інженерія програмного забезпечення»
за спеціальністю 121 Інженерія програмного забезпечення**

Особистий підпис  Володимир ТИМОШЕНКО

Науковий керівник  Світлана ПЕРЕЯСЛАВСЬКА,
кандидат педагогічних наук, доцент
кафедри інформаційних технологій
та систем

Завідувач кафедри  Микола СЕМЕНОВ,
кандидат педагогічних наук, доцент
кафедри інформаційних технологій
та систем

Полтава – 2026

Міністерство освіти і науки України
Державний заклад „Луганський національний університет
імені Тараса Шевченка”

Факультет (інститут)	Навчально-науковий інститут математики та інформаційних технологій
Кафедра, циклова комісія	Інформаційних технологій та систем
Рівень освіти	перший (бакалаврський)
Напрямок підготовки (спеціальність)	121 «Інженерія програмного забезпечення» (код, назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри ІТС
М.А. Семенов

(підпис) _____
(ініціали, прізвище)
“ ” 2023 р.

ЗЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЕКТ (РОБОТУ) СТУДЕНТУ

Тимошенку Володимиріу Олеговичу
(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) Реалізація клієнт-серверної системи
асинхронної передачі повідомлень із застосуванням черг

Керівник кваліфікаційної роботи _____
Переляславська С.О. к.п.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджена наказом по університету Від “ ” 2026 року №_

2. Строк подання студентом проекту (роботи) _____
3. Вихідні дані до роботи (проекту) _____ у результаті виконання роботи
повинно бути розроблено клієнт-серверну систему асинхронної
передачі повідомлень із застосуванням черг

(визначаються кількісні або (та) якісні показники, яким повинен відповідати об'єкт розробки)

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) МЕТОДОЛОГІЯ РОЗРОБКИ КЛІЄНТ-СЕРВЕРНОЇ СИСТЕМИ
АСИНХРОННОЇ ПЕРЕДАЧІ ПОВІДОМЛЕНЬ ІЗ ЗАСТОСУВАННЯМ
ЧЕРГ.

(визначаються назви розділів або (та) перелік питань, які повинні увійти до тексту ПЗ)

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

6. Консультанти розділів проекту (роботи)

Розділ	Прізвище, ініціали та посада Консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання „_____” _____ 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
	Вибір теми роботи, вивчення наукової літератури, затвердження теми та керівника.	До 15 жовтня	
	Аналіз літературних джерел за темою роботи. Розробка та апробація методики дослідно-експериментальної роботи. Подання структури теоретичної частини роботи та плану експериментальних досліджень.	Другий тиждень листопада (10 листопада)	
	Робота над теоретичною частиною. Подання теоретичної частини роботи для першого читання науковим керівником.	До 15 грудня	
	Усунення зауважень, урахування рекомендацій наукового керівника. Подання теоретичної частини роботи на друге читання.	До 28 січня	
	Проведення експериментальної роботи. Поетапний аналіз та обговорення її результатів. Перевірка стану виконання роботи.	Перший тиждень березня	
	Урахування рекомендацій наукового керівника, усунення недоліків, підготовка варіанта роботи до передзахисту. Розробка презентації.	До 31 березня	
	Попередній захист роботи на кафедрі	квітень	
	Доопрацювання роботи з урахуванням рекомендацій після передзахисту. Подання роботи науковому керівникові та рецензентові на підготовку відгуку та рецензії	За 10 днів до державної атестації	
	Подання на кафедру остаточного варіанта роботи, переплетеного та підписаного автором, науковим керівником і рецензентом.	За 5 днів до державної атестації	

Студент

підпис

В. О. Тимошенко

(ініціали, прізвище)

Керівник проекту (роботи)

підпис

С. О. Переяславська

АНОТАЦІЯ

Тимошенко В. О.

Тема: Реалізація клієнт-серверної системи асинхронної передачі повідомлень із застосуванням черг.

Спеціальність: 121 «Інженерія програмного забезпечення»

Установа: ЛНУ імені Тараса Шевченка, 2026 р.

Бакалаврська робота містить: 104 сторінки, 17 рис., 4 табл., 17 джерел.

Об'єктом дослідження є процес асинхронної передачі повідомлень у клієнт-серверних програмних системах.

Предметом дослідження є технології, архітектурні підходи та протоколи побудови брокерів повідомлень із застосуванням черг.

Метою роботи є проектування та програмна реалізація брокера повідомлень з підтримкою гарантованої доставки, маршрутизації та персистентності даних.

Результати роботи. Досліджено концепції асинхронного обміну даними та шаблони проектування розподілених систем. Обґрунтовано вибір технологічного стека на базі мови Java, фреймворка Spring Boot, протоколу gRPC (HTTP/2) та СУБД PostgreSQL. Розроблено архітектуру сервера (брокера), реалізовано механізми гнучкої маршрутизації повідомлень за допомогою дерева маршрутів (RouteTree), деревоподібне логування подій, а також клієнтську бібліотеку (SDK). Проведено експериментальні дослідження швидкодії та навантажувальне тестування розробленої системи.

Висновки. В результаті розробки було створено легкий брокер повідомлень.

Ключові слова: БРОКЕР ПОВІДОМЛЕНЬ, ЧЕРГИ ПОВІДОМЛЕНЬ, АСИНХРОННИЙ ОБМІН, gRPC, PROTOCOL BUFFERS, SPRING BOOT, POSTGRES SQL, PERSISTENCE.

ABSTRACT

Tymoshenko V. O.

Theme: Implementation of a client-server system for asynchronous message transmission using queues.

Speciality: 121 "Software Engineering"

Institution: Luhansk Taras Shevchenko National University (LTSNU), 2026.

Diploma work contains: 104 pages, 17 Fig., 4 tables, 17 sources.

A research object is the process of asynchronous message transmission in client-server software systems.

The article of research is technologies, architectural approaches and protocols for building message brokers using queues.

An aim of work is the design and software implementation of a message broker with support for guaranteed delivery, routing and data persistence.

Job performances. The thesis researched the concepts of asynchronous data exchange and design patterns of distributed systems. The choice of the technology stack based on the Java programming language, the Spring Boot framework, the gRPC protocol (HTTP/2) and the PostgreSQL database management system was substantiated. The architecture of the server part of the message broker was developed. Mechanisms for flexible message routing using a route tree (RouteTree), message queues, acknowledgements, repeated delivery and persistent data storage were implemented. A client library for interaction with the broker was also developed. Experimental research of performance and load testing of the developed system were carried out.

Conclusions. As a result of the development, a lightweight message broker for asynchronous message transmission using queues was created.

Keywords. MESSAGE BROKER, MESSAGE QUEUES, ASYNCHRONOUS COMMUNICATION, gRPC, PROTOCOL BUFFERS, SPRING BOOT, POSTGRESQL, PERSISTENCE.

Міністерство освіти і науки України
Державний заклад «Луганський національний університет
імені Тараса Шевченка»

Факультет (інститут)

Технологій та інформаційних систем
(повна назва)

Кафедра

Інформаційних технологій та систем
(повна назва)
(код, назва)

ТЕХНІЧНЕ ЗАВДАННЯ

на виконання програмної розробки (ПР):

РЕАЛІЗАЦІЯ КЛІЄНТ-СЕРВЕРНОЇ СИСТЕМИ АСИНХРОННОЇ
ПЕРЕДАЧІ ПОВІДОМЛЕНЬ ІЗ ЗАСТОСУВАННЯМ ЧЕРГ
ІТС.4ІПЗ.8326.02-ТЗ

ПОГОДЖЕНО

Керівник кваліфікаційної
роботи

Переяславська С.О.



“ ” 2026р

ВИКОНАВЕЦЬ

Студент групи 4ІПЗ

Тимошенко В. О.



“ ” 2026р

Лубни – 2026

ЗМІСТ

ВСТУП	3
1. ХАРАКТЕРИСТИКА ОБ'ЄКТА.....	3
2. ПРИЗНАЧЕННЯ ПРОГРАМНОЇ РОЗРОБКИ.....	4
3. ОСНОВНІ ВИМОГИ ДО ПРОГРАМНОГО КОМПЛЕКСУ	4
4. ТЕХНІКО-ЕКОНОМІЧНІ ВИМОГИ ДО КІНЦЕВОГО ПРОДУКТУ5	
5. ВИМОГИ ДО МАТЕРІАЛІВ І КОМПЛЕКТУЮЧИХ	5
6. ЕТАПИ ВИКОНАННЯ ПР	6
7. ПРИЙОМ	6
8. ПОРЯДОК ВНЕСЕННЯ ЗМІН ДО ТЕХНІЧНОГО ЗАВДАННЯ....	7

ВСТУП

1.1. Найменування: Клієнт-серверна система асинхронної передачі повідомлень із застосуванням черг.

1.2. Шифр ПР: ІТС.4ІПЗ.8326.02-ТЗ

1.3. Підстава до виконання ПР: Підставою для виконання програмної розробки є затверджена тема кваліфікаційної роботи бакалавра за спеціальністю 121«Інженерія програмного забезпечення».

1.4. Терміни розробки:

Початок: _____ 2025 р.

Закінчення: _____ 2026 р.

1.5. Джерело фінансування:

Робота виконується на некомерційній основі як індивідуальна кваліфікаційна робота здобувача вищої освіти.

1. ХАРАКТЕРИСТИКА ОБ'ЄКТА

Об'єктом розробки є програмна система для асинхронного обміну повідомленнями між клієнтськими застосунками.

До складу програмного комплексу входять:

1. серверна частина брокера повідомлень;
2. клієнтський Java-модуль для взаємодії з брокером;

Серверна частина повинна приймати повідомлення від відправників, розміщувати їх у чергах і доставляти отримувачам. Клієнтська частина повинна надавати зручний програмний інтерфейс для роботи з брокером

2. ПРИЗНАЧЕННЯ ПРОГРАМНОЇ РОЗРОБКИ

Програмна розробка призначена для організації асинхронної взаємодії між окремими компонентами програмних систем.

Система повинна забезпечувати:

1. передавання повідомлень між відправниками та отримувачами;
2. тимчасове зберігання повідомлень у чергах;

3. маршрутизацію повідомлень;
4. підтвердження обробки повідомлень;
5. можливість відновлення частини даних після перезапуску брокера.

3. ОСНОВНІ ВИМОГИ ДО ПРОГРАМНОГО КОМПЛЕКСУ

3.1. Загальні вимоги

Програмний комплекс повинен працювати як клієнт-серверна система. Серверна частина виконує роль брокера повідомлень, а клієнтська частина забезпечує доступ до його функцій.

Система повинна бути реалізована засобами Java.

3.2. Функціональні вимоги

Система повинна забезпечувати:

1. створення та видалення черг;
2. публікацію повідомлень із ключем маршрутизації;
3. доставку повідомлень отримувачам;
4. підтвердження успішної або неуспішної обробки повідомлення;
5. повторну доставку повідомлень у разі помилки;
6. збереження повідомлень для черг, які працюють у режимі збереження;
7. відновлення збережених повідомлень після перезапуску брокера.

3.3. Вимоги до клієнтської частини

Клієнтська частина повинна надавати засоби для:

1. підключення до брокера;
2. створення черг;
3. публікації повідомлень;
4. підписки на чергу;
5. обробки отриманих повідомлень;

6. надсилання підтверджень обробки.

3.4. Вимоги до якості та надійності

Система повинна коректно обробляти помилкові запити, втрату з'єднання з клієнтом і повторну доставку непідтверджених повідомлень.

Система повинна мати модульні тести для перевірки основних компонентів і сценаріїв роботи.

4. ТЕХНІКО-ЕКОНОМІЧНІ ВИМОГИ

Розробка виконується в межах кваліфікаційної роботи бакалавра та не передбачає комерційного впровадження.

Для реалізації системи використовуються безкоштовні або відкриті засоби розробки: Java, Spring Boot, gRPC, Protocol Buffers, PostgreSQL, Maven та JUnit.

5. ВИМОГИ ДО МАТЕРІАЛІВ І КОМПЛЕКТУЮЧИХ

5.1. Вимоги до екологічної безпеки при експлуатації.

Не пред'являються.

5.2. Спеціальні вимоги до кінцевого продукту.

Не пред'являються.

6. ЕТАПИ ВИКОНАННЯ ПР

№	Етап роботи	Зміст роботи	Звітні матеріали
1	Аналіз і проектування	Аналіз предметної області, визначення вимог, проектування архітектури системи	План роботи, опис архітектури
		Розробка брокера	
2	Реалізація серверної частини	повідомлень, черг, маршрутизації, доставки та збереження даних	Серверний модуль
3	Реалізація	Розробка	Клієнтський

№	Етап роботи	Зміст роботи	Звітні матеріали
	клієнтської частини	клієнтського Java-модуля для роботи з брокером	модуль
		Проведення	
4	Тестування та оформлення	тестування, аналіз результатів, підготовка пояснювальної записки	Пояснювальна записка, результати тестування

7. ПРИЙОМ

Приймання програмної розробки здійснюється керівником кваліфікаційної роботи та екзаменаційною комісією.

перелік документів:

1. пояснювальна записка;
2. вихідний код програмного комплексу;
3. результати тестування;
4. графічні матеріали;
5. презентація до захисту.

8. ПОРЯДОК ВНЕСЕННЯ ЗМІН ДО ТЕХНІЧНОГО ЗАВДАННЯ

Технічне завдання може уточнюватися в процесі виконання роботи за погодженням із керівником кваліфікаційної роботи.

Зміни не повинні суперечити темі роботи та основній меті програмної розробки.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЗ «ЛУГАНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА»

Факультет технологій та інформаційних систем

(назва факультету, інституту)

Інформаційних технологій та систем

(назва кафедри)

Пояснювальна записка

до кваліфікаційної роботи

БАКАЛАВРА

(освітньо-кваліфікаційний рівень)

на тему:

**РЕАЛІЗАЦІЯ КЛІЄНТ-СЕРВЕРНОЇ СИСТЕМИ АСИНХРОННОЇ
ПЕРЕДАЧІ ПОВІДОМЛЕНЬ ІЗ ЗАСТОСУВАННЯМ ЧЕРГ**

Виконав: студент 4 курсу,
напряму підготовки (спеціальності)
121 «Інженерія програмного
ззабезпечення»

Тимошенко В.О.

(прізвище та ініціали)

Керівник Переяславська С.О.

(прізвище та ініціали)

Рецензент Козуб Ю. Г.

(прізвище та ініціали)

Лубни – 2026 року

ЗМІСТ

ВСТУП.....	3
Розділ 1. Аналіз предметної області та існуючих рішень	5
1.1. Основи асинхронного обміну повідомленнями	5
1.2. Архітектурні підходи до побудови систем асинхронного обміну повідомленнями	7
1.3. Черги повідомлень.....	9
1.4. Аналіз існуючих систем обміну повідомленнями	10
1.5. Постановка задачі	14
Висновки до розділ 1	18
Розділ 2. Проектування системи асинхронного обміну повідомленнями....	19
2.1. Визначення вимог до системи.....	19
2.2. Розробка загальної архітектури системи	19
2.3. Проектування структури повідомлень та їх життєвого циклу	21
2.4. Проектування механізмів обробки повідомлень.....	22
2.5. Алгоритми маршрутизації та розподілу повідомлень	23
Висновки до розділу 2	26
Розділ 3. Реалізація серверної частини брокера повідомлень.....	27
3.1. Реалізація серверної частини брокера.....	27
3.1.1. Вибір технологій та інструментів розробки	27
3.1.2. Структура серверного проєкту	28
3.1.3. gRPC-контракт взаємодії з брокером	29
3.1.4. Командна обробка запитів	30
3.1.5. Реалізація черг і налаштувань черги.....	31
3.1.6. Маршрутизація повідомлень	34
3.1.7. Створення та видалення черг	34
3.1.8. Публікація повідомлень	35
3.1.9. Підписки отримувачів і облік незавершених доставок	36
3.1.10. Доставка повідомлень отримувачам.....	37

3.1.11. Підтвердження, відхилення та повторна доставка.....	39
3.1.12. Збереження даних і відновлення після перезапуску	42
3.1.13. Фонові задачі, keep-alive та обробка помилок.....	42
3.2. Реалізація клієнтської частини	45
3.3. Тестування та експериментальне дослідження.....	51
3.4. Аналіз результатів.....	55
Висновки до розділу 3.....	56
ВИСНОВКИ	57
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	59
ДОДАТКИ.....	61
Додаток А gRPC-контракт	61
Додаток Б Фрагменти реалізації серверної частини брокера.....	66
Додаток В Фрагменти реалізації клієнтського модуля	89

ВСТУП

У сучасних інформаційних системах дедалі частіше використовуються архітектури, що складаються з кількох незалежних програмних компонентів. Такі компоненти можуть виконувати різні функції, працювати на різних серверах, використовувати різні технології зберігання даних і мати різну швидкість обробки запитів. У таких умовах важливого значення набуває задача організації надійної взаємодії між окремими частинами системи.

Традиційна синхронна модель взаємодії, за якої один компонент безпосередньо викликає інший і очікує негайної відповіді, не завжди є ефективною. Вона створює тісну залежність між компонентами, ускладнює масштабування системи та знижує стійкість до тимчасових відмов. Якщо один із сервісів недоступний або працює повільно, це може вплинути на роботу всієї системи.

Одним із поширених способів зменшення такої залежності є асинхронний обмін повідомленнями. У цій моделі компонент, який створює повідомлення, не очікує негайного завершення обробки іншим компонентом. Повідомлення передається через проміжний елемент системи, який приймає його, розміщує в черзі та доставляє отримувачу відповідно до обраної логіки маршрутизації й доставки.

Черги повідомлень дозволяють тимчасово зберігати дані між моментом їх створення та моментом фактичної обробки. Завдяки цьому відправник і отримувач можуть працювати незалежно один від одного, а система може краще витримувати нерівномірне навантаження, тимчасову недоступність окремих компонентів і затримки під час обробки повідомлень.

Актуальність теми полягає в тому, що брокери повідомлень широко застосовуються в сучасних мікросервісних, хмарних і подієво-орієнтованих системах. Вони використовуються для інтеграції сервісів, передавання подій, виконання фонових задач, розподілу навантаження та підвищення надійності програмних комплексів.

Метою проєкту є розробка клієнт-серверної системи асинхронної передачі повідомлень із застосуванням черг, яка забезпечує публікацію, маршрутизацію, доставку, підтвердження обробки та відновлення повідомлень у визначених режимах роботи.

Для досягнення поставленої мети необхідно розв'язати такі **завдання**:

- проаналізувати принципи асинхронного обміну повідомленнями;
- розглянути архітектурні підходи до побудови систем обміну повідомленнями;
- дослідити роль черг у забезпеченні незалежності між відправником і отримувачем;
- проаналізувати існуючі брокери повідомлень та їхні архітектурні особливості;
- сформулювати вимоги до системи асинхронного обміну повідомленнями;
- спроектувати загальну архітектуру брокера повідомлень;
- передбачити механізми маршрутизації, підписки, підтвердження, повторної доставки, зберігання та відновлення;
- реалізувати та протестувати розроблену систему.

Об'єктом дослідження є процес асинхронної передачі повідомлень у клієнт-серверних програмних системах.

Предметом дослідження є архітектурні та програмні механізми побудови брокера повідомлень із підтримкою черг, маршрутизації, підтверджень доставки, політик зберігання та відновлення після перезапуску.

Практичне значення роботи полягає у створенні програмної системи, яка демонструє основні принципи функціонування брокера повідомлень і може бути використана як основа для подальшого дослідження асинхронної взаємодії між сервісами.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ

1.1. Основи асинхронного обміну повідомленнями

Асинхронний обмін повідомленнями є способом взаємодії між програмними компонентами, за якого один компонент передає повідомлення до проміжного компонента — брокера повідомлень, а інший компонент отримує його тоді, коли може виконати обробку. На відміну від синхронної моделі, де клієнт очікує відповідь одразу після виклику, асинхронна модель дозволяє компонентам працювати незалежно у часі.

У загальному вигляді така взаємодія складається з трьох основних учасників: відправника (publisher), брокера повідомлень (message broker) і отримувача (consumer). Відправник створює повідомлення та передає його брокеру. Брокер приймає повідомлення, визначає, куди його потрібно направити, і зберігає його до моменту доставки. Отримувач підписується на відповідну чергу або канал і обробляє повідомлення після отримання.

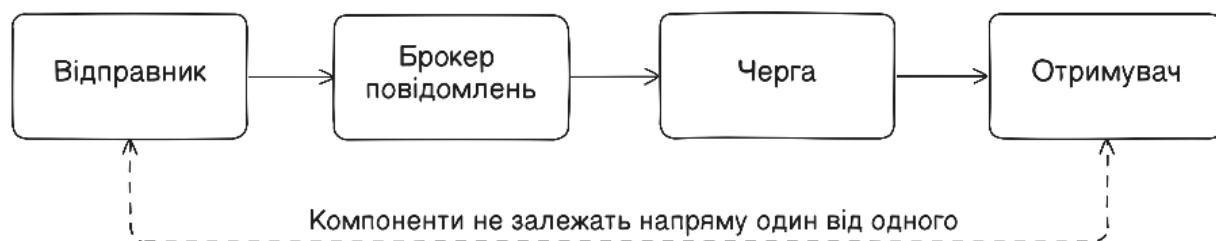


Рис. 1.1. — Загальна схема асинхронної взаємодії між відправником, брокером і отримувачем

Основною перевагою асинхронного підходу є слабке зв'язування компонентів. Відправник не повинен знати, який саме отримувач обробить повідомлення, чи доступний він у конкретний момент часу та з якою швидкістю він працює. Його завдання полягає у передаванні повідомлення брокеру. Отримувач, у свою чергу, може отримати повідомлення пізніше, коли буде готовий до обробки.

Асинхронний обмін повідомленнями також дозволяє підвищити стійкість системи до тимчасових відмов. Якщо отримувач недоступний, повідомлення

може залишатися в черзі та бути доставленим пізніше. Якщо навантаження на систему різко зростає, черга може виконувати роль буфера, який згладжує різницю між швидкістю створення та швидкістю обробки повідомлень.

У практичних системах асинхронна взаємодія використовується для виконання віддалених фонових задач, передавання подій, оновлення аналітичних даних, інтеграції мікросервісів і організації взаємодії між незалежними частинами програмного комплексу. Наприклад, після створення замовлення один сервіс може опублікувати повідомлення, а інші сервіси поступово виконують резервування товару, оновлення статистики та надсилання сповіщення користувачу.

Важливо враховувати, що асинхронна модель не усуває всі складності розподілених систем. Вона переносить частину відповідальності до брокера повідомлень, який повинен коректно приймати повідомлення, зберігати їх, обирати отримувача, контролювати підтвердження обробки та обробляти помилки. Тому розробка брокера повідомлень потребує чіткого проектування його внутрішньої логіки.

Для систем асинхронного обміну особливе значення мають гарантії доставки. У найпростішому випадку повідомлення може бути доставлене один раз без повторних спроб. У більш надійних сценаріях брокер повинен зберігати повідомлення до підтвердження та повторно доставляти його у разі відмови отримувача. Вибір гарантії доставки впливає на продуктивність, складність системи та допустимість дублювання повідомлень.

Повідомлення в асинхронній системі можна розглядати як самостійну одиницю обміну. Воно містить корисні дані та службову інформацію, яка допомагає брокеру виконувати маршрутизацію і контроль доставки. До службових даних можуть належати ідентифікатор повідомлення, ключ маршрутизації, час створення, заголовки, інформація про спробу доставки або параметри обробки.

У таких системах важливо розрізняти факт доставки повідомлення та факт його успішної обробки. Передавання повідомлення отримувачу ще не означає,

що воно було оброблене. Отримувач може завершити роботу з помилкою, втратити з'єднання або не встигнути надіслати підтвердження. Саме тому в багатьох брокерах використовуються підтвердження обробки.

Підтвердження дозволяє брокеру зрозуміти, чи можна вважати повідомлення завершеним. Якщо підтвердження отримане, повідомлення може бути видалене з черги або зі сховища. Якщо підтвердження не отримане, брокер може залишити повідомлення незавершеним і доставити його повторно. Цей механізм є основою для побудови більш надійних сценаріїв доставки.

1.2. Архітектурні підходи до побудови систем асинхронного обміну повідомленнями

Системи асинхронного обміну повідомленнями можуть будуватися за різними архітектурними підходами. Найпоширенішими є модель point-to-point та publish/subscribe. Кожен із цих підходів визначає, як повідомлення передається від відправника до отримувача та яку роль виконує брокер.

У моделі point-to-point повідомлення надсилається до певної черги, з якої його отримує один отримувач. Якщо до черги підключено кілька отримувачів, кожне окреме повідомлення зазвичай обробляється лише одним із них. Така модель добре підходить для розподілу задач між кількома виконавцями, наприклад для обробки замовлень, генерації звітів або виконання віддалених фонових операцій.

У моделі publish/subscribe відправник публікує повідомлення до певної теми або каналу, а брокер доставляє його всім зацікавленим підписникам. Такий підхід використовується тоді, коли одна подія має бути оброблена кількома незалежними компонентами. Наприклад, подія створення замовлення може бути потрібна сервісу складу, сервісу платежів, сервісу сповіщень і системі аналітики.

Є дві моделі доставки повідомлень брокером, це push і pull. У push-моделі брокер сам надсилає повідомлення підключеному отримувачу, коли той готовий приймати дані. У pull-моделі отримувач сам звертається до брокера та запитує

нові повідомлення. Вибір моделі впливає на контроль навантаження, затримку доставки та відповідальність отримувача за швидкість читання повідомлень.

Для побудови надійної системи важливими є механізми підтвердження. Позитивне підтвердження означає, що повідомлення було успішно оброблене. Негативне підтвердження означає, що повідомлення не вдалося обробити або його потрібно повернути в чергу. Завдяки цьому брокер може відрізнати завершені повідомлення від тих, які потребують повторної доставки.

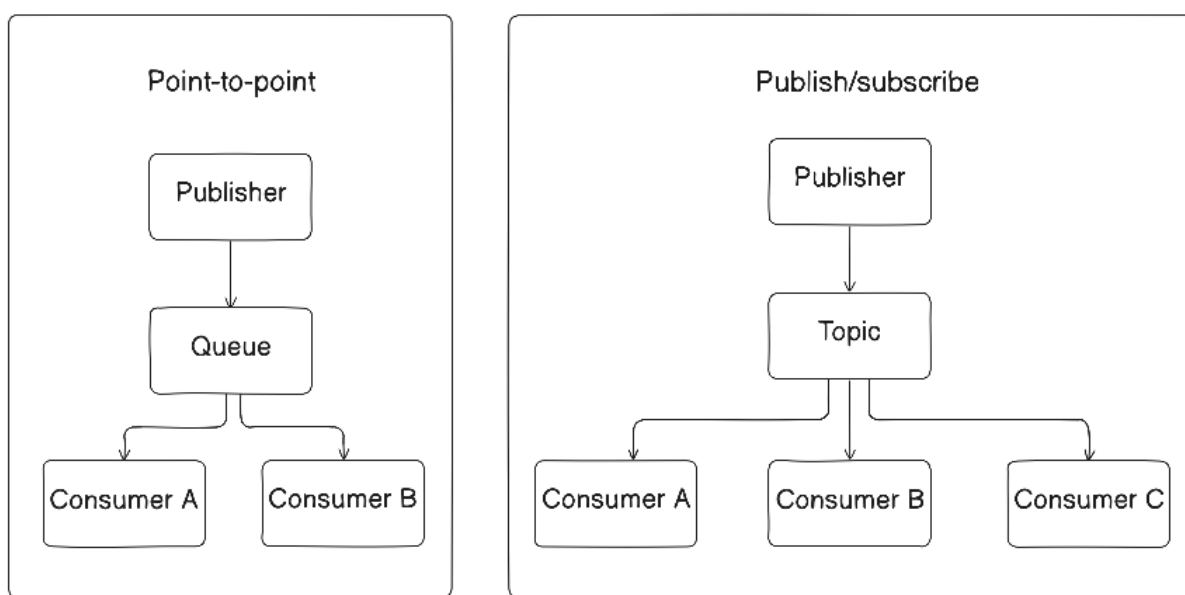


Рис. 1.2. — Моделі point-to-point та publish/subscribe

У практичних системах часто використовується комбінована модель point-to-point та publish/subscribe. Наприклад, повідомлення може бути опубліковане за певним ключем маршрутизації, після чого брокер визначає одну або кілька черг, у які воно має потрапити. Далі кожна черга може працювати за принципом point-to-point і розподіляти повідомлення між кількома отримувачами.

Важливим архітектурним рішенням є вибір способу маршрутизації. У простих системах повідомлення надсилається безпосередньо в конкретну чергу. У складніших системах використовується ключ маршрутизації або шаблон. Це дозволяє відправнику не знати точну назву всіх черг, а лише вказувати логічний тип повідомлення.

1.3. Черги повідомлень

Черга повідомлень є одним із основних механізмів асинхронної взаємодії між програмними компонентами. Її призначення полягає в тому, щоб тимчасово зберігати повідомлення між моментом їх створення відправником і моментом обробки отримувачем.

Використання черги повідомлень дозволяє змінити характер взаємодії. Відправник не надсилає повідомлення безпосередньо отримувачу, а передає його брокеру. Брокер розміщує повідомлення в черзі та доставляє його тоді, коли отримувач буде готовий прийняти нові дані.

Загальну схему роботи черги можна подати так:

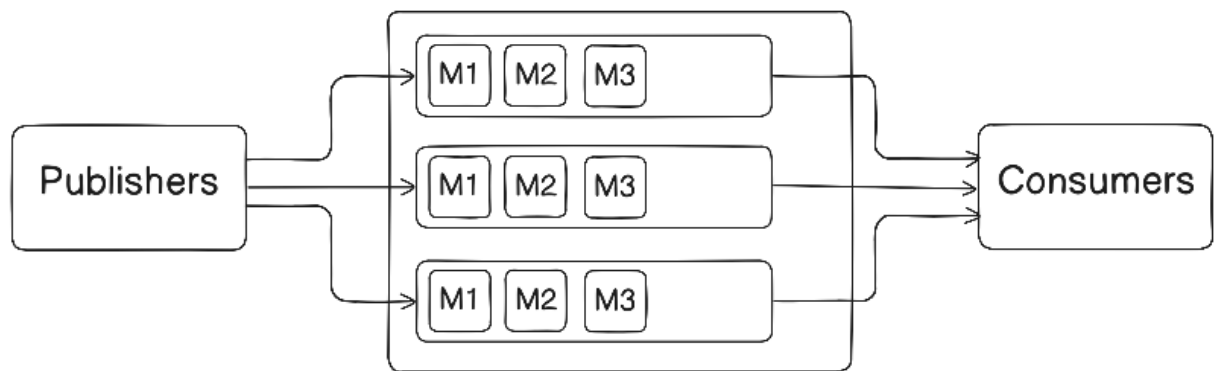


Рис. 1.3. — Черга повідомлень як посередник між відправником і отримувачем

У цій схемі черга виконує роль проміжного буфера. Якщо відправник створює повідомлення швидше, ніж отримувач здатний їх обробляти, повідомлення накопичуються в черзі. Це дозволяє системі стабільніше працювати під час короткочасних піків навантаження. Наприклад, якщо за короткий проміжок часу надходить велика кількість замовлень, система може швидко прийняти повідомлення про ці замовлення, а їх фактичну обробку виконувати поступово.

Черга також зменшує залежність між відправником і отримувачем. Відправник не повинен знати, який саме отримувач обробить повідомлення. Йому достатньо передати повідомлення брокеру. Отримувач, у свою чергу, може

бути запущений пізніше, працювати з іншою швидкістю або тимчасово бути недоступним.

Обробку повідомлень можна масштабувати горизонтально. Наприклад, якщо один отримувач не встигає обробляти всі повідомлення, можна запустити кілька екземплярів такого самого сервісу, і брокер буде розподіляти повідомлення між ними.

Використання черг має певні обмеження. Асинхронна взаємодія ускладнює логіку системи, оскільки результат обробки повідомлення не завжди відомий одразу. Відправник може знати, що повідомлення було прийняте брокером, але не знає, коли саме воно буде оброблене отримувачем. Через це у клієнтській логіці можуть знадобитися додаткові механізми відстеження стану операцій або обробки результатів із затримкою.

Черга не усуває проблему перевантаження повністю, а лише переносить її в окремий компонент системи. Якщо повідомлення надходять швидше, ніж обробляються, розмір черги буде постійно зростати. У такому випадку потрібно застосовувати механізми керування навантаженням.

1.4. Аналіз існуючих систем обміну повідомленнями

Для кращого розуміння предметної області доцільно розглянути кілька відомих систем обміну повідомленнями.

Apache Kafka — це розподілена система моделі publish/subscribe для передавання та обробки потоків подій. Вона часто використовується в мікросервісних і подієво-орієнтованих архітектурах, де потрібно обробляти велику кількість повідомлень, зберігати історію подій і забезпечувати стійкість до відмов окремих вузлів. Kafka більше орієнтована на роботу з потоками даних, які можуть зчитуватися різними групами споживачів незалежно одна від одної.

Дані в Kafka організовуються за темами. Тема може бути поділена на кілька розділів, що дозволяє розподіляти навантаження та забезпечувати паралельну обробку. Кожен розділ містить впорядковану послідовність повідомлень. Нові повідомлення додаються до цієї послідовності, а кожне

повідомлення отримує власну позицію, за якою споживач визначає, які дані вже були прочитані.

Архітектура Kafka передбачає реплікацію даних. Розділи теми можуть мати копії на кількох брокерах. Одна копія обслуговує основні операції читання та запису, а інші синхронізуються з нею. У разі відмови брокера система може обрати іншу копію для продовження роботи. Це підвищує відмовостійкість і дозволяє використовувати Kafka в системах, де втрата подій є небажаною.

Kafka не видаляє повідомлення одразу після їх прочитання. Дані зберігаються відповідно до налаштованої політики: протягом певного часу або до досягнення заданого обсягу. Завдяки цьому кілька груп споживачів можуть незалежно читати одну й ту саму тему.

У Kafka споживач сам звертається до брокера та запитує нові дані. Це дозволяє споживачу самостійно контролювати швидкість читання. Якщо сервіс тимчасово не встигає обробляти повідомлення, вони залишаються у брокері, а споживач може продовжити читання з попередньої позиції.

RabbitMQ — це брокер повідомлень, побудований на моделі point-to-point. Його основне призначення полягає в тому, щоб приймати повідомлення від відправників, маршрутизувати їх до відповідних черг і доставляти отримувачам.

У RabbitMQ відправник зазвичай не надсилає повідомлення безпосередньо в чергу. Спочатку повідомлення потрапляє до обмінника. Обмінник визначає, у яку або в які черги потрібно передати повідомлення. Для цього використовуються правила зв'язування між обмінником і чергами, а також ключ маршрутизації.

Обмінники можуть працювати за різними правилами. Один тип передає повідомлення в чергу з точним збігом ключа маршрутизації. Інший надсилає повідомлення в усі прив'язані черги. Також можливе маршрутування за шаблонами, коли повідомлення з певними ключами потрапляють до черг, пов'язаних із відповідним типом.

Після маршрутизації повідомлення зберігається в черзі до моменту доставки споживачу. RabbitMQ працює за моделлю, у якій брокер сам надсилає

повідомлення підключеному споживачу. Щоб не перевантажувати отримувача, використовується обмеження на кількість повідомлень, які вже були доставлені, але ще не підтверджені.

RabbitMQ підтримує підтвердження обробки повідомлень і повторну доставку. Якщо отримувач успішно обробив повідомлення, він надсилає підтвердження, після чого повідомлення може бути видалене з черги. Якщо підтвердження не отримано, брокер може повернути повідомлення в чергу або доставити його іншому отримувачу.

RabbitMQ зручний у випадках, коли потрібно гнучко направляти повідомлення в різні черги. Обмінники дозволяють змінювати правила маршрутизації без прямої зміни відправника, можна додати нову чергу, прив'язавши її до існуючого обмінника.

NATS — це легка система обміну повідомленнями, орієнтована на швидку взаємодію між сервісами. Вона часто використовується в мікросервісних системах, хмарних середовищах, IoT-рішеннях і застосунках, де важливими є низька затримка, простота підключення клієнтів і мінімальні накладні витрати на передавання повідомлень.

NATS працює як система публікації та підписок. Відправник публікує повідомлення за певною темою, а сервер NATS передає його підписникам. Теми можуть мати ієрархічну структуру, наприклад `orders.created` або `payments.failed`, що дозволяє логічно групувати повідомлення. Також підтримуються шаблони підписки, завдяки яким споживач може отримувати групу пов'язаних повідомлень.

Базовий NATS робить акцент на швидкості та простоті, тому не завжди передбачає довготривале збереження повідомлень. Для задач, де потрібні збереження, повторне читання і підтвердження отримання, використовується JetStream. Завдяки цьому NATS може застосовуватися як для швидкої службової взаємодії, так і для надійнішої передачі потоків повідомлень.

NATS доцільно використовувати там, де важливі простота, швидкість і невеликі накладні витрати. Базова модель добре підходить для службових подій,

внутрішніх сигналів між сервісами та систем, у яких допустима втрата окремих повідомлень.

Розширення JetStream наближає NATS до більш надійних систем обміну повідомленнями. Воно дозволяє зберігати повідомлення, повторно їх читати та використовувати підтвердження.

Apache ActiveMQ — це брокер повідомлень, орієнтований на модель корпоративного обміну повідомленнями. Він підтримує черги, теми, підтвердження доставки, збереження повідомлень і роботу з різними протоколами. ActiveMQ часто використовується в системах, де потрібна інтеграція різних застосунків і надійна доставка між компонентами.

Архітектура ActiveMQ дозволяє використовувати дві основні моделі взаємодії: черги та теми. Черга застосовується тоді, коли повідомлення має бути оброблене одним отримувачем. Тема використовується тоді, коли одне повідомлення потрібно передати кільком підписникам. Завдяки цьому ActiveMQ може використовуватися як для розподілу задач, так і для поширення подій.

Для збереження повідомлень ActiveMQ може використовувати постійне сховище. Також він підтримує транзакції, відкладену доставку, черги для проблемних повідомлень і різні політики повторної доставки. Це робить його придатним для систем, де важливими є контроль життєвого циклу повідомлення та передбачувана поведінка при збоях.

ActiveMQ також важливий з погляду підтримки стандартів. Завдяки JMS він добре інтегрується з Java-застосунками, а підтримка різних протоеколів дозволяє підключати клієнтів, побудованих на різних технологіях.

На відміну від легких брокерів, ActiveMQ більше орієнтований на корпоративні сценарії, де важливі сумісність, транзакційність, надійна доставка та можливість інтеграції з уже існуючими системами.

Розглянуті системи демонструють різні підходи до асинхронного обміну повідомленнями. Kafka більше орієнтована на потокову обробку подій, RabbitMQ — на черги та маршрутизацію, NATS — на легку й швидку взаємодію, ActiveMQ — на корпоративну інтеграцію. Для розроблюваної системи найбільш

релевантною є модель брокера повідомлень із чергами, маршрутизацією, підтвердженнями, політиками доставки та можливістю відновлення після перезапуску.

1.5. Постановка задачі

У межах бакалаврського проєкту необхідно розробити клієнт-серверну систему асинхронної передачі повідомлень із застосуванням черг. Система повинна виконувати роль брокера повідомлень, який приймає повідомлення від відправників, визначає відповідні черги, розміщує повідомлення в них і доставляє повідомлення отримувачам, що підписані на ці черги.

Загальна модель роботи системи може бути подана таким чином:

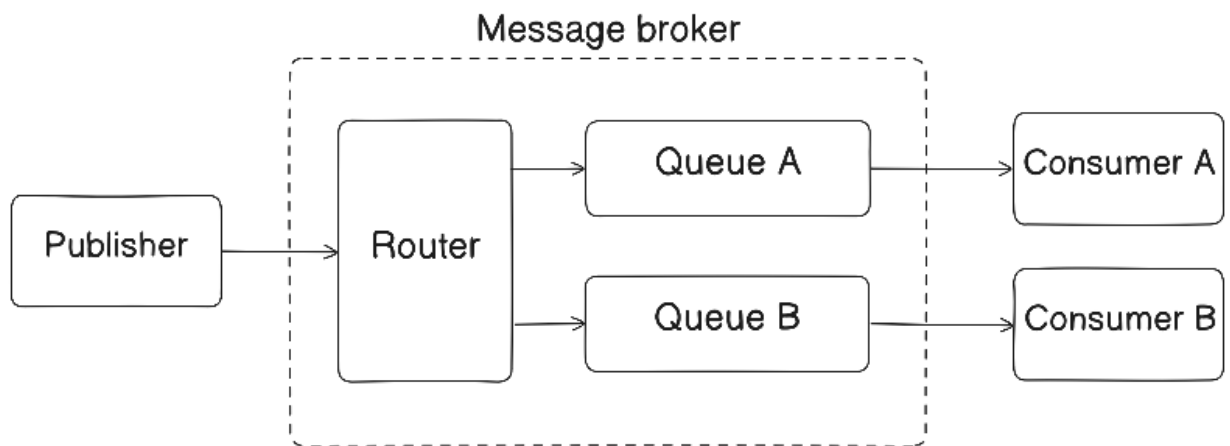


Рис. 1.5. — Загальна модель задачі розроблюваної системи брокера повідомлень

У цій моделі відправник створює повідомлення та передає його брокеру. Брокер визначає, до яких черг має бути направлено повідомлення. Черга тимчасово зберігає повідомлення до моменту доставки. Отримувач підписується на конкретну чергу та отримує повідомлення для обробки.

Задача проєкту полягає не у створенні повного аналога промислових брокерів повідомлень, а в розробці системи, яка демонструє основні принципи їх роботи. До таких принципів належать приймання повідомлень, маршрутизація за ключами, керування чергами, доставка отримувачам, підтвердження обробки,

повторна доставка, обмеження кількості непідтверджених повідомлень, постійне зберігання та відновлення після перезапуску.

Система повинна підтримувати різні параметрами роботи. До таких параметрів належать спосіб зберігання, правила підтвердження, політика повторної доставки, обмеження місткості, час життя повідомлень, правила автоматичного видалення та стратегія вибору отримувача.

Для публікації повідомлень необхідно передбачити використання ключа маршрутизації. На основі цього ключа брокер повинен визначати одну або кілька черг, у які потрібно направити повідомлення. Це дозволить відокремити відправника від конкретних черг і робить систему гнучкішою порівняно з прямим надсиланням повідомлення в одну фіксовану чергу.

Для взаємодії з отримувачем необхідно реалізувати підписку на чергу. Отримувач повинен мати можливість підключитися до брокера, вказати чергу, з якої він хоче отримувати повідомлення, і задати допустиму кількість повідомлень, які можуть перебувати в обробці без підтвердження. Після цього брокер повинен доставляти повідомлення відповідно до доступності отримувача.

Система повинна підтримувати підтвердження обробки повідомлень. АСК означає, що повідомлення успішно оброблене, після чого брокер може завершити його життєвий цикл. NACK означає, що повідомлення не було оброблене. У такому випадку брокер має прийняти рішення: повернути повідомлення в чергу для повторної доставки або завершити його без повторної постановки. Якщо повідомлення некоректне, перевищує допустимий розмір, не може бути маршрутизоване або черга переповнена, брокер повинен повернути негативну відповідь із поясненням причини.

Підтримка контролю повідомлень, які були доставлені отримувачу, але ще не підтверджені. Такі повідомлення не можна вважати завершеними, оскільки брокер не має підтвердження їх успішної обробки. Для цього система повинна вести облік незавершених доставок і використовувати його при АСК, NACK, відключенні отримувача та обмеженні кількості повідомлень у роботі.

У разі відключення отримувача брокер повинен обробити всі повідомлення, які залишилися без підтвердження. Якщо політика повторної доставки це дозволяє, такі повідомлення мають бути повернуті в чергу. Якщо повторна доставка заборонена або досягнуто максимальну кількість спроб, повідомлення має бути завершене відповідно до правил черги.

Для захисту отримувачів від перевантаження передбачити обмеження кількості непідтверджених повідомлень. Брокер не повинен надсилати нові повідомлення отримувачу, якщо той уже має максимально допустиму кількість повідомлень у роботі. Це дозволяє узгодити швидкість доставки зі швидкістю фактичної обробки.

Система повинна підтримувати постійне зберігання. Збереженими мають бути відповідно налаштовані черги, їхні налаштування та повідомлення, які ще не були завершені. Після перезапуску брокер повинен відновити черги, завантажити незавершені повідомлення та продовжити їх доставку.

Висновки до розділу 1

У першому розділі було розглянуто предметну область асинхронного обміну повідомленнями та визначено її значення для побудови сучасних розподілених програмних систем. Було показано, що асинхронна модель дозволяє зменшити залежність між відправником і отримувачем, оскільки повідомлення передається не напряму, а через проміжний компонент — брокер повідомлень.

Проаналізовано основні архітектурні підходи до передавання повідомлень, зокрема моделі point-to-point, publish/subscribe та ролі черг повідомлень, які виконують функцію буфера між моментом створення та фактичної обробки повідомлення отримувачем.

Розглянуто існуючі системи обміну повідомленнями, зокрема Apache Kafka, RabbitMQ, NATS та Apache ActiveMQ. Аналіз цих систем показав, які механізми є найбільш важливими для розроблюваного проєкту. На основі проведеного аналізу було сформульовано задачу розробки власної клієнт-серверної системи асинхронної передачі повідомлень із застосуванням черг.

Розділ 2. Проєктування системи асинхронного обміну повідомленнями

2.1. Визначення вимог до системи

На основі аналізу предметної області було визначено вимоги до системи асинхронного обміну повідомленнями.

До основних функціональних вимог належать:

- створення та видалення черг із заданими параметрами роботи;
- публікація повідомлень із ключем маршрутизації;
- маршрутизація повідомлень до однієї або кількох черг за шаблонами ключів;
- підписка отримувачів на конкретні черги;
- доставка повідомлень підписаним отримувачам;
- підтримка підтверджень від відправника про прийняття або відхилення публікації;
- підтримка підтвердження від отримувача або відхилення повідомлень;
- облік повідомлень, які доставлені отримувачу, але ще не підтверджені;
- повторне повернення повідомлень у чергу відповідно до політики повторної доставки;
- обмеження кількості непідтверджених повідомлень для одного отримувача;
- збереження черг і повідомлень, які мають бути відновлені після перезапуску;
- автоматичне очищення черг або повідомлень відповідно до заданих політик.

2.2. Розробка загальної архітектури системи

Під час проєктування системи було обрано багаторівневу архітектуру.

У загальному вигляді система складається з таких логічних частин:

- транспортний рівень;
- командний рівень;
- рівень основних сервісів брокера;

- рівень черг і маршрутизації;
- рівень підписок і незавершених доставок;
- рівень збереження та відновлення;
- рівень фонових задач;
- рівень доставки

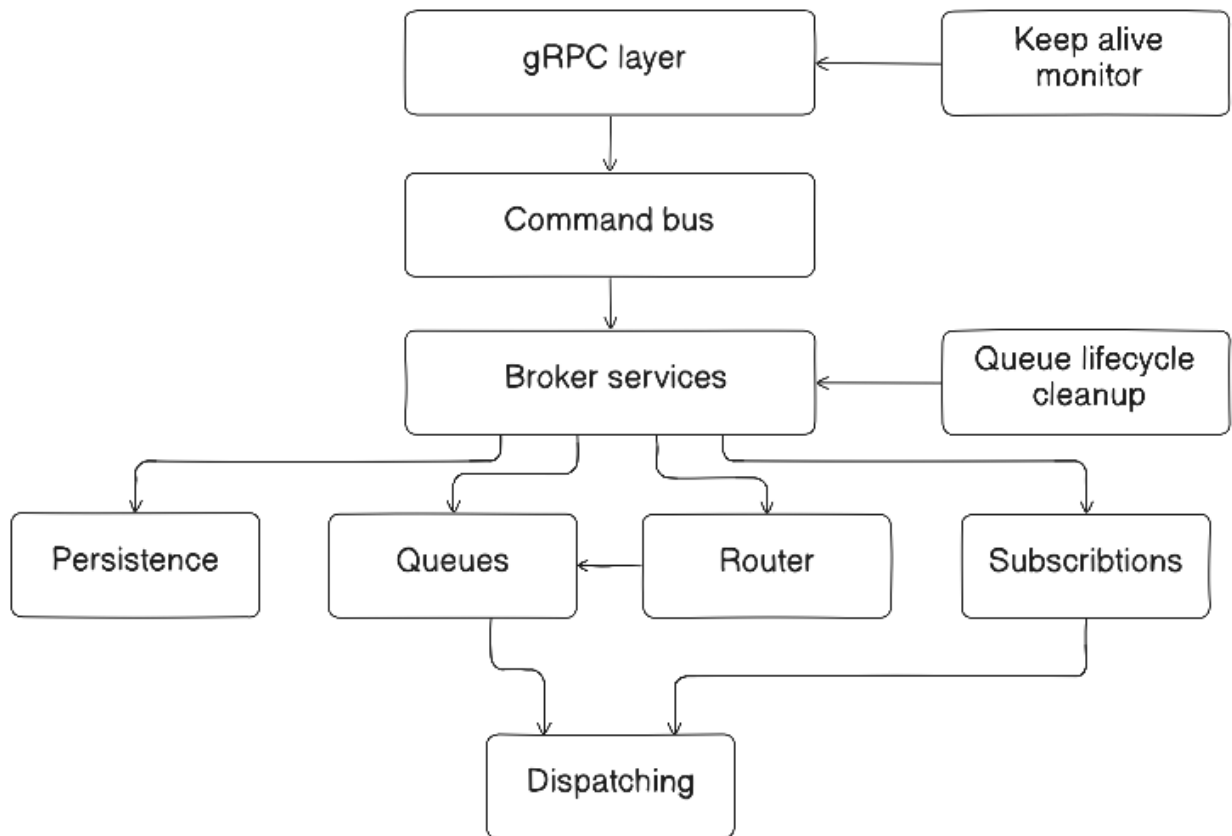


Рис. 2.2.— Загальна архітектура системи асинхронного обміну повідомленнями

Транспортний рівень відповідає за взаємодію з клієнтами. Він відповідає за всю комунікацію між клієнтами за допомогою технології gRPC. Цей рівень не повинен містити основну бізнес-логіку брокера.

Командний рівень перетворює запити клієнтів на внутрішні команди. Завдяки цьому кожна дія, може оброблятися окремо та незалежно від транспортного протоколу.

Рівень основних сервісів брокера відповідає за виконання сценаріїв роботи системи: публікацію повідомлень, керування життєвим циклом повідомлення, обробку підтверджень, повторну доставку та відновлення після перезапуску.

Рівень черг і маршрутизації відповідає за керування чергами, додавання, видалення повідомлень, маршрутизація черг по ключу

Рівень підписок і незавершених доставок відповідає за облік активних отримувачів, вибір отримувача для доставки та контроль повідомлень, які вже передані в обробку, але ще не підтверджені.

Рівень збереження та відновлення використовується для довготривалого зберігання черг і повідомлень у постійній пам'яті, які повинні бути доступні після перезапуску брокера.

Загальний потік роботи системи при публікації повідомлення можна описати так: відправник надсилає повідомлення з ключем маршрутизації; брокер перевіряє запит; система знаходить відповідні черги; повідомлення додається до цих черг; за потреби воно зберігається; після цього запускається процес доставки отримувачам.

Потік роботи отримувача складається з відкриття з'єднання, підписки на чергу, отримання повідомлень, обробки та надсилання позитивного підтвердження (ACK) або негативного (NACK). Якщо отримувач відключається до підтвердження, брокер повинен обробити незавершені доставки відповідно до політики черги.

2.3. Проєктування структури повідомлень та їх життєвого циклу

Повідомлення є основною одиницею обміну в системі. Воно створюється відправником, передається брокеру, маршрутизується до однієї або кількох черг і доставляється отримувачу. Тому під час проєктування потрібно визначити структуру повідомлення та правила його обробки.

Внутрішня модель повідомлення повинна містити такі основні дані:

- `message id` — внутрішній глобальний ідентифікатор повідомлення всередині брокера, який може бути відновлений після перезапуску.
- `routing key` - ключ маршрутизації, з яким повідомлення було опубліковане, він визначає, за яким шаблоном повідомлення має бути зіставлене з чергами.
- `payload` - тіло повідомлення, містить основні дані, які потрібно доставити отримувачу.
- `headers` - додаткові метадані, дозволяють передавати службову або прикладну інформацію, наприклад тип повідомлення або формат даних.
- `timestamp` - час створення повідомлення.
- `delivery tag` - ідентифікатор доставки повідомлення в межах одного підключення між брокером та отримувачем

Життєвий цикл повідомлення залежить від налаштувань черги. Якщо черга працює лише в пам'яті та не очікує підтвердження від отримувача, повідомлення після успішної відправки може вважатися завершеним. Така поведінка відповідає менш надійному, але швидшому сценарію.

Якщо черга налаштована як `persistent` і очікує підтвердження від отримувача, повідомлення повинно залишатися у сховищі до моменту завершення. Після доставки воно потрапляє до набору незавершених доставок. Після АСК повідомлення видаляється з цього набору та зі сховища. Після NACK воно або повертається в чергу, або завершується без повторної доставки, залежно від політики.

У разі перезапуску брокера всі повідомлення, які залишилися у сховищі, розглядаються як незавершені. Під час відновлення вони знову додаються до відповідних черг і можуть бути доставлені повторно. Це дозволяє уникнути складної фіксації кожного проміжного стану в базі даних і водночас зберігає можливість відновлення.

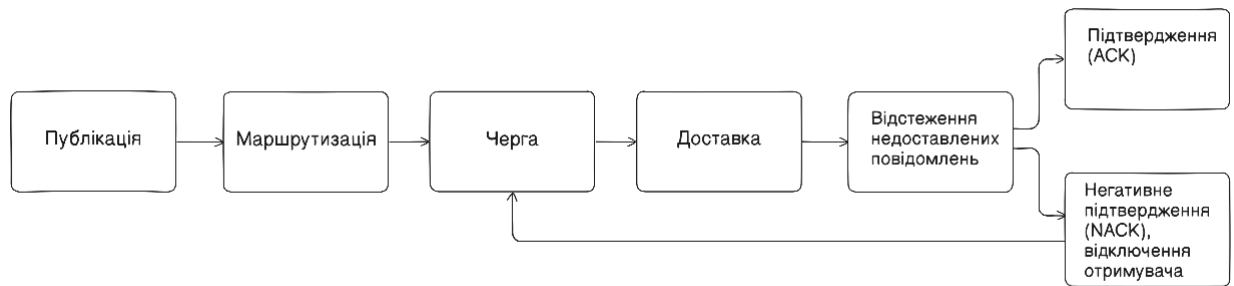


Рис. 2.3.— Життєвий цикл повідомлення від публікації до завершення доставки

2.4. Проєктування механізмів обробки повідомлень

Обробка повідомлень у системі охоплює кілька взаємопов'язаних механізмів: створення черг, публікацію, маршрутизацію, додавання в чергу, доставку, підтвердження, негативне підтвердження, повторну доставку та відновлення після перезапуску.

Перед початком роботи з повідомленнями має бути створена черга. Під час створення задаються її параметри: спосіб зберігання, правила підтвердження, політика повторної доставки, обмеження місткості, політика зберігання повідомлень і правила життєвого циклу черги.

Публікація повідомлення починається з того, що відправник надсилає повідомлення з ключем маршрутизації. Брокер перевіряє коректність запиту, знаходить черги, які відповідають ключу маршрутизації, і додає повідомлення до кожної знайденої черги. Якщо повідомлення не може бути направлено в жодну чергу, відправник має отримати негативну відповідь.

Якщо черга передбачає постійне зберігання, повідомлення записується у сховище до завершення його обробки. Якщо черга працює лише в пам'яті, повідомлення існує в оперативній пам'яті та не відновлюється після перезапуску.

Доставка повідомлень виконується окремим механізмом, який бере повідомлення з черги та передає його доступному отримувачу. Отримувач вважається доступним, якщо він підписаний на відповідну чергу, має активний канал зв'язку та не перевищив ліміт непідтверджених повідомлень.

Якщо для черги увімкнені підтвердження від отримувача, повідомлення після доставки потрапляє до набору незавершених доставок. Після АСК воно вважається успішно обробленим. Після NACK система приймає рішення про повторне повернення в чергу або завершення повідомлення без повторної доставки.

Повторна доставка використовується у випадках, коли повідомлення не було підтверджене, отримувач відключився або сталася помилка доставки. Якщо ліміт досягнуто, брокер тимчасово не надсилає цьому отримувачу нові повідомлення. Кількість повторних доставок має обмежуватися політикою, щоб уникнути нескінченного повторення одного й того самого повідомлення.

Відновлення після перезапуску передбачає завантаження збережених черг, їхніх налаштувань і повідомлень, які не були завершені. Після цього брокер може продовжити доставку повідомлень, прийнятих до перезапуску.

2.5. Алгоритми маршрутизації та розподілу повідомлень

Маршрутизація визначає, до яких черг має потрапити повідомлення після публікації. Під час проєктування було обрано підхід, за якого черга зберігається в ієрархічній послідовності подібно до того як зберігаються файли у файловій системі, де окремі сегменти шляху розділяють логічні рівні, а останній сегмент є назвою конкретної черги. Сегменти шляху розділяються крапкою. Наприклад, `orders.payment.failed` означає чергу `failed`, яка належить до гілки `orders.payment`.

Для підтримки гнучкої маршрутизації можуть використовуватися шаблони з спеціальними символами. Символ «*» означає всі черги які знаходяться в поточній гілці. Символ «#» означає всі черги які знаходяться в поточній гілці та в усіх дочірніх гілках. Наприклад, шаблон `orders.*` може відповідати чергам `orders.created` і `orders.deleted`, а шаблон `orders.#` може відповідати також глибшим шляхам, таким як `orders.payment.failed`, що дозволяє одному повідомленню бути направленим до кількох черг.

Алгоритм маршрутизації повідомлення можна подати так:

1. Отримати повідомлення від відправника.

2. Перевірити ключ маршрутизації.
3. Зіставити ключ або шаблон із зареєстрованими чергами.
4. Якщо відповідних черг немає, повернути відправнику негативну відповідь.
5. Для кожної знайденої черги створити логічну копію повідомлення.
6. Запустити процес доставки для черг, у які потрапило повідомлення.

Розподіл повідомлень між отримувачами залежить від політики доставки. Якщо до черги підключений один отримувач, брокер може доставляти повідомлення саме йому за умови, що він доступний. Якщо до черги підключено кілька отримувачів, брокер має вибрати одного з них.

Для вибору отримувача можуть використовуватися різні стратегії:

Round-robin забезпечує послідовний розподіл повідомлень між отримувачами.

First available обирає першого доступного отримувача.

Least inflight обирає отримувача з найменшою кількістю незавершених доставок.

Вибір стратегії впливає на рівномірність навантаження та ефективність доставки.

Незалежно від стратегії, перед доставкою брокер має перевірити, чи може отримувач прийняти нове повідомлення. Для цього враховується активність підписки, стан каналу та кількість непідтверджених повідомлень. Якщо жоден отримувач не готовий, повідомлення залишається в черзі до появи доступного підписника.

Висновки до розділу 2

У другому розділі було виконано проєктування системи асинхронного обміну повідомленнями. На основі аналізу предметної області сформульовано функціональні вимоги до системи, серед яких створення та видалення черг, публікація повідомлень, маршрутизація за ключ, доставка і повторна доставка, підтвердження обробки, та відновлення після перезапуску.

Було спроектовано загальну архітектуру програмної системи, у вигляді багаторівневої структури. Спроектовано структуру повідомлення та його життєвого циклу, а також алгоритми маршрутизації та розподілу повідомлень між отримувачами з використанням ієрархічної структури шляхів черг і шаблонів маршрутизації.

Розділ 3. Реалізація серверної частини брокера повідомлень

3.1. Реалізація серверної частини брокера

Серверний брокер реалізовано як Java-застосунок, у якому окремі частини системи поділено за відповідальністю. Такий поділ спрощує розуміння роботи проєкту, оскільки транспортний рівень, обробка команд, робота з чергами, маршрутизація, доставка, збереження даних і службові задачі не змішуються між собою.

3.1.1. Вибір технологій та інструментів розробки

Серверну частину брокера повідомлень реалізовано мовою Java. Для побудови проєкту використано Spring Boot, для роботи з даними — Spring Data JPA, для мережевої взаємодії — gRPC і Protocol Buffers. Як основна база даних використовується PostgreSQL.

Таблиця 3.1.1 — Основні технології серверної частини

Технологія	Використання
Java	основна мова реалізації брокера, класів повідомлень, черг, команд і сервісів
Spring Boot	запуск серверного застосунку, створення компонентів і керування залежностями
Spring Data JPA	доступ до таблиць із чергами та повідомленнями через persistence-рівень
gRPC	поточкова взаємодія з відправниками й отримувачами
Protocol Buffers	опис контракту сервісів, повідомлень, налаштувань і помилок
PostgreSQL	збереження довготривалих черг і незавершених повідомлень

Spring Boot використовується для організації проєкту як набору компонентів. Основні класи брокера позначені як сервіси або компоненти, тому вони створюються контейнером Spring і можуть отримувати залежності через конструктори.

gRPC використовується для організації взаємодії із зовнішніми клієнтами. Для брокера повідомлень зручним є те, що gRPC підтримує двонапрямлені потоки. Відправник може тримати відкритий канал і надсилати кілька повідомлень підряд, а отримувач може через постійне з'єднання приймати доставки та надсилати підтвердження.

Protocol Buffers застосовується для формального опису контракту взаємодії. У файлі `broker.proto` визначено сервіси публікації, отримання повідомлень, керування чергами, структуру повідомлень, підтверджень, помилок і налаштувань черги. Після генерації Java-класів серверна частина працює з типізованими об'єктами, а не з довільними текстовими структурами.

PostgreSQL використовується як основна база даних для збереження черг та повідомлень. У ній зберігаються тільки ті дані, які потрібні для відновлення роботи брокера: визначення довготривалих черг, їхні налаштування та незавершені повідомлення.

3.1.2. Структура серверного проєкту

Структура серверного проєкту побудована навколо пакета `com.volo.broker`. Усередині нього виділено кілька великих частин: основна логіка брокера, транспортний рівень, `persistence`-рівень, відновлення, фонові задачі та конфігурація. Такий поділ дає змогу швидко визначити, де розміщена логіка певної операції.

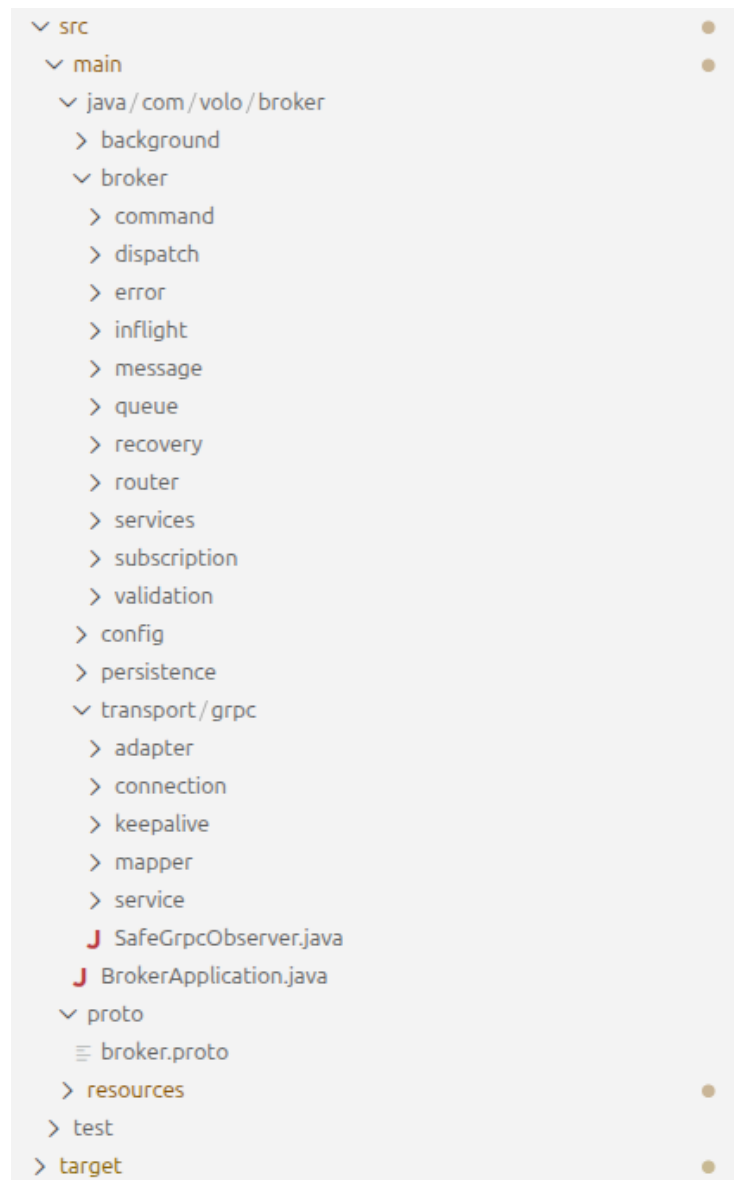


Рис. 3.1.1 — Структура серверного проєкту

У пакеті `broker.command` описано команди та їхні обробники. Команди виступають внутрішнім представленням дій, які надходять від клієнтів: публікація повідомлення, підписка, підтвердження, відхилення, створення або видалення черги. Завдяки цьому транспортний рівень не виконує основну логіку, а лише перетворює зовнішній запит у команду.

Пакет `broker.services` містить сервіси, у яких розміщена основна бізнес-логіка брокера. До них належать `PublishService`, `QueueDeclarationService`, `ConsumerAcknowledgementService`, `MessageLifecycleService`, `QueueLifecycleService` і `QueuePersistenceService`. Саме ці класи координують роботу черг, збереження, підтвердження та завершення повідомлень.

Пакет `broker.queue` відповідає за представлення черг у процесі роботи сервера. У ньому розміщені `BrokerQueue`, `QueueManager`, `InMemoryQueueManager`, `EnqueueResult`, винятки та набір налаштувань. Черга не є простою колекцією повідомлень: вона має власні правила місткості, зберігає час активності, підтримує повернення повідомлень і працює з політиками, заданими під час створення.

Пакет `broker.router` відповідає за маршрутизацію. У ньому реалізовано представлення шляху черги, шаблону маршруту, перевірку сегментів і деревоподібну структуру маршрутів. Цей пакет не займається доставкою повідомлень. Його задача полягає в тому, щоб за ключем маршрутизації знайти відповідні черги.

Пакети `broker.subscription` та `broker.inflight` відповідають за підключених отримувачів і повідомлення, які вже передані на обробку, але ще не підтверджені.

Пакет `broker.dispatch` містить компоненти доставки. `QueueDispatchScheduler` запускає доставку для конкретної черги, а `DispatchManager` виконує цикл вибору отримувача, отримання повідомлення з черги та відправку його у канал отримувача.

Пакет `transport.grpc` містить серверні gRPC-сервіси, адаптери, реєстри з'єднань, перетворювачі об'єктів і keep-alive механізми. Цей рівень працює з мережевими потоками.

У пакеті `persistence` розміщено JPA-сутності, репозиторії та сховища для черг і повідомлень. Пакет `broker.recovery` містить сервіс відновлення, який під час запуску застосунку завантажує збережені черги та незавершені повідомлення.

Узагальнений шлях виконання запиту має такий вигляд: gRPC-сервіс приймає зовнішній запит, створює команду, команда передається в `CommandBus`, далі викликається відповідний обробник і сервіс основної логіки. Після цього сервіс працює з чергами, маршрутизатором, підписками, сховищем або планувальником доставки.

3.1.3. gRPC-контракт взаємодії з брокером

Взаємодія із сервером описана у файлі `broker.proto`. У ньому визначено три сервіси: сервіс публікації повідомлень, сервіс отримання повідомлень і сервіс керування чергами. Перші два сервіси працюють через двонапрямлені потоки, оскільки і відправник, і отримувач взаємодіють із брокером не одноразово, а протягом відкритого з'єднання.

```
service PublisherService {
  rpc OpenPublisherChannel(stream PublisherRequest) returns (stream PublisherResponse);
}

service ConsumerService {
  rpc OpenChannel(stream ConsumerRequest) returns (stream BrokerResponse);
}

service QueueService {
  rpc DeclareQueue(DeclareQueueRequest) returns (QueueOperationResponse);
  rpc DeleteQueue(DeleteQueueRequest) returns (QueueOperationResponse);
}
```

Рис. 3.1.— Фрагмент `broker.proto` з сервісами

Сервіс `PublisherService` відкриває канал для відправників. Через цей канал клієнт може надсилати повідомлення для публікації, службові `ping`-повідомлення та команду закриття з'єднання. У відповідь брокер повертає підтвердження прийняття, повідомлення про помилку або відповідь на `keep-alive`.

Сервіс `ConsumerService` призначений для отримувачів. Отримувач відкриває канал, підписується на чергу, отримує повідомлення й надсилає результат обробки. Через цей же канал передаються підтвердження, відхилення та службові повідомлення для перевірки активності з'єднання.

Сервіс `QueueService` використовується для адміністративних операцій із чергами. На відміну від потокових сервісів публікації та отримання, він працює через звичайні запити: створення та видалення черги.

У `broker.proto` також описано структури даних. Для публікації використовується повідомлення з ключем маршрутизації, корисним навантаженням і заголовками. Для доставки отримувачу передається повідомлення разом зі службовими даними доставки. Для помилок використовується спільна структура з кодом, текстом і додатковими деталями.

```

message MessageDelivery {
    int64 delivery_tag = 1;
    Message message = 2;
    bool redelivered = 3;
    int32 delivery_attempt = 4;
}

message PublishAction {
    int64 client_publish_id = 1;
    string routing_key = 2;
    bytes payload = 3;
    map<string, string> headers = 4;
}

message QueueSettings {
    PersistenceMode persistence_mode = 1;
    PublishAckMode publish_ack_mode = 2;
    ConsumerAckMode consumer_ack_mode = 3;
    RedeliveryPolicy redelivery_policy = 4;
    MessageRetentionPolicy retention_policy = 5;
    QueueLifecyclePolicy lifecycle_policy = 6;
    QueueCapacityPolicy capacity_policy = 7;
    DispatchPolicy dispatch_policy = 8;
}

message ErrorInfo {
    ErrorCode code = 1;
    string message = 2;
    map<string, string> details = 3;
}

```

Рис. 3.1.— Фрагмент broker.proto з основними структурами

Через QueueSettings клієнт може задати режим зберігання, підтвердження публікації, підтвердження отримувачем, повторну доставку, обмеження часу життя повідомлень, життєвий цикл черги, місткість і параметри доставки.

На транспортному рівні gRPC-запити не обробляються напряму як бізнес-операції. Grpc сервіси після отримання запитів перетворюють їх у відповідні об'єкти команд, а потім передають в командний рівень.

Для перетворення між gRPC-структурами та внутрішніми об'єктами використовуються окремі mapper-класи. Завдяки чому сервіси брокера не мають залежності від деталей protobuf-повідомлень. Основна логіка працює з власними командами та моделями, а транспортний рівень займається тільки прийманням і перетворенням даних.

Під час роботи з довготривалими потоками важливо безпечно надсилати відповіді клієнту. Для цього використовується `SafeGrpcObserver`, який запобігає повторному використанню вже закритого потоку та синхронізує надсилання відповідей.

3.1.4. Командна обробка запитів

Командний рівень реалізовано за допомогою патерну `Command Bus`.

Після прийняття запиту транспортний рівень створює внутрішню команду. Команда є простим об'єктом, який описує дію, що має бути виконана брокером. Наприклад, для публікації використовується `PublishCommand`.

Усі команди реалізують спільний інтерфейс `TransportCommand`. Це дозволяє передавати різні типи дій через один механізм виконання. Команда не містить складної логіки. Вона тільки переносить підготовлені дані від транспортного рівня до відповідного обробника.

Центральним елементом командного рівня є клас `CommandBus`. Під час запуску застосунку він отримує список усіх обробників команд і зберігає їх у колекції за типом команди. Коли в систему надходить команда, `CommandBus` знаходить відповідний обробник і викликає його метод `handle`.

```
@Component
public class CommandBus {
    private final Map<Class<? extends TransportCommand>, CommandHandler<?, ?>> handlers = new HashMap<>();

    public CommandBus(List<CommandHandler<?, ?>> handlers){
        for (CommandHandler<?, ?> handler : handlers) {
            this.handlers.put(handler.getCommandType(), handler);
        }
    }

    @SuppressWarnings("unchecked")
    public <C extends TransportCommand, R> R execute(C command){
        CommandHandler<C, R> handler = (CommandHandler<C, R>) handlers.get(command.getClass());

        if (handler == null) {
            throw new IllegalArgumentException("No handler found for command type: " + command.getClass());
        }

        return handler.handle(command);
    }
}
```

Рис. 3.1.4 — Фрагмент реалізації `CommandBus` і обробників команд

Деякі обробники перетворюють винятки на результат операції. Наприклад, під час створення або видалення черги обробники повертають

QueueOperationResult, у якому міститься ознака успішності або код помилки, а результат можна одразу перетворити у gRPC-відповідь.

3.1.5. Реалізація черг і налаштувань черги

Робота з чергами реалізована в пакеті `broker.queue`. У ньому розміщені класи для представлення окремої черги, керування набором створених черг, результату додавання повідомлення, винятків і налаштувань. Черга зберігає повідомлення, контролює розмір, підтримує повернення повідомлення та працює відповідно до заданих політик.

Основним класом є `BrokerQueue`. Він описує одну чергу брокера. Черга має шлях, налаштування, внутрішню колекцію повідомлень, час створення, час останньої активності та сумарний розмір повідомлень у байтах.

Для зберігання повідомлень у `BrokerQueue` використовується двонаправлена черга. Нові повідомлення додаються в кінець, а для доставки береться повідомлення з початку. Якщо повідомлення потрібно повернути на повторну обробку, воно додається на початок.

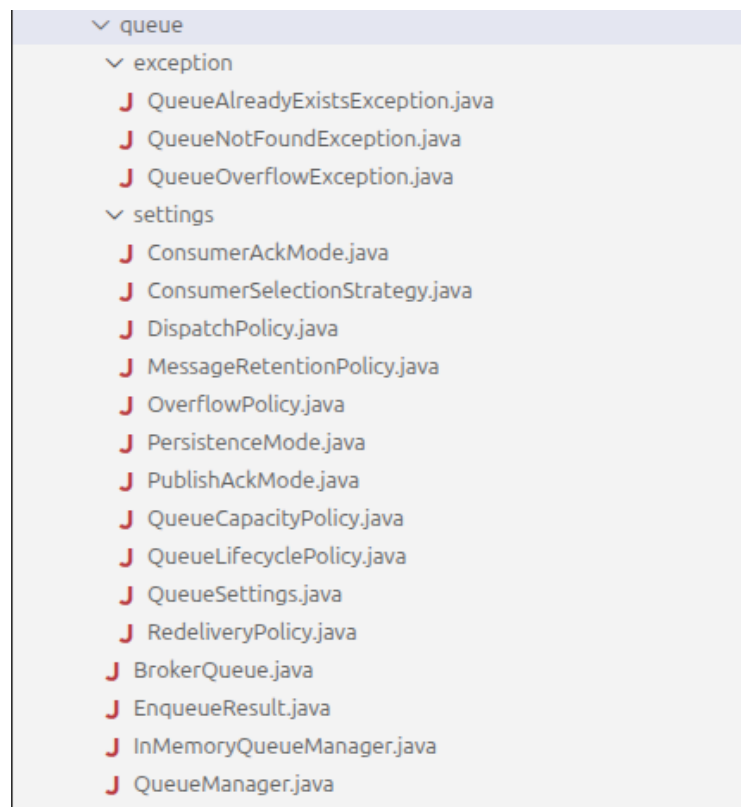


Рис. 3.1.5 — Структура пакета `broker.queue`

Перед додаванням застосовується політика переповнення. Черга може мати обмеження за кількістю повідомлень або за сумарним обсягом даних. Якщо ліміт перевищується, брокер діє відповідно до налаштувань: відхиляє публікацію, відкидає найстаріше або найновіше повідомлення.

Клас `EnqueueResult` фіксує результат додавання повідомлення. У більшості випадків повідомлення просто приймається. Але якщо використовується політика видалення старих повідомлень, результат може містити список повідомлень, які були відкинуті для звільнення місця. Це дає змогу іншим компонентам знати, що під час додавання були побічні наслідки.

Керування набором черг винесено в інтерфейс `QueueManager`. Він визначає операції створення, видалення, пошуку, отримання черги, маршрутизованого пошуку і отримання всіх черг.

Реалізацією цього інтерфейсу є `InMemoryQueueManager`. Він зберігає створені черги в потокобезпечній колекції, де ключем є шлях черги, а значенням — об'єкт `BrokerQueue`.

Якщо користувач намагається створити чергу, яка вже існує, генерується `QueueAlreadyExistsException`. Якщо під час пошуку або видалення вказано неіснуючу чергу, використовується `QueueNotFoundException`. Помилка переповнення подається через `QueueOverflowException`, що робить причину помилки зрозумілою для подальшого перетворення у відповідь клієнту.

Налаштування черги зібрані в класі `QueueSettings`. До складу налаштувань входять режим зберігання, підтвердження публікації, підтвердження отримувачем, повторна доставка, час життя повідомлень, життєвий цикл черги, місткість і параметри доставки.

Режим зберігання визначає, чи повідомлення повинні записуватися в базу даних. Для черг із постійним зберіганням незавершені повідомлення записуються у сховище і можуть бути відновлені після перезапуску сервера. Для менш критичних повідомлень можливий режим зберігання тільки в оперативній пам'яті.

Режим підтвердження публікації визначає, чи отримає відправник відповідь про результат publish-запиту. Якщо підтвердження увімкнене, клієнт бачить, чи було повідомлення прийняте, чи сталася помилка. У відповіді також вказується кількість черг, до яких повідомлення було направлено.

Режим підтвердження отримувачем визначає, чи потрібно чекати ACK або NACK після доставки повідомлення. Якщо підтвердження увімкнене, повідомлення після відправлення не вважається завершеним. Воно залишається в обліку незавершених доставок до моменту, коли отримувач повідомить результат обробки.

Політика повторної доставки визначає, чи можна повертати повідомлення в чергу після помилки. У ній задається максимальна кількість спроб, а також правила поведінки при розриві з'єднання з отримувачем або помилці надсилання. Це захищає брокер від нескінченного повторення одного проблемного повідомлення.

Політика часу життя повідомлень дозволяє обмежити строк перебування повідомлення в черзі. Якщо повідомлення стало застарілим, воно не передається отримувачу, а завершується під час обробки. Це корисно для повідомлень, які втрачають актуальність через певний час.

Політика життєвого циклу черги описує поведінку самої черги. Вона визначає, чи черга є довготривалою, чи може автоматично видалятися і чи враховується час простою. Для таких правил у проєкті передбачено окрему службу очищення неактивних черг.

Політика місткості задає обмеження кількості повідомлень або їхнього сумарного розміру. Якщо обмеження увімкнене, але не задано жодного ліміту, налаштування вважаються некоректними.

Параметри доставки містять значення prefetch і стратегію вибору отримувача. Значення prefetch обмежує кількість повідомлень, які можуть бути передані одному отримувачу без підтвердження. Стратегія вибору визначає, як брокер обирає отримувача серед кількох підписаних клієнтів.

3.1.6. Маршрутизація повідомлень

Маршрутизація повідомлень реалізована в пакеті `broker.router`. Його задача полягає в тому, щоб представити шляхи черг, перевірити маршрути та знайти черги, які відповідають ключу маршрутизації. Цей пакет відповідає тільки за зіставлення маршруту з наявними чергами.

Для конкретної черги використовується клас `QueuePath`. Він зберігає повний шлях, коротку назву черги, маршрутну частину та список сегментів. З

Для ключа або шаблону маршрутизації використовується `RoutePattern`. На відміну від шляху черги, він може містити спеціальні символи. Символ `*` відповідає одному рівню шляху, а символ `#` охоплює поточний і вкладені рівні. Ці правила дозволяють одній публікації потрапити не тільки в одну конкретну чергу, а й у групу черг.

Перевірка шляхів і шаблонів винесена в `RouteSegments`. Цей клас нормалізує рядок, перевіряє його довжину, розбиває на сегменти та перевіряє кожен сегмент.

Основна структура пошуку реалізована в `RouteTree`. Вона зберігає шляхи черг у вигляді дерева, де кожен сегмент є окремим вузлом. Під час створення черги її шлях додається в дерево. Під час видалення черги відповідний шлях прибирається, щоб він більше не брав участі в пошуку.

Коли брокер отримує ключ маршрутизації, `RouteTree` перетворює його на `RoutePattern` і запускає пошук. Якщо сегмент звичайний, алгоритм переходить тільки до відповідного дочірнього вузла.

Результатом маршрутизації є набір шляхів. Далі ці шляхи використовуються менеджером черг для отримання відповідних об'єктів `BrokerQueue`. Якщо жодної черги не знайдено, публікація завершується негативним результатом, оскільки повідомлення немає куди розмістити.

3.1.7. Створення та видалення черг

Створення та видалення черг реалізовано через `QueueDeclarationService`. Цей сервіс є основною точкою, де поєднуються перевірка шляху черги,

створення об'єкта черги, реєстрація в менеджері та збереження визначення черги за потреби.

Перед створенням черги виконується перевірка її шляху. Для цього використовується `QueuePathValidator`, який повертає об'єкт `QueuePath`. Якщо шлях порожній або має некоректний формат, черга не створюється, а клієнт отримує помилку некоректного запиту. Якщо налаштування черги не передані, використовується набір значень за замовчуванням.

Після підготовки шляху та налаштувань сервіс викликає менеджер черг. Менеджер створює об'єкт `BrokerQueue`, додає його до колекції створених черг і реєструє шлях у дереві маршрутів. Якщо черга з таким шляхом уже існує, операція завершується помилкою. Якщо черга позначена як довготривала, сервіс `QueuePersistenceService` зберігає її.

Видалення черги проходить у зворотному напрямі. Спочатку шлях що прийшов з команди перевіряється і перетворюється на `QueuePath`. Далі сервіс переконується, що така черга існує. Після цього видаляються збережені повідомлення цієї черги, видаляється її сутність зі сховища і тільки після цього черга прибирається з менеджера.

Результат адміністративної операції повертається як `QueueOperationResult`. Він містить ознаку успіху або код помилки.

3.1.8. Публікація повідомлень

Публікація повідомлення починається на транспортному рівні. `GrpcPublisherService` приймає запит від відправника, перевіряє тип та через команду дані відправляються до сервісу публікації `PublishService`. Повідомлення не приймається, якщо команда відсутня, ключ маршрутизації порожній або корисне навантаження не містить даних. Такі помилки повертаються відправнику як некоректний запит.

Після перевірки виконується пошук черг. Для цього `PublishService` звертається до `QueueManager`, який через маршрутизатор знаходить черги, що

відповідають ключу маршрутизації. Якщо список порожній, повідомлення не додається в жодну чергу, а відправник отримує негативний результат.

Якщо знайдено кілька черг, для кожної з них створюється окремий об'єкт повідомлення. Це означає, що одна публікація може створити кілька незалежних повідомлень у різних чергах. Кожне з них має власний ідентифікатор і власний життєвий цикл.

Внутрішній ідентифікатор повідомлення створюється через `MessageIdGenerator`. Реалізація використовує атомарний лічильник. Це забезпечує послідовну генерацію числових ідентифікаторів під час роботи сервера.

Перед додаванням у чергу сервіс перевіряє режим зберігання. Якщо черга налаштована на постійне зберігання, повідомлення записується у базу даних. Після чого повідомлення додається в чергу. На цьому етапі спрацьовують правила місткості самої черги. Якщо повідомлення прийняте, запускається планувальник доставки для цієї черги.

Планувальник доставки отримує сигнал через компонент `QueueDispatchScheduler`. Це означає, що в черзі з'явилася робота, яку потрібно передати отримувачам. Механізм публікації лише додає повідомлення в чергу та повідомляє відповідний компонент.

Результат публікації описується класом `PublishResult`. У разі успіху в ньому зберігається ідентифікатор клієнтського запиту та кількість черг, до яких було направлено повідомлення. У разі помилки результат містить код і текстове пояснення.

3.1.9. Підписки отримувачів і облік незавершених доставок

Отримувач починає роботу з брокером через підписку на чергу. На транспортному рівні gRPC-запит перетворюється у `SubscribeCommand`. У команді передається шлях черги, бажане значення `prefetch`, службовий тег отримувача та канал, через який брокер зможе надсилати повідомлення.

Значення `prefetch` визначає кількість повідомлень, які можуть бути доставлені отримувачу без підтвердження. Якщо клієнт не передав власне значення або передав некоректне, використовується значення з налаштувань черги. Якщо клієнт передав менше значення, воно може бути використане як більш обережне обмеження.

Клас `Subscription` описує одну активну підписку. Він зберігає тег отримувача, шлях черги, `prefetch`-ліміт, канал для відправлення повідомлень і облік незавершених доставок.

Облік незавершених доставок реалізовано через `InFlightTracker`. Для кожного повідомлення, яке було передане отримувачу і потребує підтвердження, створюється службовий тег доставки. За цим тегом потім можна знайти повідомлення під час `ACK` або `NACK`.

Реалізація використовує потокобезпечну колекцію. Вона дозволяє згенерувати новий тег доставки, зареєструвати повідомлення, видалити повідомлення після підтвердження або отримати всі незавершені повідомлення під час відключення отримувача.

3.1.10. Доставка повідомлень отримувачам

Доставка повідомлень виконується окремими компонентами пакета `broker.dispatch`. За запуск доставки відповідає `QueueDispatchScheduler`, а за сам цикл доставки — `DispatchManager`.

```

14 public class QueueDispatchScheduler {
15     private final DispatchManager dispatchManager;
16     private final QueueManager queueManager;
17     private final ExecutorService executor;
18     private final Set<String> dispatchingQueues = ConcurrentHashMap.newKeySet();
19
20     public QueueDispatchScheduler(
21         DispatchManager dispatchManager,
22         QueueManager queueManager,
23         @Value("${broker.dispatch.threads:2}") int threads
24     ) {
25         this.dispatchManager = dispatchManager;
26         this.queueManager = queueManager;
27         this.executor = Executors.newFixedThreadPool(Math.max(a: 1, threads));
28     }
29
30     public void signal(String queueName){
31         if(!dispatchingQueues.add(queueName)) return;
32
33         executor.execute(() -> {
34             try{
35                 dispatchManager.dispatchLoop(queueName);
36             } finally {
37                 dispatchingQueues.remove(queueName);
38
39                 if(hasWork(queueName)){
40                     signal(queueName);
41                 }
42             }
43         });
44     }

```

Рис. 3.1. — фрагмент QueueDispatchScheduler

Коли в черзі з'являється повідомлення або реєструється новий отримувач, викликається метод `signal` планувальника доставки. Планувальник перевіряє, чи для цієї черги вже не виконується доставка. Якщо обробка вже запущена, повторний запуск не створюється. Це запобігає одночасній обробці однієї черги кількома потоками.

Планувальник використовує пул потоків. Кількість потоків задається через конфігурацію. Це дозволяє обробляти кілька черг паралельно, але не створювати необмежену кількість потоків для кожного нового повідомлення.

Після запуску планувальник викликає `DispatchManager.dispatchLoop`. Цей цикл працює доти, доки є повідомлення, активний отримувач і можливість передати наступне повідомлення. Якщо будь-яка з цих умов не виконується, цикл завершується.

На початку кожної ітерації `DispatchManager` отримує чергу та шукає готову підписку. Вибір отримувача виконується відповідно до політики доставки,

заданої в налаштуваннях черги. Якщо немає жодного отримувача, який може прийняти повідомлення, доставка припиняється.

Перед відправкою перевіряється prefetch-обмеження. Якщо отримувач уже має максимальну кількість незавершених доставок, нове повідомлення йому не передається. Це не дозволяє брокеру перевантажити отримувача та забезпечує більш контрольований розподіл повідомлень.

Якщо для черги увімкнено підтвердження отримувачем, перед відправленням створюється тег доставки, а повідомлення додається до обліку незавершених доставок. Якщо підтвердження вимкнене, повідомлення після успішного надсилання може бути завершене одразу.

Фактичне надсилання виконується через `ConsumerChannel`. Для gRPC-з'єднання використовується адаптер `GrpcConsumerChannelAdapter`, який перетворює внутрішню модель доставки у protobuf-повідомлення та надсилає її у відкритий канал отримувача.

Після завершення циклу планувальник прибирає чергу зі списку активних обробок. Якщо після цього в черзі ще залишаються повідомлення, доставка запускається повторно. Це дозволяє обробляти нові повідомлення без одночасного запуску кількох однакових циклів для однієї черги.

3.1.11. Підтвердження, відхилення та повторна доставка

Після отримання повідомлення отримувач повинен повідомити брокер про результат обробки, якщо для черги увімкнено підтвердження. Для цього використовуються дві дії: позитивне підтвердження та негативне. На транспортному рівні вони перетворюються у `AckCommand` і `NackCommand`.

Обробку цих команд виконує `ConsumerAcknowledgementService`. Він знаходить підписку за тегом отримувача, видаляє повідомлення з обліку незавершених доставок і далі діє залежно від типу команди.

Позитивне підтвердження означає, що повідомлення успішно оброблене. У цьому випадку брокер завершує життєвий цикл повідомлення через `MessageLifecycleService`. Якщо повідомлення було збережене в базі даних, воно

видаляється зі сховища. Після цього запускається доставка наступних повідомлень цієї черги.

Відхилення повідомлення обробляється інакше. Якщо отримувач просить повернути повідомлення в чергу і політика повторної доставки це дозволяє, повідомлення повертається назад як повторно доставлюване. Якщо повторна доставка не дозволена або кількість спроб вичерпана, повідомлення завершується.

```

33     public void ack(AckCommand command) {
34         subscriptionManager.findByConsumerTag(command.consumerTag())
35             .ifPresent(subscription -> {
36             QueuedMessage message = subscription.remove(command.deliveryTag());
37             if (message == null) {
38                 return;
39             }
40
41             BrokerQueue queue = queueManager.require(subscription.queueName());
42             lifecycleService.complete(queue, message);
43             queue.touch();
44
45             dispatchScheduler.signal(queue.queuePath());
46         });
47     }
48
49     public void nack(NackCommand command) {
50         subscriptionManager.findByConsumerTag(command.consumerTag())
51             .ifPresent(subscription -> {
52             QueuedMessage message = subscription.remove(command.deliveryTag());
53             if (message == null) {
54                 return;
55             }
56
57             BrokerQueue queue = queueManager.require(subscription.queueName());
58
59             if (command.requeue() && lifecycleService.canRedeliver(queue, message)) {
60                 queue.requeue(message.asRedelivered());
61             } else {
62                 lifecycleService.complete(queue, message);
63             }
64
65             queue.touch();
66             dispatchScheduler.signal(queue.queuePath());
67         });
68     }
69 }

```

Рис. 3.1. — Фрагмент реалізації ConsumerAcknowledgementService

Перевірка можливості повторної доставки виконується в MessageLifecycleService. Сервіс перевіряє, чи увімкнена політика повторної доставки і чи не перевищено максимальну кількість спроб. Така перевірка винесена в окремий сервіс, щоб однакові правила використовувалися під час NACK, помилки надсилання та відключення отримувача.

Якщо повторна доставка при відключенні дозволена, повідомлення повертається в чергу. Якщо політика не дозволяє повторне повернення або ліміт спроб уже вичерпано, повідомлення завершується. Для збереженого повідомлення завершення означає видалення з бази даних.

```

30  @Override
31  public void handle(DisconnectCommand cmd) {
32      subscriptionManager.findByConsumerTag(cmd.consumerTag())
33          .ifPresent(sub -> {
34              subscriptionManager.unregister(cmd.consumerTag());
35
36              List<QueuedMessage> drained =
37                  new ArrayList<>(sub.getInFlightTracker().drainAll());
38
39              dispatchScheduler.signal(sub.queueName());
40
41
42
43              for (int i = drained.size() - 1; i >= 0; i--) {
44                  QueuedMessage message = drained.get(i);
45
46                  var queue = queueManager.find(sub.queueName()).get();
47
48                  boolean shouldRequeue = queue.settings().redeliveryPolicy().enabled()
49                      && queue.settings().redeliveryPolicy().requeueOnDisconnect()
50                      && message.getDeliveryAttempt() < queue.settings().redeliveryPolicy().maxAttempts();
51
52                  if (shouldRequeue) {
53                      queueManager.find(sub.queueName()).ifPresent(q -> q.requeue(message.asRedelivered()));
54                  } else {
55                      messageLifecycleService.complete(queue, message);
56                  }
57              }
58          });
59
60      return null;
61  }

```

Рис. 3.1.— Фрагмент обробки відключення отримувача

Повернення незавершених повідомлень під час відключення виконується у зворотному порядку. Це потрібно для того, щоб після додавання на початок черги порядок подальшої доставки залишався передбачуваним. Такий підхід зменшує ризик випадкового перемішування повідомлень, які перебували в обробці одного отримувача.

Після ACK, NACK або відключення брокер сигналізує планувальнику доставки. Це потрібно тому, що після зменшення кількості незавершених повідомлень отримувач або інші отримувачі можуть знову приймати нові повідомлення. Таким чином, підтвердження не лише завершує одне повідомлення, а й запускає подальший рух черги.

Повторна доставка не гарантує, що повідомлення буде оброблене рівно один раз. Якщо отримувач встиг виконати дію, але не встиг надіслати ACK, брокер може доставити повідомлення повторно. Тому реалізована поведінка

відповідає практичному підходу, у якому брокер забезпечує контроль втрати повідомлень, але не усуває повністю можливість дублювання.

3.1.12. Збереження даних і відновлення після перезапуску

Для підтримки відновлення після перезапуску в проєкті реалізовано persistence-рівень. Він складається з JPA-сутностей, репозиторіїв, інтерфейсів сховищ та їх реалізацій. У базі даних зберігаються визначення черг і повідомлення, які ще не були завершені.

Для черг використовується інтерфейс QueueStore. Його реалізація JpaQueueStore зберігає чергу як StoredQueueEntity. У таблиці черг зберігається шлях черги та всі налаштування, необхідні для повторного створення черги після запуску сервера.

Збереження черги виконується тільки тоді, коли політика життєвого циклу позначає її як довготривалу.

Для повідомлень використовується інтерфейс MessageStore. Для кожного запису зберігається ідентифікатор повідомлення, шлях черги, ключ маршрутизації, корисне навантаження, заголовки у JSON-форматі, час створення, ознака повторної доставки та номер спроби.

Повідомлення записується в базу даних під час публікації, якщо воно потрапляє до черги з постійним зберіганням. Після цього повідомлення залишається у сховищі доти, доки не буде завершене. Завершенням може бути успішний АСК, НАСК без повторної доставки, закінчення строку актуальності або вичерпання спроб повторної доставки.

Відновлення стану брокера виконує BrokerRecoveryService. Він реалізує ApplicationRunner, тому запускається автоматично під час старту Spring Boot-застосунку. Відновлення складається з трьох етапів: завантаження черг, завантаження повідомлень і відновлення генератора ідентифікаторів.

3.1.13. Фонові задачі, keep-alive та обробка помилок

Крім основної логіки публікації й доставки, у брокері реалізовано кілька службових механізмів. Вони не створюють повідомлення і не виконують маршрутизацію, але потрібні для стабільної роботи сервера: очищення неактивних черг, контроль довготривалих з'єднань і перетворення внутрішніх помилок у зрозумілі відповіді клієнтам.

Життєвий цикл черг контролює QueueLifecycleService. Він переглядає всі створені черги та визначає, чи потрібно видалити чергу. Черга не видаляється, якщо в ній є повідомлення або якщо до неї підключені отримувачі. Далі враховуються параметри життєвого циклу: автоматичне видалення та час простою.

Для періодичного запуску очищення використовується фонові задача. Вона викликає сервіс життєвого циклу та дозволяє прибирати черги, які більше не використовуються. Такий механізм особливо корисний для тимчасових черг, створених для короткочасної взаємодії.

```

23     public void cleanupIdleQueues() {
24         long now = System.currentTimeMillis();
25
26         for (BrokerQueue queue : queueManager.all()) {
27             if (shouldDelete(queue, now)) {
28                 queueManager.delete(queue.path());
29             }
30         }
31     }
32
33     boolean shouldDelete(BrokerQueue queue, long now) {
34         if (!queue.isEmpty()) {
35             return false;
36         }
37
38         if (subscriptionManager.hasSubscribers(queue.queuePath())) {
39             return false;
40         }
41
42         if (queue.settings().lifecyclePolicy().autoDelete()) {
43             return true;
44         }
45
46         if (!queue.settings().lifecyclePolicy().idleTimeoutEnabled()) {
47             return false;
48         }
49
50         long idleForMs = now - queue.lastActivityAt();
51
52         return idleForMs >= queue.settings().lifecyclePolicy().idleTimeoutMs();
53     }

```

Рис. 3.1. — Фрагмент реалізації очищення неактивних черг

Контроль активності з'єднань реалізовано через `keep-alive` механізм. Для відправників і отримувачів існують окремі реєстри з'єднань. Кожне з'єднання має ідентифікатор, час останньої активності, метод перевірки закриття, метод закриття та метод оновлення активності.

Для перевірки неактивних з'єднань використовується абстрактний монітор `AbstractKeepAliveMonitor`. Він періодично переглядає всі активні з'єднання в реєстрі. Якщо з'єднання вже закрито, воно прибирається з реєстру. Якщо час з останньої активності перевищив допустиме значення, монітор закриває це з'єднання.

Для відправника закриття з'єднання простіше: достатньо завершити потоковий канал і прибрати запис із реєстру. Проте і для відправника, і для отримувача час останньої активності оновлюється при надходженні запитів або `ping`-повідомлень.

3.2. Реалізація клієнтської частини

Клієнтська частина системи реалізована як окремий Java-модуль, який спрощує взаємодію зовнішніх програм із брокером повідомлень. Її призначення полягає не в дублюванні логіки брокера, а в наданні зручного програмного інтерфейсу для створення черг, публікації повідомлень, підписки на черги, обробки отриманих повідомлень і виконання допоміжних сценаріїв перевірки роботи системи.

Структура клієнтської частини поділена на кілька пакетів. У кореневому пакеті розміщені основні класи для створення підключення та конфігурації. Пакет `publisher` містить реалізацію відправника повідомлень. Пакет `consumer` відповідає за підписку на черги та обробку доставлених повідомлень. Пакет `queue` використовується для адміністративних операцій із чергами. Пакет `connection` містить політику повторного підключення. У пакеті `internal` розміщені допоміжні класи, які не повинні напряму використовуватися ззовні.

Окремо винесений пакет `simulation`, який застосовується для запуску сценаріїв навантаження та збору статистики.

Центральним класом клієнтської частини є `BrokerClient`. Він відповідає за створення gRPC-каналу до сервера та надає доступ до основних можливостей клієнтської бібліотеки. Підключення створюється на основі конфігурації `BrokerClientConfig`, у якій задаються адреса брокера, порт, режим з'єднання, інтервали `keep-alive`, час очікування відповіді на публікацію, час завершення роботи каналу та максимальний розмір вхідного повідомлення.

Клас `BrokerClient` приховує від користувача створення gRPC-заглушок. Для адміністративних операцій використовується `blocking stub` сервісу черг, а для відправника й отримувача створюються асинхронні `stub`-и. Завдяки цьому користувач клієнтської бібліотеки не працює напяму з низькорівневими gRPC-об'єктами. Він створює клієнт, після чого може отримати об'єкт для керування чергами, відкрити відправника або створити отримувача.

Для зручності створення об'єктів використано `builder`-підхід. Це дозволяє задавати тільки потрібні параметри, а решту залишати типовими. Наприклад, для локального запуску достатньо вказати адресу та порт брокера. Додаткові параметри використовуються тоді, коли потрібно змінити час очікування відповіді, інтервал службових повідомлень або максимальний розмір повідомлення.

Для роботи з чергами реалізований клас `QueueAdminClient`. Він використовує `QueueService` і дозволяє створювати та видаляти черги. Під час створення черги клієнт передає повний шлях черги та її налаштування. Якщо черга вже існує, поведінка залежить від `QueueDeclarationOptions`. У режимі `FAIL` така ситуація вважається помилкою, а в режимі `USE_EXISTING` клієнт може продовжити роботу з уже створеною чергою.

Налаштування черги на стороні клієнта формуються за допомогою `QueueSettingsBuilder`. Цей клас дозволяє поступово задати режим

зберігання, підтвердження публікації, підтвердження отримувачем, правила повторної доставки, TTL, життєвий цикл черги, обмеження місткості та параметри доставки. Також передбачені готові методи для типових сценаріїв: надійної *persistent*-черги та швидкої *memory-only*-черги. Це спрощує створення черг, оскільки користувач не повинен вручну формувати всі *protobuf*-структури.

У *QueueSettingsBuilder* також реалізовано перевірку сумісності параметрів. Наприклад, черга з постійним зберіганням не повинна створюватися без підтверджень з боку отримувача, оскільки в такому випадку брокер не зможе коректно визначити момент завершення повідомлення. Також повторна доставка не має сенсу без підтверджень, адже брокер повинен знати, чи повідомлення було успішно оброблене.

Публікація повідомлень реалізована в пакеті *publisher*. Основним класом є *PublisherClient*. Під час створення він відкриває потоковий *gRPC*-канал до брокера і надалі використовує його для надсилання повідомлень. Для кожної публікації створюється службовий ідентифікатор запиту. Він потрібен для того, щоб зіставити відповідь брокера з конкретною публікацією, оскільки в одному потоці може одночасно перебувати кілька повідомлень, які очікують результату.

Запит на публікацію описаний класом *PublishRequest*. Він містить ключ маршрутизації, тіло повідомлення та заголовки. У конструкторі виконується базова перевірка. Масив байтів копіюється, щоб зовнішній код не міг змінити вміст повідомлення після створення запиту.

Після надсилання повідомлення *PublisherClient* очікує відповідь від брокера. Якщо брокер підтвердив публікацію, клієнт отримує *PublishReceipt*. У ньому міститься ідентифікатор клієнтського запиту та кількість черг, до яких було маршрутизовано повідомлення. Якщо брокер повертає помилку, вона перетворюється на виняток клієнтської бібліотеки. Таким чином, код користувача не залежить напряду від *protobuf*-представлення помилок.

Для зберігання запитів, які вже надіслані, але ще не отримали відповіді, використовується колекція очікуваних публікацій. Кожному ідентифікатору відповідає `CompletableFuture`. Коли від брокера надходить АСК, відповідний `future` завершується успішно. Якщо надходить НАСК або потік завершується помилкою, очікування завершується винятком. Це дозволяє використовувати асинхронну модель публікації та не блокувати потік виконання на час очікування відповіді.

Окрім звичайної публікації з очікуванням результату, у клієнті передбачено режим `fire-and-forget`. У цьому випадку повідомлення надсилається в потік, але клієнт не очікує підтвердження. Такий режим може бути корисним для службових повідомлень або сценаріїв, де швидкість важливіша за контроль результату. Проте для надійної доставки доцільніше використовувати режим з очікуванням підтвердження.

Для підтримки активності з'єднання `PublisherClient` періодично надсилає `ping`-повідомлення. Якщо надсилання службового повідомлення завершується помилкою, клієнт закриває потік і завершує всі незавершені публікації з помилкою. Під час закриття клієнт також намагається передати брокеру команду завершення каналу.

Отримання повідомлень реалізовано в пакеті `consumer`. Основним класом є `ConsumerClient`. Під час запуску він відкриває потоковий канал до брокера, надсилає запит на підписку та очікує відповіді від сервера. Параметри підписки задаються через `ConsumerOptions`. У них вказуються шлях черги, значення `prefetch`, режим підтвердження, кількість потоків для обробки повідомлень, обробник повідомлення та додаткові параметри поведінки при помилці.

Доставлене брокером повідомлення перетворюється на об'єкт `Delivery`. Цей клас приховує `protobuf`-представлення повідомлення та надає доступ до основних даних: ідентифікатора доставки, ідентифікатора повідомлення, тіла, ключа маршрутизації, заголовків, ознаки повторної доставки та номера спроби.

У клієнті передбачено три режими підтвердження. У режимі `MANUAL` користувач сам викликає `ack()` або `nack()`. У режимі `AUTO` клієнт автоматично підтверджує повідомлення після успішного виконання обробника. Якщо під час обробки виникає помилка, клієнт може надіслати `NACK` і, залежно від налаштування, попросити брокер повернути повідомлення в чергу. У режимі `NONE` підтвердження вимкнені, а виклик `ACK` або `NACK` на стороні клієнта вважається помилкою.

Обробка повідомлень виконується в окремому пулі потоків. Кількість потоків задається в `ConsumerOptions`, тому для простих сценаріїв можна використовувати один потік, а для більш інтенсивної обробки — кілька.

Для повторного підключення реалізовані класи `ReconnectingPublisherClient` і `ReconnectingConsumerClient`. Вони використовують `ReconnectPolicy`, де задається, чи дозволені повторні спроби, максимальна кількість спроб, початкова затримка та максимальна затримка. Політика може бути вимкнена, обмежена певною кількістю спроб або необмежена.

У випадку отримувача `reconnect`-клієнт працює у власному службовому потоці. Він відкриває підключення, створює `ConsumerClient` і стежить за його станом. Якщо з'єднання закривається або виникає помилка, клієнт закриває поточні ресурси, очікує відповідну затримку та повторно відкриває підключення. Після повторного підключення підписка створюється заново з тими самими параметрами.

Для відправника повторне підключення має обмеження, пов'язані з можливою невизначеністю результату. Якщо повідомлення було надіслане, але відповідь від брокера не отримана через обрив з'єднання, клієнт не завжди може знати, чи брокер прийняв це повідомлення. Тому за замовчуванням повторна публікація в такому випадку не виконується автоматично. Окремий режим дозволяє повторювати такі публікації, але його потрібно використовувати обережно, оскільки це може призвести до дублювання повідомлень.

Помилки, які повертає брокер, перетворюються на об'єкт класу `BrokerRemoteException`. Він зберігає код помилки, текстове пояснення та додаткові деталі. Локальні помилки клієнтської бібліотеки представлені через `BrokerClientException`. Завдяки цьому можна відрізнити помилку, яку повернув брокер, від помилки, що виникла всередині клієнта.

Окремим пакетом клієнтського модуля є пакет `simulation`. Він використовується для перевірки брокера під навантаженням. Клас `BrokerSimulation` запускає сценарій для однієї черги: створює чергу, запускає задану кількість отримувачів, створює відправників і надсилає визначену кількість повідомлень. Для кількох черг використовується `MultiQueueBrokerSimulation`, який дозволяє окремо задавати параметри для кожної черги.

У симуляції повідомлення містить службові заголовки з часом публікації, номером відправника та номером повідомлення. Після отримання повідомлення клієнт обчислює затримку від моменту публікації до моменту обробки. Ці дані зберігаються в `SimulationStats`. Це дозволяє оцінити кількість успішно опублікованих і доставлених повідомлень, середню та максимальну затримку доставки.

За допомогою симуляції можна перевіряти різні конфігурації: кількість відправників, кількість отримувачів, розмір повідомлення, `prefetch`, кількість черг і час очікування завершення обробки. Це дає змогу порівняти поведінку брокера за різних умов без написання окремих тестових програм для кожного сценарію.

3.3. Тестування та експериментальне дослідження

Після реалізації серверної та клієнтської частин було виконано перевірку працездатності системи. Тестування проводилося у двох напрямках: модульна

перевірка компонентів брокера та експериментальне дослідження роботи системи під навантаженням за допомогою клієнтської симуляції.

Модульне тестування використовувалося для перевірки окремих класів і сервісів серверної частини. Було перевірено командний рівень, обробники команд, черги, маршрутизацію, підписки, облік незавершених доставок, доставку повідомлень, persistence-рівень, keep-alive та gRPC-компоненти. За результатами запуску всі тести були виконані успішно: 225 тестів із 225 завершилися без помилок.

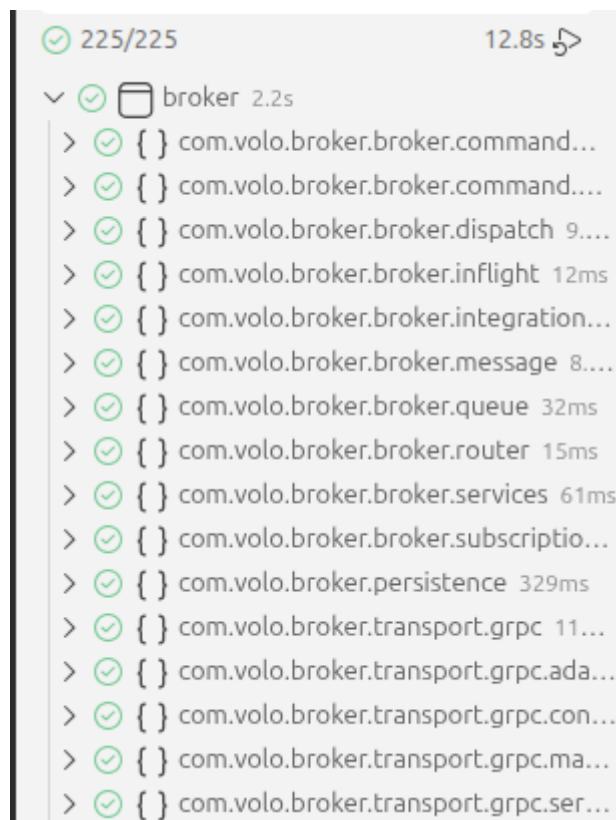


Рис. 3.3. — Результат запуску unit-тестів серверної частини

Окремо перевірялася робота системи як єдиного програмного комплексу з використанням реалізованої клієнтської частини. У такому сценарії перевіряється повний шлях повідомлення: публікація, маршрутизація, додавання в чергу, доставка отримувачу та підтвердження обробки.

Основні сценарії експериментального дослідження наведено в таблиці 3.3.1.

Таблиця 3.3. — Сценарії експериментального дослідження

№	Сценарій	Конфігурація	Повідомлень
1	Одна черга в пам'яті	1000 відправників, 1500 отримувачів	500000
2	50 черг у пам'яті	20 відправників і 15 отримувачів на кожну чергу	500000
3	10 persistent-черг	20 відправників і 15 отримувачів на кожну чергу	100000
4	Одна persistent-черга	1000 відправників, 1500 отримувачів	100000

У всіх сценаріях розмір корисного навантаження одного повідомлення становив 200 байт. Для першого та другого сценаріїв використовувалися налаштування без збереження в базу даних, а для третього та четвертого — зі збереженням.

Перший сценарій перевіряв роботу однієї черги без постійного зберігання. Тут все навантаження було зосереджене в межах однієї черги. Такий сценарій дозволяє оцінити поведінку брокера, коли велика кількість відправників і отримувачів одночасно працює з одним джерелом повідомлень.

```

28     public static void main( String[] args ) throws InterruptedException, ExecutionException
29     {
30         SimulationConfig config = SimulationConfig.builder()
31             .brokerConfig(BrokerClientConfig.plaintext(host: "localhost", port: 9090))
32             .queuePath(queuePath: "orders.created")
33             .queueSettings(QueueSettingsBuilder.memoryOnlyFast())
34             .queueDeclarationOptions(QueueDeclarationOptions.failIfExists())
35             .publishers(publishers: 1000)
36             .consumers(consumers: 1500)
37             .messagesPerPublisher(messagesPerPublisher: 500)
38             .payloadSizeBytes(payloadSizeBytes: 200)
39             .consumerPrefetch(consumerPrefetch: 50)
40             .consumerStartupDelay(Duration.ofSeconds(seconds: 1))
41             .consumptionTimeout(Duration.ofSeconds(seconds: 60))
42             .build();
43
44
45         SimulationStatsPrinter.print(new BrokerSimulation(config).run(), config.expectedMessages());

```

Рис. 3.3.2 — Фрагмент конфігурації симуляції для однієї memory-only-черги

Другий сценарій перевіряв роботу брокера з багатьма незалежними чергами. Було створено 50 черг, і кожна з них мала власних відправників та отримувачів. Такий варіант ближчий до ситуації, коли брокер обслуговує кілька різних напрямів повідомлень одночасно.

Третій і четвертий сценарії виконувалися з persistent-чергами. У цьому режимі повідомлення не тільки додаються в чергу, а й записуються у сховище до моменту завершення обробки, що дозволяє перевірити, як на продуктивність впливає постійне зберігання повідомлень.

Для перевірки коректності роботи основним показником була відповідність кількості опублікованих і доставлених повідомлень. Результати наведено в таблиці 3.3.2.

Таблиця 3.3.2 — Перевірка кількості повідомлень

№	Очікувано	Опубліковано	Доставлено	Помилки
1	500000	500000	500000	0
2	500000	500000	500000	0
3	100000	100000	100000	0
4	100000	100000	100000	0

У всіх сценаріях кількість доставлених повідомлень збіглася з кількістю опублікованих. Помилки публікації та помилок обробки не було. Це означає, що в межах проведених експериментів брокер коректно приймав повідомлення, знаходив відповідні черги, додавав повідомлення до них, доставляв отримувачам і завершував обробку після підтвердження.

Крім кількості повідомлень, під час симуляції фіксувалися показники продуктивності. До них належать загальний час виконання сценарію, пропускну здатність, середня затримка підтвердження публікації та середня затримка від публікації повідомлення до його обробки отримувачем.

Таблиця 3.3.— Продуктивність брокера під час симуляції

№	Час, мс	Повід./с	Середній час	Середній час
			підтвердження, мс	доставки, мс
1	36392	13739.00	52.323	805.746
2	21565	23184.76	11.213	18.962
3	44203	2262.28	78.464	2274.244
4	45865	2180.28	355.132	358.215

У таблиці 3.3.3 показано загальний час виконання сценарію, кількість оброблених повідомлень за секунду, середню затримку підтвердження публікації та середню затримку від публікації повідомлення до його обробки отримувачем.

У сценарії з 50 чергами кожна черга отримала по 10000 повідомлень. Для всіх черг кількість опублікованих і доставлених повідомлень збіглася, помилок публікації та обробки не було.

3.4. Аналіз результатів

Результати модульного тестування показали, що основні компоненти брокера працюють коректно в ізольованих сценаріях. Успішне виконання 225 тестів підтверджує правильність роботи командного рівня, черг, маршрутизації, доставки, підтверджень, повторної доставки, збереження даних і gRPC-компонентів.

Результати симуляційного дослідження також показали коректну роботу системи. У всіх чотирьох сценаріях брокер доставив усі повідомлення, які були опубліковані відправниками. Жодної помилки публікації або обробки не було зафіксовано. Це підтверджує, що реалізовані механізми публікації, маршрутизації, черг, доставки та підтверджень працюють узгоджено.

Найвищу пропускну здатність було отримано в сценарії з 50 memory-only-чергами. У цьому випадку брокер обробив 500000 повідомлень за 21565 мс, а пропускну здатність становила 23184.76 повідомлення за секунду. Такий

результат пояснюється тим, що навантаження було розподілене між багатьма чергами. Кожна черга мала власних відправників і отримувачів, тому система могла ефективніше використовувати паралельну обробку.

Сценарій з однією memory-only-чергою також показав стабільну роботу, але його пропускна здатність була нижчою — 13739 повідомлень за секунду. У цьому випадку всі відправники й отримувачі працювали з однією чергою, тому зростала конкуренція за доступ до одного набору повідомлень. Це вплинуло і на середню затримку доставки, яка становила 805.746 мс.

Persistent-сценарії показали нижчу пропускну здатність. Для 10 persistent-черг система обробляла 2262.28 повідомлення за секунду, а для однієї persistent-черги — 2180.28 повідомлення за секунду. Таке зниження є очікуваним, оскільки persistent-режим додає операції запису повідомлень у базу даних і видалення після завершення обробки. Тобто система виконує більше операцій для кожного повідомлення.

Затримка підтвердження публікації також залежить від режиму роботи черги. У сценарії з 50 memory-only-чергами середня АСК-затримка становила 11.213 мс. Для однієї memory-only-черги вона була більшою — 52.323 мс. У persistent-сценаріях цей показник зріс у середньому до 78.464 мс, а максимальна затримка займає 355.132 мс.

Затримка доставки показує повний час проходження повідомлення через систему. Найменше значення було отримано в сценарії з 50 memory-only-чергами — 18.962 мс. Це свідчить про те, що при розподілі навантаження між багатьма чергами повідомлення швидко проходили шлях від публікації до обробки отримувачем. У сценарії з однією memory-only-чергою цей показник був вищим через концентрацію великого навантаження на одному об'єкті.

У persistent-сценарії з однією чергою середня затримка доставки становила 358.215 мс. У persistent-сценарії з 10 чергами вона була більшою, аж цілих 2274.244 мс. На цей показник впливали одночасна робота кількох черг, велика кількість клієнтів і операції зі сховищем. При цьому всі повідомлення були

доставлені й оброблені, тому збільшення затримки не призвело до втрати повідомлень.

За результатами дослідження можна зробити кілька висновків. По-перше, брокер коректно працює з великою кількістю повідомлень і клієнтів. По-друге, розподіл навантаження між кількома чергами дає кращу пропускну здатність, ніж робота з однією перевантаженою чергою. По-третє, *persistent*-режим помітно знижує швидкість обробки, але забезпечує можливість збереження повідомлень і подальшого відновлення.

Таким чином, тестування та експериментальне дослідження підтвердили, що розроблений брокер повідомлень виконує основні функції: приймає повідомлення, маршрутизує їх до черг, доставляє отримувачам, обробляє підтвердження та працює з різними режимами зберігання.

Висновки до розділу 3

У третьому розділі було описано практичну реалізацію клієнт-серверної системи асинхронної передачі повідомлень. Серверну частину реалізовано мовою Java з використанням таких технологій як: Spring Boot, gRPC, Protocol Buffers, Spring Data JPA та PostgreSQL.

Реалізовано клієнтську частину системи у вигляді окремого Java проекту. Вона надає засоби для комунікації з реалізованим брокером, завдяки чому зовнішні програми можуть працювати з брокером через зручний програмний інтерфейс без прямої роботи з транспортним рівнем.

Після реалізації було проведено модульне тестування та експериментальне дослідження роботи системи під навантаженням. Результати показали, що брокер коректно приймає повідомлення, маршрутизує їх до черг, доставляє отримувачам, обробляє підтвердження, а продуктивність брокера залежить від режиму роботи.

ВИСНОВКИ

У ході кваліфікаційної роботи було розроблено клієнт-серверну систему асинхронної передачі повідомлень із застосуванням черг. Розроблена система виконує роль брокера повідомлень, який приймає повідомлення від клієнтів, визначає відповідні черги, зберігає повідомлення до моменту доставки та передає їх підключеним отримувачам.

Під час виконання роботи було розглянуто принципи асинхронної взаємодії між програмними компонентами та визначено основні проблеми, які виникають при прямому обміні даними між сервісами. До таких проблем належать тісна залежність між компонентами, складність масштабування, чутливість до тимчасової недоступності окремих сервісів і нерівномірність навантаження. Використання проміжного брокера з чергами дозволяє зменшити ці проблеми.

У роботі було проаналізовано підходи, які використовуються в сучасних системах обміну повідомленнями. На основі цього було сформовано вимоги до власної системи: підтримка публікації повідомлень, маршрутизації, створення черг, підписки отримувачів, підтверджень обробки, повторної доставки, обмеження кількості непідтверджених повідомлень, збереження даних і відновлення після перезавантаження. Визначено загальну структуру системи та розподілено її на окремі логічні частини.

Окремо було реалізовано клієнтську частину, яка спрощує роботу із серверним брокером. Вона надає засоби для створення черг, публікації повідомлень, підписки на черги та обробки отриманих повідомлень без необхідності напряду працювати з низькорівневими gRPC-запитами. Також клієнтський модуль використовувався для проведення симуляційних сценаріїв і перевірки роботи системи під навантаженням.

У результаті реалізації було отримано систему, яка підтримує різні режими роботи черг. Для швидких сценаріїв може використовуватися зберігання

повідомлень лише в оперативній пам'яті. Для надійніших сценаріїв передбачено постійне зберігання повідомлень у базі даних до моменту завершення їх обробки.

Працездатність розробленої системи була перевірена за допомогою модульних тестів і симуляційного навантаження. Перевірка показала, що брокер коректно виконує основні операції: приймає повідомлення, розподіляє їх між чергами, доставляє отримувачам і завершує обробку після підтвердження. Експериментальні результати також показали очікувану різницю між режимом роботи в оперативній пам'яті та режимом із постійним зберіганням: перший забезпечує вищу швидкість, а другий додає витрати на роботу зі сховищем.

Таким чином, мету кваліфікаційної роботи було досягнуто. Розроблено програмну систему, яка демонструє основні принципи функціонування брокера повідомлень і може використовуватися для вивчення асинхронної взаємодії між сервісами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol: O'Reilly Media, 2017. 616 p.
2. Hohpe G., Woolf B. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Boston: Addison-Wesley, 2003. 735 p.
3. Tanenbaum A. S., van Steen M. Distributed Systems. 3rd ed. CreateSpace Independent Publishing Platform, 2017. 704 p.
4. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol: O'Reilly Media, 2021. 616 p.
5. Richardson C. Microservices Patterns: With Examples in Java. Shelter Island: Manning Publications, 2018. 520 p.
6. Richards M., Ford N. Fundamentals of Software Architecture: An Engineering Approach. Sebastopol: O'Reilly Media, 2020. 432 p.
7. Burns B. Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services. Sebastopol: O'Reilly Media, 2018. 166 p.
8. Fowler M. Patterns of Enterprise Application Architecture. Boston: Addison-Wesley, 2002. 560 p.
9. Kreps J., Narkhede N., Rao J. Kafka: a Distributed Messaging System for Log Processing. Proceedings of the NetDB Workshop, 2011.
10. Apache Kafka Documentation. URL: <https://kafka.apache.org/documentation/>
11. RabbitMQ Documentation. URL: <https://www.rabbitmq.com/docs>
12. NATS Documentation. URL: <https://docs.nats.io/>
13. Apache ActiveMQ Documentation. URL: <https://activemq.apache.org/components/classic/documentation/>
14. gRPC Documentation. URL: <https://grpc.io/docs/>
15. Protocol Buffers Documentation. URL: <https://protobuf.dev/overview/>
16. Fowler M. Event-Driven Architecture. URL: <https://martinfowler.com/articles/201701-event-driven.html>

17. Fowler M. Microservices. URL:
<https://martinfowler.com/articles/microservices.html>

Додаток А

gRPC-контракт broker.proto

```

syntax = "proto3";

package broker;

option java_multiple_files = true;
option java_package = "com.custombroker.grpc";
option java_outer_classname = "BrokerProto";

service PublisherService {
    rpc OpenPublisherChannel(stream PublisherRequest) returns (stream PublisherResponse);
}

service ConsumerService {
    rpc OpenChannel(stream ConsumerRequest) returns (stream BrokerResponse);
}

service QueueService {
    rpc DeclareQueue(DeclareQueueRequest) returns (QueueOperationResponse);
    rpc DeleteQueue(DeleteQueueRequest) returns (QueueOperationResponse);
}

enum PersistenceMode {
    PERSISTENCE_MODE_UNSPECIFIED = 0;
    MEMORY_ONLY = 1;
    PERSISTENT = 2;
}

enum PublishAckMode {
    PUBLISH_ACK_MODE_UNSPECIFIED = 0;
    PUBLISH_ACK_NONE = 1;
    PUBLISH_ACK_ENABLED = 2;
}

enum ConsumerAckMode {
    CONSUMER_ACK_MODE_UNSPECIFIED = 0;
    CONSUMER_ACK_NONE = 1;
    CONSUMER_ACK_ENABLED = 2;
}

enum OverflowPolicy {
    OVERFLOW_POLICY_UNSPECIFIED = 0;
    REJECT_PUBLISH = 1;
    DROP_OLDEST = 2;
    DROP_NEWEST = 3;
}

enum ConsumerSelectionStrategy {
    CONSUMER_SELECTION_STRATEGY_UNSPECIFIED = 0;
    ROUND_ROBIN = 1;
}

```

```

    FIRST_AVAILABLE = 2;
    LEAST_INFLIGHT = 3;
}

enum ErrorCode {
    ERROR_CODE_UNSPECIFIED = 0;
    INVALID_REQUEST = 1;
    PAYLOAD_TOO_LARGE = 2;
    ROUTE_NOT_FOUND = 3;
    QUEUE_ALREADY_EXISTS = 4;
    QUEUE_NOT_FOUND = 5;
    QUEUE_OVERFLOW = 6;
    UNSUPPORTED_DELIVERY_GUARANTEE = 7;
    INTERNAL_ERROR = 8;
}

message RedeliveryPolicy {
    bool enabled = 1;
    int32 max_attempts = 2;
    bool requeue_on_disconnect = 3;
    bool requeue_on_send_failure = 4;
}

message MessageRetentionPolicy {
    bool ttl_enabled = 1;
    int64 ttl_ms = 2;
}

message QueueLifecyclePolicy {
    bool auto_delete = 1;
    bool idle_timeout_enabled = 2;
    int64 idle_timeout_ms = 3;
    bool durable = 4;
}

message QueueCapacityPolicy {
    bool limited = 1;
    int32 max_messages = 2;
    int64 max_bytes = 3;
    OverflowPolicy overflow_policy = 4;
}

message DispatchPolicy {
    int32 prefetch_count = 1;
    ConsumerSelectionStrategy consumer_selection_strategy = 2;
}

message MessageDelivery {
    int64 delivery_tag = 1;
    Message message = 2;
    bool redelivered = 3;
    int32 delivery_attempt = 4;
}

```

```

message PublishAction {
    int64 client_publish_id = 1;
    string routing_key = 2;
    bytes payload = 3;
    map<string, string> headers = 4;
}

message QueueSettings {
    PersistenceMode persistence_mode = 1;
    PublishAckMode publish_ack_mode = 2;
    ConsumerAckMode consumer_ack_mode = 3;
    RedeliveryPolicy redelivery_policy = 4;
    MessageRetentionPolicy retention_policy = 5;
    QueueLifecyclePolicy lifecycle_policy = 6;
    QueueCapacityPolicy capacity_policy = 7;
    DispatchPolicy dispatch_policy = 8;
}

message ErrorInfo {
    ErrorCode code = 1;
    string message = 2;
    map<string, string> details = 3;
}

message PublisherRequest {
    oneof request {
        PublishAction publish = 1;
        KeepAlive ping = 2;
        ClosePublisherAction close = 3;
    }
}

message PublisherResponse {
    oneof response {
        PublishAck ack = 1;
        PublishNack nack = 2;
        KeepAlive pong = 3;
    }
}

message PublishAck {
    int64 client_publish_id = 1;
    int32 routed_queue_count = 2;
}

message PublishNack {
    int64 client_publish_id = 1;
    ErrorInfo error = 2;
}

message ClosePublisherAction {
    string reason = 1;
}

```

```

}

message ConsumerRequest {
  oneof request {
    SubscribeAction subscribe = 1;
    AckAction ack = 2;
    NackAction nack = 3;
    KeepAlive ping = 4;
  }
}

message BrokerResponse {
  oneof response {
    MessageDelivery delivery = 1;
    KeepAlive pong = 2;
    ErrorInfo error = 3;
  }
}

message SubscribeAction {
  string queue_name = 1;
  int32 prefetch_count = 2;
}

message AckAction {
  int64 delivery_tag = 1;
  int64 message_id = 2;
}

message NackAction {
  int64 delivery_tag = 1;
  int64 message_id = 2;
  bool requeue = 3;
}

message Message {
  int64 id = 1;
  bytes payload = 2;
  string routing_key = 3;
  map<string, string> headers = 4;
  int64 timestamp = 5;
}

message KeepAlive {
  int64 timestamp = 1;
}

message DeclareQueueRequest {
  string queue_path = 1;
  QueueSettings settings = 2;
}

message DeleteQueueRequest {

```

```
    string queue_path = 1;
}

message QueueOperationResponse {
    bool success = 1;
    ErrorInfo error = 2;
}
```

Додаток Б

Фрагменти реалізації серверної частини брокера

Лістинг Б.1 — Командна обробка запитів

```

public interface TransportCommand {}
public record PublishCommand(
    public interface CommandHandler<C extends TransportCommand, R> {
        Class<C> getCommandType();

        R handle(C command);
    }

@Component
public class CommandBus {
    private final Map<Class<? extends TransportCommand>, CommandHandler<?, ?>> handlers = new
    HashMap<>();
    public CommandBus(List<CommandHandler<?, ?>> handlers){
        for (CommandHandler<?, ?> handler : handlers) {
            this.handlers.put(handler.getCommandType(), handler);
        }
    }
    @SuppressWarnings("unchecked")
    public <C extends TransportCommand, R> R execute(C command){
        CommandHandler<C, R> handler = (CommandHandler<C, R>) handlers.get(command.getClass());

        if (handler == null) {
            throw new IllegalArgumentException("No handler found for command type: " +
            command.getClass());
        }
        return handler.handle(command);
    }
}

public record PublishCommand(
    Long clientPublishId,
    String routingKey,
    byte[] payload,
    Map<String, String> headers
) implements TransportCommand {

    public PublishCommand(
        Long clientPublishId,
        String routingKey,
        byte[] payload,
        Map<String, String> headers) {
        this.clientPublishId = clientPublishId;
        this.routingKey = routingKey == null ? "" : routingKey.trim();
        this.payload = payload == null ? new byte[0] : payload.clone();
        this.headers = headers == null ? Map.of() : Map.copyOf(headers);
    }
}

```

```
}
```

Лістинг Б.2 — Реалізація черги та її налаштувань

```
public final class BrokerQueue {
    private final QueuePath path;
    private final QueueSettings settings;
    private final Deque<QueuedMessage> messages = new ArrayDeque<>();
    private final long createdAt;

    private long lastActivityAt;
    private long totalBytes;

    private BrokerQueue(QueuePath path, QueueSettings settings) {
        this.path = Objects.requireNonNull(path, "path must not be null");
        this.settings = Objects.requireNonNull(settings, "settings must not be null");
        this.createdAt = System.currentTimeMillis();
        this.lastActivityAt = createdAt;
    }

    public static BrokerQueue create(QueuePath path, QueueSettings settings) {
        return new BrokerQueue(path, settings);
    }

    public synchronized EnqueueResult enqueue(QueuedMessage message) {
        Objects.requireNonNull(message, "message must not be null");

        List<QueuedMessage> dropped = applyOverflowPolicy(message);

        messages.addLast(message);
        totalBytes += message.sizeBytes();
        touch();

        return dropped.isEmpty()
            ? EnqueueResult.ofAccepted()
            : EnqueueResult.ofAcceptedWithDropped(dropped);
    }

    public synchronized Optional<QueuedMessage> poll() {
        QueuedMessage message = messages.pollFirst();

        if (message == null) {
            return Optional.empty();
        }
        totalBytes -= message.sizeBytes();
        touch();
        return Optional.of(message);
    }

    public synchronized void requeue(QueuedMessage message) {
        Objects.requireNonNull(message, "message must not be null");
        ensureCapacityBeforeEnqueue(message);
        messages.addFirst(message);
        totalBytes += message.sizeBytes();
        touch();
    }
}
```

```

    }

    public synchronized int size() {
        return messages.size();
    }

    public synchronized long sizeBytes() {
        return totalBytes;
    }

    public synchronized boolean isEmpty() {
        return messages.isEmpty();
    }

    public synchronized void touch() {
        lastActivityAt = System.currentTimeMillis();
    }

    public QueuePath path() {
        return path;
    }

    public String queuePath() {
        return path.value();
    }

    public QueueSettings settings() {
        return settings;
    }

    public long createdAt() {
        return createdAt;
    }

    public synchronized long lastActivityAt() {
        return lastActivityAt;
    }

    private void ensureCapacityBeforeEnqueue(QueuedMessage message) {
        if (!settings.capacityPolicy().limited()) {
            return;
        }

        boolean messageLimitExceeded = settings.capacityPolicy().maxMessages() > 0
            && messages.size() >= settings.capacityPolicy().maxMessages();

        boolean bytesLimitExceeded = settings.capacityPolicy().maxBytes() > 0
            && totalBytes + message.sizeBytes() > settings.capacityPolicy().maxBytes();

        if (!messageLimitExceeded && !bytesLimitExceeded) {
            return;
        }

        OverflowPolicy overflowPolicy = settings.capacityPolicy().overflowPolicy();

```

```

        if (overflowPolicy == OverflowPolicy.DROP_OLDEST) {
            QueuedMessage dropped = messages.pollFirst();
            if (dropped != null) {
                totalBytes -= dropped.sizeBytes();
            }
            return;
        }

        if (overflowPolicy == OverflowPolicy.DROP_NEWEST) {
            throw new QueueOverflowException("Incoming message was dropped because queue is full: " + queuePath());
        }

        throw new QueueOverflowException("Queue capacity exceeded: " + queuePath());
    }

    private List<QueuedMessage> applyOverflowPolicy(QueuedMessage incomingMessage) {
        if (!settings.capacityPolicy().limited()) {
            return List.of();
        }

        if (!capacityExceededAfterAdding(incomingMessage)) {
            return List.of();
        }

        return switch (settings.capacityPolicy().overflowPolicy()) {
            case REJECT_PUBLISH -> throw new QueueOverflowException(
                "Queue capacity exceeded: " + queuePath()
            );
            case DROP_NEWEST -> throw new QueueOverflowException(
                "Incoming message was dropped because queue is full: " + queuePath()
            );
            case DROP_OLDEST -> dropOldestUntilFits(incomingMessage);
        };
    }

    private boolean capacityExceededAfterAdding(QueuedMessage incomingMessage) {
        QueueCapacityPolicy policy = settings.capacityPolicy();

        if (!policy.limited()) {
            return false;
        }

        boolean maxMessagesExceeded = policy.maxMessages() > 0
            && messages.size() + 1 > policy.maxMessages();

        boolean maxBytesExceeded = policy.maxBytes() > 0
            && totalBytes + incomingMessage.sizeBytes() > policy.maxBytes();

        return maxMessagesExceeded || maxBytesExceeded;
    }

    private List<QueuedMessage> dropOldestUntilFits(QueuedMessage incomingMessage) {
        List<QueuedMessage> droppedMessages = new ArrayList<>();

```

```

        while (!messages.isEmpty() && capacityExceededAfterAdding(incomingMessage)) {
            QueuedMessage dropped = messages.pollFirst();

            if (dropped != null) {
                totalBytes -= dropped.sizeBytes();
                droppedMessages.add(dropped);
            }
        }

        if (capacityExceededAfterAdding(incomingMessage)) {
            throw new QueueOverflowException(
                "Incoming message is too large for queue capacity: " + queuePath()
            );
        }
        return droppedMessages;
    }
}

@Component
public class InMemoryQueueManager implements QueueManager {
    private final ConcurrentMap<String, BrokerQueue> queues = new ConcurrentHashMap<>();
    private final RouteTree routeTree = new RouteTree();

    public RouteTree getRouteTree() {
        return routeTree;
    }

    @Override
    public BrokerQueue declare(QueuePath path, QueueSettings settings) {
        BrokerQueue queue = BrokerQueue.create(path, settings);

        BrokerQueue existing = queues.putIfAbsent(path.value(), queue);
        if (existing != null) {
            throw new QueueAlreadyExistsException("Queue already exists: " + path.value());
        }

        routeTree.attachQueue(path);

        return queue;
    }

    @Override
    public void delete(QueuePath path) {
        BrokerQueue removed = queues.remove(path.value());

        if (removed == null) {
            throw new QueueNotFoundException("Queue does not exist: " + path.value());
        }

        routeTree.detachQueue(path);
    }
}

```

```

@Override
public BrokerQueue require(String queueName) {
    return find(queueName)
        .orElseThrow(() -> new QueueNotFoundException("Queue does not exist: " +
queueName));
}

@Override
public Optional<BrokerQueue> find(String queueName) {
    if (queueName == null || queueName.isBlank()) {
        return Optional.empty();
    }

    return Optional.ofNullable(queues.get(queueName.trim()));
}

@Override
public List<BrokerQueue> resolve(String routingKey) {
    return routeTree.resolve(routingKey)
        .stream()
        .map(this::require)
        .toList();
}

@Override
public Collection<BrokerQueue> all() {
    return List.copyOf(queues.values());
}
}

public record QueueSettings(
    PersistenceMode persistenceMode,
    PublishAckMode publishAckMode,
    ConsumerAckMode consumerAckMode,
    RedeliveryPolicy redeliveryPolicy,
    MessageRetentionPolicy retentionPolicy,
    QueueLifecyclePolicy lifecyclePolicy,
    QueueCapacityPolicy capacityPolicy,
    DispatchPolicy dispatchPolicy
) {
    public QueueSettings {
        persistenceMode = Objects.requireNonNullElse(persistenceMode,
PersistenceMode.PERSISTENT);
        publishAckMode = Objects.requireNonNullElse(publishAckMode, PublishAckMode.ENABLED);
        consumerAckMode = Objects.requireNonNullElse(consumerAckMode, ConsumerAckMode.ENABLED);
        redeliveryPolicy = Objects.requireNonNullElse(redeliveryPolicy,
RedeliveryPolicy.defaults());
        retentionPolicy = Objects.requireNonNullElse(retentionPolicy,
MessageRetentionPolicy.disabled());
        lifecyclePolicy = Objects.requireNonNullElse(lifecyclePolicy,
QueueLifecyclePolicy.defaults());
        capacityPolicy = Objects.requireNonNullElse(capacityPolicy,
QueueCapacityPolicy.unlimited());
        dispatchPolicy = Objects.requireNonNullElse(dispatchPolicy, DispatchPolicy.defaults());
    }
}

```

```

        validateCompatibility(persistenceMode, consumerAckMode, redeliveryPolicy);
    }

    public static QueueSettings defaults() {
        return new QueueSettings(
            PersistenceMode.PERSISTENT,
            PublishAckMode.ENABLED,
            ConsumerAckMode.ENABLED,
            RedeliveryPolicy.defaults(),
            MessageRetentionPolicy.disabled(),
            QueueLifecyclePolicy.defaults(),
            QueueCapacityPolicy.unlimited(),
            DispatchPolicy.defaults()
        );
    }

    public boolean persistent() {
        return persistenceMode == PersistenceMode.PERSISTENT;
    }

    public boolean publishAckEnabled() {
        return publishAckMode == PublishAckMode.ENABLED;
    }

    public boolean consumerAckEnabled() {
        return consumerAckMode == ConsumerAckMode.ENABLED;
    }

    private static void validateCompatibility(
        PersistenceMode persistenceMode,
        ConsumerAckMode consumerAckMode,
        RedeliveryPolicy redeliveryPolicy
    ) {
        if (persistenceMode == PersistenceMode.PERSISTENT && consumerAckMode ==
ConsumerAckMode.NONE) {
            throw new IllegalArgumentException("PERSISTENT queue requires consumer
acknowledgements");
        }

        if (consumerAckMode == ConsumerAckMode.NONE && redeliveryPolicy.enabled()) {
            throw new IllegalArgumentException("Redelivery requires consumer acknowledgements");
        }
    }
}

```

Лістинг Б.3 — Маршрутизація повідомлень

```

public record QueuePath(
    String value,
    String routePath,
    String queueName,
    List<String> segments
) {

```

```

public QueuePath {
    if (value == null || value.isBlank()) {
        throw new IllegalArgumentException("queue path is required");
    }

    if (queueName == null || queueName.isBlank()) {
        throw new IllegalArgumentException("queue name is required");
    }

    routePath = routePath == null ? "" : routePath;
    segments = List.copyOf(segments);
}

public static QueuePath of(String rawValue) {
    RouteSegments.ParsedRoute parsed = RouteSegments.parseQueuePath(rawValue);

    List<String> segments = parsed.segments();

    String queueName = segments.get(segments.size() - 1);

    String routePath = segments.size() == 1
        ? ""
        : String.join(".", segments.subList(0, segments.size() - 1));

    return new QueuePath(
        parsed.value(),
        routePath,
        queueName,
        segments
    );
}

public boolean hasRoutePath() {
    return !routePath.isBlank();
}
}

public record RoutePattern(
    String value,
    List<String> segments
) {
    private static final String SINGLE_SEGMENT_WILDCARD = "*";
    private static final String MULTI_SEGMENT_WILDCARD = "#";

    public RoutePattern {
        if (value == null || value.isBlank()) {
            throw new IllegalArgumentException("routing key is required");
        }

        segments = List.copyOf(segments);
    }

    public static RoutePattern of(String rawValue) {

```

```

RouteSegments.ParsedRoute parsed = RouteSegments.parseRoutePattern(rawValue);

return new RoutePattern(
    parsed.value(),
    parsed.segments()
);
}

public boolean hasWildcard() {
    return segments.stream()
        .anyMatch(RoutePattern::isWildcardSegment);
}

public static boolean containsWildcard(String routingKey) {
    return routingKey != null
        && (routingKey.contains(SINGLE_SEGMENT_WILDCARD)
            || routingKey.contains(MULTI_SEGMENT_WILDCARD));
}

private static boolean isWildcardSegment(String segment) {
    return SINGLE_SEGMENT_WILDCARD.equals(segment)
        || MULTI_SEGMENT_WILDCARD.equals(segment);
}

@Override
public String toString() {
    return value;
}
}

final class RouteSegments {
    private static final int MAX_LENGTH = 255;
    private static final String SEGMENT_SEPARATOR = "\\.";
    private static final String SINGLE_SEGMENT_WILDCARD = "*";
    private static final String MULTI_SEGMENT_WILDCARD = "#";

    private static final Pattern NORMAL_SEGMENT_PATTERN =
        Pattern.compile("^[a-zA-Z0-9_-]+$");

    private RouteSegments() {
    }

    static ParsedRoute parseQueuePath(String rawValue) {
        String normalized = normalize(rawValue);

        validateRequired(normalized, "queue path");
        validateMaxLength(normalized, "queue path");

        String[] segments = split(normalized);

        validateQueuePathSegments(segments);

        return new ParsedRoute(normalized, List.of(segments));
    }
}

```

```

    }

    static ParsedRoute parseRoutePattern(String rawValue) {
        String normalized = normalize(rawValue);

        validateRequired(normalized, "routing key");
        validateMaxLength(normalized, "routing key");

        String[] segments = split(normalized);

        validateRoutePatternSegments(segments);

        return new ParsedRoute(normalized, List.of(segments));
    }

    // private methods ...
}

public class RouteTree {
    private final RouteNode root = new RouteNode("");

    public void attachQueue(QueuePath queuePath) {
        RouteNode current = root;

        for (String segment : queuePath.segments()) {
            current = current.childOrCreate(segment);
        }

        current.addQueue(queuePath.value());
    }

    public void detachQueue(QueuePath queuePath) {
        detachQueue(root, queuePath.value());
    }

    public Set<String> resolve(String routingKey) {
        RoutePattern pattern = RoutePattern.of(routingKey);
        Set<String> result = new LinkedHashSet<>();

        collectMatches(root, pattern.segments(), 0, result);

        return Set.copyOf(result);
    }

    private void detachQueue(RouteNode node, String queuePath) {
        node.removeQueue(queuePath);

        for (RouteNode child : node.children()) {
            detachQueue(child, queuePath);
        }
    }
}

```

```

private void collectMatches(RouteNode node, List<String> patternSegments, int index,
Set<String> result) {
    if (index == patternSegments.size()) {
        result.addAll(node.queueNames());
        return;
    }

    String segment = patternSegments.get(index);

    if (".".equals(segment)) {
        collectMatches(node, patternSegments, index + 1, result);
        for (RouteNode child : node.children()) {
            collectMatches(child, patternSegments, index, result);
        }
        return;
    }

    if ("*".equals(segment)) {
        for (RouteNode child : node.children()) {
            collectMatches(child, patternSegments, index + 1, result);
        }
        return;
    }

    RouteNode child = node.child(segment);
    if (child != null) {
        collectMatches(child, patternSegments, index + 1, result);
    }
}

```

Лістинг Б.4 — Публікація та доставка повідомлень

```

@Service
public class PublishService {
    private final QueueManager queueManager;
    private final MessageStore messageStore;
    private final QueueDispatchScheduler dispatchScheduler;
    private final MessageIdGenerator messageIdGenerator;

    public PublishService(
        QueueManager queueManager,
        MessageStore messageStore,
        QueueDispatchScheduler dispatchScheduler,
        MessageIdGenerator messageIdGenerator
    ) {
        this.queueManager = Objects.requireNonNull(queueManager);
        this.messageStore = Objects.requireNonNull(messageStore);
        this.dispatchScheduler = Objects.requireNonNull(dispatchScheduler);
        this.messageIdGenerator = Objects.requireNonNull(messageIdGenerator);
    }

    @Transactional
    public PublishResult publish(PublishCommand command) {

```

```

try {
    validate(command);

    List<BrokerQueue> queues = queueManager.resolve(command.routingKey());

    if (queues.isEmpty()) {
        return PublishResult.nack(
            command.clientPublishId(),
            BrokerErrorCode.QUEUE_NOT_FOUND, // todo: consider
using a different error code for this scenario

```

```

        "No queue matched routing key: " + command.routingKey()
    );
}

List<QueuedMessage> messages = queues.stream()
    .map(queue -> createMessage(command))
    .toList();

for (int i = 0; i < queues.size(); i++) {
    BrokerQueue queue = queues.get(i);
    QueuedMessage message = messages.get(i);

    if (queue.settings().persistenceMode() == PersistenceMode.PERSISTENT) {
        messageStore.save(queue.queuePath(), message);
    }

    queue.enqueue(message);
    dispatchScheduler.signal(queue.queuePath());
}

return PublishResult.ack(command.clientPublishId(), queues.size());
} catch (QueueOverflowException exception) {
    return PublishResult.nack(
        command.clientPublishId(),
        BrokerErrorCode.QUEUE_OVERFLOW,
        exception.getMessage()
    );
} catch (IllegalArgumentException exception) {
    return PublishResult.nack(
        command.clientPublishId(),
        BrokerErrorCode.INVALID_REQUEST,
        exception.getMessage()
    );
} catch (BrokerException exception) {
    return PublishResult.nack(
        command.clientPublishId(),
        exception.code(),
        exception.getMessage()
    );
} catch (Throwable throwable) {
    return PublishResult.nack(
        command.clientPublishId(),

```

```

        BrokerErrorCode.INTERNAL_ERROR,
        "Message was not accepted by broker"
    );
}

private void validate(PublishCommand command) {
    if (command == null) {
        throw new IllegalArgumentException("publish command must not be null");
    }

    if (command.routingKey() == null || command.routingKey().isBlank()) {
        throw new IllegalArgumentException("routingKey is required");
    }

    if (command.payload().length == 0) {
        throw new IllegalArgumentException("payload must not be empty");
    }
}

@Component
public class QueueDispatchScheduler {
    private final DispatchManager dispatchManager;
    private final QueueManager queueManager;
    private final ExecutorService executor;
    private final Set<String> dispatchingQueues = ConcurrentHashMap.newKeySet();

    public QueueDispatchScheduler(
        DispatchManager dispatchManager,
        QueueManager queueManager,
        @Value("${broker.dispatch.threads:2}") int threads
    ) {
        this.dispatchManager = dispatchManager;
        this.queueManager = queueManager;
        this.executor = Executors.newFixedThreadPool(Math.max(1, threads));
    }

    public void signal(String queueName){
        if(!dispatchingQueues.add(queueName)) return;

        executor.execute(() -> {
            try{
                dispatchManager.dispatchLoop(queueName);
            } finally {
                dispatchingQueues.remove(queueName);

                if(hasWork(queueName)){
                    signal(queueName);
                }
            }
        });
    }
}

```

```

private boolean hasWork(String queueName){
    return queueManager.find(queueName).get().size() > 0;
}

public void shutdown() {
    executor.shutdown();
}
}

@Component
@AllArgsConstructor
public class DispatchManager {
    private final QueueManager queueManager;
    private final SubscriptionManager subscriptionManager;
    private final MessageLifecycleService lifecycleService;

    public void dispatchLoop(String queueName) {
        while (true) {
            var queue = queueManager.find(queueName).get();
            Optional<Subscription> maybeSubscription = subscriptionManager.nextReady(
                queueName,
                queue.settings().dispatchPolicy().consumerSelectionStrategy()
            );

            if (maybeSubscription.isEmpty()) {
                return;
            }

            Subscription subscription = maybeSubscription.get();

            if (!subscription.canAcceptMore()) {
                return;
            }

            Optional<QueuedMessage> maybeMessage = queue.poll();
            if (maybeMessage.isEmpty()) {
                return;
            }

            QueuedMessage message = maybeMessage.get();

            if (lifecycleService.isExpired(queue, message)) {
                lifecycleService.complete(queue, message);
                continue;
            }

            dispatch(queue, subscription, message);
        }
    }

    private void dispatch(
        BrokerQueue queue,
        Subscription subscription,

```

QueuedMessage message

```

    ) {
        long deliveryTag = subscription.nextDeliveryTag();
        boolean consumerAckRequired = queue.settings().consumerAckMode() ==
ConsumerAckMode.ENABLED;

        if (consumerAckRequired) {
            subscription.track(deliveryTag, message);
        }

        try {
            subscription.channel().send(MessageDelivery.of(deliveryTag, message));

            queue.touch();

            if (!consumerAckRequired) {
                lifecycleService.complete(queue, message);
            }

        } catch (Throwable throwable) {
            if (consumerAckRequired) {
                subscription.remove(deliveryTag);
            }

            boolean shouldRequeue = queue.settings().redeliveryPolicy().requeueOnSendFailure()
                && lifecycleService.canRedeliver(queue, message);

            if (shouldRequeue) {
                queue.requeue(message.asRedelivered());
                return;
            }

            lifecycleService.complete(queue, message);
        }
    }
}

```

Лістинг Б.5 — Підтвердження та повторна доставка

```

@Service
public class ConsumerAcknowledgementService {
    private final SubscriptionManager subscriptionManager;
    private final QueueManager queueManager;
    private final QueueDispatchScheduler dispatchScheduler;
    private final MessageLifecycleService lifecycleService;

    public ConsumerAcknowledgementService(
        SubscriptionManager subscriptionManager,
        QueueManager queueManager,
        QueueDispatchScheduler dispatchScheduler,
        MessageLifecycleService lifecycleService
    ) {
    }
}

```

```

    ) {
        this.subscriptionManager = Objects.requireNonNull(subscriptionManager);
        this.queueManager = Objects.requireNonNull(queueManager);
        this.dispatchScheduler = Objects.requireNonNull(dispatchScheduler);
        this.lifecycleService = Objects.requireNonNull(lifecycleService);
    }

    public void ack(AckCommand command) {
        subscriptionManager.findByConsumerTag(command.consumerTag())
            .ifPresent(subscription -> {
                QueuedMessage message = subscription.remove(command.deliveryTag());
                if (message == null) {
                    return;
                }

                BrokerQueue queue = queueManager.require(subscription.queueName());
                lifecycleService.complete(queue, message);
                queue.touch();

                dispatchScheduler.signal(queue.queuePath());
            });
    }

    public void nack(NackCommand command) {
        subscriptionManager.findByConsumerTag(command.consumerTag())
            .ifPresent(subscription -> {
                QueuedMessage message = subscription.remove(command.deliveryTag());
                if (message == null) {
                    return;
                }

                BrokerQueue queue = queueManager.require(subscription.queueName());

                if (command.requeue() && lifecycleService.canRedeliver(queue, message)) {
                    queue.requeue(message.asRedelivered());
                } else {
                    lifecycleService.complete(queue, message);
                }

                queue.touch();
                dispatchScheduler.signal(queue.queuePath());
            });
    }
}

public class DefaultInFlightTracker implements InFlightTracker {
    private final AtomicLong counter = new AtomicLong(0);
    private final ConcurrentMap<Long, QueuedMessage> inflight = new ConcurrentSkipListMap<>();

    @Override
    public long nextTag() {
        return counter.incrementAndGet();
    }
}

```

```

@Override
public void track(long deliveryTag, QueuedMessage message) {
    inflight.put(deliveryTag, message);
}

@Override
public QueuedMessage remove(long deliveryTag) {
    return inflight.remove(deliveryTag);
}

@Override
public Collection<QueuedMessage> drainAll() {
    var drained = java.util.List.copyOf(inflight.values());
    inflight.clear();
    return drained;
}

@Override
public int size() {
    return inflight.size();
}
}

@Component
@AllArgsConstructor
public class DisconnectCommandHandler implements CommandHandler<DisconnectCommand, Void> {

    private final SubscriptionManager subscriptionManager;
    private final QueueManager queueManager;
    private final QueueDispatchScheduler dispatchScheduler;
    private final MessageLifecycleService messageLifecycleService;

    @Override
    public Class<DisconnectCommand> getCommandType() {
        return DisconnectCommand.class;
    }

    @Override
    public Void handle(DisconnectCommand cmd) {
        subscriptionManager.findByConsumerTag(cmd.consumerTag())
            .ifPresent(sub -> {
                subscriptionManager.unregister(cmd.consumerTag());

                List<QueuedMessage> drained =
                    new ArrayList<>(sub.getInFlightTracker().drainAll());

                dispatchScheduler.signal(sub.queueName());

                for (int i = drained.size() - 1; i >= 0; i--) {
                    QueuedMessage message = drained.get(i);

                    var queue = queueManager.find(sub.queueName()).get();

```

```

        boolean shouldRequeue = queue.settings().redeliveryPolicy().enabled()
            && queue.settings().redeliveryPolicy().requeueOnDisconnect()
            && message.getDeliveryAttempt() <
queue.settings().redeliveryPolicy().maxAttempts();

        if (shouldRequeue) {
            queueManager.find(sub.queueName()).ifPresent(q ->
q.requeue(message.asRedelivered()));
        } else {
            messageLifecycleService.complete(queue, message);
        }
    }
});

return null;
}
}

```

Лістинг Б.6 — Збереження та відновлення

```

@Component
public class JpaMessageStore implements MessageStore {

    private final StoredMessageRepository repository;
    private final ObjectMapper objectMapper;

    private static final TypeReference<Map<String, String>> HEADERS_TYPE =
        new TypeReference<>() {
        };

    public JpaMessageStore(
        StoredMessageRepository repository
    ) {
        this.repository = repository;
        this.objectMapper = new ObjectMapper();
    }

    @Override
    @Transactional
    public QueuedMessage save(String queuePath, QueuedMessage message) {
        StoredMessageEntity entity = new StoredMessageEntity();

        entity.setId(message.getMessageId());
        entity.setQueuePath(queuePath);
        entity.setRoutingKey(message.getRoutingKey());
        entity.setPayload(message.getPayload());
        entity.setHeadersJson(writeHeaders(message.getHeaders()));
        entity.setCreatedAt(message.getCreatedAt());
        entity.setDeliveryAttempt(message.getDeliveryAttempt());
        entity.setRedelivered(message.isRedelivered());

        repository.save(entity);
    }
}

```

```

        return message;
    }

    @Override
    @Transactional
    public void delete(long messageId) {
        int deleted = repository.deleteByMessageId(messageId);

        if (deleted != 1) {
            throw new IllegalStateException("Persistent message was not deleted: " + messageId);
        }
    }

    @Override
    @Transactional(readOnly = true)
    public List<StoredMessageDto> loadAllPending() {
        return repository.findAllByOrderByIdAsc()
            .stream()
            .map(this::toDto)
            .toList();
    }

    @Override
    @Transactional(readOnly = true)
    public long findMaxMessageId() {
        return repository.findMaxMessageId();
    }

    @Override
    public void deleteByQueuePath(String queueName) {
        repository.deleteByQueuePath(queueName);
    }
}

@Component
public class JpaQueueStore implements QueueStore {
    private final StoredQueueRepository repository;

    public JpaQueueStore(StoredQueueRepository repository) {
        this.repository = Objects.requireNonNull(repository);
    }

    @Override
    @Transactional
    public void save(BrokerQueue queue) {
        repository.save(
            StoredQueueEntity.from(
                queue.queuePath(),
                queue.settings()
            )
        );
    }
}

```

```

@Override
@Transactional
public void delete(String queuePath) {
    repository.deleteById(queuePath);
}

@Override
@Transactional(readOnly = true)
public List<StoredQueueDto> loadAll() {
    return repository.findAll()
        .stream()
        .map(entity -> new StoredQueueDto(
            entity.getQueuePath(),
            entity.toQueueSettings()
        ))
        .toList();
}
}

@Component
public class BrokerRecoveryService implements ApplicationRunner {
    private final QueueStore queueStore;
    private final MessageStore messageStore;
    private final QueueManager queueManager;
    private final MessageIdGenerator messageIdGenerator;

    public BrokerRecoveryService(
        QueueStore queueStore,
        MessageStore messageStore,
        QueueManager queueManager,
        MessageIdGenerator messageIdGenerator
    ) {
        this.queueStore = Objects.requireNonNull(queueStore);
        this.messageStore = Objects.requireNonNull(messageStore);
        this.queueManager = Objects.requireNonNull(queueManager);
        this.messageIdGenerator = Objects.requireNonNull(messageIdGenerator);
    }

    @Override
    public void run(ApplicationArguments args) {
        restoreQueues();
        restoreMessages();
        restoreMessageIdGenerator();
    }

    private void restoreQueues() {
        for (StoredQueueDto queueDto : queueStore.loadAll()) {
            QueuePath queuePath = QueuePath.of(queueDto.queuePath());

            if (queueManager.find(queuePath.value()).isEmpty()) {
                queueManager.declare(queuePath, queueDto.settings());
            }
        }
    }
}

```

```
}

private void restoreMessages() {
    for (StoredMessageDto messageDto : messageStore.loadAllPending()) {
        queueManager.find(messageDto.queuePath())
            .ifPresentOrElse(
                queue -> queue.enqueue(messageDto.message().asRedelivered()),
                () -> messageStore.delete(messageDto.message().getMessageId())
            );
    }
}

private void restoreMessageIdGenerator() {
    messageIdGenerator.initialize(messageStore.findMaxMessageId());
}
}
```

Додаток В

Фрагменти реалізації клієнтського модуля

Лістинг В.1 — Підключення до брокера

```

public final class BrokerClient implements AutoCloseable {
    private final ManagedChannel channel;
    private final BrokerClientConfig config;
    private final AtomicBoolean closed = new AtomicBoolean(false);

    private final QueueAdminClient queueAdminClient;

    private BrokerClient(
        ManagedChannel channel,
        BrokerClientConfig config) {
        this.channel = Objects.requireNonNull(channel, "channel is required");
        this.config = Objects.requireNonNull(config, "config is required");

        this.queueAdminClient = new QueueAdminClient(
            QueueServiceGrpc.newBlockingStub(channel)
        );
    }

    public static BrokerClient connect(BrokerClientConfig config) {
        Objects.requireNonNull(config, "config is required");

        ManagedChannelBuilder<> builder = ManagedChannelBuilder
            .forAddress(config.host(), config.port())
            .maxInboundMessageSize(config.maxInboundMessageSizeBytes());

        if (config.plaintext()) {
            builder.usePlaintext();
        }

        return new BrokerClient(builder.build(), config);
    }

    public QueueAdminClient queues() {
        ensureOpen();
        return queueAdminClient;
    }

    public PublisherClient openPublisher() {
        ensureOpen();

        return new PublisherClient(
            PublisherServiceGrpc.newStub(channel),
            config
        );
    }
}

```

```

}

public ConsumerClient openConsumer(ConsumerOptions options) {
    ensureOpen();

    ConsumerClient consumerClient = new ConsumerClient(
        ConsumerServiceGrpc.newStub(channel),
        config,
        options
    );

    consumerClient.start();

    return consumerClient;
}

public boolean isClosed() {
    return closed.get();
}

private void ensureOpen() {
    if (closed.get()) {
        throw new BrokerClientException("BrokerClient is closed");
    }
}

@Override
public void close() {
    if (!closed.compareAndSet(false, true)) {
        return;
    }

    channel.shutdown();

    try {
        boolean terminated = channel.awaitTermination(
            config.shutdownTimeout().toMillis(),
            TimeUnit.MILLISECONDS
        );

        if (!terminated) {
            channel.shutdownNow();
        }
    } catch (InterruptedException exception) {
        Thread.currentThread().interrupt();
        channel.shutdownNow();
    }
}

```

```

public ReconnectingPublisherClient openReconnectingPublisher() {
    ensureOpen();

    return new ReconnectingPublisherClient(
        config,
        ReconnectingPublisherOptions.defaults()
    );
}

public ReconnectingPublisherClient openReconnectingPublisher(
    ReconnectingPublisherOptions options) {
    ensureOpen();

    return new ReconnectingPublisherClient(
        config,
        options
    );
}

public ReconnectingConsumerClient openReconnectingConsumer(
    ConsumerOptions consumerOptions) {
    ensureOpen();

    ReconnectingConsumerClient client = new ReconnectingConsumerClient(
        config,
        consumerOptions,
        ReconnectingConsumerOptions.defaults()
    );

    client.start();

    return client;
}

public ReconnectingConsumerClient openReconnectingConsumer(
    ConsumerOptions consumerOptions,
    ReconnectingConsumerOptions reconnectOptions) {
    ensureOpen();

    ReconnectingConsumerClient client = new ReconnectingConsumerClient(
        config,
        consumerOptions,
        reconnectOptions
    );

    client.start();

    return client;
}

```

```

}

public final class QueueAdminClient {
    private final QueueServiceGrpc.QueueServiceBlockingStub stub;

    public QueueAdminClient(QueueServiceGrpc.QueueServiceBlockingStub stub) {
        this.stub = Objects.requireNonNull(stub, "stub is required");
    }

    public void declareQueue(String queuePath) {
        declareQueue(
            queuePath,
            QueueSettingsBuilder.persistentReliable(),
            QueueDeclarationOptions.failIfExists()
        );
    }

    public void declareQueue(String queuePath, QueueSettings settings) {
        declareQueue(
            queuePath,
            settings,
            QueueDeclarationOptions.failIfExists()
        );
    }

    public void declareQueueIfAbsent(String queuePath, QueueSettings settings) {
        declareQueue(
            queuePath,
            settings,
            QueueDeclarationOptions.useExisting()
        );
    }

    public void declareQueue(
        String queuePath,
        QueueSettings settings,
        QueueDeclarationOptions options
    ) {
        validateQueuePath(queuePath);
        Objects.requireNonNull(settings, "settings is required");

        QueueDeclarationOptions effectiveOptions = options == null
            ? QueueDeclarationOptions.failIfExists()
            : options;
    }
}

```

```

QueueOperationResponse response = stub.declareQueue(
    DeclareQueueRequest.newBuilder()
        .setQueuePath(queuePath)
        .setSettings(settings)
        .build()
);

if (response.getSuccess()) {
    return;
}

BrokerRemoteException exception = GrpcErrorMapper.toException(response.getError());

if (effectiveOptions.existsPolicy() == QueueExistsPolicy.USE_EXISTING
    && exception.errorCode() == ErrorCode.QUEUE_ALREADY_EXISTS) {
    return;
}

throw exception;
}

public void deleteQueue(String queuePath) {
    validateQueuePath(queuePath);

    QueueOperationResponse response = stub.deleteQueue(
        DeleteQueueRequest.newBuilder()
            .setQueuePath(queuePath)
            .build()
    );

    if (response.getSuccess()) {
        return;
    }

    throw GrpcErrorMapper.toException(response.getError());
}

private void validateQueuePath(String queuePath) {
    if (queuePath == null || queuePath.isBlank()) {
        throw new IllegalArgumentException("queuePath is required");
    }
}

}

public final class PublisherClient implements AutoCloseable {
    private final PublisherServiceGrpc.PublisherServiceStub stub;

```

```

private final BrokerClientConfig config;
private final ClientPublishIdGenerator publishIdGenerator = new ClientPublishIdGenerator();

private final ScheduledExecutorService keepAliveExecutor =
    Executors.newSingleThreadScheduledExecutor(
        new NamedThreadFactory("broker-publisher-keepalive", true)
    );

private final ConcurrentMap<Long, CompletableFuture<PublishReceipt>> pendingPublishes =
    new ConcurrentHashMap<>();

private final Object sendLock = new Object();
private final AtomicBoolean closed = new AtomicBoolean(false);

private volatile StreamObserver<PublisherRequest> requestObserver;

public PublisherClient(
    PublisherServiceGrpc.PublisherServiceStub stub,
    BrokerClientConfig config
) {
    this.stub = Objects.requireNonNull(stub, "stub is required");
    this.config = Objects.requireNonNull(config, "config is required");

    openStream();
    startKeepAlive();
}

public CompletableFuture<PublishReceipt> publish(PublishRequest request) {
    Objects.requireNonNull(request, "request is required");
    ensureOpen();

    long clientPublishId = publishIdGenerator.nextId();

    CompletableFuture<PublishReceipt> result = new CompletableFuture<>();
    pendingPublishes.put(clientPublishId, result);

    sendPublish(clientPublishId, request, result);

    return result.orTimeout(
        config.publishResponseTimeout().toMillis(),
        TimeUnit.MILLISECONDS
    );
}

public long publishFireAndForget(PublishRequest request) {

```

```

Objects.requireNonNull(request, "request is required");
ensureOpen();

long clientPublishId = publishIdGenerator.nextId();

PublisherRequest grpcRequest = toGrpcPublishRequest(clientPublishId, request);

send(grpcRequest);

return clientPublishId;
}

public int pendingPublishes() {
    return pendingPublishes.size();
}

public boolean isClosed() {
    return closed.get();
}

// private methods...

@Override
public void close() {
    if (!closed.compareAndSet(false, true)) {
        return;
    }

    keepAliveExecutor.shutdownNow();

    synchronized (sendLock) {
        try {
            requestObserver.onNext(PublisherRequest.newBuilder()
                .setClose(ClosePublisherAction.newBuilder()
                    .setReason("client closed")
                    .build())
                .build());
        } catch (Throwable ignored) {
        }

        try {
            requestObserver.onCompleted();
        } catch (Throwable ignored) {
        }
    }
}

```

```

        failAllPending(new BrokerClientException("PublisherClient closed"));
    }
}

public record BrokerClientConfig(
    String host,
    int port,
    boolean plaintext,
    Duration keepAliveInterval,
    Duration publishResponseTimeout,
    Duration shutdownTimeout,
    int maxInboundMessageSizeBytes,
    int keepAliveThreads,
    int handlerThreads,
    int reconnectThreads
) {
    private static final int DEFAULT_MAX_INBOUND_MESSAGE_SIZE = 16 * 1024 * 1024;

    public BrokerClientConfig {
        if (host == null || host.isBlank()) {
            throw new IllegalArgumentException("host is required");
        }

        if (port <= 0 || port > 65_535) {
            throw new IllegalArgumentException("port is invalid");
        }

        keepAliveInterval = requirePositive(keepAliveInterval, "keepAliveInterval");
        publishResponseTimeout = requirePositive(publishResponseTimeout,
"publishResponseTimeout");
        shutdownTimeout = requirePositive(shutdownTimeout, "shutdownTimeout");

        if (maxInboundMessageSizeBytes <= 0) {
            throw new IllegalArgumentException("maxInboundMessageSizeBytes must be greater than
zero");
        }
    }

    public static Builder builder() {
        return new Builder();
    }

    public static BrokerClientConfig plaintext(String host, int port) {
        return builder()
            .host(host)

```

```

        .port(port)
        .plaintext(true)
        .build();
    }

    private static Duration requirePositive(Duration value, String name) {
        Objects.requireNonNull(value, name + " is required");

        if (value.isZero() || value.isNegative()) {
            throw new IllegalArgumentException(name + " must be positive");
        }

        return value;
    }

    public static final class Builder {
        private String host = "localhost";
        private int port = 9090;
        private boolean plaintext = true;
        private Duration keepAliveInterval = Duration.ofSeconds(10);
        private Duration publishResponseTimeout = Duration.ofSeconds(30);
        private Duration shutdownTimeout = Duration.ofSeconds(5);
        private int maxInboundMessageSizeBytes = DEFAULT_MAX_INBOUND_MESSAGE_SIZE;
        private int keepAliveThreads = 2;
        private int handlerThreads = 2;
        private int reconnectThreads = 2;

        // Builder methods...

        public BrokerClientConfig build() {
            return new BrokerClientConfig(
                host,
                port,
                plaintext,
                keepAliveInterval,
                publishResponseTimeout,
                shutdownTimeout,
                maxInboundMessageSizeBytes,
                keepAliveThreads,
                handlerThreads,
                reconnectThreads
            );
        }
    }
}

```

```

}

public final class Delivery {
    private final long deliveryTag;
    private final long messageId;
    private final byte[] payload;
    private final String routingKey;
    private final Map<String, String> headers;
    private final boolean redelivered;
    private final int deliveryAttempt;
    private final AckOperations ackOperations;
    private final ConsumerAckMode ackMode;

    private final AtomicBoolean completed = new AtomicBoolean(false);

    public Delivery(
        long deliveryTag,
        Message message,
        boolean redelivered,
        int deliveryAttempt,
        ConsumerAckMode ackMode,
        AckOperations ackOperations
    ) {
        this.deliveryTag = deliveryTag;
        this.messageId = message.getId();
        this.payload = message.getPayload().toByteArray();
        this.routingKey = message.getRoutingKey();
        this.headers = Map.copyOf(message.getHeadersMap());
        this.redelivered = redelivered;
        this.deliveryAttempt = deliveryAttempt;
        this.ackMode = ackMode == null ? ConsumerAckMode.MANUAL : ackMode;
        this.ackOperations = Objects.requireNonNull(ackOperations, "ackOperations is required");
    }

    public void ack() {
        ensureAckAllowed();

        if (!completed.compareAndSet(false, true)) {
            throw new BrokerClientException("Delivery already completed");
        }

        ackOperations.ack(deliveryTag, messageId);
    }

    public void nack(boolean requeue) {

```

```

ensureAckAllowed();

if (!completed.compareAndSet(false, true)) {
    throw new BrokerClientException("Delivery already completed");
}

ackOperations.nack(deliveryTag, messageId, requeue);
}

// Getters...

private void ensureAckAllowed() {
    if (ackMode == ConsumerAckMode.NONE) {
        throw new BrokerClientException("Ack/Nack is disabled for this consumer");
    }
}
}
}

```

Лістинг В.2 — Повторне підключення клієнтів

```

public record ReconnectPolicy(
    boolean enabled,
    int maxAttempts,
    Duration initialDelay,
    Duration maxDelay
) {
    public ReconnectPolicy {
        if (maxAttempts < 0) {
            throw new IllegalArgumentException("maxAttempts must not be negative");
        }

        initialDelay = requirePositive(initialDelay, "initialDelay");
        maxDelay = requirePositive(maxDelay, "maxDelay");

        if (maxDelay.compareTo(initialDelay) < 0) {
            throw new IllegalArgumentException("maxDelay must be greater than or equal to
initialDelay");
        }
    }

    public static ReconnectPolicy disabled() {
        return new ReconnectPolicy(
            false,
            0,
            Duration.ofSeconds(1),
            Duration.ofSeconds(1)
        );
    }
}

```

```

public static ReconnectPolicy unlimited() {
    return new ReconnectPolicy(
        true,
        0,
        Duration.ofSeconds(1),
        Duration.ofSeconds(30)
    );
}

public static ReconnectPolicy limited(int maxAttempts) {
    return new ReconnectPolicy(
        true,
        maxAttempts,
        Duration.ofSeconds(1),
        Duration.ofSeconds(30)
    );
}

public boolean shouldRetry(int failedAttempts) {
    if (!enabled) {
        return false;
    }

    return maxAttempts == 0 || failedAttempts < maxAttempts;
}

public Duration delayForAttempt(int failedAttempts) {
    long multiplier = Math.max(1, failedAttempts);
    long delayMs = initialDelay.toMillis() * multiplier;

    return Duration.ofMillis(Math.min(delayMs, maxDelay.toMillis()));
}

private static Duration requirePositive(Duration value, String name) {
    Objects.requireNonNull(value, name + " is required");

    if (value.isZero() || value.isNegative()) {
        throw new IllegalArgumentException(name + " must be positive");
    }

    return value;
}
}

public final class ReconnectingConsumerClient implements AutoCloseable {
    private final BrokerClientConfig config;
    private final ConsumerOptions consumerOptions;
    private final ReconnectingConsumerOptions reconnectOptions;

    private final ExecutorService lifecycleExecutor =
        Executors.newSingleThreadExecutor(
            new NamedThreadFactory("broker-reconnecting-consumer", true)
        );
}

```

```

private final AtomicBoolean started = new AtomicBoolean(false);
private final AtomicBoolean closed = new AtomicBoolean(false);

private final Object lifecycleLock = new Object();

private BrokerClient brokerClient;
private ConsumerClient consumerClient;

public ReconnectingConsumerClient(
    BrokerClientConfig config,
    ConsumerOptions consumerOptions,
    ReconnectingConsumerOptions reconnectOptions
) {
    this.config = Objects.requireNonNull(config, "config is required");
    this.consumerOptions = Objects.requireNonNull(consumerOptions, "consumerOptions is
required");
    this.reconnectOptions = reconnectOptions == null
        ? ReconnectingConsumerOptions.defaults()
        : reconnectOptions;
}

public void start() {
    if (!started.compareAndSet(false, true)) {
        throw new BrokerClientException("ReconnectingConsumerClient already started");
    }

    lifecycleExecutor.submit(this::runLoop);
}

private void runLoop() {
    int failedAttempts = 0;

    while (!closed.get()) {
        try {
            openConsumer();

            failedAttempts = 0;

            waitUntilConsumerStops();

        } catch (Throwable throwable) {
            failedAttempts++;

            if (!shouldRetry(failedAttempts)) {
                closed.set(true);
                return;
            }

            sleepBeforeRetry(failedAttempts);
        } finally {
            closeCurrentResources();
        }
    }
}

```

```

    }
}

private void openConsumer() {
    synchronized (lifecycleLock) {
        closeCurrentResources();

        brokerClient = BrokerClient.connect(config);
        consumerClient = brokerClient.openConsumer(consumerOptions);
    }
}

private void waitUntilConsumerStops() {
    while (!closed.get()) {
        ConsumerClient current;

        synchronized (lifecycleLock) {
            current = consumerClient;
        }

        if (current == null || current.isClosed()) {
            return;
        }

        try {
            Thread.sleep(250);
        } catch (InterruptedException exception) {
            Thread.currentThread().interrupt();
            throw new BrokerClientException("Consumer wait interrupted", exception);
        }
    }
}

private boolean shouldRetry(int failedAttempts) {
    ReconnectPolicy policy = reconnectOptions.reconnectPolicy();
    return policy.shouldRetry(failedAttempts);
}

private void sleepBeforeRetry(int failedAttempts) {
    Duration delay = reconnectOptions.reconnectPolicy().delayForAttempt(failedAttempts);

    try {
        Thread.sleep(delay.toMillis());
    } catch (InterruptedException exception) {
        Thread.currentThread().interrupt();
        throw new BrokerClientException("Reconnect interrupted", exception);
    }
}

private void closeCurrentResources() {
    synchronized (lifecycleLock) {
        if (consumerClient != null) {
            try {

```

```

        consumerClient.close();
    } catch (Throwable ignored) {
    }

    consumerClient = null;
}

if (brokerClient != null) {
    try {
        brokerClient.close();
    } catch (Throwable ignored) {
    }

    brokerClient = null;
}
}

}

public boolean isClosed() {
    return closed.get();
}

@Override
public void close() {
    if (!closed.compareAndSet(false, true)) {
        return;
    }

    closeCurrentResources();
    lifecycleExecutor.shutdownNow();
}
}

```